CROSSMINER
Developer-Centric Knowledge Mining from Large Open-Source Software Repositories

Project Number 732223

D3.3 Text Representation - Final Version

Version 1.0
27 December 2018
Final

Public Distribution

Edge Hill University

Project Partners: **Athens University of Economics & Business**, **Bitergia**, **Castalia Solutions**, **Centrum Wiskunde & Informatica**, **Eclipse Foundation Europe**, **Edge Hill University**, **FrontEndART**, **OW2**, **SOFTEAM**, **The Open Group**, **University of L'Aquila**, **University of York**, **Unparallel Innovation**

Project Partner Contact Information

Athens University of Economics & Business Diomidis Spinellis Patision 76 104-34 Athens Greece Tel: +30 210 820 3621 E-mail: dds@aueb.gr	**Bitergia** José Manrique Lopez de la Fuente Calle Navarra 5, 4D 28921 Alcorcón Madrid Spain Tel: +34 6 999 279 58 E-mail: jsmanrique@bitergia.com
Castalia Solutions Boris Baldassari 10 Rue de Penthièvre 75008 Paris France Tel: +33 6 48 03 82 89 E-mail: boris.baldassari@castalia.solutions	**Centrum Wiskunde & Informatica** Jurgen J. Vinju Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 4102 E-mail: jurgen.vinju@cwi.nl
Eclipse Foundation Europe Philippe Krief Annastrasse 46 64673 Zwingenberg Germany Tel: +33 62 101 0681 E-mail: philippe.krief@eclipse.org	**Edge Hill University** Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk
FrontEndART Rudolf Ferenc Zászló u. 3 I./5 H-6722 Szeged Hungary Tel: +36 62 319 372 E-mail: ferenc@frontendart.com	**OW2 Consortium** Cedric Thomas 114 Boulevard Haussmann 75008 Paris France Tel: +33 6 45 81 62 02 E-mail: cedric.thomas@ow2.org
SOFTEAM Alessandra Bagnato 21 Avenue Victor Hugo 75016 Paris France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L'Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	

Document Control

Version	Status	Date
0.1	Table of Contents	5 October 2018
0.2	Background Section	23 October 2018
0.4	Sentiment and Emotion Sections	25 November 2018
0.9	Full draft Interim Version - For Internal Review	10 December 2018
1.0	Final QA version	27 December 2018

Table of Contents

Executive Summary

In order to use Natural Language Processing (NLP) tools, it is necessary to represent text in a format that computers are able to understand. Normally, text is represented using mathematical vectors, however there are other methods to represent text such as graphs. The choice depends on the task at hand, as some representations allow solving particular tasks faster and easier than others.

In this deliverable we focus on the design, development and evaluation of Natural Language Processing tools in CROSSMINER. Within our experiments, we explore how different vectorial representation can affect the performance of machine learning algorithms. Text vectors can be sparse, i.e. consist of a majority of zero values, or dense, i.e. contain mainly non-zero values. For many years, sparse representations were widely used for machine learning, especially in classification tasks. Nevertheless, in the last lustrum, dense representations have become more popular. Each vector representation model exhibits different characteristics, strengths and weaknesses. Therefore, in this Task 3.3, we explore how text representation affects the behaviour and performance of machine learning models, built for a variety of different text processing tasks.

More specifically, in this final report of the text representation study, we present the comparison between sparse and dense text representations with respect to five different Natural Language Processing tools developer for CROSSMINER: a sentiment analyser, an emotion classifier, a request/reply classifier, a content classifier and a severity classifier. These tools allow analysing different aspects of the information located in communication means commonly used by developers, such as issue trackers, forums and NNTP Newsgroups. Furthermore, the tools presented here are the core of multiple metrics that are used in CROSSMINER[1].

As part of the development of Natural Language tools for CROSSMINER, we designed and developed *VastText*, a novel and versatile neural network model specialised for text classification. VastText can address diverse classification problems, either single or multi-label, easily, quickly and efficiently. In this deliverable, we present how VastText has been used to solve multiple classification tasks found in CROSSMINER. Our experimental results have shown that VastText performs better or comparably well with existing state-of-the-art classification methods in a variety of tasks, while at the same time its computational and time requirements are lower that most neural networks used for text classification.

[1]Details about the integration, the functionality and the usability of the tools for the CROSSMINER platform are available on D3.4.

1 Introduction

Natural Language Processing (NLP) is a multi-disciplinary sub-field of Artificial Intelligence, Computer Science and Computational Linguistics that concerns methods and tools for analysing natural language text. A *natural language* is a language created by humans for communicating that evolved naturally through use and repetition without conscious planning. Some examples of natural languages are Spanish, English, Chinese and Mayan.

The research area of NLP was the result of the interest, and in many cases the necessity, of analysing text automatically and rapidly. In the beginning, just a few tasks were explored, such as machine translation and automatic text summarisation, however, with the evolution of computation technology and the increasing volume of readily accessible digital text, NLP was developed further. Nowadays, popular, actively researched NLP tasks are sentiment analysis, word sense disambiguation, multilingual summarisation and automatic term extraction, among others.

In CROSSMINER, the use of NLP tools is of high relevance, as the analysis of the text written by developers and users provides information that would be expensive and laborious to process manually. The outcome of the text processing tools developed and presented in this deliverable comprise the basis to compute all metrics of the CROSSMINER platform related to text analysis, such as the number of requests per day and the emotions or sentiments expressed in issue tracker comments.

This deliverable focuses on research and development of NLP tools designed for achieving CROSSMINER goals, i.e. extract useful information about the quality of support offered by the community of an open source software project to be made available not only to decision makers but also to software developers. More specifically, we explore sentiment analysis, classification of emotions, detection of request and replies among messages posted in a communication channel, bug tracker or forum, categorisation of messages according to their content type and the classification of threads of messages according to the severity of the issue that they express. These topics are in parallel used as applications where we evaluate how different text representation models affect performance.

Details about the implementation of the NLP tools discussed in the present deliverable and their integration into the CROSSMINER platform can be found in Deliverable 3.4. Moreover, deliverable 3.4 describes in depth how the NLP tools are used by the CROSSMINER platform to compute useful metrics and to enrich the knowledge base and by the users to run bespoke text processing workflows.

1.1 Overview

The remaining of this deliverable consists of 6 sections. In Section 2, we review background research related to the methods explored in this deliverable. In Section 3, we introduce VastText, a novel, versatile neural network model for text classification that was developed in house. Section 4 presents a sentiment analyser that was designed and developed specifically for the domain of software engineering and software development. In Section 5, we introduce an emotion classifier, i.e. a tool that determines the sentiments expressed in text that comes from sources frequently used by software developers. In succession, Section 6 presents the classifier that we have developed for the detection of request and replies among messages in forums, communication channels and, issue and bug trackers. Section 7 introduces the content classifier, a tool developed to identify the various types of content within comments, news group messages and forum posts. Section 8 presents a classifier capable of predicting the severity of a software bug/defect using information from the textual content of a bug report.

Then, in Section 9, we discuss the risks associated to this deliverable and ways to prevent them or to reduce their impact. Finally, Section 10 concludes this deliverable.

1.2 Intentions

The objective of this deliverable is to present five different NLP tools developed specifically for CROSSMINER and at the same time, to explore how text representations can affect the performance of NLP tools. In particular, we designed and developed three single-label classifiers, i.e. sentiment analyser, request vs. reply classifier and severity classifier, and two multi-label classifiers, i.e. emotion and content classifiers.

To achieve our objective, we have executed multiple experiments to comparatively evaluate state-of-the-art methods and resources. We have also developed a novel neural network model able to integrate features of diverse nature, executable on non-specialised equipment and flexible enough to address a variety of different classification tasks.

1.3 Outcome

The main outcome of this final version of the deliverable is a set of NLP tools that analyse different aspects of text that are posted in sources frequently used by software developers. More specifically, we have developed five classifiers: a sentiment analyser, an emotion classifier, a request and reply classifier, a content classifier and a severity classifier.

A second, equally important outcome of this work is a novel neural network model that can be easily employed in single-label and multi-label text classification tasks.

2 Background

This section presents a review concerning the background information that is relevant to all following sections of this deliverable. In particular, we discuss methods and tools that are in succession used or employed while designing and developing CROSSMINER NLP tools. We further elaborate, in subsequent sections, on how specific methods are adapted in the context of the different problems that these tools address.

2.1 Single-label Classification

Classification in machine learning is the task of categorising an instance, such as a document or set of values, with respect to previously defined categories or classes [46, page 119]. Single-label classification is used to define a process in which the instances are computationally categorised into only one of two or more classes. Thus, a single-label classification can be either binary (i.e. consist of two classes) or multi-class (i.e. consist of more than two classes). An example binary classification task is to determine the presence or absence of code within a document. With regard to multi-class classification, examples such as language detection or the classification of sentiment in a document are likely to produce more than two class labels. An alternative to this method is called multi-label classification, where one or more class labels may be assigned to each instance to be classified. In other words, there is no constraint on how many of the available classes an instance can be assigned to. Detailed description of multi-label classification is provided in Section 2.2.

In the literature, single-label classification is considered as a conventional classification task [31, 114]. There exists a large variety of methods that address the problem using different approaches. In this deliverable, which is focused on the comparison of sparse and dense text representation models, we make use of two classification methods, *Support Vector Machine* and *FastText*, which demonstrate the effect of different text representations on a given classification task. The classifiers are presented in depth in the following sections.

2.1.1 Support Vector Machine

A *Support Vector Machine (SVM)* [25] is a machine learning algorithm designed for binary classification problems. However, it can be used in multi-class tasks using a one-vs-rest strategy. The idea behind SVMs consists in mapping vectors, from a training data set, into a high dimensional space, i.e. a hyper-space, and finding the optimal hyperplane that can separate the vectors according to their class. New examples, such those found in a testing data set, are mapped into the same hyper-space to predict their class based on which side of the hyper-plane they fall into. The mapping of vectors is done using linear algebra operations through the application of kernels such as the Radial Basis Function (RBF), the linear or the polynomial. Due to their robustness and relative simplicity, SVMs have become a classic method in the literature for performing single-label classification tasks. Furthermore, SVMs support both dense or sparse vector representations, in line with our requirements.

In this deliverable, the experiments based on SVMs are done using LibSVM [17]. LibSVM, is a library that has been ported to different languages, like Java, Python and Perl. It is multi-thread and capable of implementing automatically a one-vs-rest strategy for multi-class problems.

More importantly, in this deliverable we make use of a linear SVM. In other words, we employ an SVM that uses a linear kernel for mapping the vectors into a high dimensional space. The reason to choose a linear SVM, instead of others such as the polynomial or the RBF kernel, is that the number of features to be represented into the hyperspace is always much greater than the number of classes [44].

2.1.2 FastText

Similarly to Deliverable 3.1, in this deliverable we explore FastText [49, 11], a library with twofold functionalities: creation of word embeddings and document classification. This classifier is based on a simple neural network which implements an improved version of Word2Vec [70, 69]. More specifically, in this work, we use FastText for document classification.

The neural network used by the FastText classifier consist of an input layer, an embedding layer, a global average pooling, a hidden layer and an output layer. FastText is able to create word embeddings specific to the classification task. Furthermore, it implements a linear classifier based on probabilities computed by a softmax function. One advantage of FastText is that it performs comparably better than the more complex deep learning algorithms, in less training time and without using a GPU [131]. Presently, FastText only supports input features in the form of text.

2.2 Multi-label Classification

Multi-label classification is the process in which a data instance can be categorised as belonging to one or more classes at the same time [31]. It contrasts single-label classification in which any given instance can only be categorised into one class. Examples of multi-label classification tasks are the genres of movies or books, the sentiments in a text and the classification of elements in images among others.

There are many methods for addressing multi-label classification tasks. In [30], the authors group these methods in three different groups, namely:

- **Algorithm adaptation:** This group consists of methods that have been created or have modified other algorithms to perform multi-label classification. Examples of these methods are *(Backpropagation for Multilabel Learning (BP-MLL)* [128] (neural network), *Clus* [10] (decision trees) or the *Multi-Label K-Nearest Neighbour (ML-kNN)* method [127] (k-nearest neighbours).

- **Problem transformation:** This group consists of methods that transform the multi-label task into multiple single label classification problems. There are two types of transformation. In the first transformation, better known as *Binary Relevance*, each instance annotated with several labels is duplicated multiple times, however, each of these instances is just annotated with one label. The second transformation is called *Label Powerset*, through which each of the possible multi-label combinations is given a unique label; normally, the possible combinations are only those seen during the training process.

- **Ensemble**: These algorithms try to improve the methods used in problem transformation by creating multiple classifiers that are trained on subsets of the original training data set. Examples of these methods are *HOMER*[115], *Rakel* [116] and *ECC* [95].

As it is impossible to experiment with all the multi-label classifier methods, we have decided to select two representative methods: *HOMER*, which implements a Binary Relevance transformation, and *RaKel*, that solves the multi-label problem through an Label Powerset transformation. These two methods have been implemented in the library Mulan [117]. Mulan was built over Weka [40], another library for machine learning, in order to extend its capabilities to multi-labels problems.

In the following subsections, we present the selected multi-label classification methods in detail.

2.2.1 HOMER

Hierarchy Of Multilabel classifiERs (HOMER) [115] is a machine learning algorithm that follows the principle of *divide-and-conquer*, in the sense that it uses multiple multi-label classifiers, arranged hierarchically, that only deal with a small set of labels. HOMER supports different multi-label algorithms, which in fact are single-label ones that uses transformed entries. In other words, HOMER uses a Binary Relevance transformation to convert the multi-label task into multiple single-label tasks. Once the transformation is done, it can apply other classical methods of machine learning suitable for single-label tasks. By default, *HOMER*, in the library *Mulan*, uses a C4.5 Decision Tree as classification method.

2.2.2 RaKel

RAndom k-labELsets (RAkEL) [116] is a multi-label classifier, that uses the same principle with HOMER, but finds correlations between labels to efficiently perform improved predictions. More precisely, RAkEL builds an ensemble of Label Powerset classifiers where the sets of labels have been selected randomly. As it happens with HOMER, RAkEL supports different single-label classifiers, however, in Mulan, RAkEL, uses a C4.5 Decision tree as its main classification method.

2.3 Optimisation of Classifiers

In order to get the best performance of any machine learning method, it is necessary to determine a series of parameters. These parameters can range from the size of n-grams to the learning rate used for the neural-network-based models. Specific combinations can generate models with different performance, therefore, a correct selection of these is essential. Numerous optimisation methods exist within the literature for parameter tuning. For the experiments presented in this deliverable, we used *Bayesian Optimisation* [71]. The Bayesian Optimisation method is based on a lazy learning that tries to model the hyper-surface that is generated by an objective function and a set of parameters. Then, the Bayesian Optimisation can determine the points in which an objective function is maximised. This optimisation method is known to perform better and be more efficiently than manual or brute force methods [104].

2.4 Evaluation Methods of Classifiers

In this section, we present the evaluation metrics used to evaluate the classifiers presented in this deliverable. These have been divided into two groups as different metrics apply to single and multi-label classifiers.

2.4.1 Metrics for Single-label classifiers

Single-label classifiers are commonly evaluated using metrics based on confusion matrices. The work reported in this deliverable is no exception. In Table 1, we present the structure of a general confusion matrix, where I_{nm} denotes the number of instances that belong to class n and were predicted as class m.[2] A confusion matrix with zero off-diagonal values corresponds to the evaluation of a method that performs ideally, without any classification errors. Although Table 1 shows a confusion matrix for two classes, it can be extended to accommodate as many classes as necessary, depending on the dataset being evaluated.

[2] In some works, the predicted and true conditions can be positioned at the opposite axes.

Table 1: General structure of the confusion matrix used to define evaluation metrics for single-label classification. I_{nm} denotes the number of instances that belong to class n and were predicted as class m.

		Predicted condition	
		Class_1	Class_2
True Condition	Class_1	I_{11}	I_{21}
	Class_2	I_{12}	I_{22}

We present in the following paragraphs, a brief description of each evaluation metric that is used in this deliverable for evaluating single-label classifiers and that are deduced from confusion matrices.

Precision of a single-label classifier for a determined class n is defined in Equation 1.

$$\text{Precision}(n) = \frac{I_{nn}}{\sum_{m=0}^{c} I_{nm}} \tag{1}$$

where I_{nn} is the number of instances that were correctly predicted for class n, c is the total number of classes present in the data set and I_{nm} is the number of instances of class n that were predicted as class m.

The *Recall* of a class n in a single-label classifier is defined in Equation 2.

$$\text{Recall}(n) = \frac{I_{nn}}{\sum_{m=0}^{c} I_{mn}} \tag{2}$$

where I_{nn} is the number of instances that were correctly predicted for class n, c is the total number of classes present in the data set and I_{mn} is the number of instances of class m that were predicted as class n.

The *F-Score* or *F-1* is the harmonic mean of Precision and Recall. We present its definition in Equation 3.

$$\text{F-Score}(n) = 2 \cdot \frac{\text{Precision}(n) \times \text{Recall}(n)}{\text{Precision}(n) + \text{Recall}(n)} \tag{3}$$

In the literature, we can also find *Macro* and *Micro* versions of F-Score[3]. The former is the averaged F-score. More specifically, we present its definition in Equation 4.

$$\text{Macro F-Score} = \frac{\sum_{n=0}^{c} \text{F-Score}(n)}{c} \tag{4}$$

where c is the number of classes present in the data set. The Macro F-Score indicates how well the classifier performs on all classes regardless of the class size.

As shown in Equation 5, *Micro F-Score* is a weighted average, i.e. considers the proportion of each class in the data set.

$$\text{Micro F-Score} = \frac{\sum_{n=0}^{c} I_{nn}}{\sum_{n=0}^{c} \sum_{n=0}^{c} I_{nm}} \tag{5}$$

It should be noted that the Micro F-score is also known as *Accuracy*.

[3]There are also *Micro* and *Macro* versions of Precision and Recall. Macro versions calculate the average over all classes. Micro versions work similarly to Micro F-Score, as shown in Equation 5. However, the Micro and Macro versions of Precision and Recall are not used in this deliverable.

2.4.2 Metrics for Multi-label classifiers

Difficulties of multi-label classification tasks reside not only in the development of methods, but also in the evaluation of the results. A first problem arises when determining which labels are predicted by the classifier. This is because the great majority of methods are probabilistic, i.e. expresses the output as a vector with weights from 0 to 1. Therefore, it is necessary to establish a threshold of activation above which a label will be considered as predicted or not [114]. By convention a threshold of 0.5 is set to activate a label. However, it might be the case that multiple thresholds should be used for different labels. A second problem is evaluating the correctness of multiple labels assigned to an instance. In [121], the authors summarise the problem of evaluating multi-label classifiers as "it is difficult to tell which of the the following mistakes is more serious: one instance with three incorrect labels vs. three instances each with one incorrect label".

Therefore, in the literature we can find different evaluation metrics that are used for the assessment of multi-label classifiers. The evaluation metrics of a multi-label classifier, unlike single-label ones, do not use confusion matrices, but sets of vectors that represent all instances and labels. Below, we summarise how these sets of vectors are used, along with the most representative evaluation metrics for multi-label classification:

- ***Hamming Loss***: This evaluation metric is based on the Hamming distance. It calculates how different the prediction is from the expected outcome. Equation 6 shows its mathematical definition:

$$\text{Hamming Loss}(T, P) = \frac{1}{nl} \sum_{i=1}^{n} \sum_{j=1}^{l} t_{ij} \oplus p_{ij} \tag{6}$$

  where n is the number of instances tested, l is the number of different labels, $T = \{t_1, t_2, \ldots, t_n\}$ is the ground truth for every instance t and $P = \{p_1, p_2, \ldots, p_n\}$ is the prediction p of every instance. Each t or p is a binary vector of size l, where 1 means that the label is activated and 0 that the label is not triggered off. When a label is predicted incorrectly, a 1 is added into the loss function.

  Hamming Loss considers all mistakes as equally important. For example, predicting that a text expresses anger instead of love is equally wrong as predicting that a text expresses love instead of joy. Furthermore, Hamming Loss can be affected by unbalanced corpora, i.e. wrong prediction of less frequent labels can be underestimated.

- ***Subset 0/1 Loss***: In contrast to Hamming Loss, this loss metric is strict. More specifically, a prediction with at least one incorrect label is considered as wrong. Mathematically it is defined as shown in Equation 7:

$$\text{Subset 0/1 Loss}(T, P) = \frac{1}{n} \sum_{i=1}^{n} t_i \oplus p_i \tag{7}$$

  where n is the number of instances tested, $T = \{t_1, t_2, \ldots, t_n\}$ is the ground truth for every instance t and $P = \{p_1, p_2, \ldots, p_n\}$ is the prediction p of every instance. When the instance p_i is different to t_i, a 1 is added into the loss function.

  Similarly to Hamming Loss, this metric is also problematic in corpora with unbalanced occurrences of labels.

- ***Multi-label Macro F-score***: It is a metric that evaluates how accurately every label has been predicted by the classifier. Despite the *macro* in its name, this metric can be considered strict as it takes into account the proportion of labels within the dataset. Equation 8 shows its mathematical definition:

$$\text{Multi-label Macro F-Score}(T, P) = \frac{1}{l}\sum_{i=1}^{l} 2\frac{\sum_{j=1}^{n} t_{ij} \cdot p_{ij}}{\sum_{j=1}^{n} t_{ij} + p_{ij}} \tag{8}$$

  where n is the number of instances tested, l is the number of different labels, $T = \{t_1, t_2, \ldots, t_n\}$ is the ground truth for every instance t and $P = \{p_1, p_2, \ldots, p_n\}$ is the prediction p of every instance. The elements t and p represent binary vectors of length l, in which an activated label is indicated with 1.

- ***Multi-label Micro F-score***: It is a metric that evaluates how well every instance has been predicted by the classifier. It is a less strict metric in comparison to Multi-label Macro F-score. Equation 9 presents its definition mathematically:

$$\text{Multi-label Micro F-Score}(T, P) = 2 \cdot \frac{\sum_{i=1}^{l}\sum_{j=1}^{n} t_{ij} \cdot p_{ij}}{\sum_{i=1}^{l}\sum_{j=1}^{n} t_{ij} + p_{ij}} \tag{9}$$

  where n is the number of instances tested, l is the number of different labels, $T = \{t_1, t_2, \ldots, t_n\}$ is the vector representing the ground truth for each instance and $P = \{p_1, p_2, \ldots, p_n\}$ is the vector of predictions related to each instance. The elements t and p represent binary vectors of length l, in which an activated label is indicated with a 1.

2.5 Core Natural Language Processing Tools

Complex NLP tools can be developed using functionalities from cores ones. Below, we discuss core NLP tools that are relevant for the implementation of the methods and experiments present in this deliverable:

- *Tokeniser*: Also known as lexical analyser, is a tool designed to split a string into tokens. A token is a set of characters that have a meaning by themselves [46][Page 10]. Although a simple tokeniser would split a string by white space, a more advanced tokeniser may use other sources and techniques to separate elements such as punctuation and abbreviations.

- *Part-of-Speech Tagger*: It is a tool that tags every token with its own Part-of-Speech (PoS) tag, e.g. noun, verb, preposition. A Part-of-Speech tagger can either be based on rules or be trained on large annotated datasets to determine probabilities of tokens belonging to a specific POS category.

- *Lemmatiser*: It is a tool for identifying the lemmas, i.e. the dictionary forms of words. Lemmatisers are frequently based on rules and dictionaries.

- *Stemmer*: It is a tool that uses heuristics to delete prefixes and suffixes of words in order to find its root. Stemmers are faster than a lemmatisers, however they are less accurate, especially in highly inflected languages, such as Spanish or French.

In the literature we can find multiple libraries that offer core NLP tools, such as *Freeling* [85], *Stanford CoreNLP* [62] and *NLTK* [57]. However, among all the libraries and to the best of our knowledge, only

Apache OpenNLP[4] and *NLP4J*[5] are Java-based and have an open source licence that is compatible with the CROSSMINER project.

Apache OpenNLP, is a project that was originally created by *Jason Baldridge* and *Thomas Morton*. The goal was to create a Java-based open source library of NLP tools. In 2010 the project became part of the *Apache Foundation*. Currently, Apache OpenNLP is a library that allows training and using different basic NLP tools, like tokenisers, part-of-speech taggers, named entity recognisers, co-reference resolution systems and parsers. For some languages, such as English, Danish and Spanish, Apache OpenNLP provides a series of pre-trained models[6]. They are considered just for testing purposes and, therefore, Apache OpenNLP developers suggest users to train their own models.

NLP4J, is the successor of a library called *ClearNLP* which was originally created by *Jinho D. Choi*. Currently, it is a project supported by the NLP group of Emory University. It provides the following tools: tokeniser, lemmatiser, part-of-speech tagger [20], name-entity recogniser and dependency parser [21]. Although, it provides fewer tools than Apache OpenNLP, NLP4J offers pre-trained models for English.

We have decided to use NLP4J instead of Apache OpenNLP for the following reasons: firstly, the models of Apache OpenNLP are indicated to be available just for testing purposes. Moreover, it is not clear which dataset was used for training the models as there are no publications about them. In contrast, we know that NLP4J's models were trained using different sources that cover domains like news, web and health[7]. Secondly, during some experiments for the sentiment analyser, we found out that part-of-speech tagger of Apache OpenNLP was considerably slower than the one provided in NLP4J.

[4] opennlp.apache.org

[5] emorynlp.github.io/nlp4j/

[6] It should be noted that there is a lack of publications or clear indication about which corpora the models were trained on. In fact, the project's website states: "Trained on OpenNLP training data." In the old forums of OpenNLP, we found a discussion in which Thomas Morton indicates vaguely that the version 1.3 of the models were trained using the Penn Treebank [63], portions of the Brown Corpus [52] and other files that were not specified. This discussion can be found in: sourceforge.net/p/opennlp/discussion/9942/thread/efd75b3b/

[7] The list of sources used for training can be found at: emorynlp.github.io/nlp4j/supplements/english.html

3 VastText

As part of Task 3.1, we have developed a neural network called VastText. This neural network has been conceived to solve classification problems of either single or multi-label nature. VastText is complementary to the classification methods previously described in Section 2.

VastText is a simple neural network based on the ideas of FastText [49, 11]. It has been extended to support features other than text, which is often useful in a variety of classification problems. More specifically, VastText uses word embeddings, created particularly for a classification task, similarly to FastText, but at the same time that can take into account extra numerical features, i.e. observations that can improve document classification.

Besides, unlike FastText that is a single-label classifier, VastText has been designed to solve either single and multi-label classification problems. Due to VastText's implementation in DeepLearning4j[8], it can run either on CPU or GPU, making it a versatile neural network. Owing to its characteristics, the name of this neural network includes the word "vast".

The architecture of VastText consist in two input layers, one dedicated for text and one for the extra numerical features. Furthermore, it includes one embeddings layer, one global average pooling layer, a merger layer, a dense layer and an output layer. In Figure 1 we present the architecture of VastText. As it can be observed in Figure 1, the architecture of VastText can be divided in 3 sectors:

- The first one is dedicated to the creation of embeddings. Thus, its input is a set of one-hot vectors[9] of tokenised texts that are converted into embeddings using the embedding layers. Documents are represented using a unique embedding vector through the global average pooling layer.
- The second sector is dedicated to the extra numerical features and receives a numeric vector.
- The third sector consists of three interconnected layers, i.e. the *merger*, the *dense* and the *output* layer. These are explained as follows:
  - The *merger layer* concatenates the document embedding vector and the numeric vector that represents the extra numerical features.
  - The *dense layer* reduces the size of the merged vector by halving the size of the document embeddings.[10]
  - Finally, the *output layer* provides the prediction vector that represents which classes have been activated. The output layer, can be modified depending on whether the classification task is single or multi-label:
    - In the former case, we make use of Softmax as activation function and categorical cross-entropy as loss function.
    - In the latter case, we use a Sigmoid as activation function and a binary cross-entropy as loss function.

[8] Eclipse DeepLearning4j is an open-source and Java-based library for the creation of neural networks. deeplearning4j.org

[9] A one-hot vector is such that only one component of the vector is activated, i.e. value set to one. In our context, the activated component makes reference to one token, i.e. word or n-gram, found in the text. The set of one-hot vectors is as large as the number of tokens found in a text.

[10] Although the reduction of the vector size by half the size of the document embeddings vector was chosen empirically, it was done to condense the information merged in the previous layer. In other words, we allow the neural network finding the relations between the numeric and textual features by reducing the output of the dense layer.

The selection of the activation and loss functions are based on practices found in the state-of-the-art [22].

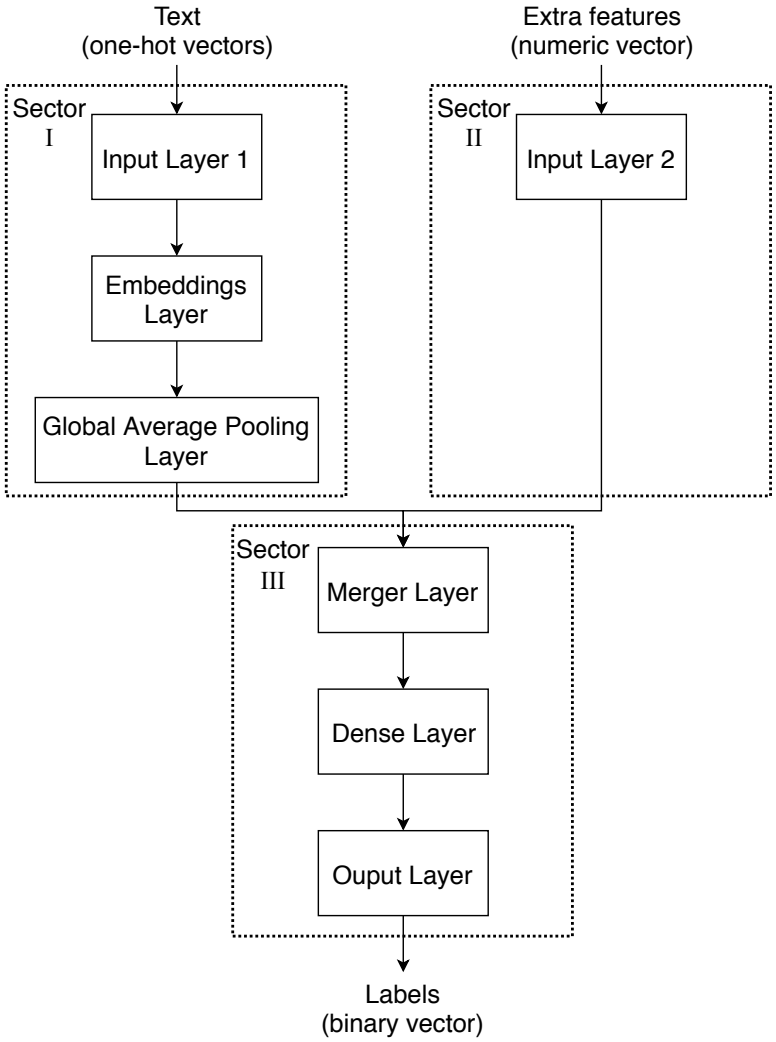

Figure 1: Architecture of the neural network VastText. It is composed of three sectors, the first one for creating word embeddings, the second one for accepting extra numeric features, and the third one to merge both inputs and solve the classification problem.

Following the philosophy of FastText to provide easy access to text classification tasks using neural networks, we include in VastText a vectoriser, a stop-word remover, an n-gram generator, a skip-bigrams generator and a library with different evaluation metrics used in single and multi-label classification problems. In the following list, we present a description of the extra elements that we provide in VastText:

- *Vectoriser*: It is the tool responsible for converting the document text into a numerical representation that can be fed into the neural network. The vectoriser can reduce the vocabulary used for training by setting a minimum occurrence frequency threshold but also through the stop-word remover. In addition, the vocabulary can be extended by calling the n-gram and skip-bigram generator.
- *Stop-word remover*: This optional tool is responsible for removing words that are frequently considered as non content-bearing, such as prepositions.

- *n-gram and skip-bigram generator*: Depending on the parameters set by the users, this tool can generate n-grams of different size and skip-bigrams with holes of different sizes.
- *Evaluation library*: In VastText, we created a library of the evaluation metrics that we have indicated in Section 2.

VastText has been used in most tasks of this deliverable to address different types of classification problems, such as sentiment analysis and emotion classification.

4 Sentiment Analysis

According to [81], *Sentiment Analysis* is the process of computationally determining the polarity orientation, i.e. positive, negative or neutral, of opinions expressed in a text. For instance, a sentiment analyser tool should determine that a phrase such as "I really like CROSSMINER project" has a positive polarity, while a phrase like "I have a bug in my code that I can't solve" has a negative polarity.

Sentiment analysis tools have been applied in areas such as movies reviews [87], tweets [98] or politics [119] to determine the writer's attitude towards the topic of interest. However, the increasing popularity of bug tracking systems, software repositories and specialised question and answering websites, have drawn the attention of researchers towards the software engineering domain. Among them, Guzman et al. [39] and Pletea et al. [90] explored how sentiments are expressed in Github software repositories. However, they quickly noticed that applying sentiment analysis tools originally developed for general domains do not perform well on technical areas like software engineering [48]. Therefore, in the last couple of years, the scientific community has started to developed specialised tools that are created specifically for the software engineering domain [45, 3, 13, 55].

In this section, we present a sentiment analyser, designed for the software engineering and software development domain. The sentiment analyser developed for CROSSMINER is based on VastText, a neural network model that makes use of word embeddings and extra numerical features. The extra numerical features are based on values regarding textual aspects such as number of exclamations and question marks, as well as the polarity strength of words defined by a lexicon.

The rest of this section is subdivided as follows. In Section 4.1, we present the state-of-the-art regarding the computational analysis of sentiments. Section 4.2 introduces a series of lexica that can be found in the literature and form the basis of multiple sentiment analysers. In Section 4.3, we explain the approach utilised by OSSMETER for detecting sentiments. In succession, Section 4.4 and Section 4.5 introduce respectively the methodology and the datasets utilised for the development and evaluation of the sentiment analyser. The experimental, comparative and evaluative settings are presented in Section 4.6. In Section 4.7 we show the results obtained by the developed sentiment analysers, and discuss their outcomes in Section 4.8. Finally, we conclude in Section 4.9.

4.1 State-of-the-Art

As indicated earlier, sentiment analysis is an important area of research within the natural language processing. Therefore, it is not surprising that numerous methods have been proposed in the state-of-the-art to solve the associated problems.

According to [123], sentiment analysers can be developed using two types of data: annotated and unannotated. The former lends itself to machine learning techniques because researchers have access to datasets with possible polarity of textual content. In the latter, the polarity of texts are unknown *a priori* so techniques like lexicon expansion or domain adaptation are used. The lack of annotated text happens frequently when a sentiment analyser is being developed for a new domain or language. In the following paragraphs, we introduce the most representative tools and methods that have been developed in the last decade regarding the computational analysis of sentiment in texts.

Semantic Orientation Calculator better known as *SO-CAL* [110] is a lexicon-based tool for sentiment analysis. Its development started in 2004 [111] and multiple versions of SO-CAL have been published, always trying to improve its performance. The last version of SO-CAL, makes use of different lexica for adjectives, verbs, nouns and adverbs, and also includes a list of intensifiers, i.e. a list of terms that can indicate if a context is non-

factual. In addition, SO-CAL can modify how scores are calculated on different portions of text, e.g. boosted weights at the end of a phrase and negative expressions can be boosted.

Among the most used tools for sentiment analysis is *SentiStrength*[11] [112] which works similarly to *SO-CAL*. SentiStrength is designed to determine the strength of positive and negative sentiments in short but informal communication channels, like Twitter, YouTube or MySpace. The tool is unsupervised and based on different lexica, e.g. idioms, booster words, emoticons, but also on other linguistic features as well as rules, e.g. sentence with exclamations marks or repeated letters.

Stanford CoreNLP is a suit of tools designed by the NLP group of Stanford University which contains among its tools a sentiment analyser developed by [105]. The tool was developed to address the problem that words can have different polarity depending on their surrounding textual elements such as other words, emoticons or even punctuation. More importantly the position of the surrounding elements play a key role in determining the correct sentiment of a text. To solve the problem the authors firstly annotated a set of *parse trees* obtained from film reviews; and then created a recursive neural network called *Recursive Neural Tensor Network*. Thanks to parsing trees and the recursive neural network, the sentiment analyser is able to improve the correct prediction of phenomena as sentences with a contrastive conjunction ("Repetitive movie but it has very good jokes") or the negation of negative sentences ("The movie wasn't that terrible").

Apart from these sentiment analysers, in the literature we can find other tools developed during challenge campaings such as *SemEval 2017* [98], in which researchers test different approaches and methods to determine the sentiment of short texts like tweets. According to the event organisers, there were 48 participants of which 24 used deep neural networks, like *Convolution Neural Networks (CNN)* or *Long-Short Term Memory Networks (LSTM)*, and a good number of the remainder used a combination of SVM and neural networks. The top three analysers from the event are as follows:

- *DataStories* [8] is a system that implements a Long Short-Term Memory (LSTM) network using pre-trained word embeddings as input. One characteristic of this system is that it does not use any kind of manual features, like lexicon-based elements. Instead, it performs a series of text preprocessing to identify spelling mistakes and normalise the text.
- *BB_twtr* [23] is a sentiment analyser built using two different deep neural networks, a Convolutional Neural Network (CNN) and a LSTM. The system uses word embeddings that were trained on unlabelled data but fine-tuned using a labelled corpus.
- *LIA* [99] is similar to BB_twtr, as it is formed by a CNN and a LSTM, which are merged at a score level. As input, the neural networks needs a set of word embeddings but also some extra features which are lexicon-based and token-based (e.g. elongated words, capital letters, among others).

Concerning sentiment analysers developed for the domains of software development and software engineering we can name *SentiStrength-SE* [45], *SentiCR* [3], *Senti4SD* [13] and *Stanford CoreNLP SO* [55]. These tools are described in the following paragraphs:

In first place we have *SentiStrength-SE*. This tool developed by [45] is a modified version of *SentiStrength* [112] that was created to increase the sentiment detection in technical domains like software engineering. To develop SentiStrength-SE, the authors analysed how SentiStrength behaved when it was applied on an annotated corpus based on comments extracted from JIRA issue tracker[12]. Based on this analysis, the authors identified the

[11] In this deliverable, we focus uniquely on the second version of SentiStrength, which is the current one and has been tested in different domains and datasets.

[12] The corpus used was initially created by [83] for the emotion detection in software development domain. The authors of SentiStrength-SE extrapolated the emotions from the corpus to represent sentiments, either positive or negative

difficulties or reasons why SentiStrength misclassified the sentiment of JIRA's comments. They concluded that SentiStrength had problems with aspects like domain-specific vocabulary, misinterpretation of the letter *x*, repetition of consecutive numbers, among others. To surpass the problems, the authors proposed SentiStrength-SE, which uses the following modifications: domain-specific dictionary, a contextualiser, extended lexicon of words/sentiments, list of neutralising words, improvement of the preprocessor and a parameter to change the handling of negations.

In the case of *SentiCR* [3], this tool is a supervised sentiment analyser that is trained on code review comments using a *Gradient Boosting Tree* and an oversampling technique. SentiCR removes stops words and code from the text; as well as pre-process the text in order to handle different emoticons, negations, contractions and inflections. The pre-processing is done using a stemmer. SentiCR represents texts in form of sparse vectors weighted using a TF-IDF approach.

Senti4SD [13] is another tool developed specifically for the detection of sentiment in the domain of software development. To develop Senti4SD, the authors generated a gold standard[13] based on StackOverflow comments. Senti4SD is based on a linear SVM which uses different types of features, namely:

- Lexicon-based: Set of features derived from SentiStrength lexicon for determining textual aspects like number of positive tokens, detection of emoticons, score of positive tokens, among others.
- Keywords-based: These features determine whether the document to analyse contains uppercased words, expressions of laugh, elongated words, or repetitions of exclamation marks. In addition, these features include the use of unigrams and bigrams.
- Semantic-based: For the semantic-based features, the authors generated a Word2Vec model based on 20M posts from StackOverflow. This model was trained using CBOW and 600 dimensions. With the Word2Vec model, the authors firstly generated prototype vectors that represents positive, neutral, negative and subjective polarities. These prototype vectors are the sum of the vectors given by the Word2Vec model of each word found in SentiStrength lexicon, to be either positive, neutral or negative[14]. Then, for each document to analyse, they generate a prototype vector using all the words and calculate the similarity with the prototype vectors of each polarity.

Although Senti4SD uses word embeddings for the calculation of semantic-based features, the vectors used for the linear SVM are sparse.

In [55], the authors explore the retraining of the sentiment analysis tool from Stanford CoreNLP on an annotated corpus based on StackOverflow, to create *Stanford CoreNLP SO*. To achieve this, the authors annotated 1,500 sentences from StackOverflow posts. Despite being trained and tested on domain-specific texts the performance of Stanford CoreNLP SO never surpassed the performance of other tools from the state-of-the-art, like Stanford CoreNLP and SentiStrength-SE.

4.2 Lexica as extra resource

As observed in the state-of-the-art, a great number of systems make use of lexica to work. A lexicon, is a specialised dictionary that gathers a series of entries that have been annotated previously. In the domain of sentiment analysis, the entries can be annotated according to their polarity i.e. negative, neutral or positive, and

[13] A gold standard is a corpus, frequently annotated by experts manually, that it is considered as a high quality reference.
[14] The subjective is the set of positive and negative words

according to their polarity strength, among other elements. Numerous lexica have been developed through the time, with some using manual techniques while others use completely automated methods. In the following paragraphs we present the most representative lexica found in the state-of-the-art:

General Inquirer (GI) [106] is a lexicon that was annotated to reflect different semantic aspects of words. Among the different annotations are: positivity, pain, domain, human collectivities, animal-related, persistence, quality. Its richness and annotations have been extended over the years.

Another lexicon created for sentiment analysis is the one presented in [56]. This lexicon was created manually with approximately 6.5K entries that are classified either as positive or negative. To create the lexicon, the authors analysed documents related to e-commerce and used *WordNet* [34] to enrich the vocabulary with synonyms. The lexicon includes misspelled words to increase its coverage.

In [120] the authors presented a lexicon that was created to determine the subjectivity of sentences as well as their polarity. The lexicon was annotated with different elements, for example, it indicates the subjectivity of the word (weak or strong), if it belongs to a specific *Part-of-Speech* and the word's polarity (positive, negative, both or neutral). The data used to create this lexicon comes from different sources, such as the General Inquirer.

The *NRC Word-Emotion Association Lexicon* [75] is a crowd-sourced annotated lexicon that indicates the emotions (anger, fear, anticipation, trust, surprise, sadness, joy and disgust) and polarity (positive or negative) associated to a list of words from other lexica: *WordNet Affect Lexicon* [108], General Inquirer and *Affective Norms for English Words (ANEW)* [12]. Similarly, the *NRC Hashtag Sentiment Lexicon* [74] is an automatically annotated lexicon created specifically for the sentiment analysis of tweets. The annotation was done using a large set of tweets, a set of 78 seeds, i.e. words with a well-known polarity, the detection of hashtags and by using *Pointwise Mutual Information* as a measure.

Another example of lexicon is the one created by [37] and currently available as part of Sentiment140[15]. This lexicon was created automatically using tweets, similarly to the NRC Hashtag Sentiment Lexicon. The main difference is that instead of using hashtags they made use of emotions to determine the polarity of tweets.

SentiWordNet 3.0 [5] is a lexicon created for the development of sentiment analysis tools. It is the results of the automatic annotation of WordNet according to their degree of positivity, negativity or neutrality. The automatic annotation consists of three processes: a weak-supervision, a semi-supervised learning and a random-walk. In SentiWordNet 3.0 only words considered as verbs, adverbs, nouns and adjectives are annotated. It should be noted that a word in SentiWordNet 3.0 can have both negative or positive polarity. However, the sum of its scores cannot be greater than 1 or less than 0. If the score is 0 for positivity and negativity, it means that the words is neutral.

In *SentiStrength*, the authors make use of different lexica, among them we can name those related to positive and negative polarities. The lexica come from the first version of SentiStrength and also parts of the General Inquirer lexicon. Every entry of the lexicon, either for positive or negative polarity, has a score between 1-5, where 1 means weak or subjective polarity, and 5 expresses a strong polarity. As SentiStrength and SO-CAL work in similar ways, their lexica have similar characteristics. In other words, SO-CAL's lexicon was built using different documents in addition to the General Inquirer lexicon.

Another lexicon is SenticNet, which is currently in its 5th version [16] and contains $100,000$ different entries, that range from unigrams to pentagrams. SenticNet5 uses a LSTM neural network to extend the number of entries from previous versions by discovering *conceptual primitives*, i.e. ensembles of verb-noun pairs. Each entry of SenticNet5 is annotated with different aspects such as polarity (positive or negative); polarity strength; moods such as surprise, interest, disgust; and related concepts. Moreover, it includes elements from

[15] sentiment140.com

the Hourglass of Emotions (HOE) [15], thus entries are annotated with weights in terms of *pleasantness*, *attention*, *sensitivity* and *aptitude*.[16]

Evidently from the preceding discussion, several lexica used for sentiment analysis have been created to cover general, domain-independent contexts. However, the polarity of the words can vary for many reasons such as language evolution, demographics or even domain [41, 126, 43]. Thus, when lexica created for general, domain-independent topics are applied in certain and specific contexts, they can mislead or bias the sentiment as they do not consider language variations. For example, [41] indicate that the word *soft* does not have the same connotation, and in consequence, polarity, in domains related to sports and to kids toys. Moreover, according to [43], the inclusion of domain-specific aspects in a sentiment analyser can improve their performance.

Taking into account these aspects, we have decided to explore whether the use of a domain-specific lexicon can improve the performance of a sentiment analyser. More specifically, we have decided to use the lexicon for the *programming* domain created by SocialSent [41]. This specialised lexicon was generated automatically by using domain-specific word embeddings, label propagation and a small set of seed words. The specialised texts came from Reddit, a social medium, which includes varied communities, one of which is related to programming.

4.3 OSSMETER Approach

OSSMETER, CROSSMINER's predecessor project, included a sentiment analyser based on machine learning methods and used external resources and manually selected features. More specifically, the sentiment analyser present in OSSMETER is a Radial Basis Function SVM trained on the dataset used in SemEval 2013 [78], i.e. tweets labelled either as positive, negative or neutral. The tool made use of the following manual features:

- Word n-grams- This feature indicates only the presence or absence of words unigrams, bigrams, trigrams and tetragrams.
- Non-contiguous n-grams- This binary feature that indicated the existence or absence of the following non-contiguous n-grams: unigram-unigram pairs, bigram-bigram pairs, unigram-bigram pairs, bigram-unigram pairs.
- Character n-grams- This feature indicates the presence or absence of trigrams, tetragrams and pentagrams of characters.
- Capital letters- This numeric feature indicated the umber of words that are all in capital letters, like "LOL" or "AMAZING".
- Part-of-Speech- This feature counts the occurrences of each Part-of-Speech.
- Contiguous punctuation- This feature indicates the number of contiguous exclamation or question marks.
- Last punctuation- This binary feature that indicates whether the last token is linked to an exclamation or question mark.

[16] According to [15], there is a relation between pleasantness, attention, sensitivity and aptitude in terms of polarity and emotions.

- Elongated words- This numeric feature indicates the numbers of words that are elongates, e.g. “Thaaaanks”.
- Negation- This feature specifies the number of negations found in the text.

Regarding the use of external resources, the sentiment analysis tool in OSSMETER made use of the following lexica, previously described in this deliverable: [56], [120], [75], [74] and [37]. In addition, the sentiment analyser present in OSSMETER used the CMU words clusters [84], to determine the presence or absence of tokens. The CMU words cluster provides $1,000$ groups of words that are spelled similarly. It was created using the Brown Corpus [52] and tweets.

4.4 Methodology

In this deliverable we explore three different machine learning methods: FastText, Linear SVM and VastText. As we explained in Section 2, FastText is a simple neural network based on embeddings; a linear SVM is a classic machine learning method based on the mapping of vectors into a high dimensional space using a linear kernel; and VastText, Section 3, is a neural network that uses embeddings similarly to FastText, but supports the use of extra numerical features.

For every machine learning method, we test whether the use of lemmatised text has an effect on the performance.[17] In addition, we determine how different numbers of word n-grams (from unigrams to trigrams) and word skip-bigrams[18] (a skip hole between 2-4), along with a minimum threshold of occurrence, affect the outcome of a sentiment analyser created for software engineering. The input of every method is a tokenized text, which has been previously normalised. The normalisation consist of aspects such as deleting extra spacing, converting angle and curved quotation marks into English quotation marks and transforming emoticons into UTF-8 emojis[19].

In the case of the Linear SVM and VastText, we have decided to enrich their input vectors with numerical elements that represent the following aspects:

- Number of consecutive positive emoticons
- Number of consecutive negative emoticons
- Number of consecutive exclamations marks
- Number of consecutive questions marks
- Number of ellipsis (. . .)
- Number of question marks
- Number of exclamation marks
- Number of combinations of questions and exclamation marks, e.g. “!?!?”, “?!?!”

[17] Although a stemmer, could be faster than a lemmatiser, NLP4J only provides a lemmatiser. Furthermore, several lexica use lemmatised entries. Therefore, at least, to query the lexica, we need lemmas.

[18] This feature is not valid in FastText, which only provides a n-gram generator

[19] More specifically, we converted emojis, like *:)*, into emoticons, such as ☺. Emoticons and emojis can be a great source of information regarding the sentiments. However, taking into account that a happy face in emoji can be represented with variants like *(:* and *:-)*, we considered pertinent to normalise them into UTF-8 emojis.

- Number of questions words, e.g. *how*, *what*, *why*
- Number of positive emoticons
- Number of negative emoticons
- Number of words written completely in capital letters, e.g. “HAPPY”, “VERY”
- Number of elongated words, for instance, “soooo” “looong”
- Number of negations words, for example, “not”, “never”, “without”

In addition to the numerical elements highlighted above, there are the following binary features that are evaluated according to the first token or last token of the text:

- Presence of question mark
- Presence of exclamation mark
- Presence of negative emoticons
- Presence of positive emoticons
- Presence of ellipsis
- Presence of full stop (this case only for the last token)

It should be indicated that in the presence of question or exclamation marks and ellipsis, in the first token does not mean that the exact first token must be one of those characters, but can be instead next to the first word token. Therefore, texts starting with “What?!” or “so. . .” would activate the corresponding binary features.

Furthermore, we have decided to explore the semantic features, like in [13], but using the embeddings that our partners from AUEB created using StackOverflow [33]. More specifically, these word embeddings were trained using word2vec model using as input the dump of StackOverflow of December 2017. The word embeddings were trained on text without code and stop words.

Apart from the previous numerical elements, we decided to explore different lexica to determine which one provides the best performance to our sentiment analyser. Due to the different nature and annotations found in each lexicon, the number and types of extra features based on these knowledge resources are varied. In Table 2 we present the extra features used in accordance to each lexicon explored.

4.5 Datasets

Despite the interest from the research community in creating sentiment analysers for the domains of software development or engineering, the number of annotated datasets available is quite minimal. The main reason is that corpora annotation is expensive and time-consuming. In the following paragraphs we describe the datasets that, to our knowledge, are freely available.

In [83], the authors annotated with emotions a set of 5.9K JIRA issue tracking system posts. This same corpus was extrapolated into positive and negative sentiments by [45]. The extrapolation was done considering the emotions joy and love as positive, while the emotions anger, sad and fear are considered as negative. In the case of the emotion surprise, it could be either positive or negative depending on the evaluation given by external

Table 2: This table shows the numerical features that were obtained with respect to each lexicon explored for the sentiment analyser.

Feature	Lexica			
	SentiWordNet 3.0	SenticNet 5	SentiStrength	SocialNet
Number of positive words	X	X	X	X
Positive strength	X	X	X	X
Maximum positive strength	X	X	X	X
Last positive strength	X	X	X	X
Number of negative words	X	X	X	X
Negative strength	X	X	X	X
Maximum negative strength	X	X	X	X
Last negative strength	X	X	X	X
Number of subjective words	X			
Subjective strength	X			
Maximum subjective strength	X			
Last subjective strength	X			
Number of neutral words	X			
Number of objective words			X	
Objective strength			X	
Number of pleasantness words		X		
Pleasantness strength		X		
Maximum pleasantness strength		X		
Minimum pleasantness strength		X		
Last pleasantness strength		X		
Number of pleasantness words		X		
Attention strength		X		
Maximum attention strength		X		
Minimum attention strength		X		
Last attention strength		X		
Number of pleasantness words		X		
Sensitivity strength		X		
Maximum sensitivity strength		X		
Minimum sensitivity strength		X		
Last sensitivity strength		X		
Aptitude strength		X		
Maximum aptitude strength		X		
Minimum aptitude strength		X		
Last aptitude strength		X		

annotators. As the authors do not provide the extrapolated corpus, and we do not have the means to evaluate manually, we have decided not to use this corpus, at least for the sentiment analyser[20].

In [13] the authors presented an annotated corpus based on StackOverflow posts, that are either questions, answers or comments. Numerous posts contain only natural language text. However, few traces of code can be

[20]In Section 5, we describe how we make use of this corpus for training an emotion classifier

found in some posts. The corpus has three different labels, *positive*, *negative* and *neutral*. More specifically, it is composed of 4423 posts and it is quite well balanced: 35% positive, 27% negative, and 38% neutral. Furthermore, the authors provided a split version of the corpus into train and test, allowing to reproduce the results but more importantly to compare their tool on the same dataset. We decided to use the training data set from this corpus to train our methods; the test part of the corpus will be used to determine the performance.

The authors from Stanford CoreNLP SO, [55], also provided an annotated corpus from StackOverflow posts. As it happens with the gold-standard from [13], small traces of code can be found. Nevertheless, this corpus is not balanced as 12% of the data belong to the positive label, 8% to the negative category and 80% to the neutral label. This corpus was used for training Stanford CoreNLP SO, but also for comparing it with other state-of-the-art tools. We expected to use this corpus for testing, due to its unbalanced aspect and because it was used in [55] to evaluate multiple sentiment analysers from the state-of-the-art. However, as none of the sentiment analysers tested by the authors on this corpus achieved good scores, we wondered whether it was correctly annotated. A further analysis done by us, considered that this dataset was not correctly annotated, therefore we excluded it from the analysis.[21]

4.6 Settings

We present in the following subsections the settings that were used for doing the experiments, but also those used to compare the results with the tools from the state-of-the-art.

4.6.1 Experimental

In Table 3, we present the total number of experiments that we explore in this deliverable for Sentiment Analysis. As it can be observed, we explore different combinations of lexica and in most cases, numerical features.

The column *ID* in Table 3, indicates the name that will be used throughout this deliverable to identify the experiment. We must indicate, that to the ID we will concatenate the letters *L* or *R*, to indicate whether the text in the experiment was lemmatized or not.

There might be two questions that can be posed regarding the experiments done. The first one, is why we only use semantic vectors for SentiStrength. Although the goal was to experiment with these vectors for all the lexica, the use of the semantic vectors pose a problem *vis-à-vis* its possible usefulness for CROSSMINER. The word2vec model is around 1Gb but, when it is loaded in memory, it uses 4Gb of RAM. Using this amount of RAM for only one component might reduce the attractiveness of CROSSMINER. Furthermore, the calculation of vectors is slow and as evident from the results presented in Section 4.7, they did not provide the expected boost.[22] The second question might be why SenticNet 5 was not explored also with the specialised lexicon from SocialNet. The reason is that when we analysed the former, we found out that it contained already concepts like *memory leak* and *software bug*.

To obtain the best model possible, we optimized the training of each experiment described in Table 3 with a *Bayesian Optimisation*. The objective function for the Bayesian Optimisation was defined as either the median

[21] Some examples, that are neutral and factual, are wrongly annotated. For example, with negative polarity we found the following sentences “That is what is causing your null pointer exception”, “This line gives the error”. With positive polarity were annotated the following two sentences “API_NAME can be used”, “Javolution Structs supports mapping of off heap memory as a data structure.”

[22] The loading of the model is slow as well, but as this task would be done only once, we do not consider this aspect into account.

Table 3: We present the experiments done with respect to each machine learning algorithm explored. As well, we indicate how we combined the use of lexica and extra numeric features in every experiment. Experiments are linked to IDs that will be used along this deliverable.

Method	ID	Numeric Features	Semantic Similarity	Lexica utilised			
				SentiWordNet 3.0	SenticNet 5	SentiStrength	SocialNet
FastText	S1						
SVM	S2	X		X			
	S3	X		X			X
	S4	X			X		
	S5	X				X	
	S6	X				X	X
	S7	X	X			X	X
VastText	S8	X		X			
	S9	X		X			X
	S10	X			X		
	S11	X				X	
	S12	X				X	X
	S13	X	X			X	X

or average, whichever is the lesser, Micro F-score (see Section 2.4) of a 10-fold cross validation. We preferred this objective function as our task is multi-class. Moreover, we decided to use this objective function to avoid to an extent, possible outliers that could mislead us away from the best-performing parameters; either due to underfitting or overfitting.[23]

In Table 4, we present the parameters, found by the Bayesian Optimisation for each experiment using the linear SVM with non lemmatised text. The parameters for the linear SVM and lemmatised text are shown in Table 5.

Table 4: We present the Linear SVM parameters used for training the models using non-lemmatised texts.

Method	ID	n-grams	Skip-bigrams	Min. Freq.	C
SVM	S2R	3	2	10	2^{16}
	S3R	1	0	10	2^{32}
	S4R	1	0	1	2^{32}
	S5R	1	1	10	2^{8}
	S6R	3	2	1	2^{8}
	S7R	1	2	1	2^{64}

We present in Table 6, the parameters used by FastText and VastText in each experiment where the text was not lemmatised. In Table 7, we do the same but for lemmatized text.

[23] Outliers can make an average F-score look better (or worse). For example, in a 5-fold cross validation process we could get as F-score the following values $\{0.50, 0.52, 0.55, 0.95, 0.58\}$. The mean, affected by the outlier 0.95, is equal to 0.62. However, the median, which is 0.55, represents the set of F-scores better.

Table 5: We present the Linear SVM parameters used for training the models using lemmatised texts.

Method	ID	n-grams	Skip-bigrams	Min. Freq.	C
SVM	S2L	1	0	10	2^5
	S3L	3	2	1	2^{16}
	S4L	1	2	2	2^{13}
	S5L	2	2	4	2^8
	S6L	3	2	1	2^{30}
	S7L	1	2	20	2^8

Table 6: We present the parameters used for training the models using non-lemmatised texts and related to each experiment using FastText and VastText. It shold be noted that FastText does not accept skip-bigrams.

Method	ID	n-grams	Skip-bigrams	Min. Freq.	Epoch	LR	Vector dimemsion
FastText	S1R	3	-	10	30	0.2018	75
VastTet	S8R	2	0	50	5	0.4462	20
	S9R	3	1	40	30	0.0132	40
	S10R	2	2	1	5	0.0511	200
	S11R	3	2	5	25	0.0157	30
	S12R	2	1	2	30	0.0043	175
	S13R	1	1	50	25	0.0062	75

Table 7: We present the parameters used for training the models using lemmatised texts and related to each experiment using FastText and VastText. It shold be noted that FastText does not accept skip-bigrams.

Method	ID	n-grams	Skip-bigrams	Min. Freq.	Epoch	LR	Vector dimension
FastText	S1L	2	-	3	10	0.4255	100
VastTet	S8L	2	1	2	5	0.1383	20
	S9L	1	0	10	5	0.0721	75
	S10L	2	2	2	5	0.1262	20
	S11L	2	2	1	30	0.1829	175
	S12L	1	1	2	40	0.0046	50
	S13L	1	1	50	25	0.0062	75

4.6.2 Comparative

To have a point of comparison, the sentiment analysers explored here are compared against 3 baselines from the state-of-the-art; one for general, domain-independent texts and two for specialised-domain texts. In the first group, we use SentiStrength.[24] For the second group, we compare with SentiStrength-SE and Senti4SD.

[24]We wanted to compare against Stanford CoreNLP, however, this tool analyses the text sentence by sentence, instead at a document level. Each document can have multiple sentences and in consequence multiple sentiments. The data sets used for testing, and training too, are composed of posts, which in turn, may have multiple sentences, but each post is annotated with only one sentiment at a document level. It is not an obvious task to determine which would be the main sentiment. Many questions are open regarding this problem, should we consider the most frequent label? How do the neutral labels affects the others? Should we consider the length of the sentence?

The rationale behind selecting these tools, among other available tools is that they have been used previously to compare similar tools and more importantly, can be downloaded and tested easily.

4.6.3 Evaluative

All the baselines and the methods developed by us have been evaluated using a precision, recall and, micro and macro F-score. These are reported in the next section.

4.7 Results

We present, in Table 8 the results from applying the trained models on the testing dataset provided by [13] without using a lemmatiser.

Table 8: Results obtained in terms of Precision (P), Recall (R) F-score (F1), Micro and Macro F-Score, for the experiments using non-lemmatised texts.

Method	ID	Positive			Negative			Neutral			Global F-score	
		P	R	F1	P	R	F1	P	R	F1	Micro	Macro
FastText	S1R	0.781	0.938	0.853	0.620	**0.827**	0.709	**0.902**	0.525	0.664	0.750	0.742
SVM	S2R	0.839	0.731	0.835	0.704	0.463	0.559	0.692	**0.866**	0.769	0.745	0.721
	S3R	0.860	0.886	0.873	0.715	0.677	0.696	0.777	0.785	0.781	0.791	0.783
	S4R	0.888	0.853	0.870	0.747	0.675	0.709	0.7433	0.820	0.780	0.792	0.786
	S5R	0.899	**0.941**	**0.919**	0.799	0.763	0.781	0.827	0.818	0.822	0.846	0.841
	S6R	**0.920**	0.903	0.911	0.774	**0.827**	0.800	0.820	0.793	0.806	0.840	0.839
	S7R	0.902	0.925	0.913	0.789	0.772	0.780	0.817	0.811	0.814	0.840	0.836
VastText	S8R	0.852	0.897	0.874	**0.814**	0.572	0.672	0.732	0.852	0.787	0.791	0.778
	S9R	0.884	0.888	0.886	0.753	0.697	0.724	0.782	0.820	0.801	0.810	0.804
	S10R	0.889	0.893	0.891	0.764	0.786	0.775	0.804	0.785	0.794	0.822	0.820
	S11R	0.918	0.912	0.915	0.793	0.819	0.806	0.833	0.818	0.826	0.851	0.849
	S12R	0.913	0.917	0.915	**0.814**	0.816	**0.815**	0.839	0.834	**0.837**	**0.858**	**0.855**
	S13R	0.891	0.910	0.900	0.782	0.800	0.791	0.834	0.805	0.819	0.840	0.837

In Table 8, we can observe that in general, all the methods using extra features surpass the performance of FastText, with the exception of *S2R*. Also, the results show that the lexicon of SentiStrength boosts the performance of all the classifiers (*S6R*-*S8R* and *S13R*-*S15R*). However, the use of semantic vectors in *S8R* and *S15R* did not boost the performance of the sentiment analyser. In fact, it slightly reduced the performance, although this difference may not be statistically significant. Regarding the use of a sparse or dense representation for the text, i.e. SVM vs VastText, this does not seem to have an effect on the performance of the classifiers. In general, for all the classifiers, it is easy to identify texts with positive polarity, but much harder to correctly identify texts with negative and neutral polarities.

We present, in Table 9, the results from applying the trained models on the testing data set provided by [13] after being lemmatised.

From the results presented in Table 9 we can conclude that lemmatisation does not improve the results of the classifiers, except for *S2L* and *S8L*. In some cases, the utilisation of lemmas decreased the performace of the classifiers, like in *S1L*, *S5L*, *S6L*, *S11L* and *S12L*. The comments that we did for Table 8 concerning the text representation, the semantic vectors and the lexica are valid for the lemmatised classifiers as well.

Table 9: Results obtained in terms of Precision (P), Recall (R) F-score (F1), Micro and Macro F-Score, for the experiments using lemmatised texts.

Method	ID	Positive			Negative			Neutral			Global F-score	
		P	R	F1	P	R	F1	P	R	F1	Micro	Macro
FastText	S1L	0.746	**0.956**	0.838	0.639	0.833	0.723	**0.903**	0.480	0.627	0.740	0.729
SVM	S2L	0.874	0.882	0.878	0.738	0.641	0.686	0.760	0.724	0.791	0.794	0.785
	S3L	0.877	0.895	0.886	0.733	0.666	0.698	0.768	0.805	0.786	0.798	0.790
	S4L	0.890	0.871	0.880	0.760	0.697	0.727	0.760	0.820	0.789	0.804	0.799
	S5L	0.898	0.932	0.915	0.794	0.750	0.771	0.810	0.814	0.812	0.837	0.833
	S6L	0.899	0.921	0.910	0.770	0.791	0.780	0.811	0.777	0.793	0.831	0.828
	S7L	**0.918**	0.908	0.913	0.783	0.813	0.798	0.813	0.799	0.806	0.840	0.839
VastText	S8L	0.886	0.906	0.896	0.795	0.669	0.726	0.772	0.844	0.807	0.818	0.810
	S9L	0.889	0.912	0.900	**0.830**	0.705	0.762	0.787	**0.852**	0.818	0.833	0.827
	S10L	0.896	0.906	0.901	0.807	0.733	0.768	0.779	0.822	0.800	0.827	0.823
	S11L	0.894	0.910	0.902	0.764	0.802	0.783	0.796	0.755	0.775	0.822	0.820
	S12L	0.915	0.919	**0.917**	0.781	0.825	**0.802**	0.845	0.809	0.826	**0.851**	**0.848**
	S13L	0.900	0.910	0.905	0.779	0.816	0.797	0.837	0.801	0.818	0.843	0.840

In Table 10, we show the performance of tools from the state-on-the-art when they are applied on the data set of [13]. It is shown that SentiStrength-SE, despite being a tool designed to address the problems of SentiStrength in the domain of software engineering, produced the lowest performance, in comparison to other tools from the state-of-the-art. SentiStrength has a good performance, but it is still below the level of Senti4SD.

Table 10: Results obtained from applying state-of-the-art sentiment analysers on the testing corpus. Results are expressed in terms of Precision (P), Recall (R) F-score (F1), Micro and Macro F-score.

Method	Negative			Positive			Neutral			Global F-score	
	P	R	F1	P	R	F1	P	R	F1	Micro	Macro
SentiStrength	**0.958**	0.675	0.792	**0.925**	0.892	0.908	0.639	**0.955**	0.766	0.825	0.822
SentiStrength-SE	0.730	0.790	0.760	0.900	0.790	0.840	0.770	0.730	0.750	0.780	0.780
Senti4SD	0.798	**0.891**	**0.841**	0.923	**0.923**	**0.923**	**0.872**	0.799	**0.833**	**0.867**	**0.865**

We can determine, from Table 10, that the models created by us do not surpass the state-of-the-art, but we are not far from it; particularly *S12R*, which has a Micro F-Score of 0.855 and a Macro F-Score of 0.853. The difference with respect to Senti4SD is unlikely to be significant.

4.8 Discussion

The fact that a neural network and a classical machine learning method performs very similar is not surprising, especially when we optimise them. In other words, thanks to the optimisation, we know, to an extent[25], the parameters that will generate the best model possible, regardless of the advantages or disadvantages of the methods used. Furthermore, one element to take into consideration is that VastText and the linear SVM, are both linear classifiers. In the literature we can find multiple examples in which a classical machine learning

[25]We cannot ensure that the parameters found through the Bayesian Optimisation correspond to the absolute maximum. The only method that could give us, in theory, the absolute maximum is a grid, however grids are not performing at all.

method performs equally, or even better, than a neural network, e.g. [67], [92], [77]. Aspects like, type of task, dataset size and selected parameters, can have a significant impact on the performance of machine learning algorithms.

One interesting point to discuss is the fact that *S2R* is at least 6 points lower than other methods that use extra features. It is the detection of negative texts that affected the performance of the sentiment analyser. We do not have a specific reason for this performance, but there are two possible causes. The first one is that the quality of the lexicon SentiWordNet 3.0 is not very good, and the lack of other resources or lemmas, makes it hard to achieve a better performance. The second reason is that the Bayesian Optimisation did not find the optimal paramaters. This supposition is based on the fact that lemmatisation did not play a big role for others models and that experiments using equally SentiWordNet 3.0 performed similarly. Nonetheless, it exists the possibility that the lemmatisation of text does not affect the outcome as expected due to a tendency of misspelling words in informal communications. We will need to perform a deeper analysis to determine the exact reasons of this poorer performance.

We observed in Table 8 and Table 9, that the selection of the lexicon plays an important role in the machine learning performance. The quality of SentiStrength is irrefutable, even when it is 14 times smaller than SenticNet 5, i.e. $7,000$ entries vs. $100,000$, the former is more powerful than the latter. This means that the number of entries do not affect as much as the entries' polarity weights. Using the specialised lexicon from SocialNet did not change the performance of the models. After doing an inspection of this lexicon, we find out, that some entries are noise, especially the ones with negative weight. Examples of these entries are: "e", "bs", "5" or "10x". We think that these noisy entries, especially those containing a number, are related to specific versions of software or events related to general failures with a technology product, which cannot be generalised to truly express a positive or negative polarity. In other words, we cannot consider than the version *5* of a software will be always faulty no matter the product.

Concerning the use of lemmatisation as part of the text pre-processing, we observed that only *S2L* was improved around 6 points. In the case that the lemmatisation is the main reason of this change, this could be because the vocabulary was reduced but concentrated, thus, the model could use better textual features. For the rest of the models, the use of the lemmatisation did not provide an improvement in terms of performance. This can reinforce the idea that the text has a limited amount of effect on the classification task, but it is more related to the quality of the lexica used. With respect to the small differences found between models trained with lemmas and without, we expected them and, they might not be significant. We were expecting some differences, as the training data, despite being the exact same one for all the methods, the technique used for loading it could have introduced some non-deterministic changes on the order of element.[26]

There is still work to do, especially for improving the detection of texts with a negative polarity. However, this is recurrent problem found in the literature, phenomena like negation of negative words (e.g."I can't fire you"), or negation and "but" (e.g. "I liked the movie, but I wouldn't watch it again") can pose several problems. The use of n-grams can help, but these are not a complete solution. Techniques like the one used by Stanford CoreNLP, i.e. the use of a parsing trees and recursive neural networks, can be very powerful for detecting negations correctly. But if we consider that for running the Stanford CoreNLP's sentiment analyser, it is necessary to use at least 5GB of RAM, we can conclude that better performing methods can be computational expensive.

[26] In other words, the data before being stored in files, it was loaded in memory using an associative array. The access to the data stored in these arrays, is by definition, not deterministic. Therefore, when the data had to be printed, the order of the elements could vary from one file to another. This difference, although minor, can have a small effect on how machine learning methods will train.

Moreover, the correct detection of neutral texts is challenging due to combinations of words and contexts being negative or lose polarity in certain scenarios. For example, the word *off* in "Please turn *off* the lights before leaving" does not have a negative polarity like in "The party is definitely *off*". This phenomenon happens especially in factual texts, such as "Here is the problematic line of code" or "Good is a positive adjective".

With respect to which model is the most adequate for CROSSMINER, there are different aspects to take into account. In first place, and the most restrictive one, is the compatibility, in terms of licence, of the resources and tools utilised. In this aspect, DeepLearning4J and NLP4J are made public with an Apache Licence, therefore these do not cause such problems. However, some lexica explored for the creation of a sentiment analyser can only be used for research purposes or their use in commercial components means to pay a fee. Considering this aspect, only SentiWordNet 3.0, SenticNet 5 and SocialNet are compatible with CROSSMINER.

The second aspect to consider is the size of the models. As we have been explaining along this deliverable, to make CROSSMINER attractive to the greatest number of developers, we need to provide natural language tools that can be run in a great variety of computers, either low or high-end. Therefore, in Table 11, we present the size (on hard disk) of the models including the dictionary used to vectorize the texts.

Table 11: Size of the model generated by each experiment regarding the sentiment analyser.

Method	ID	Lemmatisation	
		No	Yes
FastText	S1	72M	72M
SVM	S2	1M	2.4M
	S3	12.7M	1M
	S4	3.3M	1.5M
	S5	2.5M	1M
	S6	12.4M	11M
	S7	1.2M	5.8M
VastText	S8	83K	310K
	S9	116K	346K
	S10	42.1M	1.6M
	S11	1M	33M
	S12	2.6M	653K
	S13	152K	141K

As we can observe in Table 11, the models with the dictionaries can be tiny, using only some hundreds of kilobytes. The largest model is just 72M, which does not impose challenges to ubiquitous hardware. It should be noted that experiments $S8$ and $S1$ need indirectly a word2vec model of 1GB.

Taking into account the licence limitations, the size of the models and the results in terms of Micro and Macro F-score, we consider that the most adequate model for CROSSMINER would be the one related to experiment $S9L$, which gives a Micro and Macro F-score of 0.827 and has a size in hard disk of 346K.

4.9 Conclusions

Sentiment analysis is a natural language processing task that has for objective to determine whether a text reflects a polarity, either positive or negative, or none, i.e. neutral.

The automatic analysis of sentiments has been mainly explored in non-technical domains, like movies reviews or tweets. However, there are other possible applications of sentiment analysis in which the topics are tech-

nical. This is the case of communications done within the communities of software engineering and software development. Being able to know the sentiments located in messages exchanged by these specialised communities can be vital for the success of a project. In other words, if users normally post messages with a negative polarity, it might be a signal of frequent errors or a community that feels that it is ignored by the developers. Knowing which sentiments are triggered in a community automatically, can make developers take actions that will solve the problem.

Due to the fact that sentiment analysis has been mainly explored and utilised in non-technical domains, the number of tools available for technical domains, like the one exposed in CROSSMINER, is reduced. Researchers have observed that sentiment analysers for non-technical domains do not perform well, therefore they have started to create tools focused on domains like software engineering.

In this part of the deliverable, we have presented different sentiment analysers trained on a corpus based on posts from StackOverflow. These sentiment analysers were trained using a classic machine learning method, i.e. a Support Vector Machine (SVM), and neural networks, FastText and VastText, an in-house neural network architecture.

Depending on the support of features, other than text, in each machine learning method, we enriched the text with lexical and semantic aspects. In some cases, we used different lexica, as they provide varied types of information related to words' polarity.

The results showed that the use of appropriate lexica plays an important role in the performance of a sentiment analyser. However, it is not the case of SVM or neural networks, in other words, these can perform equivalently.

Nonetheless, many of the lexica found in the literature can only be used freely for research or non-commercial purposes, we have arrived to create a sentiment analyser, that has shown a performance, in terms of F-score, equal to 0.827. Moreover, the model size is tiny, 346K, making it useful for CROSSMINER.

There is still work to do, in order to improve the correct detection of negative and neutral texts. We have some ideas, like the negation of n-grams beginning with a negation token. Other techniques may be explored in the future, like the use of a discourse or dependency parsing tool. However, to use these techniques, we may need to increment the workflow's complexity or the computational resources necessary to run them.

5 Emotion Classification

Emotion classification is the task that consists in determining the emotions expressed in a text [123][27]. Nevertheless, although it is easy to define what an emotion classifier is in the area of natural language processing, it is complicated to define what an emotion is. The problem resides on the fact that emotions are complex states of mind raised due to internal or external events [15, 123].

For at least 70 years, researchers in psychology have proposed different theories which try to define what an emotion is and which different emotions exist. Nevertheless, there is still not a consensus among researchers. In the following paragraphs we will introduce, at superficial level, the most relevant theories regarding emotions.

One of the first theories regarding emotions is the one of Ekman [89], which states that there are 6 basic emotions universally recognised: *anger*, *disgust*, *fear*, *joy*, *sadness* and *surprise*.

In 1980, Plutchik and Kellerman proposed a model currently known as the *Wheel of Emotions* [91]. This theory indicates that there are 8 primary bipolar emotions arranged in 4 axes: *joy vs. sadness*, *anger vs. fear*, *trust vs. disgust* and *surprise vs. anticipation*. Secondary emotions revolve around the primary ones due to variations in their intensity or combinations.

Another theory is the one proposed by Shaver et al. in [102], where a tree is used to represent emotions. More specifically, this theory proposes 6 main branches that correspond to the emotions: *anger*, *fear*, *joy*, *love*, *sadness* and *surprise*. These 6 branches are in turn divided in secondary and tertiary emotions.

More recently, [15] proposed a theory based on the Wheel of Emotions called Hourglass of Emotions (HOE). In this theory, the authors categorise the emotions in 4 dimensions: *pleasantness*, *attention*, *sensitivity* and *aptitude*. Depending on the level of activation of these dimensions, different emotions are activated. Moreover, according to the authors, these dimensions can be related to positive and negative polarities.

Although the creation of theories regarding emotions is an area concerning uniquely psychology, researchers in computer science have been interested on their potential applications. For example, emotions can be used for marketing aspects and to know whether consumers are satisfied about a product or service [38]. In addition, emotions can be used to prevent cases of suicide by analysing conversations or texts [32]. Nowadays, it is possible to find in the literature works describing the classification of emotions in domains like newspapers [30], tweets [73] and *apps* reviews [65].

The exploration of emotions in the domain of software engineering and software development is not the exception [82, 83, 14]. However, it is a domain that it has been less explored, not only due to the few resources available for training machine learning systems, but also because text can contain more than one emotion. This means that it is necessary to use multi-label classifiers, which, as we stated in Section 2, poses a series of challenges.

In the following sections, we present a more detailed state-of-the-art concerning the creation of tools related to emotion classifiers. We will describe as well the lexica used in many tools, which was the approach used in OSSMETER for the classification of emotions. Furthermore, we present the method explored in this deliverable for a new tool based on machine learning, capable of identifying which emotions are present in documents related to software engineering and software development.

[27] Some authors use the term *emotion detection* as synonym for *emotions classification*, while some others consider that the former only indicates whether a text is neutral or not.

5.1 State-of-the-Art

In the last years, the scientific community has shown an interest on the problem of emotion classification as a multi-label task. In the following paragraphs we describe the most representative articles from the state-of-the-art.

Affect Analysis Model [79] is a rule-based system, i.e. non-supervised method, for the classification of emotions. Apart from the rules used for the classification of emotions, the system also relies on an affect database created by the authors. The database contains entries that correspond to emoticons, affect words, common abbreviations and acronyms, among other elements; each entry of the database was manually annotated with their respective affective strength.

Feeler [28] is a unsupervised method that is based on the cosine similarity of vectors represented in a high dimensional space. Among the features used by Feeler, is the weighting of unigrams using TF-IDF, and the enrichment of the vectors using lexica such as *WordNet Affect Lexicon*.

In [50], the authors present an unsupervised method for the classification of emotions. This system is based on the use of the lexicon *WordNet Affect Lexicon* [108] and tools for reducing the dimensionality of the text, such as Latent Semantic Analysis and Non-negative Matrix Factorisation.

The approach of [42] differs from the previously described systems for emotion classification, as it uses not only a supervised method but a psychological approach. More specifically, the authors trained a *Hidden Markov Model (HMM)* that is based on the process of how a sequence of mental states affect or causes the emotion of a person.

In [58], the authors explored the utilisation of a multi-label method to detect the emotions in texts written by people that committed suicide. The multi-label classifier consisted on a Radial SVM merged using a one-vs-all along with a Label Powerset strategy. Although they tested with different extra features, such as emotion-related keywords or the context-related aspects[28], their best approach consisted on the use uniquely of unigrams. The classifier detected 15 different kinds of emotions, such as *hopelessness*, *love*, *guilt* and *fear*. The classifier could also detect the lack of emotion, i.e. neutral.

The work of [30] presents numerous experiments using different multi-label classification methods for the identification of emotions in short texts written in Brazilian Portuguese. More specifically, the authors make use of SenticNet [16] to determine the strength of 4 emotional dimensions: pleasantness, attention, sensitivity and aptitude. These dimensions are mapped into 24 different perceptions of emotions described in HOE but also were mapped into 16 combinations of 4 independent HOE dimensions [29]. Apart from SenticNet, the authors made use of TF-IDF and removed stop words. Regarding the methods used, the researchers made use of algorithms such as BP-MLL, RAkEL and HOMER.

As it happens in the domain of sentiment analysers, we can find in the literature systems that have been developed for solving a problem proposed in challenge campaings, such as *SemEval*.

For example, in *SemEval 2007* task 4 consisted of the classification of emotions and polarity of news headlines [107]. Only 3 teams participated in the sub task related to the classification of emotions. The system that achieved the best evaluation for this sub task is *UPAR7* [19]. UPAR7 is a rule-based system that utilises dependency graphs enriched with information from the lexica WordNet Affect and SentiWordNet.

In *SemEval 2018*, the organisers of Task 1 [73] proposed different challenges that revolved around emotions, with the classification of emotions being one task. In this aspect, the organisers indicate that for the classification of emotions, most of the participants used deep neural networks based on architectures such as *Convolution Neural Networks (CNN)*, *Recurrent Neural Networks (RNN)* or *Long-Short Term Memory Networks (LSTM)*,

[28] These features consisted in indicating the emotion of the n preceding and following sentences.

along with external resources such as lexica, word embeddings or word n-grams. In the following list we summarise the methods used by the first top three teams in the classification of emotions for English:

- *NTUA-SLP* [7]: The systems consists of a Bidirectional LSTM (BiLSTM) that uses embeddings and an attention mechanism. The embeddings were trained on a large corpus of tweets that were unlabelled. The attention mechanism is neural network layer that obtains a better estimation of the importance of each word in a text. Due to the small training data set given by the organisers, these participants used a transfer learning approach by pre-training the models using the corpus for sentiment analysis given in *Semeval 2017 Task 4*.
- *TCS Research* [66]: In this system, the authors trained a SVM that uses features from lexica, a deep learning model and an external pre-trained model. The pre-trained model uses the output of a neuron from an LSTM trained on Amazon reviews. Concerning the lexica, the authors used different sources like *NRC Word-Emotion Association Lexicon* [75], *NRC Hashtag Sentiment Lexicon* [74] and Sentiment140 [37]. The deep learning model consist of a BiLSTM that had an embeddings layer[29] and also an attention layer.
- *PlusEmo2Vec* [88]: This system is based on three large features sets, an emoji sentence representation, emotional word vectors and textual features. The emoji sentence representation consisted on using a one-layered BiLSTM pre-trained on tweets corpora for predicting the emojis that should be used clusters of texts. The emotional word vectors were trained using a CNN on tweets that were related to emotions. The textual features consisted in determining aspects such as the number of upper-cased words and number of exclamations, among others; the authors did not use any kind of lexica. For the classification of emotions, the authors used a regularized linear regression and a logistic regression classifier chain, that used as input the previously described features.

In the state-of-the-art, we found only two tools created for the detection of emotions in the domains of software engineering and software development. In the following paragraphs we describe the tools that we have found in the literature.

The first system is [82] presents a tool was created for the classification of emotions in technical texts coming from Jira, an issue tracker system.[30] The tool consists in multiple Linear SVMs that were trained to detect 4 possible emotions: *anger*, *love*, *sadness* and *joy*; it is not indicated whether the authors considered the multi-label aspect or not for the SVM, and if it was the case, which technique was used to deal it. The SVM used as features, not only the text, but also elements from *WordNet Affect Lexicon* and different external tools, like SentiStrength [112] and a system that detects politeness [27].

The second system is *EmoTxt* [14] based on the tool presented by [82]. It is a emotion classifier that was trained, separately, on two different corpora. The corpora were based on Jira issue tracker system [83] and on StackOverflow posts.[31] EmoTxt uses 6 different binary SVM for classifying the following emotions: *joy*, *love*, *sadness*, *anger*, *surprise*, *fear*. Apart from the features used in [82], the authors added TF-IDF and did not used any kind of stemming or lemmatization. EmoTxt has made available, as open source, the code for training a sentiment classifier[32]. However, the models and the exact corpora for training and testing were not provided, making it difficult to reproduce the results.

[29]This embeddings layer consisted of grouping different pre-trained embeddings models that focused on different domains and unit levels, i.e. word and character levels.

[30]The authors do not describe in deep the data set, except that its size is of $4,000$ entries. We suspect that it is the same data set present in [83].

[31]This corpus was made available after the publication of EmoTxt, in the following paper [80].

[32]github.com/collab-uniba/Emotion_and_Polarity_SO

As it can be seen, in the state-of-the-art the majority of systems developed for the classification of emotions are non-supervised; few of them explore the utilisation of supervised machine learning methods, and those that use these methods mostly rely on single-label classifiers.

5.2 Lexica as External Resources

One common element in multiple emotion classifiers is the use of external resources called lexicon, i.e. a specialised dictionary in which the entries have been annotated according to varied aspects. For the emotion detection, we can find different lexica that are annotated in accordance to the different emotions theories. In the following paragraphs we present the most representative lexica found in the state-of-the-art:

Affective Norms for English Words (ANEW) [12] is a lexicon that has for objective to provide a normative and standardised emotional rating for words in English. The lexicon has been annotated in terms of *pleasure*, *arousal* and *dominance*. It includes, as well, how frequently occurs each annotated word in the Brown Corpus [52].

WordNet Affect Lexicon [108] is an extension of WordNet [34] which includes entries that are correlated with affection, i.e. emotions, and polarity.

The *NRC Word-Emotion Association Lexicon* [75] is a crowd-sourced annotated lexicon that indicates the emotions (anger, fear, anticipation, trust, surprise, sadness, joy and disgust) and polarity (positive or negative) associated to a list of words come from other lexica: WordNet Affect Lexicon, General Inquirer and Affective Norms for English Words (ANEW). Similar to this source, we can name the *NRC Hashtag Sentiment Lexicon* [74] an automatic annotated lexicon created specifically for the sentiment analysis of tweets. The annotation was done using a large set of tweets, a set of 78 seeds, i.e. words with a well-known polarity, the detection of hashtags and the measure *Pointwise Mutual Information*.

A more recent lexicon, is the one proposed by [72], called *NRC Affect Intensity Lexicon*. This lexicon is a collection of words linked to their intensities regarding an emotion. The current lexicon has around $6,000$ entries and only covers the focus basic emotions: anger, fear, joy and sadness. It is expected to be increased in the future. The lexicon was built using a technique called Best-Worst Scaling (BWS), an annotation scheme that bypasses problems such as inter and intra annotations inconsistencies.

There are some lexica that not only indicate the emotions to which are related, but also other semantic aspects. In this case, we can name *SenticNet*[16] and *General Inquirer (GI)* [106]. The former, is a lexicon currently in its 5th version that is composed of $100,000$ entries. Each entry is annotated with semantic aspects like the polarity (positive or negative), polarity strength, moods (e.g. surprise, interest, disgust) and related concepts. However, its main and original characteristic, is that this lexicon is annotated with elements from the Hourglass of Emotions (HOE), therefore, each entry has its own level of *pleasantness*, *attention*, *sensitivity* and *aptitude*. Concerning GI, it is a lexicon that has been enriched through the years and contains semantic aspects of words such as positivity, domain, quality, human collectivities and animal-related.

5.3 OSSMETER's Approach

In OSSMETER, the classification of emotions was done through an unsupervised method. More specifically, it consisted on determining whether a text contained entries found in *NRC Word-Emotion Association Lexicon* [75]. If a word from the text matched an entry, then the associated emotion was indicated as the one present in the text.

5.4 Methodology

In contrast to OSSMETER, we expect to provide an emotion classifier based on supervised machine learning for CROSSMINER. More specifically, we explore different approaches from the state-of-the-art for creating a multi-label classifier. These approaches are VastText, our neural network based on extra numerical features and word embeddings, and two classifiers designed for multi-label classification: Homer and RAkEL.

As we did for the sentiment analyser, we explore how an emotion classifier is affected due to the use of a lemmatiser as part of the pre-processing. We include in this exploration the use of a minimum threshold of occurrence, word n-grams (from unigrams to trigrams) and word skip-bigrams (a skip hole between 2-4). We tokenize the text and apply a normalisation consisting on the conversion of emojis to emoticons, the suppression of vertical spacing characters, such as newline, and the lower case of the text.

We used as extra numerical features for our emotion classifier the same features that were used in the sentiment analyser, namely:

- Number of consecutive positive emoticons
- Number of consecutive negative emoticons
- Number of consecutive exclamation marks
- Number of consecutive question marks
- Number of ellipsis (. . .)
- Number of question marks
- Number of exclamation marks
- Number of combinations of questions and exclamation marks, e.g. “!?!?”, “?!?!”
- Number of question words, e.g. *how*, *what*, *why*
- Number of positive emoticons
- Number of negative emoticons
- Number of words written completely in capital letters, e.g. “HAPPY”, “VERY”
- Number of elongated words, for instance, “soooo” “looong”
- Number of negation words, for example, “not”, “never”, “without”

In addition, there are some binary features that are evaluated according to the first token or last token of the test. These features are as follows:

- Presence of question mark
- Presence of exclamation mark
- Presence of negative emoticons
- Presence of positive emoticons

- Presence of ellipsis
- Presence of full stop (this case only for the last token)

We decided to enrich the vectors for the machine learning methods by including numerical features that come from the use of lexica specialised in emotions or affectiveness. In Table 12 and Table 13, we present the numerical features used in accordance to the lexicon explored.

Table 12: Numeric features obtained from the lexica NRC Affect Intensity Lexicon and NRC Word-Emotion Association. The two lexica were used in parallel for full coverage of emotions.

Feature	Lexica	
	NRC Affect Intensity Lexicon	NRC Word-Emotion Association
Number of anger words	X	X
Anger strength	X	
Maximum anger strength	X	
Strength of last anger word	X	
Number of joy words	X	X
Joy strength	X	
Maximum joy strength	X	
Strength of last joy word	X	
Number of sadness words	X	X
Sadness strength	X	
Maximum sadness strength	X	
Strength of last sadness word	X	
Number of fear words	X	X
Fear strength	X	
Maximum fear strength	X	
Strength of last fear word	X	
Number of surprise words		X
Number of love words		X
Number of neutral words		X

5.5 Data sets

To the best of our knowledge, there are only two data sets that have been created for the emotion detection in the domains of Software Development and Software Engineering. These are described in the following paragraphs.

This first data set is the one created by [83], which is a collection of comments from Jira Issue Tracker. It is composed of almost 6k entries divided in three groups. These three groups were annotated using different granularity, sentence vs full issue comment. The emotions that are explored in the data set are *joy*, *love*, *anger*, *sadness*, *surprise*, *fear*. It also includes *neutral* texts, i.e. no emotion is present. In Table 14, we present a list showing the statistics of the corpus. This data set has been previously used by [45] to create a tool for sentiment analysis, and it was used for creating EmoTxt [14].

The second data set, is the one presented in [80], in which the authors annotated $4,800$ posts from StackOverflow. The posts were annotated with the following emotions: *joy*, *love*, *anger*, *sadness*, *surprise*, *fear*. It also

Table 13: Numeric features obtained from SenticNet 5.

Feature	Lexicon SenticNet 5
Number of positive words	X
Positive strength	X
Maximum positive strength	X
Strength of last positive word	X
Number of negative words	X
Negative strength	X
Maximum negative strength	X
Strength of last negative word	X
Number of pleasantness words	X
Pleasantness strength	X
Maximum pleasantness strength	X
Minimum pleasantness strength	X
Strength of last pleasantness word	X
Number of pleasantness words	X
Attention strength	X
Maximum attention strength	X
Minimum attention strength	X
Strength of last attention word	X
Number of pleasantness words	X
Sensitivity strength	X
Maximum sensitivity strength	X
Minimum sensitivity strength	X
Strength of last sensitivity word	X
Aptitude strength	X
Maximum aptitude strength	X
Minimum aptitude strength	X
Strength of last aptitude word	X

includes texts annotated as *neutral* i.e. no emotion present. In Table 14, we present the statistics regarding the corpus.

Table 14: Statistics regarding each corpus explored for the training and testing of the emotion classifiers. Only 80% of the corpus from Ortu et al. 2016 [83] was used for training, the rest, including the one from Novielli et al. 2018 [80] was used for testing

Corpus	Emotions						Neutral
	Joy	Love	Anger	Surprise	Sadness	Fear	
Ortu et al. 2016 [83]	362	834	346	32	457	13	3885
Novielli et al. 2018 [80]	491	1,220	882	45	230	106	1959

In this deliverable, we present the experimental approach for training models based only on the first described data set. The main reason, is that we found out about the emotion corpus of StackOverflow too late for training

a model.[33]. However, we expect that Jira's data will be useful for creating a performing tool for emotion classification, because it presents texts with different granularity levels. In other words, varied granularities may help us to determine the emotions even in small pieces of text. We removed the neutral entries from the corpus as our goal is to create an emotion classifier.[34] Furthermore, Jira's data set was randomized and then divided in training and testing sets using a proportion of 80%-20% respectively. The emotion corpus based on StackOverflow is used only for testing.

5.6 Settings

We present in the following subsections the experimental and evaluative settings that were used for experiments concerning the emotion classifier.

5.6.1 Experimental

For the experiments regarding the emotion classifier, we compare different machine learning methods and three lexica. In Table 15, we present the series of experiments that were done for this deliverable regarding the emotion classification.

Table 15: Experiments realised using 3 different multi-label classifiers. We explored in each experiment multiple parameters, such as the use of lemmatisation and a different combination of lexica.

Method	ID	Lemmas	Numerical Features	Lexica		
				NRC Word-Emotion Association	NRC Affect Intensity Lexicon	SenticNet 5
HOMER	E1R		X	X	X	
	E1L	X	X	X	X	
RaKel	E2R		X	X	X	
	E2L	X	X	X	X	
VastText	E3R		X	X	X	
	E3L	X	X	X	X	
	E4R		X			X
	E4L	X	X			X

We are not including in this deliverable the experiments using SenticNet 5 on HOMER and RAkEL, as these take longer than expected to train and subsequently optimise.[35] This is not surprising and has also been observed in the literature, like in [96, 59], where some experiments were not able to finish despite being run for a week. The main reason is that the back-end of these algorithms are decision trees implemented in *Weka*, which is not multi-threaded.

As the lexicon *NRC Affect Intensity Lexicon* does not cover all the emotions found in our training and testing data sets, we have decided to use it in parallel with the *NRC Word-Emotion Association*. Furthermore, as the lexica from NRC can only be used for research purposes, we considered that it would be appropriate not to combine it with SenticNet 5.

[33] In some cases, the optimisation of the models can take several weeks, especially those using HOMER and RAkEL.

[34] It is not possible to use neutral as a label in the classifier because we risk having wrong predictions consisting of emotions that are neutral and non-neutral at the same time.

[35] In some cases, a fold of the cross-validation could take more than one day to finish.

Regarding the optimisation of the models, we made use of the Bayesian Optimisation with an objective function based on either the median or average, whichever is the lesser, of the Multi-label Macro F-score (see Section 2.4) calculated in a 10-fold cross validation. We decided to go for a Multi-label Macro F-score, as it is a strict metric that, most importantly, takes into account the proportion of the labels in the data set. As observed in Table 14, the classes are not balanced, therefore it is important to use a metric that takes this into account.[36]

In Table 16, we present the hyper-parameters, found by the Bayesian Optimisation, for the models trained using HOMER and RAkEL. In Table 17, we show the hyper-parameters for VastText.

Table 16: Hyper-parameters used for the experiments related to the multi-label classfiers HOMER and RAkEL

Method	ID	n-grams	Skip-bigrams	Min. Freq.	Clusters/Subsets
HOMER	E1R	2	1	10	4
	E1L	2	2	15	5
RAkEl	E2R	3	1	15	2
	E2L	3	2	5	3

Table 17: Hyper-parameters utilised for training the models with the neural network VastText.

Method	ID	n-grams	Skip-bigrams	Min. Freq.	Epoch	LR	Vector dimension
VastText	E3R	1	1	22	20	0.2738	100
	E3L	1	2	40	30	0.0404	175
	E4R	1	2	25	20	0.2738	100
	E4L	2	2	20	5	0.4040	40

5.6.2 Evaluative

With respect to the evaluative settings, we decided to evaluate the trained models with 4 different evaluation metrics: Subset 0/1 Loss, Hamming Loss, Multi-label Macro F-Score and Multi-label Micro F-score. These metrics have been chosen to give a wide spectrum of which is the performance of the classifiers.

5.7 Results

In Table 18, we present the results for the experiments described in Table 15, for testing corpus based on Jira Issue Tracker posts. It should be indicated that for *Subset 0/1 Loss* and *Hamming Loss* a value closer to zero means a better performance.

We can appreciate in Table 18, by the metric Subset 0/1 Loss, that in general, models based on VastText work better than those utilising HOMER or RAkEL. In other words, all the labels of a greater number of instances have been correctly predicted. Models trained with lemmatised data perform worse than those trained without lemmas, except for "E3L", which performed better than its non lemmatised counterpart.

In Table 19, we present the outcomes of the analysis of StackOverflow corpus annotated with emotions.

[36]To use a metric that does not consider the proportion of labels, can be misleading. For example, a classifier that predicts with the most frequent label all the entries can have a high value of Subset 0/1 Loss or Hamming Loss. This might make a classifier to show a good performance, but in fact, is not doing the classification correctly for the rest of the classes.

Table 18: Results obtained from the testing corpus of Ortu et al 2016 in terms of 4 different multi-label evaluation metrics. In the case of the loss evalution metrics, the lower the value the better.

Method	ID	Macro F-score	Micro F-score	Subset 0/1 Loss	Hamming Loss
HOMER	E1R	0.562	0.788	0.331	0.071
	E1L	0.532	0.791	0.307	0.069
RaKel	E2R	0.590	**0.810**	0.290	**0.063**
	E2L	0.551	0.775	0.324	0.076
VastText	E3R	0.512	0.792	0.265	0.071
	E3L	0.581	0.796	0.285	0.068
	E4R	**0.602**	0.793	**0.260**	0.072
	E4L	0.508	0.782	0.285	0.074

Table 19: Results obtained from applying the models trained on Ortu et al. 2016 over the Stack Overflow testing corpus of Novielli et al. 2018. For the loss evaluation metrics, the lower the value the better.

Method	ID	Macro F-score	Micro F-score	Subset 0/1 Loss	Hamming Loss
HOMER	E1R	0.297	0.452	0.732	0.177
	E1L	0.296	0.466	0.704	0.156
RaKel	E2R	0.292	0.462	0.707	**0.156**
	E2L	0.290	0.455	0.740	0.164
VastText	E3R	0.282	0.467	0.670	0.176
	E3L	**0.308**	**0.481**	**0.656**	0.165
	E4R	0.273	0.448	0.663	0.189
	E4L	0.229	0.365	0.735	0.190

It can be seen in Table 19, that all methods are struggling to predict correctly the emotions found in the StackOverflow corpus. According to the values of Subset 0/1 Loss, we can determine that around 25% of the entries has been predicted completely correct. However, as models do not a high value of Hamming Loss, this means that frequently none of the labels was activated or only one label was activated but different from the correct one.

5.8 Discussion

Table 20 a summary of the number of true labels of each emotion, the number of times the label was predicted, the number of times this prediction was correct and the number of instances that were predicted without a label. The total of entries in this testing corpus is 410.

From Table 20, we can appreciate that it is a hard task to predict correctly the emotions *fear* and *surprise*. The most predicted, but also the most correctly prognosticated emotion, is *love*. The methods based in RAkEL and HOMER tend not to label more frequent entries. This means that RAkEL and HOMER are more conservative when predicting the labels.

By observing Table 20, we can understand the reason why experiment *E4R* on Jira Issue Tracker corpus, performed better than the rest of the classifiers. In first place, it predicts the greatest number of correct labels regarding *love*, *joy* and *suprise*. In second place, it produces the lowest number of instances without labels, although this does not mean that they are always correct.

Table 20: Number of labels for each emotion found in the testing corpus of Ortu et al 2016. As well, we show the total number of predictions and how many of these were correct for each model trained. We show in addition, a column concerning to the number of entries, from the testing corpus, that each model considered as without emotions.

Method	ID		Emotions						Zero Labels
			Love	Joy	Sadness	Anger	Fear	Surprise	
		Gold standard	176	81	93	71	3	10	0
HOMER	E1R	Prediction	*180*	*61*	*82*	*69*	*1*	*7*	54
		Correct	164	45	71	46	0	3	
	E1L	Prediction	*176*	*56*	*74*	*67*	*1*	*6*	56
		Correct	161	43	67	50	0	1	
RaKel	E2R	Prediction	*180*	*63*	*70*	*70*	*1*	*6*	56
		Correct	162	48	65	**56**	0	3	
	E1L	Prediction	*183*	*60*	*88*	*63*	*1*	*7*	47
		Correct	164	48	69	40	0	3	
VastText	E3R	Prediction	*186*	*71*	*95*	*65*	*0*	*0*	20
		Correct	162	53	**76**	46	0	0	
	E3L	Prediction	*187*	*64*	*77*	*62*	*0*	*5*	40
		Correct	162	53	67	45	0	3	
	E4R	Prediction	*197*	*76*	*76*	*69*	*0*	*5*	**13**
		Correct	**165**	**55**	55	51	0	**4**	
	E4L	Prediction	*178*	*61*	*98*	*70*	*0*	*0*	24
		Correct	157	41	73	50	0	0	

In Table 21 we present the same elements shown in Table 20, this time for the corpus from StackOverflow. In total, there are $2,841$ entries in the corpus grounded to at least one emotion.

This first aspect to be seen in Table 21 is the large number of instances that are predicted without any label, this number can be very large for models trained with RAkEL and HOMER. In some cases, these instances can represent 40% of the total of entries in the corpus.

In the StackOverflow corpus, the most frequently predicted emotion was *anger*, however, in the best case, *E3L*, no more than 70% of the predictions were correct. The proportion of instances totally predicted and correctly predicted with the label *love*, unlike in Jira Issue Tracker corpus, is lesser. Furthermore, we can observe that the biggest problem of the classifiers, is the detection of sadness. In some cases, the number of predictions do not reach even 40% of the total entries labelled with *sadness*, even worse, the proportion of correct predictions is at most 8%. We can observe in Table 21, that methods like HOMER and RAkEL, even though they propose fewer predictions for *love*, the proportion of the correct ones is higher than for VastText.

As it happened with the results obtained from Jira Issue Tracker corpus, the detection of *fear* and *surprise* is poor across all the methods and experiments, in some cases is null. The detection of these emotions is more noticeable due to the greater number of entries labelled with *fear* and *surprise*.

We do not have a specific reason of why the classifiers behave like this in StackOverflow corpus, especially for the emotion *sadness*. We hypothesise that the annotators had a different definition of *sadness* in the StackOverflow corpus and the Jira Issue Tracker one. In [14], they do not observe this behaviour, however, the approach

Table 21: Number of labels for each emotion found in the testing corpus of Novielli et al. 2018. As well, we show the total number of predictions and how many of these were correct for each model trained. We show in addition, a column concerning to the number of entries, from the testing corpus, that each model considered as without emotions.

Method	ID		Emotions						Zero Labels
			Love	Joy	Sadness	Anger	Fear	Surprise	
		Gold standard	176	81	93	71	3	10	0
HOMER	E1R	Prediction	*588*	*651*	*416*	*836*	*0*	*63*	859
		Correct	516	169	**97**	466	0	**4**	
	E1L	Prediction	*527*	*579*	*174*	*739*	*0*	*12*	1135
		Correct	470	157	40	**496**	0	**4**	
RaKel	E2R	Prediction	*519*	*542*	*167*	*761*	*0*	*9*	1152
		Correct	470	153	35	489	0	**4**	
	E1L	Prediction	*561*	*506*	*473*	*641*	*1*	*6*	1074
		Correct	502	124	89	458	0	2	
VastText	E3R	Prediction	*856*	*816*	*302*	*716*	*0*	*0*	482
		Correct	**603**	**225**	59	437	0	0	
	E3L	Prediction	*732*	*794*	*231*	*711*	*0*	*3*	663
		Correct	547	222	59	482	0	2	
	E4R	Prediction	*847*	*791*	*310*	*927*	*2*	*8*	**288**
		Correct	583	224	43	461	**1**	1	
	E4L	Prediction	*491*	*582*	*449*	*628*	*0*	*0*	808
		Correct	387	172	66	311	0	0	

used is different and the metrics employed are precision, recall and F-score, metrics developed for single-label classifiers.[37]

In general, VastText performed better than the rest of the classifiers, however, it is still far from being perfect. The possible reason that VastText outperformed HOMER and RAkEL, is that the neural network works like a unit instead of a set of classifiers. Moreover, VastText does not need to transform the multi-label problem into multiple single-label tasks; this means that it can consider better how different labels correlate.

The detection of the correlation of labels, such as *love* and *joy*, can improve the performance of a multi-label classifier, therefore,it is an important task. Nevertheless, it is not an obvious task for machine learning algorithms. In methods like HOMER, where the algorithm make use of a Binary Relevance approach, it loses all the information regarding dependency of labels because each label is considered separately. In RAkEL, the algorithm tries to conserve and learn the correlation between labels by using a Label Powerset strategy, i.e. considering a multi-label instance as another label. However, in algorithms based on Label Powerset, the number of instances for each individual label might get reduced while the number of possible classes increases [114]. Furthermore, Label Powerset considers the correlation of labels always as a conditional dependence which might not be always true [31].

[37] In [14] it is not clear either whether they trained two models, i.e. one per corpus, or just one. If there were two models, the authors do not provide a crossed evaluation, so we cannot determine if they saw the same phenomena as us. If it was just one model, it should be indicated that the authors considered that the corpus based on Jira Issue Tracker did not contain instances labelled with *fear* and *surprise*. However, the results do not indicate whether instances from Jira Issue Tracker were predicted with these labels, and these were assessed.

It should be noted that the optimisation of a multi-label classifier is challenging due to the evaluation of these. As we indicated in Section 2.4, it is hard to determine which case is worst, an instance with 3 wrong labels or 3 instances each with a wrong label. There are studies regarding the compatibility of metrics and how can these be optimised [31, 121], but there is still not a universal solution.

To finalise the discussion, one important element to take in consideration for selecting the best model for the detection of emotions, is the licence compatibility between CROSSMINER and the lexica/software used. The NRC's lexica, i.e. *NRC Affect Intensity Lexicon* and *NRC Word-Emotion Association*, do not have a licence suitable for CROSSMINER. With respect to the multi-label classifiers RAkEL and HOMER, the library that provides them, Mulan, is an extension of Weka. However, Weka is an open source software with a GNU General Public Licence, therefore, we are not able to use these algorithms in CROSSMINER. To the best of our knowledge, only Mulan provides a Java implementation of HOMER and RAkEL. Moreover, Meka [97], another Java-based library implementing other multi-label methods, is back-ended by Weka too. Therefore, for the emotion classifier, we do not have another option than choosing one model trained on VastText and using as lexicon SentiNet5. In this case, we have selected the one obtained with the experiment *E4R*, as it is the model giving the best performance.

5.9 Conclusion

The classification of emotions is a natural language processing task that consists in determining which emotions are present in a text.

Despite its relevance, it is a task hard to explore due to the lack of annotated data and because a text can exhibit multiple emotions at the same time. Thus, the number of tools for emotion classification in the state-of-the-art is limited, especially for text that concern domains like software engineering and software development.

In this section Deliverable 3.3, we presented the creation of a supervised emotion classifier for CROSSMINER. Being the classification of emotions a multi-label task, we explored three different approaches, one based on Binary Relevance, one based on Label Powerset, and one using a neural network.

We trained on a specialised corpus of texts from Jira Issue Tracker, and we tested on a similar corpus and on corpus from StackOverflow.

The model with the highest performance was trained with a neural network, Macro F-score of 0.602, Micro F-score of 0.793, Subset 0/1 Loss of 0.260 and Hamming Loss of 0.072. Even though the results are promising, there is still work to do, in specific, for improving the detection of emotions like *sadness*, *fear* and *surprise*. Furthermore, we need to reduce the number of cases in which the classifier provides zero emotions, however, this can be hard to achieve due the nature of multi-label classifiers, i.e. there is a risk in cases where no labels are activated.

Future work includes an investigation on how to improve the performance of the neural network by including the neutral label, not as an extra label, but as a zero label activated. This might help to the neural network to find the words or elements that truly denote an emotion. Furthermore, we expect to train a model using a StackOverflow corpus and see which is the performance.

6 Request vs. Reply Classification

It is vital for software developers, either internal or external to a project, to be aware how well communication with users is managed, as it can determine the success of a project. A way to determine this aspect of a project, is to analyse the messages from sources, such as issues trackers and forums, with tools such as the ones previously presented in this deliverable, i.e. sentiment analyser (Section 4) or emotion classifier (Section 5). Alternatively, we may determine the number of request and replies present in these sources. Put differently, for a well-managed problem, developers must be conscious whether users' requests are replied, and if not to take action.

Unlike Q&A websites, where by definition a thread can only contain one request[38] but multiple replies, in other sources of information such as newsgroups and forums, it is possible to have multiple request and replies in one same thread. Therefore, in CROSSMINER, we explore the creation of a tool that could determine, through the use of machine learning, whether a text found in any kind of communications is a request or reply.

In the following subsection, we present all the information related to the development of this tool, in terms of the state-to-the-art, the methodology and experiments conducted, along with their respective results and discussion.

6.1 State-of-the-Art

To the best of our knowledge, there is only one work in the state-of-the-art that explores the classification of messages into requests or replies [51]. In this paper, which presents the model used in OSSMETER, the authors explore different approaches for solving this classification task. More specifically, they explore unsupervised and supervised methods and series of extra numerical features, that are used along with textual features. The models, in the case of supervised methods, were trained in a cross-validation strategy using a manually annotated corpus that has messages from different sources, like GitHub, Bugzilla and NNTP Newsgroups (See Section 6.2 for more information). The final model uses an SVM with the following extra numerical features:

- Number of question marks
- Presence of previous message indicator (>)
- Subject of the message, if available, staring with the pattern *Re:*
- Presence of question words, such as *how*, *what*, *when*

Apart from the previous work, in the literature, it is possible to find researches that have tried to classify messages from software engineering and software development sources into classes, such as problems, solutions, explanations, among others. In the following paragraphs we present the most representative works in this domain.

In [6] it is presented a tool that allow distinguishing whether a post from *Linux Forums*, describes a specific problem or it is part of general discussion. For creating this tool, the authors explored different machine learning methods, such as a Radial SVM or a linear regression. Furthermore, each post was represented through a vector space model, i.e. a sparse representation, enriched with 18 extra features. These features were

[38] Users can pose questions to the person who posted, either a request or reply through comments. However, for practical reasons of explaining the Request vs. Reply Classifiers, we have omitted this aspect

lexical and contextual, such as the number of posts in a thread, Linux distribution names, URLs, exclamation marks or the presence of code.

A similar work is the one presented in [93], where the authors utilise *Conditional Random Fields (CRF)* to determine whether a post from the *SAP Community*[39] expresses a problem or a solution. Multiple CRFs were trained using as training data a corpus that was manually annotated with different levels of granularity, i.e. at sentence and paragraph levels and a varied number of label. To improve the classification of the posts, the authors explored the use of extra features that consisted in PoS tags and the position of a post in a thread. The best results were given by the model trained on the sentence-level data set using the PoS tags and the position of a post in a thread.

In [101] the authors create a system for identifying the types of issues that people post in the website *Ask Different*[40]. They classified the posts in two classes, *Anomaly description* and *Information request*, the latter is subdivided into *explanation*, *howto*, *property* and *other*. The method proposed to classify post in the previously described classes, consists in using rules and the detection of patterns. The patterns come from a discourse analyser that allows determining the discourse structure of the text.[41] Due to the use of discourse analysis and rules, the authors consider that the method is relatively domain-independent.

A related work is the one presented by [35], which consists in classifying the bugs from *OpenStack* into the following classes: *enhancement*, *bug*, *software issue*, *documentation issue*, *reliability*, *scale* and *performance*. The authors train different machine learning methods, such as *SVM*, *Hierarchical Agglomerative Clustering* and *K-Nearest Neighbour* on an manually annotated set of issues found in OpenStack. The best performing method is the one trained using the SVM.

The work of [2] proposes *CAPS (Classifying API issue Posts in StackOverflow)*, a tool that classifies whether a post in StackOverflow is related to an API issue. This tools uses a *Conditional Random Fields (CRF)* that was trained on a manually annotated corpus. The text for training the model was enriched with a set of extra features concerning aspects like the presence of the word issue in the title, links to a website or the reputation of the users.

6.2 Data sets

We make use of the corpus annotated in OSSMETER for the classification of request and replies. The corpus is a manually annotated data set that consists of messages from different sources related to the software engineering and software development. More specifically, the corpus is a collection of messages from *Bugzilla*, *GitHub* and different *NNTP Newsgroups*. In Table 22, we present the statistics regarding the corpus. The corpus was annotated by different researchers and partners from OSSMETER.

As it can be appreciated in Table 22, the size of the corpus is not very large, however it covers a series of different projects such as *Fedora*, *Red Hat Database*, *Amazon-EC2*, *Git-Wiki*, *Eclipse Platform* and *Eclipse Hudson*. It should be indicated that the annotation of a corpus is an expensive task which, in some cases, is underestimated.

Unlike the methodology followed in OSSMETER, we have split the corpus into training and testing using a proportion of 80% vs. 20%. This has been done, to know how well the model performs with unseen data.

[39] sap.com/community.html

[40] apple.stackexchange.com

[41] There are several theories regarding the discourse. For example, the *Rhetorical Structure Theory (RST)* [61] indicates that a text can be subdivided into discourse elements. These discourse elements are linked through relations like: *means*, *background* and *contrast*.

Table 22: Statistics regarding the corpus used for training and testing the Request vs. Reply classifier. This corpus was used previously in OSSMeter.

Data Source	Projects	Messages	Request	Replies
Bugzilla	8	410	131	279
GitHub	22	412	98	314
NNTP Newsgroups	4	208	77	131
Total	34	1030	306	724

6.3 Methodology

For the Request/Reply classifier, we have decided to test three different machine learning methods from the state-of-the-art (see Section 2): FastText, Linear SVM and VastText. FastText is a linear classifier based on word embeddings and a simple neural network. Linear SVM is a linear classifier that maps vectors into a high dimensional space. VastText is an in-house neural network that supports word embeddings and extra numerical features.

As we have done for the other classifiers presented in this deliverable, we tested whether the use of a lemmatiser affected the performance of the models generated by the machine learning algorithms. Furthermore, we enrich the text with n-grams (size 1-3), and in the case of VastText and the linear SVM, with Skip-bigrams (skip hole between 1-3 words). We test different minimum frequency thresholds and convert emojis into emoticons.

The vectors used in VastText and the linear SVM were enriched with numerical features too. Based on the experiments previously done in OSSMETER, we decided to use the same features explored for the classification of request and replies. However, we extended these features to detect other elements that might help to determine whether a text is a reply or request. In the following listing, we present the new features explored in this deliverable:

- Binary feature indicating if the text contains code detected with the Code Detector (Deliverable 3.1)
- Number of ellipsis (. . .)
- Number of exclamation marks
- Number of combinations of questions and exclamation marks, e.g. “!?!?”, “?!?!”
- Number of times the word upvote and their inflections occur
- Number of times the word downvote and their inflections occur
- Number of times words expressing thanks occur
- Number of emoticons
- For the first[42] and last token we detect if they contain:

  Question mark

  Exclamation mark

  Emoticons

[42]The presence of question or exclamation marks, and ellipsis, in the first token does not mean that the exact first token was those elements, but that were next to the first word token.

Ellipsis

Question words

Upvote words

Words expressing thanks

Full stop (only for the last token)

As we indicated in Section 6.2, the data that we use for the Request/Reply classifier contains portions of code. As code can introduce noise, we decided to detect it using the Code Detector developed in Deliverable 3.1. In the case a message contained a portion of text detected as code, we decided to explore three approaches for dealing it:

- Do nothing: The messages with code are kept intact.
- Suppression of code: The text portions predicted as code by the Code detector are deleted from the text. However, if a text, after deleting the code, becomes an empty string, we return to the original text. This is done to reduce the cases, in which the Code Detector wrongly detected text in English as code.
- Conversion to label: This approach consists in converting the portions of text predicted as code into a label represented by "CODE".[43] For example, the following message "I have a problem with this code:\n System.out.pritl("Hello World"); \n Can you help me?" would be converted, after pre-processing, into "i have a problem with this code : CODE can you help me ?".

It should be indicated, that even if the data for training and testing will be the same as the ones used in OSSMETER, we have improved the pre-processing of it for improving the performance of the final classifier. In the following list we present the pre-processing applied for posts coming from issue trackers like GitHub and Bugzilla:

1. Detection and suppression of previous message indicator (>): In some messages from the issue trackers is possible to find portions of text that, at the beginning of the line have the *greater than* character, i.e. ">". This character indicates that the line of text came from a previous message. The lines starting with this character are deleted, as they can introduce noise to the classifier.
2. Markdown text to plain text: It is frequent to find posts, especially from GitHub, that use the markdown syntax to format plain text. The use of it can introduce noise and we have removed through two steps. The first one consists in using the *Commonmark Parser*, a markdown parser developed by Atlassian[44], to convert the markdown text into HTML text.[45] The second step consists in using an in-house tool, back-end by *JSOUP* HTML parser, that extracts and splits the text into paragraphs[46].
3. Text normalisation: In each paragraph, we delete repetitive spacing, either vertical (e.g. newlines) or horizontal (e.g. tabulations, non-breaking space). In addition, some symbols, like quotes, are normalised and some symbols without meaning are deleted.

[43] Although it might seem hard to differentiate the label "CODE" with other occurrences of the word "code" in a text, it should be mentioned that the text is always lower case. Therefore, the label CODE will be considered as a different textual element.

[44] github.com/atlassian/commonmark-java

[45] Although the Commonmark Parser can convert from markdown to plain text, we found that some textual elements, like newline characters, were deleted. Some of these elements are vital for the next pre-processing task that consists in detecting code.

[46] We define as a paragraph, the text that was located in HTML within the tags " ".

4. Detection of code: We apply the Code Detector presented in Deliverable 3.1 and, depending on the experiment, we delete the portions predicted as code, or convert them into a label, i.e. *CODE*, that indicates the previous presence of code.

For texts belonging to newsgroups, the pre-processing varied slightly. In the following list, we indicate which were the steps followed for pre-processing newsgroups messages:

1. HTML parsing: One particular characteristic of newsgroups messages is the use of HTML to format text. In some cases, HTML is used to highlight portion of text, such as the tags "". In some other cases, an HTML version of the previous message is included at the end of the message. Therefore, it is necessary to parse all the text and delete HTML tags. [47] The text is parsed using *JSOUP*.

2. Detection and suppression of previous message indicator (>): As it happens in messages from issue trackers, in newsgroups messages it is possible to find the character *greater than* at the beginning of a line to represent text that belongs to a previous message. The lines starting with this character were deleted.

3. Splitting into paragraphs: We have split the text into paragraphs, i.e. portions of text that were separated by a newline character.

4. Detection of code: Every paragraph was analysed with the Code Detector developed for Deliverable 3.1; depending on the experiment, the code portions are deleted or converted into the label *CODE*.

The goal of the previous pre-processing steps is to get the text as clean as possible and reduce the amount of noise. It is a hard task to achieve, especially in newsgroups messages, but it might improve the performance of classifiers.

Once we have detected the code, the text is lemmatised, if indicated by the experiment, lower-cased and tokenised. The lemmatisation and tokenisation is done using NLP4J, as we explained in Section 2.5.

6.4 Settings

We present in the following subsections the settings that were used for doing the experiments, but also those used to evaluate the performance of the resulting models.

6.4.1 Experimental

In Table 23, we show the experiments effectuated regarding the Request/Reply classifier. The column *ID* in Table 23, makes reference to the unique ID that represent the experiment along this deliverable.

To obtain the best model possible for each experiment, we performed a Bayesian Optimisation, where the objective function was either the median or average, whichever is the lesser, Macro F-score (see Section 2.4) of a 10-fold cross validation. We decided to use the Macro version of the F-score, because our data sets are

[47] It is hard to differentiate both cases in which HTML tags are used. Regular expressions could be useful in some cases, especially to detect patterns like "NAME_USER wrote:". However not all the users have an e-mail in English, and the patters can be in other languages. Furthermore, some users reply a message by intercalating their own text. This process is a double-edged sword as we can delete or leave more information than necessary. After all, we consider we took the less risky decision.

Table 23: Experiments done for the Request vs. Reply classifier; each experiment was done using different machine learning algorithms, but also varying the use of lemmatisation, numeric features and approaches to deal the code.

Method	ID	Lemmatisation	Numeric Features	Code approach
FastText	RR1R	No	No	Intact
	RR1L	Yes	No	Intact
	RR2R	No	No	Label
	RR2L	Yes	No	Label
	RR3R	No	No	Deleted
	RR3L	Yes	No	Deleted
SVM	RR4R	No	Yes	Intact
	RR4L	Yes	Yes	Intact
	RR5R	No	Yes	Label
	RR5L	Yes	Yes	Label
	RR6R	No	Yes	Deleted
	RR6L	Yes	Yes	Deleted
VastText	RR7R	No	Yes	Intact
	RR7L	Yes	Yes	Intact
	RR8R	No	Yes	Label
	RR8L	Yes	Yes	Label
	RR9R	No	Yes	Deleted
	RR9L	Yes	Yes	Deleted

imbalanced. Therefore, we need to assess the performance of the classifiers using a metric that considers equally important to predict correctly both classes, in spite of the proportion is not being the same.

In Table 24, we present the parameters, found by the Bayesian Optimisation for each experiment using the linear SVM. The parameters for FastText and VastText are shown in Table 25.

Table 24: We present the linear SVM parameters used for training the models regarding the Request vs. Reply classifier.

Method	ID	n-grams	Skip-bigrams	Min. Freq.	C
SVM	RR4R	3	2	3	2^{26}
	RR4L	3	0	1	2^{16}
	RR5R	1	1	10	2^{5}
	RR5L	1	1	3	2^{6}
	RR6R	2	1	10	2^{8}
	RR6L	3	2	4	2^{30}

6.4.2 Evaluative

We evaluate the classifiers here presented using *precision*, *recall* and *F-score* for both possible classes. Furthermore, we evaluate globally the classifier through the calculation of *macro* and *micro F-score*.

Table 25: We present the parameters used for training the models related to each experiment using FastText and VastText. It shold be noted that FastText do not accept skip-bigrams.

Method	ID	n-grams	Skip-bigrams	Min. Freq.	Epoch	LR	Vector dimension
FastText	RR1R	2	-	10	40	0.1366	250
	RR1L	2	-	5	15	0.1642	250
	RR2R	2	-	50	20	0.4063	100
	RR2L	2	-	45	20	0.2353	125
	RR3R	3	-	50	40	0.5000	200
	RR3L	2	-	45	25	0.3384	175
VastText	RR7R	2	1	2	10	0.3885	75
	RR7L	2	3	4	30	0.3057	125
	RR8R	1	1	50	30	0.3489	35
	RR8L	2	2	40	5	0.3102	125
	RR9R	1	2	30	10	0.1449	75
	RR9L	3	2	5	30	0.1192	50

6.5 Results

In Table 26, we present the results regarding the experiments done for the Request/Reply classifier that were described in Table 23. It should be indicated, that for this classifier, the scores in terms of Macro F-score are the most relevant, as they indicate how well classifiers perform regardless the proportion of the classes. In other words, we consider equally important to predict correctly both classes, despite the fact that *replies* are more frequent than *requests*.

We can observe in Table 26 that leaving the code in the text allow us to achieve better performance (experiments *RR1*,*RR4*, *RR7*), in term of Macro F-score, than deleting it or converting it into labels. We can see, as well, that the lemmatisation of the text helps the SVM to identify request and replies better in all the cases. Furthermore, despite FastText do not use any kind of extra features, it has a performance, in terms of Macro F-score, similar to the SVM. Nevertheless, we can state in Table 26, that it is a hard task to predict correctly texts labelled in the ground truth as *request*. The highest value of F-score for the *request* class is 0.611 given by model *RR4L*, however, some other models give a score as low as 0.400, i.e. *RR5R*.

The results given by our neural network VastText are underperforming as we can observe Table 26. In general, the neural network has the lowest Macro F-scores, meaning that models using this approach are the worst detecting correctly the class *request*.

6.6 Discussion

It is interesting the fact that leaving the code intact in the messages offered a better performance than deleting it. There are three possible reasons for this behaviour. The first one is that the subjects discussed in the data set, training and testing, is very limited and the code presented inside is quite similar. By keeping the code, we can classify better the texts, as we have more textual elements that represent the classes request and reply. The second possible reason is that the Code Detector developed in Deliverable 3.1, is struggling to detect correctly certain expressions in English and, in consequence, we are deleting or converting into labels, information that should be kept intact. We suspect of this last reason, because, if the presence of code, as an entity, would be the decisive factor, the experiments where the code was changed by a label, should work as well as leaving the code intact. However, this may not be completely due to a faulty training of the Code Detector *per se*,

Table 26: Results obtained from applying the different trained models for the request vs. reply classification on the testing corpus. Results are expressed in terms of Precision (P), Recall (R), F-score (F1), Micro and Macro F-score.

Method	ID	Reply			Request			Global F-score	
		P	R	F1	P	R	F1	Micro	Macro
FastText	RR1R	**0.855**	0.721	0.782	0.520	**0.716**	0.605	0.720	0.694
	RR1L	0.833	0.785	0.808	0.558	0.633	0.593	0.740	0.701
	RR2R	0.839	0.635	0.723	0.457	**0.716**	0.558	0.660	0.641
	RR2L	0.813	0.714	0.760	0.480	0.616	0.540	0.685	0.650
	RR3R	0.801	0.721	0.759	0.472	0.583	0.522	0.680	0.640
	RR3L	0.809	0.728	0.766	0.486	0.600	0.537	0.690	0.652
SVM	RR4R	0.807	0.871	0.838	0.632	0.516	0.598	0.765	0.703
	RR4L	0.822	0.892	**0.856**	**0.687**	0.550	**0.611**	**0.790**	**0.733**
	RR5R	0.754	**0.942**	0.838	0.680	0.283	0.400	0.745	0.619
	RR5L	0.764	0.928	0.838	0.666	0.333	0.444	0.750	0.641
	RR6R	0.758	0.921	0.832	0.633	0.316	0.422	0.740	0.627
	RR6L	0.791	0.842	0.816	0.568	0.483	0.522	0.735	0.669
VastText	RR7R	0.816	0.764	0.789	0.521	0.600	0.558	0.715	0.558
	RR7L	0.840	0.792	0.816	0.573	0.650	0.609	0.750	0.609
	RR8R	0.778	0.828	0.802	0.529	0.450	0.486	0.715	0.486
	RR8L	0.803	0.757	0.779	0.500	0.566	0.531	0.700	0.531
	RR9R	0.796	0.728	0.761	0.472	0.566	0.515	0.680	0.515
	RR9L	0.821	0.821	0.821	0.583	0.583	0.583	0.750	0.583

but more due to the pre-processing that we applied for extracting the text from the corpus, as we explained in Section 6.3. In other words, the pre-processing is either leaving elements that generate a context similar to the one used in code, such as escaped HTML tags, or deleting essential information for considering all the context as English, e.g. reply intercalated between the original request.

Furthermore, it is an unexpected outcome that the best model is given by a combination of lemmatisation and leaving the code intact. The reason is that, in the perspective of natural language processing, programming language would be considered as a source of noise, it would be like having, for example, a text with a mixture of English and French, certain words in both languages could be shared or written similarly, but *a priori* this would make difficult to understand clearly the text.

With respect to the low performance of VastText, this is due to a poor optimisation. We never arrived to finish the optimisation process with VastText, because for an unknown reason, DeepLearning4j crashed with a certain combination of parameters. We tried to find out which was the pattern that caused the crash, but we did not arrive to determine one. In the end, we trained with the best hyper-parameters that the Bayesian Optimisation determined before the crash by analysing the logs.[48]

Although it is impossible to do an exact comparison between the tools developed in OSSMETER and the one presented in this deliverable, due to the way the data set was used[49], it should be indicated that the classifier

[48] In fact, we ran multiple times the optimisation trying to avoid certain combinations of hyper-parameters, but none of them finished. At the end, we submitted an issue ticket in DeepLearning4J GitHub.

[49] As we said previously, in OSSMETER the data was used only in a cross validation way. For CROSSMINER, we have split the data into train and test, meaning that one portion of the corpus was never seen until the testing process.

presented here has a Micro F-Score 2.54 points better than the one used in OSSMETER, which had a Micro F-score of 0.764.

6.7 Conclusion

For developers, it is important to know, along all the communications means related to a project, elements that can give an idea of how to improve the management of the project and how is the management of a project is being done. One key element for detecting this, is the detection of *request* and *replies*, for example, a project with multiple request but few replies could imply that either the project is no longer active, or that developers do not know how to manage their users community. Moreover, the detection of these elements, can provide information to generate more metrics, like those presented in Deliverable 3.4, that could improve the management of the project. For instance, it is possible to determine if a comment in an issue has been replied, or to determine how many requests have been done in a day.

Therefore, in this section of Deliverable 3.3, we have presented a tool that classifies messages posted in different sources related to software engineering and software development into two types *requests* and *replies*. We have explored three different approaches for creating a model that could improve the request and reply classifier that was used previously in OSSMETER. Being more specific, we tested with two neural networks that use internally dense representations and a Linear SVM that uses sparse representation.

The Linear SVM outperforms the other methods, but also the method proposed for OSSMETER. With the experiments presented here, we observed that, for this kind of classifiers, an approach that consists in using extra numerical features, and lemmatised text including code might be the best. Moreover, we observed, due to the poor results of the neural network VastText, that it is very important to do an optimisation.

In the future, we expect to see whether a model trained on questions and accepted answers from StackOverflow can improve the correct detection of request and replies in messages related to software engineering and software development.

7 Content Classifier

Content classification, also referred to as content analysis, provides automated context-sensitive analysis for the organisation of documents, based upon pre-established categories [64]. Content classification has numerous applications in a variety of areas such as book recommendations [76], purpose of messages in forums [9] or even assigning labels associated with particular types of failures based upon the contents of a crash report [122].

Content classifiers should not be confused with more general classifiers. In the former, the content of a source of information, such as a text or image, is split into multiple sub-elements that are analysed and classified individually. While in a more general classifier, the whole content of the information is categorised into one or multiple classes. For example, a movie content analyser would say that the movie *Metropolis* of *Fritz Lang* contains scenes of love, action and surrealism, while a general classifier of movies genre, would say that it is a sci-fi movie.

Within the context of CROSSMINER, a content classifier is a tool that allows determining which elements, such as *Request of clarification* or *Suggestion of a solution*, are being triggered within messages, posts and comments posted related to open source projects. In other words, CROSSMINER content classifier is a tool that would give developers indicators that summarise what the messages exchanged by users and developers contain. The relevance of this tool, is that the outcome can be simultaneously used by metrics that give a general idea of the quality of a project, but at the same time to create alerts for developers concerning an action that should be done as soon as possible. For example, it might be of interest to developers to know when a user has posted a message reporting a bug and, at the same time, has proposed a solution of it.

In CROSSMINER's predecessor project, OSSMETER, a similar tool was developed, however, it was only capable of indicating one of the multiple possible contents of a message, despite a text might talk about several contents. Therefore, as part of task 3.1, we use a content hierarchy, previously defined in OSSMETER but modified here for CROSSMINER, along with different multi-label classifiers, *VastText*, *HOMER* and *RAkEL*, to explore the creation of a content classifier that could indicate all the content elements found within messages related to open source projects. The results that will be presented in this section of the deliverable, show that *VastText* is the most adequate method for creating a content classifier, but some improvements should be done to increase its performance.

In the remaining sections, we explore the state-of-the-art relating specifically to content classification and, we provide an overview of the content classification approach integrated into OSSMETER. In succession, we proceed to present details of the methodology used for experimentation, followed by a discussion about the data set utilised for the experiments. In the latter part of this section we present details of the experimental and evaluative settings, results obtained from experimentation, a discussion of the results and a conclusion.

7.1 State-of-the-Art

The goal of our work is to develop a content classifier to identify types of posts, messages and comments found in open source software repositories. Presented throughout this section are various research works that demonstrate state-of-the-art content classification in diverse domains.

Bhatia *et al*. [9] studies the problem of classifying individual posts as per their role and or purpose within online forums in general. They classify posts based upon the following labels: *Question, Repeat Question, Clarification, Further Details, Solution, Positive Feedback, Negative Feedback and Junk* ,with the aim to intelligently facilitate the improvement of: how information is presented, how information is extracted, and how user roles are defined within the online forum. For each post, they enriched the feature space with various

features relating to: Content, Users, Structure and Sentiment/Emotion. Features relating to *content* include cosine similarity scores[50], presence of question marks and question words and the presence of quotes or previous posts. With regard to *user* features; the authority score, i.e. the number of posts, and an indication whether they are the thread initialiser are used. Features relating to *structure* included the location of a post within a thread and the length of post. Stemming and the removal of stop words were all performed during this stage. *Sentiment/Emotion* features included sentiment strength scores, calculated using the SentiStrength algorithm, and punctuation and keywords that are considered as emotional indicators such as *! and love*. They experimented with a variety of supervised machined learning approaches that included: Support Vector Machines, Naive Bayes classifier, Decision tree, multi-layer perceptron and logistic regression. They report that logistic regression was the best performing achieving an accuracy of ~72%.

In the work of Yamada *et al*. information contained within Japanese tweets [51] was used to extract and assign labels to information relating to localised events [124]. Events can be categorised into one or more of the following labels *seasonal*, *music*, *culture*, *pop-culture* and *life*. Overall they report that the best performance was from a feature space that consisted of both the event information and presence of keywords; achieving an overall accuracy of ~67%. This was used as a component within a recommendation system that provided users with personalised event recommendations.

In LIBRA [76], the authors created a book content analyser that uses different aspects such as the book's title, text, author along with synopses, published reviews and previous readers comments. The book content analyser is based on a Bayesian text classifier that allows LIBRA to determine the probability, and in consequence the strength, of elements of a book to belong to descriptions such as, reality, intellect or world. LIBRA, in other words, creates a profile of the book automatically in terms of its content. After that, this output is used for recommending books to readers based as well on profiles.

Argumentation refers to the action or process of reasoning systematically in support of an idea, action, or theory and consists of several components such as argument, premise and conclusion [118]. Palau and Moens [86] utilised a chain of classifiers for identifying content relating to *argumentation* in structured text, such as newspaper articles, parliamentary records and online discussion boards. Via utilising a linguist theory called *Rhetorical Structure Theory* (RST), a technique in which natural language is organised by how its parts are related [60], they were able to enrich the feature space to identify if a post is an argument, classify argumentative clauses (premise or conclusion) as well determine the limits of an argument and the relations it holds with other surrounding arguments.

The work of Agarwal *et al*. [1] exploits the textual meta-data associated with YouTube videos and explores the use of two multi-class classifiers to provide an overall multi-label classification. Their goal is to provide a more efficient mechanism for identifying *"unwanted"* video content that contains acts of violence or harassment in YouTube videos and is designed to be use by law enforcement and intelligence agencies. More specifically, it identifies if content, *i.e*., the video, posted on YouTube is a form of harassment or not and if it is, the next classifier assigns a further label, one of five labels that distinguish its type. Their feature space utilised only the meta-data associated with the YouTube video and included: the title, number of likes, number of dislikes, comment count, total views and the id of the category.

Alfaro *et al*. [4] utilise a multi-stage method that consists of both supervised and unsupervised methods for automatic detection of different opinion trends of comment posts in online blogs, relating to a particular candidate. The content classification consists of three stages, the first two of which are supervised. The first identifies the candidate, the second performs sentiment analysis to indicate the polarity (positive, neutral or negative) of

[50] How related the post is with the thread title, first post and whole thread
[51] Tweets are messages posted by users on Twitter.

the post. Finally, K-means clustering, an unsupervised machine learning method, is used to extract relevant keywords based upon the polarity characterising different opinion trends.

Xia *et al*. developed a composite mutli-label learning genetic algorithm (MLL-GA) for assigning labels associated with particular types of software failures based upon the contents of a crash report [122]. The objective of this classifier is to provide, to a software engineer, contextual knowledge relating to the specific software failure to improve the overall effectiveness of addressing the problem.

Zhou *et al*. [130] developed a composite method for assigning semantic categories to posts made in online discussions related to APIs[52]. The categories consisted of terms that were related to a particular component under discussion. For example, if a post contained a discussion and some related code, using this information their composite method could indicate it is referring to a specific element associated with the API, such as layout or dispose functions.The output is used to facilitate the retrial of useful and relevant API information for developers.

The works presented in this section have been ordered so that those in the beginning demonstrate state-of-the-art content classification in general, broad terms, whereas those towards the end are more related to our aim. With that said, to the best of our knowledge there is no state-of-the-art literature that explores content classification with sources that are directly related to the topic of our work, apart from the work conducted in OSSMETER, which is discussed in detail further details in Section 7.2.

[52] Application Programmable Interfaces

7.2 OSSMETER Approach

The aim of OSSMETER content classifier was to classify the *overall* content contained within bug tracking system comments and newsgroup articles related to open source software repositories. Initially the objective was to develop a multi-label classifier to provide information back to the developer, which would be more granular and representative of all the content within a specific comment or article. However, due to paucity of libraries in Java, with a compatible licences, during the time of its development, a multi-class classifier was developed. The multi-class classifier was a linear SVM built using the open source library LibSVM. The classifier was capable of assigning 1 of 37 potential classes (labels) to comments or articles. Each class was organised into a hierarchical structure[53] that consisted of 8 high level categories each of which are summarised below:

1. Clarification of a request or solution
2. Suggestion of a solution
3. Resolution of a request or problem
4. Request or report
5. Unsuccessful solution notification
6. Action notification
7. Resolution acceptance
8. Other information provision

The corpus used consisted of messages contained in threads extracted from NNTP newgroups that related to Eclipse. It was manually annotated by several volunteers to form a gold standard set of annotations[54]. The data was subjected to cleaning process and the feature space was enriched with the same heuristics presented in [51]. An overview of the heuristics are presented below:

- Position of each message in the thread it belongs to
- Indication whether a message contains a reply
- All parts-of-speech
- TF-IDF features
- Uni-grams filtered of stop-words
- Word Features such as n-*grams*

Overall the content classifier used in OSSMETER achieved an accuracy of ~70%.

[53] The hierarchy is discussed in more detail in Section 7.4
[54] The data set is discussed in detail in Section 7.4

7.3 Methodology

Our intention is to provide CROSSMINER with a classifier based on supervised machine learning that would determine the content of text written by a developer or user of a project. As with the emotion classification task, the same machine learning approaches were used as the basis for experimentation with the content classifier, i.e. *VastText*, *HOMER* and *RAkEL*.

Prior to enriching the feature space, the raw data was subjected to a preprocessing work flow that consisted of the following stages:

1. **Splitting text into paragraphs:** As a consequence of how the data is parsed, it was important to process the raw text to provide structure allowing more control of the volume of text retained as a result of later stages of pre-processing.
2. **Text normalisation:** This stage involves the deletion of repetitive spaces, such as new lines or tabulations. Symbols such as quotes are converted to a common consistent form and other symbols without concrete meaning are deleted.
3. **Removal of HTML Tags:** Messages from newsgroups often utilise HTML to format or emphasise text. For example the tags <*b*> Hello <*/b*> are used to format the text between them as bold. Other examples included using HTML to insert excerpts from other messages to provide context. Therefore, HTML tags were removed from each paragraph to reduce noise.
4. **Detection and suppression of reply messages (>):** The *greater-than* character is used at the beginning of a line in newsgroup messages to represent a previous message from a thread. Lines that began with > were removed to reduce noise being introduced by reply messages that were included and a flag was triggered.
5. **Detection and suppression of code:** During this stage each paragraph, was analysed by the Code Detector that was developed in Deliverable 3.1. Based on the output, the text was reconstructed removing all paragraphs that contained code and a flag was triggered to indicate the text contained code. In the some instances were the number of paragraphs in the reconstructed text was zero, the text passed into the Code Detector was kept.

The feature space associated with each message was enriched with the following binary and numerical features:

- **Binary Features:**
  - Contains code, *status of the flag triggered during preprocessing*
  - Contains reply message, *status of the flag triggered during preprocessing*
- **Numerical Features:**
  - Weight of Natural Language in text[55], *e.g. 0.8*
  - Weight of Code in text[56], *e.g. 0.2*
  - Frequency of exclamation marks, *!*

[55] Calculated based on percentage of natural language blocks Vs. code blocks in comments
[56] See footnote 55

- Frequency of question marks, *?*
- Frequency of question words, *e.g. who, what, where, why*
- Frequency of thanks words, *e.g. thanks, thank, thanx*
- Position of a message in a thread
- Frequency of help words, *e.g. advice, help, guidance*

7.4 Data set

Apart from the corpus developed and utilised in OSSMETER, to the best of our knowledge, there is currently no alternative corpus available that is suitable for assigning labels based upon the content of messages within the software engineering and software development domain. The data set used in OSSMETER consists of messages structured in threads. Threads were extracted from NNTP newsgroups that related to *Eclipse*. In OSSMETER, the data set was used for several tasks. The first task created a hierarchy of labels that consists of a diverse range of content found within the messages. The second task involved annotating the corpus, using the hierarchy developed in the first task. This was undertaken by several volunteers, whose annotations were used to produce a *gold standard* set of annotations by calculating the inter-annotator agreement scores.

The *gold standard* corpus was utilised in each of the content classification experiments. As briefly mentioned above, the data consists of a number of threads each containing numerous messages. Each message is assigned at least one label. Table 27 presents statistics about the *gold standard* corpus.

Table 27: Content Classifier gold standard statistics

Statistic	**Total**
Number of Threads	1164
Number of Messages	3335
Total number of annotations	4111
Number of messages annotated with a single label	2659
Number of messages annotated with multiple labels	676

7.4.1 Content Classification Hierarchy

As mentioned earlier, the content-based hierarchy, shown in Figure 2, covers a diverse range of content types found in on-line communication channel messages. More specifically, it consists of a total of 37 labels each grouped into 1 of 8 categories. It is worth noting that only the leaf nodes may be assigned to individual messages. Super-classes that have descendants exist for semantic integrity, grouping similar or related classes together. This means that the label *1.2. Clarification of request* can be assigned to a message , whereas *1. Clarification* cannot. Furthermore, a message may also be assigned another label that belongs to the same group. For example, a message may not only thank a developer for their help, but also indicate that they have discovered a new bug.

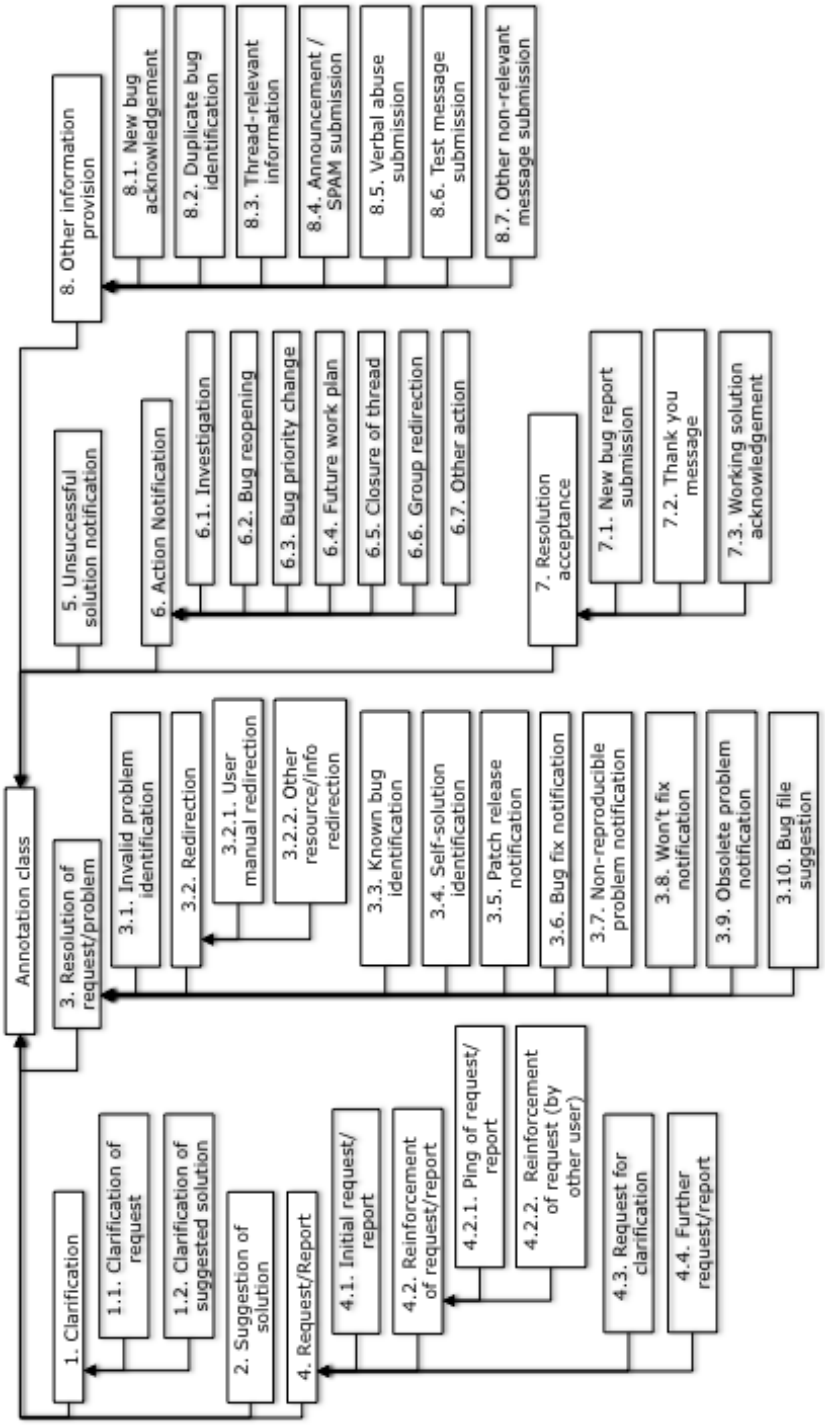

Figure 2: Content Classifier label hierarchy used in OSSMETER

7.4.2 Refactoring of Label Hierarchy

As indicated in Table 27, the data set was small, in contrast to others used in this deliverable, and the number of labels within the hierarchy is large in relation to the size of the data. This fact raised the question: *is the size of the data set sufficient in providing enough data for all 37 labels within the hierarchy?* Whilst this level of granularity would provide developers with specific knowledge regarding the content of a message, many of the labels are under-represented and some are not represented at all within the *gold standard*, as can be seen in Figure 3. It would be extremely difficult for labels such as 6.3 , 8.5 and 3.2.1 to be distinguished by a machine learner. Therefore, we decided to reduce the number of potential labels, by mapping each label back to its root ancestor, resulting in a total of eight, more general labels[57], as shown in Figure 4.

Once the data was processed following the methodology described in Section 7.3 the corpus was split randomly[58] into training and testing bins. We ensured that in both bins, the relative distribution of each label combination, either single or multi-label, is representative of the distribution found in the entire corpus.[59]

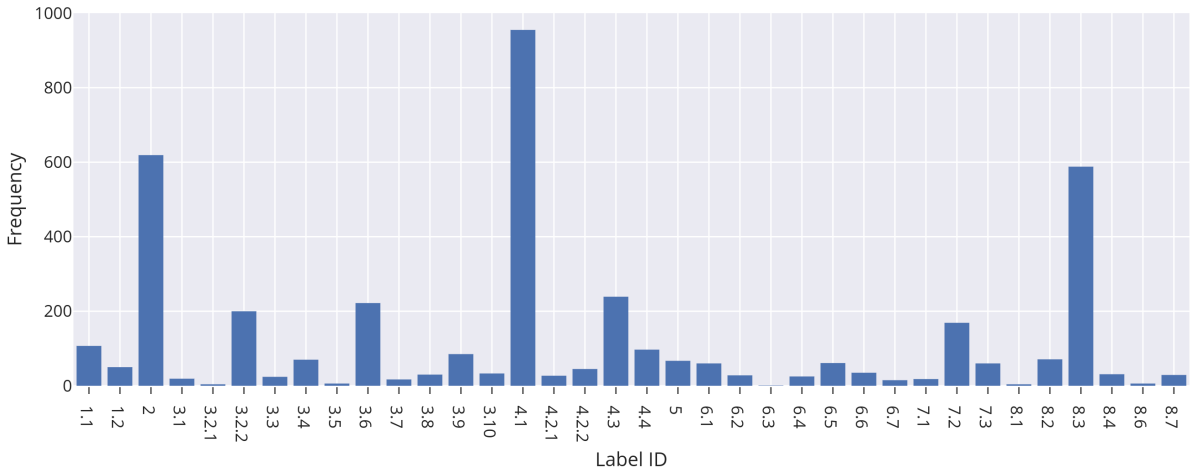

Figure 3: Graph presenting the frequency of occurrence for each label in the gold standard.

[57] A description of each label within the refactored hierarchy is available on Appendix A

[58] Training = 80%, Testing = 20%

[59] In the case a label combination occurs just once in the corpus, the instance labelled with this combination was set in the training set. If there were only two instances labelled with a specific combination of classes, one instance was used training while the other for testing.

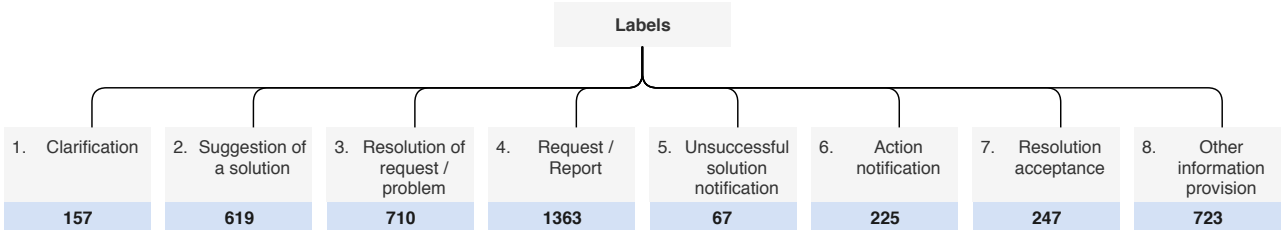

Figure 4: Refactored Content Classifier label hierarchy for CROSSMINER. Distribution values for each label in the data set are shown in the blue boxes.

7.5 Settings

The following subsections present the experimental and evaluative settings that were used for experiments associated with the content classifier.

7.5.1 Experimental

Each classification method was executed using the same experimental conditions, presented in Table 28. With respect to parameter tuning, we used Bayesian Optimisation with a objective function based upon the median or average, whichever is the least, of the *Multi-label Macro F-score* calculated in a *10-fold cross validation* for the same reasons presented in Section 5.6.1.

Table 28: Content Classifier: Methods and features used in each experiment

Method	ID	Pre-processing	Reduced Labels	Lemmatisation	Binary Features	Numerical Features
VastText	VT1	✓	✓	No	All	All
Homer	H1	✓	✓	No	All	All
RAkEL	R1	✓	✓	No	All	All

Tables 29 and 30 present the optimal parameters computed by Bayesian Optimisation for VastText and HOMER, respectively. For each of these models, 10-fold cross validation was completed successfully. For reasons still unknown, partially completing the optimisation of *RAkEL* took very long time. In particular, one cross validation fold would take several days to complete. The experimentation was terminated at iteration 25 out of 35, therefore, the parameters shown in Table 31 are not fully optimised, but instead the *best available* parameters computed in the part of the optimisation process that was completed.

Table 29: VastText: Optimised parameter values

Method	ID	n-grams	skip-bigrams	Min. Freq.	Epoch	LR	Dim
VastText	VT1	1	0	1	20	0.403	100

Table 30: HOMER: Optimised parameter values

Method	ID	n-grams	skip-bigrams	Min. Freq.	Subsets
HOMER	H1	1	2	1	2

Table 31: RAkEL: *Best available* parameter values

Method	ID	n-grams	skip-bigrams	Min. Freq.	Subsets
RAkEL	R1	3	0	2	5

7.5.2 Evaluative

Similarly to the emotion classifier, discussed in Section 5, we chose to evaluate each of the trained models with the following four evaluation metrics: *Subset 0/1 Loss*, *Hamming Loss*, *Multi-label Macro F-Score* and

Multi-label Micro F-score. These metrics have been chosen because they provide an indication of model performance, from various perspectives.

7.6 Results

Results per evaluation metric have been computed using the optimised parameter values for *VastText* and *HOMER*, and using the *best available* parameter values for *RAkEL*. Table 32 presents the results.

Table 32: Content Classifier experimentation results

Method	ID	Macro F-score	Micro F-score	Subset 0/1 loss	Hamming loss
VastText	VT1	**0.488**	**0.652**	**0.442**	0.101
Homer	H1	0.451	0.613	0.508	0.119
RAkEL	R1	0.447	0.618	0.515	**0.098**

It can be observed that all approaches performed similarly. *VastText* performed better than the other classification methods by a small margin, only. *Macro F-score* considers all labels to have equal importance, whereas *Micro F-score* takes into account the distribution of class labels. The fact that *Micro F-score* results are better than *Macro F-score* results indicates that all classification methods are more successful in assigning labels for which many training and test instances are available. *Subset loss 0/1* is a strict, direct-matching evaluation metric. It indicates the percentage of test instances for which a classification method did not predict all its correct labels according to the gold standard and only those labels.

7.7 Discussion

Table 33, provides details about the number of predictions made by each classification method, how many of them were correct and the number of instances that were not assigned any label. The highest number of correctly predicted labels is highlighted in bold.

Table 33: For each content classfication experiment, this table shows for the number of predictions made for each label, how many of them were correct and the number of instances where no labels were assigned by each classifier

			Label ID								
Method	**ID**	**P | C**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**No Label**
VastText	VT1	Prediction	8	133	133	313	3	27	18	131	35
		Correct	2	**72**	**80**	**235**	**2**	24	10	94	
Homer	H1	*Prediction*	24	136	161	290	4	44	28	175	52
		Correct	**3**	71	75	224	0	**25**	**17**	**103**	
RAkEL	R1	*Prediction*	10	103	68	268	2	16	26	87	168
		Correct	2	58	50	218	1	14	**17**	75	
		Gold standard	3	126	141	272	16	46	43	149	0

Subset 0/1 loss provides the percentage of instances that were wrongly labelled. Values closer to 0 indicate better results. As can be seen in Table 32, *VastText* achieved the best result, although still not ideal . It

performed between ~6 and 7% better than *HOMER* and *RAkEL*. Taking into consideration the Hamming Loss, which shows how different each prediction is from the corresponding expected outcome, we can observe that all methods have performed comparably.

Common among all classification methods is the ability to correctly predict label 4, which indicates that the content of a particular instance contains a *Request or Report*. This is probably because this class is present in ~34% of messages in the corpus. Furthermore, all methods found it difficult to predict label 5, *Unsuccessful solution notification*. This is most probably a consequence of its very infrequent occurrences in the corpus, although it was a leaf node in the original hierarchy.

Due to the small size of the corpus, being unable to classify many instances affects the results dramatically. It can be observed that *VastText* did not predict any labels for the fewest instances, followed closely by HOMER. *RAkEL* performed significantly worse. This could be attributed to the nature of RAkEL, which consists in transforming multi-label instances into single label instances through a Label Powerset transformation. This may have caused loss of information about the features that belong, individually, to each label. For example, consider the instance "We are investigating this bug. Thank you for telling us.", labeled in the gold standard as *Resolution Acceptance* and *Action Notification*. RAkEL keeps the former label only, and discards the latter. As a result, once this transformation has taken place, RAkEL cannot determine that "Thank you for telling us." should be uniquely related to *Resolution Acceptance*, whilst "We are investigating this bug." matches the *Action Notification* label.

This loss of information does not happen with HOMER, which transforms the labels using Binary Relevance. In this case, the aforementioned example would be repeated twice, with each copy annotated with just one label. Nonetheless, this does not solve the problem completely, especially in cases where an instance is assigned many labels.

VastText, due to its neural network nature, does not apply any transformation, which be the reason why it had the lowest number of unlabelled instances.

7.8 Conclusion

The textual content of messages, posts and comments can be analysed to provide developers with useful information about the communication taking place in Bug Trackers, communication channels and forums associated with an Open Source project. Content, in this work, refers to different functionality types, *e.g. suggestion to a solution*, *acceptance of a resolution*, *clarification*. This information can be used to summarise the characteristics of communication, which in turn can inform about the quality of support offered via these communication means and suggest ways to improve it. The overall goal of this work was to design and develop a content classifier, through exploring diverse methods that use different text representation models.

The content classifier developed for OSSMETER was a multi-class linear SVM capable of detecting 37 types of content in posts, comment and articles related to open source software. It was developed using a corpus that consisted of a *gold standard* set of annotations of messages contained in threads relating to Eclipse. Overall, the content classifier integrated into OSSMETER performed well. However, due to the nature of a multi-class classifier it was restricted to assigning only a single label to a message. This meant that some content contained within the text would remain hidden from the users of the platform.

For CROSSMINER, we explored multi-label methods to represent the content in messages better. Specifically, we used *VastText*, a neural network based machine learner, *HOMER*, a learner based on Binary Relevance, and *RAkEL*, which uses a *Label Powerset* transformation. We utilised the same corpus used in OSSMETER. However, due to the distribution of classes, we chose to reduce the hierarchy classes from 37 to 8, minimising

the impact of individual classes being under-represented. Furthermore, we experimented with multi-label classifiers to address the limitation of OSSMETER's single-label approach. Each experiment used the same methodology and the feature space was enriched with ten heuristics. Overall, the results show that *VastText* performed best in terms of *Macro F-score*, *Micro F-score* and *Subset 0/1 loss*. We discussed the results based on similarities and differences of the methods, and analysed classes that the models predicted successfully or not.

Our preliminary results indicate that the VasText method is a potential tool for content classification in CROSSMINER. Not only because, it was the best performing, but also its licence is compatible with the requirements of the project. However, further work still need to be undertaken in order to improve the overall performance. For example, we expect to expand and enrich the feature space with additional observations, such as features based on Rhetorical structure Theory (RST), in order to increase the overall performance.

8 Severity Classifier

A critical part of software maintenance is the so-called debugging, i.e. the identification and removal of concealed errors/defects. In a small software project involving a single developer, the task is relatively straightforward. However, it is a major concern in larger projects, requiring bug reports from numerous developers and a tracking system to manage and resolve each bug report. Generally, each bug report is manually assigned a severity level to indicate the degree to which the reported bug impacts software quality. However, this task is known to be inefficient as it usually takes much time and human resources [125]. In this section, we report an attempt to automatically predict the severity of a software defect using information from textual content only. In other words, this section of the deliverable addresses the task of grouping threads that discuss issues or software bugs that are similarly severe to a software application.

In the following sections, we present a synopsis of previous work related to severity prediction. This is followed by a short description of our previous work in OSSMETER project to address the same issue. Furthermore, we present the approach explored in this deliverable and compare results with the OSSMETER approach.

8.1 State-of-the-Art

Several studies have attempted to automate bug severity prediction. These studies can be grouped into fine-grained (multi-class) and coarse-grained (binary class) prediction efforts. These are discussed separately in this section.

As an early work, Menzies and Marcus [68] proposed the *SEVERity ISsue* assessment algorithm, better known as *SEVERIS*, to assist test engineers in assigning severity levels to bug reports. The approach is based on entropy and information gain, supported by a rule learner. Specifically, SEVERIS applied standard text mining techniques such as tokenisation and stemming to several bug reports; and used Cohen's RIPPER rule based classifier [24] to perform fine grained prediction involving 5 bug severity levels. An average of 775 reports composed of 79,000 terms were examined with optimisation results of individual severity level ranging from 65% to 98% in terms of F-Score.

In other related fine-grained bug severity prediction studies, Tian et al. [113] proposed an algorithm called *INSPect* (acronym for *Information Retrieval based Nearest Neighbour Severity Prediction Algorithm*). INSPect combines the K Nearest Neighbour (KNN) algorithm [26] with an information retrieval technique, particularly the extended BM25-based document similarity algorithm [109], to predict bug severity. The authors used three different sets of data to replicate and compare their approach to SEVERIS. The results show that their approach produced better results than SEVERIS. Chaturvedi and Singh [18] also applied 6 different classification algorithms to compare accuracy on the training data used for SEVERIS development. They ranked the document terms based on information gain and tested with various number of terms in intervals of 25. The method accuracy ranges from 29% to 97%. As a related study, Zhang et al. [129] extended the INSPect algorithm by adding topic modelling; a statistical approach used to discover abstract topics that occur in a collection of documents [94]. Their intuition is that bug reports in the same category share the same topic(s). By using topic modelling, the authors found the topic(s) that each bug report belongs to. These topics were introduced to INSPect as additional features to produce better prediction with F-Score ranging from 13.96% to 80.25%.

More recently, Singh et al. [103] proposed two ensemble approaches to fine-grained bug severity prediction. They used voting and bagging to combine predictions from KNN, Naïve Bayes and SVM classifiers. To our knowledge, this is the most recent study in fine-grained bug severity prediction that was conducted in a cross project-context. In an attempt to obtain results that can be generalised, the authors merged bug reports from

various projects to be used as training data set. The resulting model was validated with data set from a different project unseen during training.

From a coarse-grained bug severity perspective, Lamkanfi et al. [53] also used a combination of text mining techniques and Naïve Bayes classifiers [47] to classify bug reports into 'severe' and 'non-severe' classes. Their main contribution is that predictive performance is directly dependent on the training data size. By gradually increasing the training set during experiments, they concluded that an average of 500 reports per bug class is required to obtain a stable and reliable prediction. Indeed, their experiments with larger data sets resulted to stable and improved prediction of the severity levels with precision between 65% - 83% and recall between 62% - 84%. In a successive study, Lamkanfi et al. [54] presented a comparison of several classifiers using the same approach. This time, performance was measured with Area Under Curve (AUC) and results range from 51% to 93%. Gegick et al. [36] studied a binary bug severity prediction using SAS text miner with the aid of Singular Value Decomposition (SVD). Overall, the approach allowed to identify 77% severe bugs that were labeled as non-sever by bug reporters. Yang et al. [125] compared three feature selection algorithms to determine the best features for training a Naïve Bayes classifier. The results showed that the application of feature selection can improve the results. Starting from a baseline AUC of 74% authors were able to reach 77% on a given training data set. Roy and Rossi [100] applied feature selection on bi-grams to train a Naïve Bayes classifier. The results showed that performance is data/project dependent as the addition of bi-grams worsened performance in some cases.

As reported, there are many commonalities across the state-of-the-art, such as the usage of text mining and machine learning techniques in all related works. One of the differences is the level of granularity applied to the bug severity, for example, some use fine grained categories of severity [68, 113, 18, 129, 103], whereas others use just binary categories [53, 54, 36, 125, 100]. It must be noted that the experimental data used in most of these studies originally had seven bug categories namely *Blocker*, *Critical*, *Major*, *Normal*, *Minor*, *Trivial* and *Enhancement*. This categorisation is in accordance to the Eclipse Web Tool Platform convention and is widely adopted by researchers working on bug severity classification, because Bugzilla provides readily annotated bug reports for experiments. All the reviewed studies disregarded the *Enhancement* category as a non-bug and the *Normal* category because they are highly disproportionate in size compared to the other categories. Several researchers [68, 18, 53, 54] attribute the imbalance to error of omission by bug reporters who simply forgot to change the bug category value, which defaults to the *Normal* category. The studies that used binary categories simply aggregated and relabelled the *Blocker*, *Critical* and *Major* categories as "severe bugs", whereas the *Minor* and *Trivial* categories are considered "non-severe bugs". Two limitations we address in our work are:

- To go beyond considering single project bug severity prediction models. The majority of approaches and results in related works are project specific. We intend to build a generic model capable of assigning severity labels to bug reports from a wide range of software projects.
- To extend the granularity of automated bug severity prediction by including all seven categories. To our knowledge, no other bug severity prediction study, including those reported as fine-grained, have used all seven categories. For example one of the five available bug classes for SEVERIS [1] was discarded due to insufficient examples while the rest eliminated two out of seven categories due their high number of examples compared to others.

8.2 OSSMETER Approach

OSSMETER, CROSSMINER's predecessor project, included a bug severity predictor based on text mining techniques and machine learning. More specifically the predictor is a linear SVM classifier trained on bug re-

ports downloaded from the Bugzilla server of the Eclipse Foundation. This data source was chosen primarily because Bugzilla supports fine-grained bug severity categorisation. The severity predictor made use of information drawn from the header and textual body of the bug reports. Particularly, the information from each bug report was transformed into two types of feature vectors used to train the LibSVM classifier, namely:

- word n-grams: binary features encoding the presence or absence of unigrams, bigrams, trigrams and four-grams.
- character n-grams: binary features encoding the presence or absence of character trigrams, four-grams and five-grams.

Two different kernels for SVM were tested, the linear and the Radial Basis Function (RBF) one, with various word n-gram cut-off thresholds from 5 to 200 in intervals of 5. The overall best micro F-score of 97.47% was achieved by the linear kernel for n-gram frequency threshold 5. The RBF kernel achieved its highest micro F-score of 83.13% at a greater n-gram frequency of 65. It is important to note that OSSMETER approach utilised only a fraction of the data currently used in CROSSMINER[60].

8.3 Data set

In order to perform the intended level of bug severity prediction, fine-grained annotated data is needed to train classification models. The Bugzilla server of the Eclipse Foundation hosts bugs for numerous Eclipse related products, which provides more than enough data to train a machine learning classifier. More so, Bugzilla supports a bug metadata feature called severity, which stores information very similar to our extended fine-grained requirements. Therefore, instead of annotating threads manually, we took advantage of Bugzilla bug reports previously downloaded for the OSSMETER project. The bug severity levels and corresponding descriptions are shown in Table 34.

The corpus consists of bug reports from 38 Eclipse products with a total of $93,051$ bugs. As shown in Figure 5, each bug report contains basic elements such as *bug status*, *product*, *component*, *operating system*, *platform*, *version*, *summary* and *comment*. To ensure that our bug severity predictor will be applicable to threads from other channels, such as GitHub, that may not support this wealth of elements, only *summary* and *comment* were used. *Summary* corresponds to the title, where as *comment* corresponds to the body of the message. This data is used for presentation and evaluation purposes throughout this section of the deliverable. The distribution of bugs per severity level is shown in Figure 6. The data set was split into training (80%) and testing (20%), taking the distribution of the severity classes into account. Table 35 shows the number of instances per class used for training and testing purposes.

8.4 Methodology

As a preliminary study on our bug severity predictor, we have applied FastText and Linear SVM, introduced previously in Section 2. Both classifiers are able to handle multi-class tasks. FastText is based on a neural network model and thus it manages multi-class tasks intrinsically. SVMs, in general, tackle multi-class problems through the implementation of a *one vs. rest* strategy. The strategy involves training a single classifier per class, with the instances of that class as *positive* instances and all other instances as *negative*.

[60]The OSSMETER approach used 5 out of 55 bug files downloaded from Bugzilla. A total of 16,230 bugs were used in OSSMETER while 93,051 are currently used in CROSSMINER. Moreover, the 5 files used in OSSMETER represent bug reports on different components of a single product.

```
- <Bugs>
  - <Bug status="CLOSED" severity="normal" resolution="WONTFIX" priority="P3" id="242358">
      <product>Acceleo</product>
      <component>Core</component>
      <operatingSystem>All</operatingSystem>
      <platform>Other</platform>
      <version>0.7.0</version>
      <summary>Javadoc is not in the build zips</summary>
    - <Comments>
      - <Comment url="https://bugs.eclipse.org/bugs/xmlrpc.cgi" product="Acceleo" commentId="1300219" bugId="242358">
          <author>cedric.brun@obeo.fr</author>
          <creator>cedric.brun@obeo.fr</creator>
          <time>Tue Jul 29 10:11:45 BST 2008</time>
          <text>The javadoc creation fails, probably needing some tweaking in the build scripts...</text>
        </Comment>
      - <Comment url="https://bugs.eclipse.org/bugs/xmlrpc.cgi" product="Acceleo" commentId="1765568" bugId="242358">
          <author>laurent.goubet@obeo.fr</author>
          <creator>laurent.goubet@obeo.fr</creator>
          <time>Wed Jun 23 12:49:42 BST 2010</time>
          <text>We've removed the javadoc from the documentation altogether : it was redundant with the SDK.</text>
        </Comment>
      </Comments>
  </Bug>
```


Figure 5: Sample bug report from Bugzilla.

Table 34: Levels of severity in the header of Bugzilla bug reports and their descriptions

Severity levels	Description
Blocker	Blocks further development and/or testing work.
Critical	Crashes, loss of data in a widely used and important component.
Major	Major loss of function in an important area.
Normal	Default/average.
Minor	Minor loss of function, or other problem that does not affect many people or where an easy workaround is present.
Trivial	Cosmetic problem like misspelled words or misaligned text which does not really cause problems.
Enhancement	Request for a new feature or change in functionality for an existing feature.

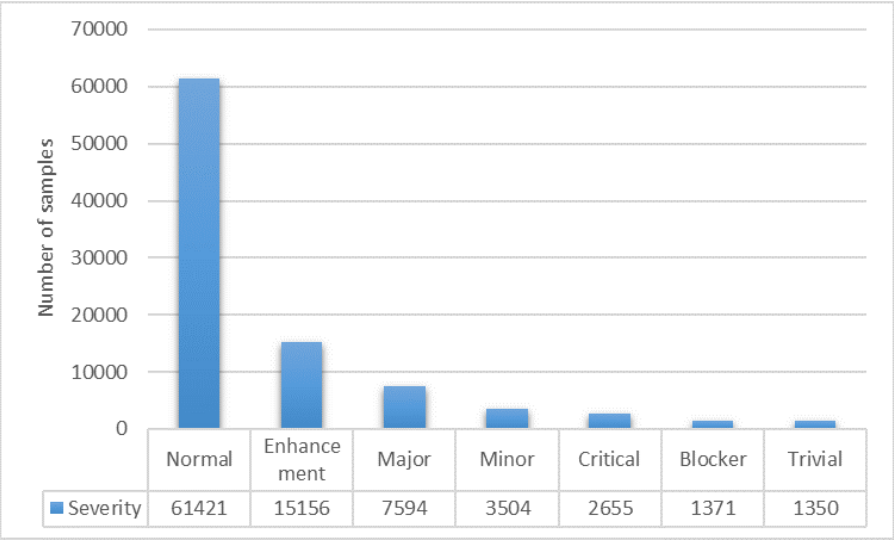

Figure 6: Total number of bugs annotated with each severity level

Table 35: Number of instances in the training and the test set per severity level.

Class labels	Training	Test
Normal	49136	12285
Enhancement	12125	3031
Major	6075	1519
Minor	2803	701
Critical	2124	531
Blocker	1097	274
Trivial	1080	270

Due to the fact that the *normal* class is disproportionally larger than all other classes, as shown in Figure 6, we considerer a different approach that would, in theory, reduce the bias of any classifier, either FastText or linear SVM, to any specific class. In other words, to mitigate bias, we designed a classifier that consists of multiple one vs. rest, i.e. binary, sub-classifiers organised hierarchically, so that instances from the dominant[61] class are trained against all other classes, but are not included in the next level of the tree. The first binary classifier is trained on instances of the *normal* class against instances of the other classes. In the second binary classifier, the *normal* class instances are eliminated from the training data, so that instances from the next dominant class, i.e. *enhancement* are trained against all *major*, *minor*, *critical*, *blocker* and *trivial* instances grouped together. The process is repeated until the a classifier trained on the least dominant classes. This can be observed in Figure 7. This hierarchically structured approach was applied for both FastText and Linear SVM classifiers.

Following the OSSMETER approach, we decided to draw classification features on the text elements of the bug reports, only. A lemmatised and tokenised version of this text was fed to the classifiers. As with other classifiers presented in this deliverable, we tested during optimisation, how different numbers of word n-grams, where $n \in [1, 3]$, word skip-bigrams, where the skip hole size ranges between 2 and 4, and a minimum threshold of occurrence affect the outcome of the classifier. Given the hierarchically structured classification approach adopted in our experiment, each binary classifier has a set of parameters which leads to optimum performance. These parameters were used to train a final model for each binary classifier, on which evaluation was performed.

8.5 Settings

The following subsections present our experimental settings. To obtain the best possible model, we optimized training using *Bayesian Optimisation*. We set as its objective function either the median or the average, whichever achieves the lowest Macro F-score on a 10-fold cross validation basis. This objective function reflects our choice to consider all classes as equally important, despite their disproportionate distribution in the corpus. We also discuss the evaluation metrics used.

8.5.1 Experimental

We applied the hierarchically structured one vs. rest approach shown in Figure 7), to both FastText and Linear SVM classifiers in order to address class imbalance. Each of the binary models in the hierarchy was trained with a specific set of parameters for optimum performance.

[61] Dominant refers to size of a class in relation to other classes. For example, our data contains more instances of the *normal* class than of any other class.

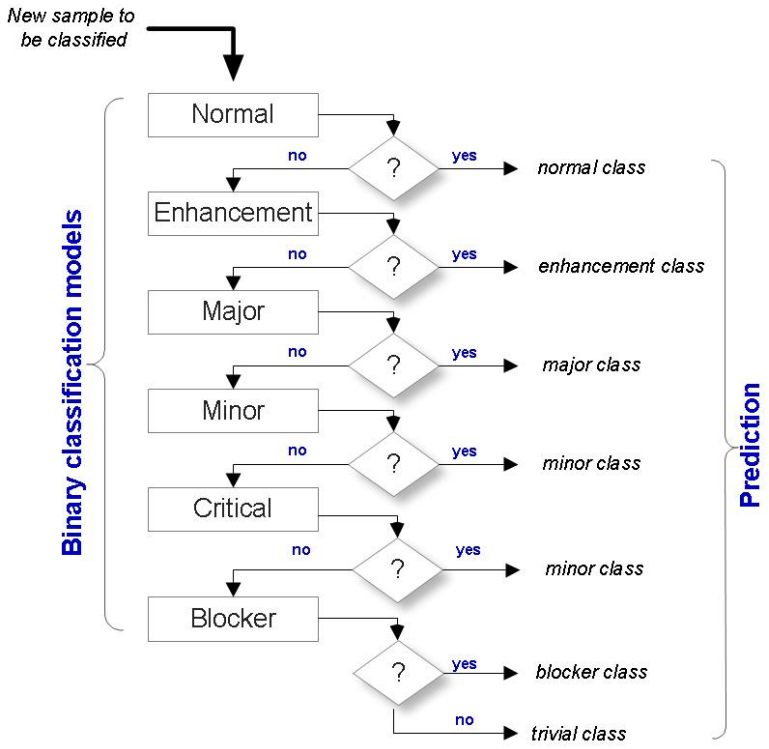

Figure 7: A hierarchically structured one vs. rest multi-label classification approach

Table 36 presents the best parameters found by Bayesian Optimisation for each of the binary SVM models. The column "classifiers" indicates the name of the predominant class vs. the other classes, in the hierarchical-tree. Notice that the table illustrates 6 classifiers only. This is because instances of the least predominant class, i.e. *trivial*, was trained using the *blocker* class parameters.

Similarly, Table 37 presents the best parameters found by Bayesian Optimisation for each of the binary FastText models.

8.5.2 Evaluative

Precision, *recall*, *micro F-Score*, and *macro F-Score* were used to evaluate all baselines and methods.

Table 36: Parameters used to train Linear SVM models

Classifiers	n-grams	skip-bigrams	min frequency	c	Micro F-Score (%)
Normal vs. Other	1	0	5	32	65.01
Enhancement vs. Other	2	1	5	33554432	84.91
Major vs. Other	2	0	2	7	63.64
Minor vs. Other	3	2	1	22	69.44
Critical vs. Other	3	2	50	7	69.31
Blocker vs. Other	3	2	20	128	86.49

Table 37: Parameters used to train FastText models

Classifiers	lr	dim	n-grams	min	epoch	Micro F-Score(%)
Normal vs. Other	0.20	100	3	10	5	66.87
Enhancement vs. Other	0.11	200	3	30	20	88.03
Major vs. Other	0.27	200	3	40	15	64.29
Minor vs. Other	0.19	135	2	30	25	70.72
Critical vs. Other	0.26	25	3	2	40	69.22
Blocker vs. Other	0.19	50	2	4	35	87.29

Table 38: Contingency table of the FastText models

	Normal	Enhancement	Major	Minor	Critical	Blocker	Trivial
Normal	10398	1003	417	161	132	71	103
Enhancement	847	2082	47	26	16	5	8
Major	1239	61	157	12	33	12	5
Minor	517	63	34	56	5	10	16
Critical	404	13	51	3	46	14	0
Blocker	211	8	24	1	6	24	0
Trivial	162	27	9	24	4	3	41

8.6 Results

Table 38 presents the contingency table of predictions using FastText. For example, the first model in the hierarchical structure, *Normal vs. Other* classified correctly 10,398 out of 12,285 *normal* bugs, but misclassified 3,380 instances as *normal*. The misclassified instances represent 18.16% of the test set. Likewise, the next model, *Enhancement vs. Other* misclassified a further 12.11% of the test set. Prediction performance degrades dramatically after the first two dominant classes. This dynamic is bound to have a knock on effect at the subsequent nodes of the hierarchical structure. Indeed, this effect can be seen clearly in Table 39, which shows massive decrease in performance when classifying the other classes. This is because, the vast majority of test instances has been lost at preceding nodes.

A similar pattern was observed in the results of experimentation with Linear SVMs, as shown in the contingency Table 40. Prediction performance can be seen to degrade dramatically after the dominant class (*normal*).

Table 39: Results of experimentation with FastText

Severity labels	Precision (%)	Recall (%)	F-Score (%)
Normal	75.47	84.64	79.79
Enhancement	63.92	68.69	66.22
Major	21.24	10.34	13.91
Minor	19.79	7.99	11.38
Critical	19.01	8.66	11.90
Blocker	17.27	8.76	11.62
Trivial	23.70	15.19	18.51
Macro F-Score: 30.48%		**Micro F-Score: 68.80%**	

Table 40: Contingency table of the Linear SVM models

	Normal	**Enhancement**	**Major**	**Minor**	**Critical**	**Blocker**	**Trivial**
Normal	11881	352	9	21	10	3	0
Enhancement	1850	1131	13	18	9	1	0
Major	1498	14	2	3	1	1	0
Minor	675	14	0	10	0	0	23
Critical	525	4	1	0	1	0	0
Blocker	274	0	0	0	0	0	0
Trivial	252	7	1	7	0	0	0

Table 41: Results of experimentation with SVMs

Severity labels	**Precision (%)**	**Recall (%)**	**F-Score (%)**
Normal	70.07	96.78	81.29
Enhancement	74.31	37.43	49.78
Major	7.70	0.13	0.26
Minor	16.95	1.39	2.56
Critical	4.76	0.19	0.36
Blocker	0.00	0.00	0.00
Trivial	0.00	0.00	0.00
Macro F-Score: 19.18%		**Micro F-Score: 69.99%**	

At this node, 27.26% of the test set was misclassified, and a further 6.29% was misclassified at the *enhancement* node. This affected the other nodes and particularly the *blocker* and *trivial* nodes reached by only 28 out of 18,611 instances to be classified. This decrease can be seen clearly in Table 41.

Both methods achieved moderate performance at the top of the hierarchical structure, which kept decreasing continuously down the structure. The Linear SVM achieved a slightly better Micro F-Score than FastText, 69.99% vs. 68.80%, respectively. We believe that both results could be higher if the prediction accuracy at the top of the hierarchical structure was better. In other words, the results in Table 39 do not represent the individual predictive power of the classifiers at each node. According to the optimisation results for FastText, shown in Table 37, the *Blocker vs. Other* classifier achieved the highest Micro-F-Score, 86%, followed by the *Enhancement vs. Other* classifier, 84%. To compute the actual predictive power of each individual classifier in the hierarchical structure, we calculated the *Precision*, *Recall* and *F-Score* at each node of the hierarchy by eliminating test instances already seen at the preceding node. For example, to calculate the true performance of the *Enhancement vs. Other* classifier, we eliminated instances relating to the *Normal vs. Other* classifier in the contingency Table 38. We present the modified results in Table 42.

Table 42 shows that both *Blocker vs. Other* and *Enhancement vs. Other* still maintained high performance relative to the other classifiers. In particular, *Enhancement vs. Other* which occurs second in the hierarchical structure performed better than the preceding *Normal vs. Other* classifier, which suggests that the poor performance of the approach is caused by the misclassification rate at the apex.

Similarly, we present in Table 43, modified results for experiments with Linear SVM in order to reflect the predictive power of each classifier in the hierarchical structure without the effect of the preceding ones. We noticed a similar pattern with the *Enhancement vs. Other* classifier which came second in both optimisation results, as shown in Table 36, and the modified results in Table 43. However, the *Blocker vs. Other* classifier

Table 42: Modified results of experimentation with FastText

Classifiers	Precision (%)	Recall (%)	F-Score (%)
Normal vs. Other	75.47	84.64	79.79
Enhancement vs. Other	92.37	95.33	93.83
Major vs. Other	57.09	71.69	63.56
Minor vs. Other	66.67	64.37	65.50
Critical vs. Other	82.14	76.67	79.31
Blocker vs. Other	88.89	100.00	94.12

Table 43: Modified results of experimentation with SVMs

Classifiers	Precision (%)	Recall (%)	F-Score (%)
Normal vs. Other	70.07	96.78	81.29
Enhancement vs. Other	96.67	96.50	96.58
Major vs. Other	50.00	28.57	36.36
Minor vs. Other	58.82	30.30	40.00
Critical vs. Other	100.00	100.00	100.00
Blocker vs. Other	0.00	0.00	0.00

which produced the highest performance during optimisation did not perform well in the table of modified results. Again, this is attributable to the poor performance of the preceding classifiers in the hierarchical structure, as only 23 out of 18,611 test instances was presented to the classifier for prediction.

8.7 Discussion

The results of both the FastText and SVM experiments were heavily biased in favour of the top binary classifier in the hierarchical structure approach that was implemented. The *Normal vs. Other* classifier is the first to encounter the test set, which is predominantly composed of instances from the *normal* class, i.e. $12,285$ out of $18,611$ instances. However, the classifier misclassified a high percentage of the other classes, 18.16% when using FastText and 27.26% when using Linear SVM. A possible explanation may be the fact that selecting a severity level is not compulsory, when filing a bug report on Bugzilla. Users that are not bothered to choose a severity level, unintentionally select the default value, which is *normal*. As a result, not all instances marked as *normal* are genuine representatives of the *normal* class, introducing noise to the classifier that is being trained on them.

Another possible reason is the class distribution of the data and importantly the level of granularity applied in our experiments. To the best of our knowledge, none of the related studies within the literature utilised the *normal* and *enhancement* classes of Bugzilla data for experimentation. The majority disregarded the *enhancement* class because its instances mainly represent suggestions for additional features to a given software rather than bugs.

A further possible reason may lie within the separability of the severity classes available in the experimental data. Instances from the *normal* and *enhancement* classes may share similar characteristics which may pose difficulties for a classifier. However, classes such as *trivial* and *blocker* are easier to differentiate as they use different vocabulary theoretically.

The cross-product approach taken in our experiment may also influence the misclassification. For example, we combined bug reports from different products whereas all the related studies presented in the state-of-the-art, such as Lamkanfi et al. [53], Chaturvedi and Singh [18], and Menzies [68] presented their results based on individual products. Their project-specific approach is unlikely to have an adverse effect on results because the presentation and annotation of the bug reports are likely to be consistent. In our case, we combined bug reports from 38 different projects which may not follow similar bug presentation and/or annotation.

8.8 Conclusions

Bug severity prediction is a natural language processing task that aims at categorising textual content of bug reports based on their impact to a given software project. Although several research works have been reported in this area, none to our knowledge has gone to the level of granularity investigated as part of Task 3.3. In addition, the majority of the approaches and results in the literature are project specific. Our work in CROSSMINER is intended to automatically extend severity annotation to bug tracking systems that do not support it inherently.

In this section of Deliverable 3.3, we presented the design and development of a supervised bug severity classifier for CROSSMINER. Being a multi-class task to be performed on highly imbalanced experimental data sets, we explored a one vs. rest strategy based on hierarchical structure of binary classifiers arranged to address the data imbalance. We investigated this method using two alternative classification models, *Linear SVM* and *FastText*, and compared the results.

FastText performed better than the Linear SVM, with a *Macro F-Score* of 30.48% compared to 19.18% achieved by Linear SVM. However, the Linear SVM produced a slightly higher *Micro F-score*, 69.99%, in comparison to 68.80% produced by FastText. In the future, we plan to investigate extend our investigation in regard to the causes of misclassification as well as improve our classifiers and in particular the classification of less dominant classes. Further experiments are under way to explore other options that might improve our results. For example, in the current experiments we took a naïve approach by using the textual content of each bug in its original form as features. Our on-going work includes the extraction of additional features from the data that may be useful for training the classifiers, such as the presence of code. In succession, we are applying further pre-processing to the data in order to remove elements,such as stack traces, that may compromise classification performance.

9 Risks and Limitations

In this section, we discuss the possible risks and limitations associated with the tools presented in this deliverable.

Limitations:

- The use of slang, neologisms and emoticons in messages exchanged by developers can reduce the performance of the emotion classifier and the sentiment analyser.

Risks:

- Discontinuation of support for libraries used for processing text, such as NLP4J, or the neural network, DeepLearning4j. In this case, new supported tools will have to be selected to replace the unsupported ones.
- Change of message formatting from certain sources, such as GitHub, can risk the correct pre-processing of the text and, in consequence, affect the performance of the tools. To alleviate this, the pre-processing step will have to be updated.

10 Conclusions

In this deliverable we presented five NLP tools that have been designed for the analysis of messages exchanged between software developers and users of open source software. The corresponding classifiers encompass different types of analysis that can be useful for understanding and managing a project. In addition, we have presented *VastText*, a novel neural network model, that has been designed and developed in house for solving single-label and multi-label classification problems.

The first text processing tool that we developed was a sentiment analyser, i.e. a tool that can determine the polarity of a message as positive, negative or neutral. The classifier uses VastText and has been trained on a corpus of JIRA Issue Tracker messages. The second tool is an emotion detector, i.e. a tool able to detect in text emotions such as *joy*, *love*, *anger* and *fear*. The classifier has been trained using VastText. The third tool identifies if a message is a request or a reply to another message. The corresponding classifier has been trained using a linear Support Vector Machine. The fourth tool we developed is a VastText classifier that determines the type(s) of content mentioned in a message. Some example content types are *clarification*,*suggestion of solution* or *action notification*. The fifth tool is the severity classifier. It aims to determine the degree of severity of a bug discussed in a thread of messages. The tool categorises the thread of messages into one of seven possible severity levels: *normal*, *minor*, *enhancement*, *major*, *critical*, *blocker* and *trivial*. We explored two different classifiers, one based on dense vector representation (FastText) and the other based on sparse vector representation (Linear SVM). We took a naive approach in the design of our experiments by we took a naive approach in the current experiments by using the textual content of each bug in its original form as features. The results were interesting as FastText performed better than the Linear SVM with Macro F-Score of 30.48% compared to 19.18% produced by Linear SVM. However, the Linear SVM produced a slightly higher Micro F-score of 69.99% compared to 68.80% produced by FastText. Further work is currently being conducted in order to improve these results.

Through the investigation, design and development of the above text processing tools, we investigated not only various machine learning methods, but also algorithms that use different text representation models, either sparse or dense. Regarding the use of sparse or dense representations, a universal conclusion cannot be made. The choice of the best text representation model possibly depends on factors such as the nature of the classification task at hand and the quantity of data available for training. It should be noted that we also focused on the optimisation of all the machine learning methods, which can improve the performance of the algorithms regardless of the text representation model used.

In the future, we plan to train a new emotion classifier using a corpus based on StackOverflow, that in this work was used uniquely for testing. Moreover, we expect that in the near future, we will be able to improve VastText for the request vs. reply classifier, by addressing the bug that currently exists in the underlying DeepLearning4j. With regard to the content classifier, we would like to perform further experiments with a richer feature space and perhaps explore more advanced linguistic techniques such as discourse analysis to improve the overall performance of the multi-label content classifier.

We note that our experiments on the severity classifier are still at preliminary stage and, we are working to improve their performance, e.g. by exploring the use of additional input features as a way to improve performance of the tool. Furthermore, we are applying further data pre-processing to the experimental data in order to remove elements that may reduce classification performance such as stack traces. We plan, as well, to validate our approach with external studies found in the literature by replication and comparison.

A Content Classifier: Refactored Labels and Definitions

This appendix presents the definitions associated with the refactored label hierarchy for the classification of content within posts, messages and comments from open source software repositories in CROSSMINER.

1. **Clarification:** Messages assigned this category provide additional information, in response to a request to clarify details of a previous post. For example the clarification of a request or clarification of a solution.

2. **Suggestion of a solution:** A component in message, submitted by a user other than the initial requestor, that describes possible ways in which a previously submitted request or problem report can be solved or resolved by the initial user. This also applies to parts of the message that correspond to background or explanatory information that is provided to put the suggested solution into content. Moreover, this label should be assigned regardless of the outcome of the solution.

3. **Resolution of a request / problem:** A component within a message that aims to provide a resolution to a previous request or problem report, which was submitted earlier in the thread. Such components may include additional information that:

    - Directly supports or elaborates upon the resolution given
    - Redirects the user to information which can help solve their problem
    - Identifies the user's problem as a known bug
    - The user who posted the original initial request / problem stating they have found their own solution
    - Details of a software patch, that address the initial request / problem
    - Details of a bug fix, that addresses the initial request / problem
    - A response from a developer that indicates that they cannot replicate the initial request / problem, resulting in them not being able to take any further action
    - A response from a developer that indicates that they wont fix the initial request / problem, resulting in them not being able to take any further action
    - A response from a developer that indicates that they are not going to fix the initial request / problem, by explicitly stating that the software have made this problem obsolete.
    - A response from a developer to an initial request / problem submitted by an other user, suggesting that the user should file a bug in some bug tracking system.

4. **Request or support:** A component within a message that requests help / further help with software, reports of a software problem or requests/suggests a software update.

5. **Unsuccessful solution notification:** A component within a message in which the requestor/problem reporter provides a notification that a particular suggested solution did not work or solve their problem.

6. **Action notification:**A component within a message that provides an indication or description of a planned type of action, ongoing action or action that is already being undertaken as a step towards resolving the problem. This includes:

    - Investigations

- Bug re-openings
- Bug priority changes
- Plan of future work
- Closure of th thread
- Redirection of support to another group or party
- As well as any other actions or description of an actions that is expressed by a user or developer, not covered by the previous actions, above.

7. **Resolution acceptance:**A component within a message that indicates that the initial requester/problem reporter accepts (one of) the resolutions that has been proposed by other developers/users. This includes:

   - Confirmation that the user has filed a new bug report
   - A thank you message
   - A users acknowledgement of a working solution

8. **Other information provision:**A component within a message that contains information that does not fit into one of the classes described previously. Information may or may not be directly relevant to the ongoing discussion. Examples include:

   - A message in which a developer responds to an initial request/problem report submitted by a user, by acknowledging the problem as an previously unencountered bug.
   - A statement in which a developer responds to an initial request/problem report submitted by a user, by acknowledging the problem as a duplicated bug which may or may not have been solved.
   - A message that provides information that supports or elaborates upon a previously introduced request/problem report or its solution, or provides relevant contextual information, but which does not fit into any other relevant category.
   - A message that can be considered as SPAM within the context of a bug tracking system or news-group whose aim is to solve software problems. Such SPAM messages include announcements of information targeted at the community of users, such as events, conferences, job opening, general software release messages, etc. These messages may appear in their own threads.
   - Verbal abuse.
   - Test messages that are used to assess if the communication channel is working.
   - Any other type of messages that is unrelated to the discussion (in that it does not provide any useful or contextual information) such as a incomplete or truncated message.

B Publications & Working papers

Below is a summary of publications and working papers. Copies of the papers are attached at the end of the present document.

1. **Title:** *Code Snippets on Stack Overflow: Are they always labelled correctly?*
   Status: The paper has been submitted and rejected at the conferences MSR 2018 and ASE 2018. Subsequently, the manuscript was improved based on the review comments.
   Abstract: The increasing size of Stack Overflow's data has attracted researchers to explore and use it. For example, recent methods have been proposed to detect duplicated questions in Stack Overflow automatically or to match natural language search queries with code portions inside an IDE. The success of these methods depends on structures integrated in Stack Overflow, such as tags and annotations, and in turn on how accurately they are used. In this paper, we focus on the code tag and analyse how these tags are used, misused or omitted by users when posting on Stack Overflow. Tagging code snippets correctly is crucial for the performance of more sophisticated tools that can help developers. For this analysis, we have developed a code detector using a classifier based on FastText and word embeddings. The detector can identify whether a block of text, of any length, is either English or code. It has been evaluated on a large Stack Overflow data set of 3.6 million posts and achieved promising results. Our analysis of the results has shown that the code tag is sometimes omitted and, in occasions, it is used to mark a variety of related but distinct concepts like: stack traces, errors and input-output text. These omissions and misuses can be corrected using the code detector.

2. **Title:** *Detection of spam-posting accounts on Twitter*
   Status: Published in Neurocomputing 315 (2018) 496-511
   Abstract: Online Social Media platforms, such as Facebook and Twitter, enable all users,independently of their characteristics, to freely generate and consume huge amounts of data. While this data is being exploited by individuals and organisations to gain competitive advantage, a substantial amount of data is being generated by spam or fake users. One in every 200 social media messages and one in every 21 tweets is estimated to be spam. The rapid growth in the volume of global spam is expected to compromise research works that use social media data, there by questioning data credibility. Motivated by the need to identify and filter out spam contents in social media data, this study presents a novel approach for distinguishing spam vs. non-spam social media posts and offers more insight into the behaviour of spam users on Twitter. The approach proposes an optimised set of features independent of historical tweets,which are only available for a short time on Twitter. We take into account features related to the users of Twitter, their accounts and their pairwise engagement with each other. We experimentally demonstrate the efficacy and robustness of our approach and compare it to a typical feature set for spam detection in the literature, achieving a significant improvement on performance. In contrast to prior research findings, we observe that an average automated spam account posted at least 12 tweets per day at well defined periods. Our method is suitable for real-time deployment in a social media data collection pipeline as an initial preprocessing strategy to improve the validity of research data.

3. **Title:** *Lexical analysis of automated accounts on Twitter*
   Status: Published In Proceedings of 17th International Conference on WWW/Internet (pp. 75-82). IADIS
   Abstract: In recent years, social bots have been using increasingly more sophisticated, challenging detection strategies. While many approaches and features have been proposed, social bots evade detection and interact much like humans making it difficult to distinguish real human accounts from bot accounts. For detection systems, various features under the broader categories of account profile, tweet content,

network and temporal pattern have been utilised. The use of tweet content features is limited to analysis of basic terms such as URLs, hashtags, name entities and sentiment. Given a set of tweet contents with no obvious pattern can we distinguish contents produced by social bots from that of humans? We aim to answer this question by analysing the lexical richness of tweets produced by the respective accounts using large collections of different datasets. Our results show a clear margin between the two classes in lexical diversity, lexical sophistication and distribution of emoticons. We found that the proposed lexical features significantly improve the performance of classifying both account types. These features are useful for training a standard machine learning classifier for effective detection of social bot accounts. A new dataset is made freely available for further exploration.

4. **Title:** *The Effect of Engagement Intensity and Lexical Richness In Identifying Bot Accounts on Twitter*
Status: Published in the IADIS International Journal on WWW/Internet – Volume 16, Issue 2 (online journal)
Abstract: The rise in the number of automated or bot accounts on Twitter engaging in manipulative behaviour is of great concern to studies using social media as a primary data source. Many strategies have been proposed and implemented, however, the sophistication and rate of deployment of bot accounts is increasing rapidly. This impedes and limits the capabilities of detecting bot strategies. Various features broadly related to account profiles, tweet content, network and temporal patterns have been utilised in detection systems. Tweet content has been proven instrumental in this process, but limited to the terms and entities occurring. Given a set of tweets with no obvious pattern, can we distinguish contents produced by social bots from those of humans? What constitutes engagement on Twitter and how can we measure the intensity of engagement? Can we distinguish between bot and human accounts based on engagement intensity? These are important questions whose answer will improve how detection systems operate to combat malicious activities by effectively distinguishing between human and social bot accounts on Twitter. This study attempts to answer these questions by analysing the engagement intensity and lexical richness of tweets produced by human and social bot accounts using large, diverse datasets. Our results show a clear margin between the two classes in terms of engagement intensity and lexical richness. We found that it is extremely rare for a social bot to engage meaningfully with other users and that lexical features significantly improve the performance of classifying both account types. These are important dimensions to explore toward improving the effectiveness of detection systems in combating the menace of social bot accounts on Twitter.

5. **Title:** *Code Detector*
Status: A draft we have been working on to be submitted at a journal or conference.
Abstract: With the Internet expansion, new means and ways to communicate arose making it easier to be in contact with different kind of people but also to exchange knowledge with them. Thanks to this, websites based on Question and Answering (Q&A) or forums became popular among Internet users. Some of these websites became specialised in specific topics, e.g. in software engineering like Stack Overflow or Code Project. The communication between software developers and experts inherits a mixture of natural language text with portions of code, from programming or markup languages. Although many of these specialised websites or services propose the use of specific labels to format differently code from natural language, such as in Stack Overflow, in many cases these labels can be misused or forgotten. Therefore, a code detector is necessary. A code detector can determine if a document, or a part of it, is code in some programming or markup language, or it is text in natural language. In this paper, we explore the use of word embeddings along to two different types of classifiers: a Support Vector Machine and a neural-network-based classifier.

References

[1] N. Agarwal, R. Gupta, S. K. Singh, and V. Saxena. Metadata based multi-labelling of YouTube videos. In *2017 7th International Conference on Cloud Computing, Data Science Engineering - Confluence*, pages 586–590, 2017.

[2] Md Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K Roy, and Kevin A Schneider. Classifying stack overflow posts on API issues. In Rocco Oliveto, Massimiliano Di Penta, and David C Shepherd, editors, *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 244–254, Campobasso, Italy, mar 2018.

[3] Toufique Ahmed, Amiangshu Bosu, Anindya Iqbal, and Shahram Rahimi. SentiCR: A customized sentiment analysis tool for code review interactions. In Grigore Rosu, Massimiliano Di Penta, and Tien N Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*, pages 106–111, Urbana-Champaign, IL, USA, oct 2017.

[4] César Alfaro, Javier Cano-Montero, Javier Gómez, Javier M. Moguerza, and Felipe Ortega. A multi-stage method for content classification and opinion mining on weblog comments. *Annals of Operations Research*, 236(1):197–213, 2016.

[5] Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. SentiWordNet 3.0: An Enhanced Lexical Resource for Sentiment Analysis and Opinion Mining. In Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of 7th Language Resources and Evaluation Conference (LREC'10)*, pages 2200–2204, La Valleta, Malta, 2010.

[6] Timothy Baldwin, David Martinez, and Richard B Penman. Automatic Thread Classification for Linux User Forum Information Access. In Amanda Spink, Andrew Turpin, and Mingfang Wu, editors, *Proceedings of The Twelfth Australasian Document Computing Symposium*, pages 72–79, Melbourne, Australia, 2007. RMIT University.

[7] Christos Baziotis, Athanasiou Nikolaos, Alexandra Chronopoulou, Athanasia Kolovou, Georgios Paraskevopoulos, Nikolaos Ellinas, Shrikanth Narayanan, and Alexandros Potamianos. NTUA-SLP at SemEval-2018 Task 1: Predicting Affective Content in Tweets with Deep Attentive RNNs and Transfer Learning. In Marianna Apidianaki, Saif M Mohammad, Jonathan May, Ekaterina Shutova, Steven Bethard, and Marine Carpuat, editors, *Proceedings of The 12th International Workshop on Semantic Evaluation*, pages 245–255, New Orleans, Louisiana, 2018. Association for Computational Linguistics.

[8] Christos Baziotis, Nikos Pelekis, and Christos Doulkeridis. DataStories at SemEval-2017 Task 4: Deep LSTM with Attention for Message-level and Topic-based Sentiment Analysis. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 747–754, Vancouver, Canada, 2017. Association for Computational Linguistics.

[9] Sumit Bhatia, Prakhar Biyani, and Prasenjit Mitra. Classifying User Messages For Managing Web Forum Data. *WebDB*, (WebDB):13–18, 2012.

[10] Hendrik Blockeel, Luc De Raedt, and Jan Ramon. Top-Down Induction of Clustering Trees. In Jude W Shavlik, editor, *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, pages 55–63, Madison, WI, USA, 1998. Morgan Kaufmann Publishers Inc.

[11] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching Word Vectors with Subword Information. *Transactions of the Association of Computational Linguistics*, 5:135–146, 2017.

[12] Margaret M Bradley and Peter J Lan. Affective norms for English words (ANEW): Instruction manual and affective ratings. Technical Report Technical Report C-1, The Center for Research in Psychophysiology, University of Florida, 1999.

[13] Fabio Calefato, Filippo Lanubile, Federico Maiorano, and Nicole Novielli. Sentiment Polarity Detection for Software Development. *Empirical Software Engineering*, 23(3):1352–1382, jun 2018.

[14] Fabio Calefato, Filippo Lanubile, and Nicole Novielli. EmoTxt: A toolkit for emotion recognition from text. In *2017 Seventh International Conference on Affective Computing and Intelligent Interaction Workshops and Demos (ACIIW)*, pages 79–80, San Antonio, TX, USA, oct 2017.

[15] Erik Cambria, Andrew Livingstone, and Amir Hussain. The Hourglass of Emotions. In Anna Esposito, Antonietta M Esposito, Alessandro Vinciarelli, Rüdiger Hoffmann, and Vincent C Müller, editors, *Cognitive Behavioural Systems*, pages 144–157, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[16] Erik Cambria, Soujanya Poria, Devamanyu Hazarika, and Kenneth Kwok. SenticNet 5: Discovering Conceptual Primitives for Sentiment Analysis by Means of Context Embeddings. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, pages 1795–1802, New Orleans, LA, USA, 2018. Association for the Advancement of Artificial Intelligence.

[17] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1—-27:27, 2011.

[18] K K Chaturvedi and V B Singh. Determining Bug severity using machine learning techniques. In *2012 CSI 6th International Conference on Software Engineering, CONSEG 2012*, pages 1–6. IEEE, 2012.

[19] François-Régis Chaumartin. UPAR7: A Knowledge-based System for Headline Sentiment Tagging. In Eneko Agirre, Lluis Marquez, and Richard Wicentowski, editors, *Proceedings of the 4th International Workshop on Semantic Evaluations*, SemEval '07, pages 422–425, Prague, Czech Republic, 2007. Association for Computational Linguistics.

[20] Jinho D Choi. Dynamic Feature Induction: The Last Gist to the State-of-the-Art. In Kevin Knight, Ani Nenkova, and Owen Rambow, editors, *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 271–281, San Diego, California, 2016. Association for Computational Linguistics.

[21] Jinho D Choi and Andrew McCallum. Transition-based Dependency Parsing with Selectional Branching. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1052–1062, Sofia, Bulgaria, 2013. Association for Computational Linguistics.

[22] François Chollet. *Deep Learning with Python*. Manning, 2017.

[23] Mathieu Cliche. BB_twtr at SemEval-2017 Task 4: Twitter Sentiment Analysis with CNNs and LSTMs. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 573–580, Vancouver, Canada, 2017. Association for Computational Linguistics.

[24] William W Cohen. Fast Effective Rule Induction. In *Machine Learning Proceedings 1995*, pages 115–123. Elsevier, 1995.

[25] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[26] T Cover and P Hart. Nearest neighbor pattern classification. *IEEE Transactions of Information Theory*, 13(1):21–27, 1967.

[27] Cristian Danescu-Niculescu-Mizil, Moritz Sudhof, Dan Jurafsky, Jure Leskovec, and Christopher Potts. A computational approach to politeness with application to social factors. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 250–259, Sofia, Bulgaria, 2013. Association for Computational Linguistics.

[28] Taner Danisman and Adil Alpkocak. Feeler: Emotion classification of text using vector space model. In Chris Mellish, editor, *AISB 2008 Convention Communication, Interaction and Social Intelligence*, volume 2, pages 53–59, Aberdeen, Scotland, UK, 2008.

[29] Alex Marino de Almeida, Sylvio Barbon Junior, and Emerson Cabrera Paraiso. Multi-class Emotions Classification by Sentic Levels as Features in Sentiment Analysis. In *5th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 486–491, oct 2016.

[30] Alex Marino de Almeida, Ricardo Cerri, Emerson Cabrera Paraiso, Rafael Gomes Mantovani, and Sylvio Barbon Junior. Applying multi-label techniques in emotion identification of short texts. *Neurocomputing*, 320:35–46, 2018.

[31] Krzysztof Dembczyński, Willem Waegeman, Weiwei Cheng, and Eyke Hüllermeier. On label dependence and loss minimization in multi-label classification. *Machine Learning*, 88(1):5–45, jul 2012.

[32] Bart Desmet and Véronique Hoste. Emotion detection in suicide notes. *Expert Systems with Applications*, 40(16):6351–6358, 2013.

[33] Vasiliki Efstathiou, Christos Chatzilenas, and Diomidis Spinellis. Word Embeddings for the Software Engineering Domain. In Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 38–41, Gothenburg, Sweden, 2018. ACM.

[34] Christine Fellbaum, editor. *WordNet: An electronic lexical database*. MIT Press, 1998.

[35] Washington Garcia and Theophilus Benson. A First Look at Bugs in OpenStack. In Murat Yuksel and Timothy Wood, editors, *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, CAN '16, pages 67–72, Irvine, CA, USA, 2016. ACM.

[36] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *Proceedings - International Conference on Software Engineering*, pages 11–20. IEEE, 2010.

[37] Alec Go, Richa Bhayani, and Lei Huang. Twitter Sentiment Classification using Distant Supervision. Technical Report CS224N, Stanford Natural Language Processing Group, Stanford University, 2009.

[38] Narendra Gupta, Mazin Gilbert, and Giuseppe Di Fabbrizio. Emotion Detection in Email Customer Care. *Computational Intelligence*, 29(3):489–505, 2013.

[39] Emitza Guzman, David Azócar, and Yang Li. Sentiment Analysis of Commit Comments in GitHub: An Empirical Study. In Premkumar Devanbu, Sung Kim, and Martin Pinzger, editors, *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 352–355, New York, NY, USA, 2014. ACM.

[40] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations Newsletter*, 11(1):10–18, nov 2009.

[41] William L Hamilton, Kevin Clark, Jure Leskovec, and Dan Jurafsky. Inducing Domain-Specific Sentiment Lexicons from Unlabeled Corpora. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 595–605, Austin, Texas, USA, 2016. Association for Computational Linguistics.

[42] Dung T Ho and Tru H Cao. A High-Order Hidden Markov Model for Emotion Detection from Textual Data. In Deborah Richards and Byeong Ho Kang, editors, *Knowledge Management and Acquisition for Intelligent Systems*, pages 94–105, Kuching, Malaysia, 2012. Springer Berlin Heidelberg.

[43] Dirk Hovy. Demographic Factors Improve Classification Performance. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 752–762, Beijing, China, 2015. Association for Computational Linguistics.

[44] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. Technical report, Department of Computer Science, National Taiwan University, 2003.

[45] Md Rakibul Islam and Minhaz F Zibran. Leveraging Automated Sentiment Analysis in Software Engineering. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 203–214, Buenos Aires, Argentina, may 2017.

[46] Peter Jackson and Isabelle Moulinier. *Natural Language Processing for Online Applications: Text Retrieval, Extraction & Categorization*, volume 5 of *Natural Language Processing*. John Benjamins, Amsterdam, 1 edition, 2002.

[47] George H John and Pat Langley. Estimating Continuous Distributions in Bayesian Classifiers. In *Uncertainty in Artificial Intelligence*, pages 338–345, 1995.

[48] Robbert Jongeling, Proshanta Sarkar, Subhajit Datta, and Alexander Serebrenik. On negative results when using sentiment analysis tools for software engineering research. *Empirical Software Engineering*, 22(5):2543–2584, oct 2017.

[49] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of Tricks for Efficient Text Classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, volume 2, pages 427–431, Valencia, Spain, 2017.

[50] Sunghwan Mac Kim, Alessandro Valitutti, and Rafael A Calvo. Evaluation of Unsupervised Emotion Models to Textual Affect Recognition. In Diana Inkpen and Carlo Strapparava, editors, *Proceedings of the NAACL HLT 2010 Workshop on Computational Approaches to Analysis and Generation of Emotion in Text*, CAAGET '10, pages 62–70, Los Angeles, California, 2010. Association for Computational Linguistics.

[51] Ioannis Korkontzelos and Sophia Ananiadou. Locating Requests among Open Source Software Communication Messages. In Nicoletta Calzolari, Hrafn Loftsson, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 1347–1354, Reykjavik, Iceland, 2014. European Language Resources Association (ELRA).

[52] Henry Kučera and Winthrop Nelson Francis. *Computational analysis of present-day American English*. Providence Brown University Press, 1967.

[53] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. Predicting the severity of a reported bug. In *Proceedings - International Conference on Software Engineering*, pages 1–10. IEEE, 2010.

[54] Ahmed Lamkanfi, Serge Demeyer, Quinten David Soetens, and Tim Verdonckz. Comparing mining algorithms for predicting the severity of a reported bug. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 249–258. IEEE, 2011.

[55] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, Michele Lanza, and Rocco Oliveto. Sentiment Analysis for Software Engineering: How Far Can We Go? In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 94–104, New York, NY, USA, 2018. ACM.

[56] Bing Liu, Minqing Hu, and Junsheng Cheng. Opinion observer: analyzing and comparing opinions on the web. In Allan Ellis and Allan Ellis, editors, *Proceedings of the 14th international conference on World Wide Web*, pages 342–351, Chiba, Japan, 2005. ACM.

[57] Edward Loper and Steven Bird. NLTK: The Natural Language Toolkit. In Chris Brew and Michael Rosner, editors, *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ETMTNLP '02, pages 63–70, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[58] Kim Luyckx, Frederik Vaassen, Claudia Peersman, and Walter Daelemans. Fine-grained emotion detection in suicide notes: a thresholding approach to multi-label classification. *Biomedical Informatics Insights*, 5(Suppl 1):61–69, 2012.

[59] Gjorgji Madjarov and Dejan Gjorgjevikj. Hybrid Decision Tree Architecture Utilizing Local SVMs for Multi-Label Classification. In Emilio Corchado, Václav Snášel, Ajith Abraham, Micha\textbackslashl Woźniak, Manuel Graña, and Sung-Bae Cho, editors, *Hybrid Artificial Intelligent Systems*, pages 1–12, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[60] William C Mann and Maite Taboada. Rhetorical Structure Theory : looking back and moving ahead. *Discourse Studies*, 8(3):423–460, 2006.

[61] William C Mann and Sandra A Thompson. Rhetorical structure theory: Toward a functional theory of text organization. *Text-Interdisciplinary Journal for the Study of Discourse*, 8(3):243–281, 1988.

[62] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In Kalina Bontcheva and Zhu Jingbo, editors, *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Baltimore, Maryland, 2014. Association for Computational Linguistics.

[63] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, jun 1993.

[64] D.F. Marks and L. Yardley. *Research Methods for Clinical and Health Psychology*. SAGE Publications, 2004.

[65] Daniel Martens and Timo Johann. On the Emotion of Users in App Reviews. In *Proceedings of the 2Nd International Workshop on Emotion Awareness in Software Engineering*, SEmotion '17, pages 8–14, Piscataway, NJ, USA, 2017. IEEE Press.

[66] Hardik Meisheri and Lipika Dey. TCS Research at SemEval-2018 Task 1: Learning Robust Representations using Multi-Attention Architecture. In Marianna Apidianaki, Saif M Mohammad, Jonathan May, Ekaterina Shutova, Steven Bethard, and Marine Carpuat, editors, *Proceedings of The 12th International Workshop on Semantic Evaluation*, pages 291–299, New Orleans, Louisiana, 2018. Association for Computational Linguistics.

[67] Vincent Menger, Floor Scheepers, and Marco Spruit. Comparing Deep Learning and Classical Machine Learning Approaches for Predicting Inpatient Violence Incidents from Clinical Text. *Applied Sciences*, 8(6):981, 2018.

[68] Tim Menzies and Andrian Marcus. Automated severity assessment of software defect reports. In *IEEE International Conference on Software Maintenance, ICSM*, pages 346–355, 2008.

[69] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[70] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[71] Jonas Močkus, Vytautas Tiešis, and Antanas Žilinskas. The application of Bayesian methods for seeking the extremum. In George Philip Szegö and Laurence Charles Ward Dixon, editors, *Towards Global Optimisation*, volume 2, pages 117–128. North-Holland, 1978.

[72] Saif Mohammad. Word Affect Intensities. In Nicoletta Calzolari, Khalid Choukri, Christopher Cieri, Thierry Declerck, Sara Goggi, Koiti Hasida, Hitoshi Isahara, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asuncion Moreno, Jan Odijk, Stelios Piperidis, and Takenobu Tokunaga, editors, *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, may 2018. European Language Resources Association (ELRA).

[73] Saif Mohammad, Felipe Bravo-Marquez, Mohammad Salameh, and Svetlana Kiritchenko. SemEval-2018 Task 1: Affect in Tweets. In Marianna Apidianaki, Saif M Mohammad, Jonathan May, Ekaterina Shutova, Steven Bethard, and Marine Carpuat, editors, *Proceedings of The 12th International Workshop on Semantic Evaluation*, pages 1–17, New Orleans, Louisiana, 2018. Association for Computational Linguistics.

[74] Saif Mohammad, Svetlana Kiritchenko, and Xiaodan Zhu. NRC-Canada: Building the State-of-the-Art in Sentiment Analysis of Tweets. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013)*, pages 321–327, Atlanta, Georgia, USA, 2013. Association for Computational Linguistics.

[75] Saif M Mohammad and Peter D Turney. Crowdsourcing a Word-Emotion Association Lexicon. *Computational Intelligence*, 29(3):436–465, 2012.

[76] Raymond J. Mooney and Loriene Roy. Content-based book recommending using learning for text categorization. In *Proceedings of the Fifth ACM Conference on Digital Libraries*, DL '00, pages 195–204, New York, NY, USA, 2000. ACM.

[77] Rodrigo Moraes, João Francisco Valiati, and Wilson P Gavião Neto. Document-level sentiment classification: An empirical comparison between SVM and ANN. *Expert Systems with Applications*, 40(2):621–633, 2013.

[78] Preslav Nakov, Sara Rosenthal, Zornitsa Kozareva, Veselin Stoyanov, Alan Ritter, and Theresa Wilson. SemEval-2013 Task 2: Sentiment Analysis in Twitter. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013)*, pages 312–320, York, UK, 2013. Association for Computational Linguistics.

[79] Alena Neviarouskaya, Helmut Prendinger, and Mitsuru Ishizuka. Textual Affect Sensing for Sociable and Expressive Online Communication. In Ana C R Paiva, Rui Prada, and Rosalind W Picard, editors, *Affective Computing and Intelligent Interaction*, pages 218–229, Lisbon, Portugal, 2007. Springer Berlin Heidelberg.

[80] Nicole Novielli, Fabio Calefato, and Filippo Lanubile. A Gold Standard for Emotion Annotation in Stack Overflow. In Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 14–17, Gothenburg, Sweden, 2018. ACM.

[81] Nicole Novielli, Daniela Girardi, and Filippo Lanubile. A Benchmark Study on Sentiment Analysis for Software Engineering Research. In Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 364–375, New York, NY, USA, 2018. ACM.

[82] Marco Ortu, Bram Adams, Giuseppe Destefanis, Parastou Tourani, Michele Marchesi, and Roberto Tonelli. Are Bullies More Productive?: Empirical Study of Affectiveness vs. Issue Fixing Time. In Massimiliano Di Penta, Martin Pinzger, and Romain Robbes, editors, *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 303–313, Piscataway, NJ, USA, 2015. IEEE Press.

[83] Marco Ortu, Alessandro Murgia, Giuseppe Destefanis, Parastou Tourani, Roberto Tonelli, Michele Marchesi, and Bram Adams. The Emotional Side of Software Developers in JIRA. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 480–483, Austin, Texas, USA, may 2016.

[84] Olutobi Owoputi, Brendan O'Connor, Chris Dyer, Kevin Gimpel, and Nathan Schneider. Part-of-speech tagging for Twitter: Word clusters and other advances. Technical Report CMU-ML-12-107, School of Computer Science Carnegie Mellon University, Pittsburgh, PA, USA, 2012.

[85] Lluís Padró and Evgeny Stanilovsky. FreeLing 3.0: Towards Wider Multilinguality. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, pages 2473–2479, Istanbul, Turkey, may 2012. ELRA.

[86] Raquel Mochales Palau and Marie Francine Moens. Argumentation mining: The Detection, Classification and Structuring of Arguments in Text. *Belgian/Netherlands Artificial Intelligence Conference*, pages 351–352, 2009.

[87] Bo Pang and Lillian Lee. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In Kevin Knight, Hwee Tou Ng, and Kemal Oflazer, editors, *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 115–124, Ann Arbor, MI, USA, 2005. Association for Computational Linguistics.

[88] Ji Ho Park, Peng Xu, and Pascale Fung. PlusEmo2Vec at SemEval-2018 Task 1: Exploiting emotion knowledge from emoji and #hashtags. In Marianna Apidianaki, Saif M Mohammad, Jonathan May, Ekaterina Shutova, Steven Bethard, and Marine Carpuat, editors, *Proceedings of The 12th International Workshop on Semantic Evaluation*, pages 264–272, New Orleans, Louisiana, 2018. Association for Computational Linguistics.

[89] Paul Ekman, Wallace V. Friesen, and Phoebe Elssworth. *Emotion in the Human Face: Guidelines for Research and an Integration of Findings*, volume 11 of *Pergamon General Psychology Series*. Pergamon, 1972.

[90] Daniel Pletea, Bogdan Vasilescu, and Alexander Serebrenik. Security and Emotion: Sentiment Analysis of Security Discussions on GitHub. In Premkumar Devanbu, Sung Kim, and Martin Pinzger, editors, *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 348–351, New York, NY, USA, 2014. ACM.

[91] R Plutchik and H Kellerman. *Emotion: theory, research, and experience*, volume 1 of *Theories of Emotion,*. Academic Press, 1980.

[92] Edwin Raczko and Bogdan Zagajewski. Comparison of support vector machine, random forest and neural network classifiers for tree species classification on airborne hyperspectral APEX images. *European Journal of Remote Sensing*, 50(1):144–154, 2017.

[93] Preethi Raghavan, Rose Catherine, Shajith Ikbal, Nanda Kambhatla, and Debapriyo Majumdar. Extracting Problem and Resolution Information from Online Discussion Forums. In P Sreenivasa Kumar, Srinivasan Parthasarathy, and Shantanu Godbole, editors, *Proceedings of the 16th International Conference on Management of Data (COMAD)*, pages 77–88, Nagpur, India, 2010. Allied Publishers.

[94] Daniel Ramage, David Hall, Ramesh Nallapati, and Christopher D Manning. Labeled LDA: A supervised topic model for credit attribution in multi-labeled corpora. In *EMNLP '09 Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 248–256, 2009.

[95] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. Classifier chains for multi-label classification. *Machine Learning*, 85(3):333, jun 2011.

[96] Jesse Read, Antti Puurula, and Albert Bifet. Multi-label Classification with Meta-Labels. In Ravi Kumar, Hannu Toivonen, Jian Pei, Joshua Zhexue Huan, and Xindong Wu, editors, *2014 IEEE International Conference on Data Mining*, pages 941–946, Shenzhen, China, dec 2014.

[97] Jesse Read, Peter Reutemann, Bernhard Pfahringer, and Geoff Holmes. MEKA: A Multi-label/Multi-target Extension to Weka. *Journal of Machine Learning Research*, 17(21):1–5, 2016.

[98] Sara Rosenthal, Noura Farra, and Preslav Nakov. SemEval-2017 Task 4: Sentiment Analysis in Twitter. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 502–518, Vancouver, Canada, 2017. Association for Computational Linguistics.

[99] Mickael Rouvier. LIA at SemEval-2017 Task 4: An Ensemble of Neural Networks for Sentiment Classification. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 760–765, Vancouver, Canada, 2017. Association for Computational Linguistics.

[100] Nivir Kanti Singha Roy and Bruno Rossi. Towards an improvement of bug severity classification. *Proceedings - 40th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2014*, pages 269–276, 2014.

[101] Agnes Sandor, Nikolaos Lagos, Ngoc-Phuoc-An Vo, and Caroline Brun. Identifying User Issues and Request Types in Forum Question Posts Based on Discourse Analysis. In Jacqueline Bourdeau, Jim A Hendler, and Roger Nkambou Nkambou, editors, *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, pages 685–691, Montréal, Québec, Canada, 2016. International World Wide Web Conferences Steering Committee.

[102] Phillip Shaver, Judith Schwartz, Donald Kirson, and Cary O'Connor. Emotion knowledge: Further exploration of a prototype approach. *Journal of Personality and Social Psychology*, 52(6):1061–1086, 1987.

[103] V B Singh, Sanjay Misra, and Meera Sharma. Bug Severity Assessment in Cross Project Context and Identifying Training Candidates. *Journal of Information & Knowledge Management*, 16(01):1750005 1 —- 30, mar 2017.

[104] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In John Shawe-Taylor, Richard S Zemel, Peter L Bartlett, Fernando Pereira, and Kilian Q Weinberger, editors, *Proceedings of the 25th International Conference on Neural Information Processing Systems*, volume 2, pages 2951–2959, Lake Tahoe, Nevada, USA, 2012. Curran Associates Inc.

[105] Richard Socher, Jeffrey Pennington, Eric H Huang, Andrew Y Ng, and Christopher D Manning. Semi-Supervised Recursive Autoencoders for Predicting Sentiment Distributions. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 151–161, Edinburgh, Scotland, UK., 2011. Association for Computational Linguistics.

[106] Philip J Stone, Robert F Bales, J Zvi Namenwirth, and Daniel M Ogilvie. The General Inquirer: A computer system for content analysis and retrieval based on the sentence as a unit of information. *Behavioral Science*, 7(4):484–498, 1962.

[107] Carlo Strapparava and Rada Mihalcea. SemEval-2007 Task 14: Affective Text. In Eneko Agirre, Lluis Marquez, and Richard Wicentowski, editors, *Proceedings of the 4th International Workshop on Semantic Evaluations*, SemEval '07, pages 70–74, Prague, Czech Republic, 2007. Association for Computational Linguistics.

[108] Carlo Strapparava and Alessandro Valitutti. WordNet-Affect: an Affective Extension of WordNet. In Maria Teresa Lino, Maria Francisca Xavier, Fátima Ferreira, Rute Costa, and Raquel Silva, editors, *In Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC'04)*, pages 1083–1086, Lisbon, Portugal, 2004. European Language Resources Association (ELRA).

[109] Chengnian Sun, David Lo, Siau Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*, pages 253–262, 2011.

[110] Maite Taboada, Julian Brooke, Milan Tofiloski, Kimberly Voll, and Manfred Stede. Lexicon-Based Methods for Sentiment Analysis. *Computational Linguistics*, 37(2):267–307, 2011.

[111] Maite Taboada and Jack Grieve. Analyzing appraisal automatically. In Yan Qu, James Shanahan, and Janyce Wiebe, editors, *In Proceedings of the AAAI Spring Symposium on Exploring Attitude and Affect in Text: Theories and Applications*, pages 158–161, 2004.

[112] Mike Thelwall, Kevan Buckley, and Georgios Paltoglou. Sentiment Strength Detection for the Social Web. *Journal of the American Society for Information Science and Technology*, 63(1):163–173, 2012.

[113] Yuan Tian, David Lo, and Chengnian Sun. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 215–224, 2012.

[114] Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)*, 3(3):1–13, 2007.

[115] Grigorios Tsoumakas, Ioannis Katakis, and Ioannis Vlahavas. Effective and efficient multilabel classification in domains with large number of labels. In *Proceedings of ECML/PKDD 2008 Workshop on Mining Multidimensional Data (MMD'08)*, pages 53–59, Antwerp, Belgium, 2008.

[116] Grigorios Tsoumakas, Ioannis Katakis, and Ioannis Vlahavas. Random k-Labelsets for Multilabel Classification. *IEEE Transactions on Knowledge and Data Engineering*, 23(7):1079–1089, jul 2011.

[117] Grigorios Tsoumakas, Eleftherios Spyromitros-Xioufis, Jozef Vilcek, and Ioannis Vlahavas. Mulan: A Java Library for Multi-Label Learning. *Journal of Machine Learning Research*, 12:2411–2414, 2011.

[118] Rob; van Eemeren, Frans H.; Grootendorst. A Systematic Theory of Argumentation. The Pragma-Dialectical Approach. *Journal of Pragmatics*, 37(4):577–583, 2005.

[119] Hao Wang, Dogan Can, Abe Kazemzadeh, François Bar, and Shrikanth Narayanan. A system for real-time twitter sentiment analysis of 2012 us presidential election cycle. In Min Zhang, editor, *Proceedings of the ACL 2012 System Demonstrations*, ACL '12, pages 115–120, Jeju Island, Korea, 2012. Association for Computational Linguistics.

[120] Theresa Wilson, Janyce Wiebe, and Paul Hoffmann. Recognizing Contextual Polarity in Phrase-level Sentiment Analysis. In Raymond J Mooney, editor, *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing (HLT '05)*, pages 347–354, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[121] Xi-Zhu Wu and Zhi-Hua Zhou. A Unified View of Multi-Label Performance Measures. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3780–3788, International Convention Centre, Sydney, Australia, aug 2017. PMLR.

[122] Xin Xia, Yang Feng, David Lo, Zhenyu Chen, and Xinyu Wang. Towards more accurate multi-label software behavior learning. *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014 - Proceedings*, pages 134–143, 2014.

[123] Ali Yadollahi, Ameneh Gholipour Shahraki, and Osmar R Zaiane. Current State of Text Sentiment Analysis from Opinion to Emotion Mining. *ACM Computing Surveys (CSUR)*, 50(2):25:1—-25:33, may 2017.

[124] Wataru Yamada, Haruka Kikuchi, Keiichi Ochiai, Shu Takahashi, Yusuke Fukazawa, Hiroshi Inamura, and Ken Ohta. Multi-label categorizing local event information from micro-blogs. *2016 9th International Conference on Mobile Computing and Ubiquitous Networking, ICMU 2016*, 2016.

[125] Cheng Zen Yang, Chun Chi Hou, Wei Chen Kao, and Ing Xiang Chen. An empirical study on improving severity prediction of defect reports using feature selection. In *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, volume 1, pages 240–249. IEEE, 2012.

[126] Yi Yang and Jacob Eisenstein. Putting Things in Context: Community-specific Embedding Projections for Sentiment Analysis. *CoRR*, abs/1511.06052, 2015.

[127] M Zhang. A k-Nearest Neighbor Based Multi-Instance Multi-Label Learning Algorithm. In Éric Grégoire, editor, *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2010)*, volume 2, pages 207–212, Arras, France, oct 2010.

[128] Min-Ling Zhang and Zhi-Hua Zhou. Multilabel Neural Networks with Applications to Functional Genomics and Text Categorization. *IEEE Transactions on Knowledge and Data Engineering*, 18(10):1338–1351, oct 2006.

[129] Tao Zhang, Jiachi Chen, Geunseok Yang, Byungjeong Lee, and Xiapu Luo. Towards more accurate severity prediction and fixer recommendation of software bugs. *Journal of Systems and Software*, 117:166–184, 2016.

[130] Bo Zhou and Xin Xia. Towards More Accurate Content Categorization of API Discussions Categories and Subject Descriptors. pages 95–105, 2014.

[131] Vladimir Zolotov and David Kung. Analysis and Optimization of fastText Linear Text Classifier. *arXiv preprint arXiv:1702.05531*, 2017.

Code Snippets on Stack Overflow: Are they always labelled correctly?

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

***Abstract*—The increasing size of Stack Overflow's data has attracted researchers to explore and use it. For example, recent methods have been proposed to detect duplicated questions in Stack Overflow automatically or to match natural language search queries with code portions inside an IDE. The success of these methods depends on structures integrated in Stack Overflow, such as tags and annotations, and in turn on how accurately they are used. In this paper, we focus on the code tag and analyse how these tags are used, misused or omitted by users when posting on Stack Overflow. Tagging code snippets correctly is crucial for the performance of more sophisticated tools that can help developers. For this analysis, we have developed a code detector using a classifier based on FastText and word embeddings. The detector can identify whether a block of text, of any length, is either English or code. It has been evaluated on a large Stack Overflow data set of 3.6 million posts and achieved promising results. Our analysis of the results has shown that the code tag is sometimes omitted and, in occasions, it is used to mark a variety of related but distinct concepts like: stack traces, errors and input-output text. These omissions and misuses can be corrected using the code detector.**

***Index Terms*—Stack Overflow, code detection, machine learning, word embeddings, automatic classification**

I. Introduction

Question-and-answer (Q&A) websites allow users to find a replies to their questions or to provide help to other users through the exchange of knowledge. During the early days of the internet, only a few Q&A websites existed, such as *Experts Exchange*[1], for IT-related topics, and *Answer Point*[2], for general questions. The success of social media revived interest in Q&A websites [1]. Currently, a variety of Q&A websites are available: general-knowledge ones, such as *Yahoo! Answers*[3], *Answers.com*[4] and *Quora*[5] and more specialised ones, such as *Stack Exchange Network*[6], which supports multiple communities by area of expertise, and the Q&A component of *ResearchGate*[7].

Stack Overflow[8] is a crowd-sourced Q&A website, very popular within the communities of software developers [2], [3]. Lately, a number of articles related to Stack Overflow have focussed on processing questions and answers [4], [5], practices for software development [3], [6] and on aspects of human behaviour on this communication means [1], [7].

The quality of Stack Overflow data has been previously investigated [8]. However, to the best of our knowledge, the reliability of data, in terms of the internal structure of posts and code tags, has not been questioned. Stack Overflow allows users to markup their posts to improve readability and comprehension. One of the most common tags in Stack Overflow is *"code"*, useful for highlighting code snippets. As users are the only responsible for formatting posts, tags may be omitted or misused.

To rectify these code markup inconsistencies, Stack Overflow supports a verification process. A series of heuristics are applied to detect code on each post posted by a user with reputation less than 50 points. If the post contains unannotated code snippets, the user is notified and is not allowed to post it until the markup is correct[9]. However, some users are unaware of the code tag, despite their high reputation[10].

Stack Overflow is supported by a large community of users and moderators, who can rectify the markup of posts to improve their readability[11]. Nonetheless, a visually well-formatted post may not be parsed correctly by an HTML/XML parser due to a "tag soup"[12], i.e. use of malformed HTML/XML markup, invalid structure and/or unescaped characters. In addition there is no guarantee that the heuristics to recover from parsing errors, such as the ones in *BeautifulSoup*[13], *Jsoup*[14] and *LibXML2*[15], will figure out the correct HTML/XML structure and markup.

[1] Experts Exchange (experts-exchange.com) went on-line in 1996.
[2] The service was originally available in *Ask Jeeves*, now Ask.com.
[3] answers.yahoo.com
[4] answers.com
[5] quora.com
[6] stackexchange.com
[7] researchgate.net
[8] stackoverflow.com
[9] meta.stackexchange.com/questions/27352
[10] meta.stackexchange.com/questions/224054
[11] meta.stackexchange.com/questions/88627
[12] Generally, new web browsers include a tag soup parser in order to interpret structurally incorrect HTML/XML, but it is not the case for every HTML/XML parsers alone.
[13] crummy.com/software/BeautifulSoup
[14] jsoup.org/
[15] xmlsoft.org

Verifying that text marked up as code is truly code and vice-versa can affect the performance of tools based on Stack Overflow data, such as duplicate question finders [5] and natural-language-based code searchers [4]. Using a code detector to correct code markup prior to any other processing can prevent later problems and increase processing performance.

Code detection in electronic means of communication, such as emails and forums, has been investigated previously [9]–[11]. However, it has not been applied to Stack Overflow, mainly because it supports a specific markup label for code that users are expected to use constantly and correctly.

In this paper, we present a code detector based on a neural network and word embeddings to analyse Stack Overflow data. More specifically, it is an automatic classifier aiming to determine if an HTML text block corresponds to programming or markup code, e.g. in *Java* or *HTML*, or natural language, e.g. English or Spanish. We applied the code detector on a large portion of the Stack Overflow Data Dump[16] and manually evaluated a sample of the output to estimate how the code tag is used in this Q&A website. Manual analysis has shown that code tags are used not only to formatting code snippets, but also to highlight commands, error traces or logs. We also present problematic cases that would limit the performance of post-processing tools, if not picked up by the code detector.

The remaining of this work consists of nine sections. In Section II, we make a review of works that are based on the use of Stack Overflow data. In Section III, we present the state-of-the-art regarding the methods to automatically detect code in text documents. We introduce the necessary background information in Section IV while explaining the methodology in Section V. Then, in Section VI, we present the data used to train the code detector and the corpus used for the experiments. After, in Section VII, we explain how the experiments were done and how they were evaluated. In Section VIII, we explain the limitations of this work. The results and their discussion are presented in Section IX and Section X, respectively. Finally, Section XI concludes this paper.

II. Related Work

Recently, literature investigating Stack Overflow users and data has flourished. State-of-the-art articles are related to: (a) text and code processing, (b) practices for software development and (c) human aspects.

Text and code processing: This area concerns tools and methods applied to the data posted by users, i.e. questions, answers and comments. They target to improve data quality or to increase the performance of developers. For example, two approaches for duplicate question detection have been proposed to improve the website's quality by reducing redundancy and response time [5]. Stack Overflow data has been made available to developers via an IDE plugin. Developers can search code through natural languages queries, e.g. "parsing an HTML document using Python" [4]. To reduce the workload of moderators, automatic detection of low quality posts has been proposed [8].

Practices for software development: Work in this area aims in understanding how developers use Stack Overflow. For instance, an analysis of the (re)use of code snippets posted on Stack Overflow in Github repositories and vice-versa is presented in [6]. It was also argued that developers do not use Stack Overflow only to improve their knowledge, but also to document bugs and publish improvements that they have implemented [3].

Human aspects: As Stack Overflow is a collaborative website governed by certain rules and principles, such as reputation points, researchers are interested on how human aspects affect or interact with it. Plagiarism, gender participation and even cases of collusion in Stack Overflow have been studied in [7]. Issues such as which questions are asked, how they are answered and how users select which questions to answer have been investigated in [1].

The present study is part of the text and code processing area, as we analyse data published on Stack Overflow. However, we do not use it to develop explorative tools, but to improve its reliability for further exploitation. In particular, we focus on analysing how users separate code from text using the code markup label.

III. Code Detectors

A code detector is a tool able to determine whether a document or a part of it, is code in some programming or markup language, e.g. Java or XML, or whether it is text in a natural language, e.g. English or French. Most code detectors in the literature aim at improving the communication among developers via electronic means, such as e-mails and forums. Knowing which parts of a document are natural language or code can improve the performance of other tools, such as search engines and indexes [9]–[12].

State-of-the-art code detectors are based on a variety of methods, ranging from basic techniques, such as heuristics, to more complex ones, such as Hidden Markov Models. These methods are summarised below.

Basic Heuristics. Stack Overflow's code detector[17] detects code using regular expressions, keywords and space indentation counting.

Heuristics and Clustering. In [9], natural language text and technical information, i.e. code, patches, stack traces and log extracts, are classified using a system based on heuristics and agglomerative hierarchical clustering. The heuristics consist of detecting programming keywords, repetitive lines, patch patterns, brackets, indentations and medial capitals[18]. Agglomerative hierarchical clustering is applied to correct the errors caused by the heuristics. For example, a string in a program is classified as natural language text. Although strictly it is natural language, it occurs in code.

[16] archive.org/details/stackexchange

[17] meta.stackoverflow.com/questions/341763

[18] Medial capitals, known as well as camel cases, consist of using capital letters in the middle of a compound word, e.g. `getValue`.

E-mail Unified Content Classification Approach[19]: Also known as MUCCA, this method was designed to classify email text lines as natural language, code, patches or stack traces [10]. MUCCA employs two different ways to determine if a text line is natural language: a Naïve-Bayes classifier that uses unigrams, bigrams and punctuation as features and *island parsers*. An island parser is a set of rules that capture the grammar of the language of interest (the "islands") but ignore other languages constructions (the "water") [12]. MUCCA was developed to detect uniquely code in Java, although it is possible to retrain the Naïve-Bayes classifier and create new island parsers for other programming languages.

InfoRmation ISlands Hmm (IRISH)[20]: This system is based on two Hidden Markov Models (HMM), one for natural language detection and one for code detection [11]. Instead of using grammars or regular expressions, it is trained on an alphabet composed of words (alphanumeric characters sequences separated by white space or underscore), keywords, underscore, numbers and other characters. The HMMs use the alphabet to learn frequent characteristics of natural language and code.

None of the existing code detectors has explored the newest technologies, such as neural networks and word embeddings. In this paper, we explore a neural network classifier that uses word embeddings. In specific, our code detector is a classifier based on FastText [13], which is in turn founded on word embeddings, a rich semantic word representation. The reason for using word embeddings for the code detector is that they capture semantic properties and characteristics [14], [15], which have been shown to improve the performance of other tools [16], [17]. Consequently, using word embeddings in a code detector is expected to bring some benefits over other methods, such as those founded on grammars.

IV. Background Information

In this section we present background definitions, theory and other information necessary to better understand how our code detector works and was trained. The first subsection introduces the notion of Word Embeddings. The second one presents FastText, the tool used for creating the neural network classifier. Finally, the third subsection is about Bayesian Optimisation, a method used to find the best hyper-parameters for FastText's neural networks.

A. Word Embeddings

A *word embedding* is a dense vector representation, in which a word keeps its semantic characteristics and properties as well as its relations with other words. Word embeddings support simple mathematical operations. For example, *vector("biggest")-vector("big")+vector("small")* produces a vector similar to the vector of *smallest* [14].

The benefits of word embeddings, over other methods to represent words or text in general, is that they can address tasks such as synonymy, polysemy or analogy identification [14]. In addition, word embeddings can contain information related to regional spelling and sentiment polarity, as for example in [15]. They have been successfully used to improve performance in other tasks, such as name entity recognition [16], automatic translation [18] and sentiment analysis [17]. In the last lustrum, word embeddings have been used widely in the field of natural language processing and have shown their benefits.

Methods to create word embeddings are based either on counting or predicting context units [19]. In the former case, embeddings are based on word co-occurrence counts and algebraic space transformations, e.g. Latent Semantic Analysis [20], [21] and Glove [22], whereas in the latter case word embeddings are based on neural networks, as in Word2Vec [14], [23] and FastText [13], [24].

B. FastText

FastText [13] is a library of tools developed by Facebook Research. Among others, it provides a linear classifier that generates word embeddings specialised for text classification. The classifier uses *Continuous Bag-of-Words (CBOW)*, a modified version of the neural network architecture that was previously developed for Word2Vec [14], [23].

The CBOW neural network architecture consists of four different layers: the input, the projection, the hidden and the output. It links the input text with the expected label, by creating an *embeddings matrix* and determining neurons' weights, located in the hidden and the output layer. The embeddings matrix is the mechanism behind the dense representation of words. Figure 1 graphically represents FastText's CBOW architecture using a training instance, where we input a text and a label. In the input layer, the text is represented as a collection of sparse vectors. The vectors are modified by the embeddings matrix to obtain word embeddings, i.e. dense vectors, in the projection layer. Words embeddings are then averaged and processed to compute a label, in the hidden and output layer. Concerning the example in Figure 1, it should be noted that although *"getDate()"* is a code snippet, it is considered as text in English due to its context.

FastText has been shown to perform comparably to more complex deep learning algorithms but takes less time to train and does not require a GPU to run [25]. This last point can be crucial to make our code detector more accessible to new tools and research communities. Although FastText computes embeddings for the classification task at hand, they can be reused in other tasks or tools.

C. Bayesian Optimisation

Bayesian Optimisation [26], as any optimisation method, determines a set of parameter values to maximise an objective function. The method evaluates the objective function by varying combinations of input parameters, and collects the input and output values into a Gaussian process. After multiple iterations, the Gaussian process is able to predict which unseen parameter values could maximise the objective function.

In this paper, we use Bayesian Optimisation to determine which FastText neural network parameters perform best. Following [27], Bayesian Optimisation is more suitable than brute

[19] mucca.inf.usi.ch

[20] rcost.unisannio.it/cerulo/dataset-irish.tgz

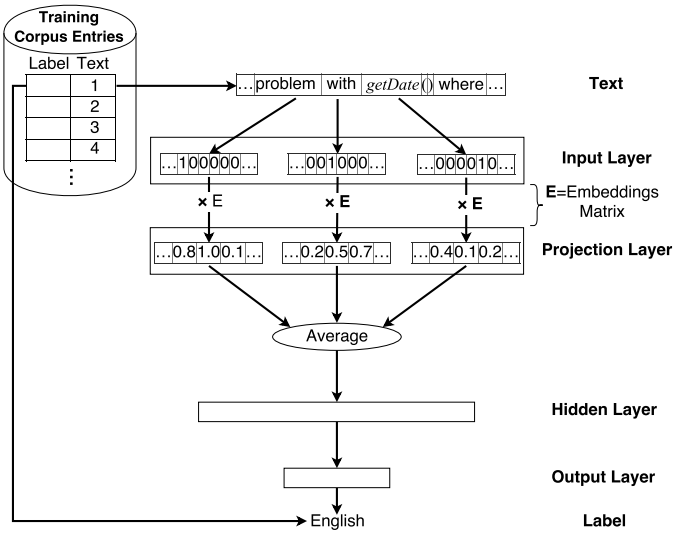

Figure 1. Architecture of FastText CBOW neural network using a training instance

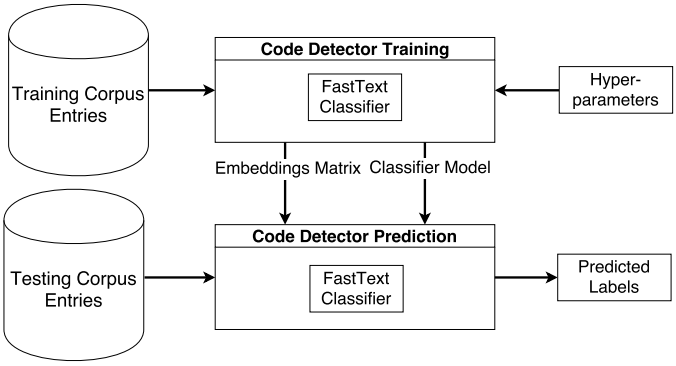

Figure 2. Workflow diagram: Training and testing the code detector

force methods, e.g. grid search, to estimate machine learning parameters. Moreover, Bayesian Optimisation can estimate parameters quicker than brute force methods. Methods, such as grid search, inspect all values in a range, while Bayesian Optimisation first evaluates parameter combinations randomly but then predicts parameters that should perform best.

V. Methodology

Developing the code detector involves training FastText neural network on a newly-induced training corpus (Section VI-A) and estimating a set of hyper-parameters (Section VII). After the training process, FastText produces an embeddings matrix and a trained classification model. The model can be loaded on any FastText instance to predict the label of instances in a testing corpus (Section VI-B). In Figure 2 we present graphically the process of training and testing the code detector.

The code detector aims at detecting whether parts of Java-related Stack Overflow posts are natural language or code. Java-related posts were selected as Java is one of three most popular programming languages on Stack Overflow[21], with more than $1,3$M questions by the end of 2017. Due to the popularity of Java, it is easy to find experts able to evaluate the results.

VI. Data

This section discusses the data used for training and testing the code detector.

A. Code Detector Training Data

For training, we created a corpus of 200K English sentences and 200K Java code lines from different resources. English sentences were randomly selected from the *Leipzig Corpora Collection*, which contains 1M sentences from Wikipedia recorded in 2012 and 2016 [28]. 200K Java code lines were randomly extracted from the *GitHub Java Corpus* [29]. Lines were split using Perl regular expressions that located newline characters such as \n and \r.

The training corpus does not contain duplicate English sentences, but it contains some very similar or identical code lines. Due to the nature of code, some duplication is expected by convention, need or coincidence. Examples of common code lines are: `{`, `import java.util.*;` and `Date date = new Date();`.

Both parts of the corpus have been pre-processed in the same manner. Every line was encoded in UTF-8 and characters that vary in English sentences, such as quotes and white space, were normalised. Brackets, question marks and dots were separated from words using spaces, e.g. *"I can't believe it!"* and *"public static void main(String[] args)"* were converted to *"I can ' t believe it !"* and *"public static void main (String [ ] args)"*, respectively[22]. While text was lowercased, following [10] we did not apply stemming[23] or lemmatisation[24] or stop-word filtering[25].

Although the size of the training corpus (400K entries) was selected empirically, thanks to the Bayesian Optimisation we can estimate the parameters that achieve the best performance, i.e. to maximise the percentage of correctly classified text lines with a model trained on the training corpus. Moreover, the size of the training corpus leads to a small model, easy to reuse in other tools. Word embeddings models, such as the ones in FastText, tend to be large. For example, the pre-trained models of Glove [22] are zipped in files with sizes between 800MB and 2.03GB. The pre-trained model of FastText [24] on the English Wikipedia is zipped in a file of 9.6GB. Although these models are useful in many NLP tasks, they may not be suitable for developing Software Engineering tools, such

[21] insights.stackoverflow.com/survey/2017

[22] We did not add spaces where a dot is not followed by space. Following the rules used in MUCCA [10], this formatting exception was introduced to treat dots in code differently than full stops in natural language. It keeps sequences such as *System.out.println* as one string, whereas splits *"The end."* as *"The end ."*.

[23] A stemmer computes the root word of each word using heuristics.

[24] A lemmatiser normalises words into their dictionary form. In contrast to a stemmer, it uses dictionaries, probabilities and rules.

[25] A stop word is a word which does not provide useful information for a given task. Examples of function words are *if*, *the*, *nonetheless*.

Table I
NUMBER OF HTML LINES, TAGGED AS CODE AND NATURAL LANGUAGE, IN THE 3.6M POSTS TAGGED AS JAVA.

	Code	Natural Language
HTML Blocks	8.5M (33.1%)	17.3M (66.9%)

Table II
HYPER-PARAMETERS USED FOR TRAINING THE CODE FETECTOR BASED ON FASTTEXT.

LR	Epoch	Dim	N-grams	
			Min	Max
0.458	30	100	1	5

as IDE plugins. The size of the trained models is discussed further in Section VII, which describes the training process in general.

B. Stack Overflow Data

The *Stack Exchange Data Dump*[26] is an anonymised dump of all user-contributed content on the Stack Exchange network. In this paper, we use the version that was published in December 2017. The dump contains data collections from different Stack Exchange sites. We focused on the Stack Overflow collection and on data related to posts in particular. The data refers to activity between 2008 and 2017, and is contained in an XML file of 47.6M entries, that offer a wealth of information, such as whether a post is a question or an answer, its unique ID, its creation date, associated tags, the number of views and its text body in HTML format.

As explained in Section V, this article focuses in the analysis of Java-related posts. Thus, we searched the posts, i.e. questions or answers, that were tagged with the keyword "Java". We accessed this information by using Java's library *Simple API for XML Parser*, better know as SAX Parser.

The HTML body of every Java-related post was parsed using *jsoup*[27]. First we deleted some uninformative HTML tags related to format, such as bold face (<*b*>), emphasis (<*em*>) or italics (<*i*>). Then, we extracted the blocks of text marked up with the HTML tag <*code*>. These blocks were considered as code tagged by Stack Overflow users. The remaining HTML blocks were considered as tagged as natural language by Stack Overflow users. Finally, every HTML block was pre-processed using the rules applied on the training data.

In total, 3.6M posts were tagged as Java and were parsed into 25.8M HTML blocks. Table I shows statistics of posts marked with the Java tag in the Stack Overflow data. Code snippets and natural language blocks were extracted to investigate how the code tag is used in Stack Overflow and whether tools based on this data should pre-process it prior to using it.

VII. EXPERIMENTAL SETTINGS

Training a neural network, as the one in FastText, requires to set a number of hyper-parameters. Their values were determined using Bayesian Optimisation in a 10-fold cross validation process. The median F-score[28] value of the 10-fold cross validation was used as the objective function.

Table II presents the hyper-parameter values as computed by the Bayesian Optimisation process, where the objective function, i.e. median F-score, reached a value of 0.994. The following hyper-parameters were considered:

- *Learning rate (LR):* The learning rate controls the frequency of updating the weights of neurons.
- *Epoch:* the number of passes that the neural network will do on the full training corpus during the training process.
- *Dimensions (Dim):* the number of dimensions of the vector space in which word embeddings are represented.
- *N-grams*: FastText allows computing embeddings of n-grams. An N-gram is a sequence of consecutive tokens, i.e. character sequences separated by white-space. For example, *"String [ ] args"* is a 4-gram, whereas *"hello"* is a unigram. FastText requires to set the minimum and maximum size of n-grams.

Using the hyper-parameter values in Table II, the classifier including the word embeddings has a size of 16.5MB. Once the code detector was trained on the full training data using the selected hyper-parameter values, it was applied to every HTML block in the Java-related posts of Stack overflow data.

A. Baselines

Besides the code detector created using FastText, three baselines were used in the evaluation process. The first one determines the label randomly. The second baseline labels all HTML blocks with the most frequent label, henceforth called *MFL*. Finally, the third baseline labels as code every HTML block that contains at least one curly bracket, i.e. "{" or "}".

B. Manual Annotation

From the total amount of HTML blocks analysed by the code detector, we randomly selected $1,000$ and evaluated them manually. 500 of these $1,000$ HTML lines (50%) were originally tagged as code by Stack Overflow users and 500 were marked up as English.

Two expert programmers annotated each HTML block with respect to two main classes: Code and Natural language. In the ideal situation, the manual annotation tags should match the tags used by Stack Overflow users. Strictly speaking, the FastText classifier should only classify as code HTML blocks that were originally tagged as *code* and the remaining HTML blocks as natural language.

Besides these classes, we asked the expert programmers to propose sub-divisions of the two main classes. For code, the experts identified the following sub-classes: *Programming code*, *Markup code*, *Command* and *Other code*. The natural language class was divided into: *English*, *Error/Stack trace*, *Log/Input/Output*, and *Other Natural Language*. The sub-classes of type *other*, represent HTML blocks that are ambiguous or can have multiple interpretations, but due to their

[26] archive.org/details/stackexchange
[27] jsoup.org
[28] The F-score is the harmonic mean of *recall* and *precision*.

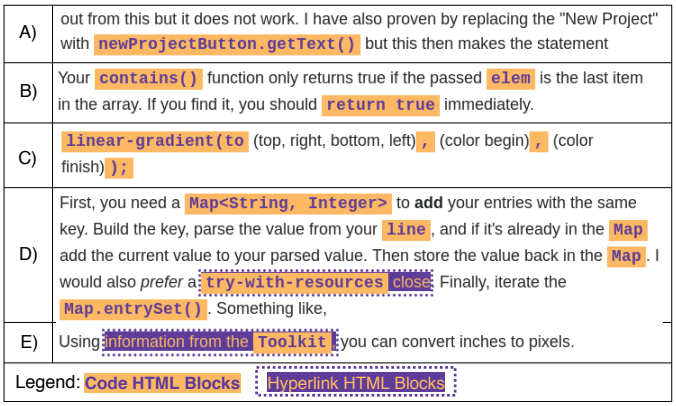

Figure 3. Examples of posts indicating use cases of the code tag in Stack Overflow. The CSS used in Stack Overflow has been modified to visually enhance the cases.

context they most probably belong either to code or natural language.

We calculated the inter-annotator agreement for the main classes and the sub-classes, using *Cohen's Kappa*. For the main classes, the inter-annotator agreement was $\kappa = 0.787$, indicating substantial agreement [30]. The instances on which annotators disagreed, in most cases, were related to function calls surrounded by natural language in the same line, e.g. example *A* in Figure 3. Concerning the sub-classes, the inter-annotator agreement was $\kappa = 0.527$, i.e. of moderate strength [30]. The lower inter-annotator agreement with respect to the main classes, is due to the fact that one annotator preferred to label variable names or types as *English*, while the other annotator as *Other Natural Language*. Examples of variable names are shown in Figure 3, instance *B*.

Annotation disagreements were resolved by the expert programmers after analysing the conflicted cases. Regarding HTML blocks surrounded by natural language but tagged as code, the experts finally considered most of them as *Other Natural Language*.

Manual annotation was performed to fulfil two objectives: firstly, to evaluate the proposed code detector with real-world data and secondly, to analyse how leniently Stack Overflow users utilise the code tag to markup text other than code snippets.

In addition to assigning annotation labels, the programming experts were asked to indicate if they observed any parsing problems. This binary annotation allows measuring possible HTML parsing errors. It sets a baseline about the number of posts whose HTML structure is well-formatted visually but structured wrongly. In Section IX, we analyse the set of $1,000$ annotated HTML blocks.

VIII. Limitations

In this work, there are some limitations regarding the possible analyses of Stack Overflow data. We focussed on tags utilised by users to mark up their post. Therefore, if an HTML block contains natural language and code, but no code tags is used, the code detector will not be able to distinguish both parts. The reason is that the input is an HTML block and not a smaller unit, like a line or a sentence. For example, in "...you should use System.err.println(error); to print ...", the portion of text `System.err.println(error);` is not tagged as code. Therefore, the code detector may consider the HTML block as English as it cannot distinguish, in the current version, that code portion.

In some other cases, users may use the code tag in an English context to highlight the name of a variable or a library, although these mentions are not strictly code. For instance, in example *D* in Figure 3, the code detector may detect `Map<String, Integer>` as code and `line` as natural language, whereas the manual annotator may consider both as natural language because of their context.

IX. Results

In this section, we present the results of analysing the manual annotated data, presented in Section VII-B. The results are divided in three parts. The first part discusses HTML parsing issues. In the second part, we analyse the use of the code tag on Stack Overflow as opposed to the manual annotations. Finally, in the third part, we focus on how the code detector, presented in this article, can help to address problems regarding the use, misuse or omission of the code tag in Stack Overflow.

A. With Respect to HTML Parsing

The annotators did not find any case in which *jsoup* could not parse a Stack Overflow post. However, they found a case, where the parser over-segmented a code block due to line break tags, (*
*), occurring within the code tag. Consequently, the parser produced multiple code blocks despite visually there was only one.

Although these cases are not parsing errors, the annotators located instances where the code tag could have been used erroneously or it can be debated if the closing tag was at the correct position. Some of these cases can be observed in examples *C*, *D* and *E* in Figure 3.

B. With Respect to Stack Overflow Users

Table III presents a confusion matrix regarding the tags utilised by Stack Overflow users and the labels assigned by the expert programmers. It can be observed that 282 HTML blocks tagged by users as code, for the annotators are in fact natural language. On the other hand, just two HTML blocks were not tagged correctly as code. Table IV presents in which sub-classes HTML blocks were distributed by the expert programmers. Thanks to Table IV, we can distinguish that the greatest part of the 282 conflicting HTML blocks were sub-classified as *Other Natural Language*. Despite being tagged as code, 199 instances should be considered as natural language due to their context. In Table IV, we can also observe that several blocks, i.e. errors, stack traces, logs, inputs and outputs, that are frequently annotated as code by Stack Overflow users, are not code strictly.

Due to the unbalanced proportionality of the annotated corpus, we referred to it as the *Unbalance* corpus.

Table III
CONFUSION MATRIX REGARDING THE TAGS USED BY STACK OVERFLOW USERS AND THE LABELS CHOSEN BY THE ANNOTATORS.

		User tags		
		Code	Natural Language	
Manual classes	Code	218	2	220
	NL	282	498	780
		500	500	1,000

Table IV
TABLE PRESENTING HOW STACK OVERFLOW USERS UTILISE THE TAGS TO ANNOTATE THE SUB-CLASSES IDENTIFIED BY THE ANNOTATORS.

		Users tags		
Manual sub-classes		Code	Natural Language	
Code	Programming	194	1	195
	Markup	17	0	17
	Command	4	1	5
	Other	3	0	3
NL	English	62	489	551
	Error/Stack trace	10	2	12
	Log/Output/Input	11	1	12
	Other	199	6	205
		500	500	1,000

Table V
CONFUSION MATRIX REGARDING THE CLASSIFICATION OF STACK OVERFLOW DATA IN THE UNBALANCED CORPUS BY THE CODE DETECTOR.

		Prediction classes		
		Code	Natural Language	
Manual classes	Code	208	12	220
	NL	184	596	780
		392	608	1,000

Table VI
EXTENDED CONFUSION MATRIX PRESENTING HTML BLOCKS IN THE UNBALANCE CORPUS CLASSIFIED BY THE CODE DETECTOR WITH RESPECT TO THE SUB-CLASSES IDENTIFIED BY THE EXPERT PROGRAMMERS.

		Prediction classes		
Manual sub-classes		Code	Natural Language	
Code	Programming	191	4	195
	Markup	13	4	17
	Command	2	3	5
	Other	2	1	3
NL	English	91	460	551
	Error/Stack trace	5	7	12
	Log/Output/Input	4	8	12
	Other	84	121	205
		392	608	1,000

C. With Respect to the Code Detector

Taking into account the results presented in Table III, we have decided to evaluate how well the code detector can match the manual annotations. In Table III we observed that the number of HTML blocks tagged as code by users surpass the overall number of code blocks that, in accordance to our experts, truly exist. The conflicted cases in the Unbalance corpus should be resolved by applying the code detector.

Table V presents the results of applying the code detector on the Unbalance corpus. It can be observed that the code detector reduces the number of HTML blocks tagged as code by the users, that in fact are natural language. Nevertheless, it fails to detect a small portion of natural language HTML blocks, which are incorrectly labelled as code.

In Table VI, we present an extended confusion matrix, which highlights more clearly which sub-classes are the most conflicting for the code detector. We can observe that the code detector classifies correctly 97.9% of the HTML blocks labelled as *Programming Code*, whereas it classifies accurately 76.4% of the HTML blocks of sub-class *Markup Code*. For the sub-classes, *Command* and *Other Code*, the code detector is less accurate. However, it should be taken into account that the code detector was only trained on Java code.

As the number of HTML blocks in the Unbalance corpus is not proportional, we evaluated the results using Sensitivity[29] and Specificity[30]. Sensitivity, in our scope, indicates how well a system classifies the HTML blocks annotated manually as code. In contrast, specificity indicates how well a system classifies the HTML blocks manually annotated as natural language.

[29]Number of HTML blocks predicted as code divided by the number of HTML blocks manually annotated as code.

[30]Number of HTML blocks predicted as NL divided by the number of HTML blocks manually annotated as NL.

In Table VII, we show the results in terms of Sensitivity and Specificity for the code detector and the respective baselines: Random, Curly Bracket and MFL. The MFL baseline assigns to every HTML block the label natural language, as it is the most frequent label according to the manual annotators. We can observe, thanks to the Sensitivity, that the code detector detects more accurately HTML blocks that contain code, than those that contain natural language (0.954 vs 0.764). Regarding the baselines, the Curly Bracket is the best performing one. Nonetheless, although good for the detection of natural language HTML blocks, Curly Bracket baseline is less successful in classifying code HTML blocks. The Random and MFL baselines performed as expected. In the former, we anticipated values around 0.500, as the probability of selecting one class was 50%. In the latter, as it classifies all HTML blocks as natural language, it was certain that sensitivity would be zero.

Although the unbalanced corpus is helpful to understand how well the code detector and the baselines classify the annotated HTML blocks, the lack of proportionality in the corpus makes harder to estimate the real performance of the system. Thus, we decided to test the code detector on a balanced subset of the $1,000$ HTML blocks corpus. In this subset, called henceforth *Balance*, the proportion of HTML blocks manually classifies as code and natural language is equal. The balanced corpus contains 440 HTML blocks. Its size was chosen to take into account the maximum number of code HTML blocks, 220, found in the manually annotated corpus. 220 HTML blocks manually tagged as natural language were randomly selected from the 780 ones in the Unbalance corpus.

Table VIII shows the confusion matrix related to the application of the code detector on the balanced corpus. The corresponding extended confusion matrix is presented

Table VII
SUMMARY OF THE RESULTS FOR THE CODE DETECTOR AND THE BASELINES, IN TERMS OF SENSITIVITY AND SPECIFICITY, ON THE UNBALANCE CORPUS.

		Sensitivity	**Specificity**
Code Detector		0.954	0.764
Baseline	Curly Bracket	0.513	0.994
	Random	0.504	0.508
	MCL	0	1

Table VIII
CONFUSION MATRIX REGARDING THE CLASSIFICATION OF STACK OVERFLOW DATA IN THE BALANCE CORPUS BY THE CODE DETECTOR.

		Prediction classes		
		Code	Natural Language	
Manual class	Code	208	12	220
	NL	52	168	220
		260	180	440

Table IX
EXTENDED CONFUSION MATRIX PRESENTING HTML BLOCKS IN THE BALANCE CORPUS CLASSIFIED BY THE CODE DETECTOR WITH RESPECT TO THE SUB-CLASSES IDENTIFIED BY THE EXPERT PROGRAMMERS.

	Prediction Classes			
Manual sub-classes		Code	Natural Language	
Code	Programming	191	4	195
	Markup	13	4	17
	Command	2	3	5
	Other	2	1	3
NL	English	28	135	163
	Error/Stack trace	0	1	1
	Log/Output/Input	1	4	5
	Other	23	28	51
		260	180	440

Table X
SUMMARY OF THE RESULTS FOR THE CODE DETECTOR AND THE BASELINES, IN TERMS OF F-SCORE, FOR THE BALANCE CORPUS.

		F-score Code	**F-score NL**
Code Detector		0.866	0.839
Baseline	Curly Bracket	0.678	0.803
	Random	0.499	0.490
	MCL	0	0.666

in Table IX. It can be observed that the number of natural language HTML blocks wrongly predicted as code is almost 4 times the number of code HTML blocks predicted incorrectly as natural language. Table IX reveals that the biggest difficulty for the code detector is to classify correctly the HTML blocks annotated manually as *Other Natural language*.

Table X presents the results, in terms of F-score, of applying the code detector and the baselines on the Balance corpus. The F-score, the harmonic mean of *recall* and *precision*, indicates how precise and robust a system is regarding the prediction of a specific class. It can be observed that the best performing method is the code detector, followed by the Curly Bracket baseline.

X. DISCUSSION

The first point to discuss in this section is how Stack Overflow users use the code tag. As we presented in Table III, the manual annotators found that many HTML blocks tagged as code are in fact natural language. Observing the original posts in Stack Overflow, it is clear why users utilise the code tag to format, apart from code snippets or code portions, element such as stack traces or libraries names. The code tag, highlights these elements differently that bold, italics or emphasis, and makes locating this information easier while reading the post. Moreover, the code tag helps to reduce, in many cases, the ambiguity of elements, i.e. some terms used in Java (or in other markup or programming language) are shared in English too. For instance, in example *B* in Figure 3, the term "*Map*" in "*Then store the value back in the* `Map`", refers to a Java term other than a cartography one. However, we believe that this excessive use of the code tag can introduce noise and reduce the performance of tools based on Stack Overflow data.

Despite these outcomes, we do not think that it is necessary to increase the number of tags in Stack Overflow. Even With the current number, users in some cases forget to use them. However, our results indicate that researchers that utilise code snippets posted on Stack Overflow should not trust the code tag completely. In fact, it is necessary to do a pre-analysis or pre-processing to verify that an HTML block tagged as code is in fact meaningful code.

Some readers may consider the annotation process as faulty, because code blocks surrounded by natural language should still be considered as code. In fact, the most difficult part of the annotation process was to determine under what circumstances a term should be considered as code. We adopted the idea that many of these code blocks should be considered as "foreign words" or "foreign phrases". They are not standard English vocabulary, however, they are used to expressing other ideas, in some cases, difficult to express in English. In the future, we intend to introduce a mixed tag to indicate that previously validated and tagged code elements are mixed with natural language elements. This mixed tag may be helpful in indexing tools or systems that create a link between code snippets and natural language.

The second aspect to discuss is the performance of the proposed code detector. The code detector achieved good performance, especially if we consider that it was only trained on sentences in English and lines of Java programs, without any exposure to stack traces, logs or other type of languages such as HTML and XML. For instance, the code detector classified as code the JSON presented in example *A* in Figure 4. In addition, despite not having seen code in Python, the code detector classified correctly as code the snippet in example *B* in Figure 4. The reason for this generalisation ability may be related to the use of word embeddings.

We observed, that in some cases, HTML blocks that describe uniquely names of variables, methods or libraries are classified as natural language by the code detector. Examples of these cases are HTML blocks such as "*TreeMap*", "*onCreate*" or "*crawler4j*". As it can be seen, many of these instances belong to a concatenation of English words. They are classified

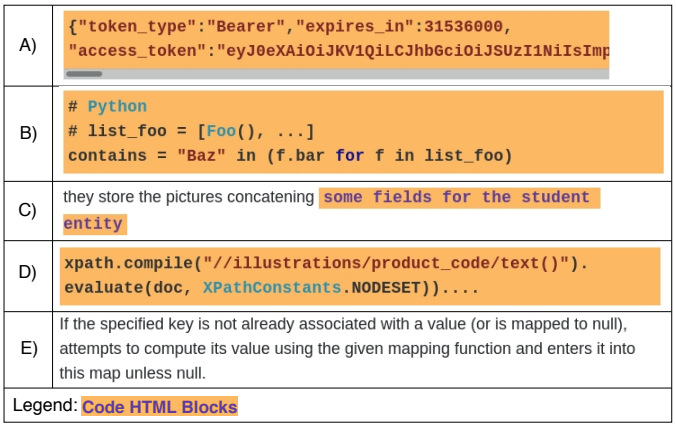

Figure 4. Examples of posts from Stack Overflow that were analysed by the code detector. The CSS used in Stack Overflow has been modified to visually enhance the cases. In some cases, like in *A*, *B* and *D*, the code tag is inside a pre-formatted tag (<*pre*>) to give the code a pretty-print style.

as English because FastText creates word embeddings of n-grams, increasing its predictive power for previously unseen words. In some other cases, the code detector seems to have learned that certain words or phrases are indicative of code, such as HTML blocks that contain "*valueOf*" or "*path*".

In some cases, it is hard to understand why the code detector failed to classify an HTML block correctly. In example *C* in Figure 4, the HTML block "*some fields for the student entity*" was tagged, for unknown reasons, as code by Stack Overflow users and by the code detector, despite its natural language essence. It should be remembered that we do not provide the tag assigned by Stack Overflow users to the code detector, therefore this case should have been classified as natural language. Another example of erroneous classification is presented in example *D* in Figure 4, where an HTML block originally tagged as code was classified as natural language. Another case of misclassification is presented in example *E* in Figure 4, where a natural language block was classified as code. The reason might be the number of Java terms used, such as *map* and *null*, or which can be found in a program, such as *function* or *mapping*.

Strange misclassification cases most commonly concern HTML blocks tagged by users as code, however they also concern natural language HTML blocks less frequently. Some curios examples are the following HTML blocks, "*Edit:*" and "*—cheerio atul*", which were considered as code by the code detector. These misclassified cases, although tough to explain, might be related to the type of English texts that were used during the training. It should be remembered that we made use of Wikipedia articles, therefore, it might be rare to find informal expressions like "*cheerio*". This might be possible to correct if we add tweets as a source of English text in the training data.

As we observed previously, applying the code detector as a pre-processing tool comes at a certain risk. In some cases, the code detector may misclassify code blocks as natural language and vice-versa. Thus, to use the code detector in applications

Table XI
THE MOST FREQUENT CASES, WHERE THE CODE DETECTOR PREDICTION DOES NOT MATCH WITH THE TAG USED BY STACK OVERFLOW USERS.

Code tagged block		Natural language tagged block	
Text	**Frequency**	**Text**	**Frequency**
`A`	12,845	}	39,886
`i`	12,010	Thanks	21,213
`HashMap`	10,284	EDIT:	20,028
`this`	9,683	Use	12,064
`for`	9,595	Try this:	6,620

where this tendency for misclassification of natural language text portions is considered risky, the detector can only be used on HTML lines tagged as code. Alternatively, the code detector could be used as a method to flag HTML blocks in Stack Overflow that may need to be checked or visually improved by the community.

Despite the classification errors, using the code detector offers a number of benefits. We can discover HTML blocks that are in fact errors, stack traces, logs or input/output texts. Moreover, we can identify HTML blocks tagged as code that in their context are natural language. This simplifies using Stack Overflow, if the target is to extract code snippets only, for example.

It is interesting how a baseline based on detecting curly brackets classifies natural language blocks better than code ones. The main reason is that curly brackets are not frequently used in natural language context, therefore they are a good anti-pattern. However, not every code block in Stack Overflow contains a curly bracket, as in some cases code snippets do not contain block statements.

In the future, an improved version may increase the benefits brought by the code detector. This improved version can be trained, not only on other programming languages, like Python or C++, but instances of other kinds of text like commands and trace errors.

Finally, it should be noted that increasing the variety of training instances can sometimes be expensive and/or difficult. We used resources that contain instances of one class only, because manually annotating resources, such as Stack Overflow, can be tedious and time-consuming, as the number of instances necessary for training a classifier can be quite large. Moreover, by increasing the training data, we risk increasing the model size, which can limit reuse and integration in other tools.

A. Frequency Analysis

The code detector has been applied on 25.8M HTML blocks in Java-related Stack Overflow posts. In addition to the manual evaluation, we analysed the most recurrent cases in which the Stack Overflow tag did not match the code detector prediction. We present the most frequent mismatches in Table XI.

The misclassification of expressions such as "*Thanks*" and "*Try this:*", must be because in our training source for English, i.e. Wikipedia, the occurrences of these expressions are very rare. It is interesting that the most misclassified natural language HTML block corresponds to a closing curly bracket.

These blocks may concern cases where Stack Overflow users closed the code tag before the end of the code snippet. Regarding the conflicting cases of HTML blocks tagged as code, many of them are frequent words in English.

XI. Conclusion and future work

With the expansion of the Internet and the use of social media, question and answer (Q&A) websites improved the exchange of knowledge. Nowadays, we can find multiple and different Q&A websites focusing on general knowledge or on specialised *savoir faire*. Stack Overflow is one of the most recognised and widely used Q&A websites. It specialises on software issues and has become very popular in the communities of software developers. Moreover, due to its popularity, Stack Overflow has become the basis of diverse research works that either use its data to create other tools or analyse human behaviour on the website.

However, despite the frequent use of Stack Overflow data for research, its quality in terms of structure has not been questioned to the best of our knowledge. Thus, we focussed on exploring how Stack Overflow users utilise the "code" tag and how a code detector could make data more reliable.

To achieve our goal, we presented in this article two interesting elements. The first one consists on a code detector that aims at determining whether a block of text, even one character long, is code or natural language. The code detector was developed using FastText, a library that allows creating classifiers based on neural networks and word embeddings. Our code detector was trained using a corpus of English sentences and code lines from Java.

The second element is an annotated corpus of $1,000$ HTML blocks obtained from Java-related posts published on Stack Overflow. The annotated corpus was used not only to evaluate the performance of the code detector on real data, but also to explore how people use the code tag. By analysing the uses of the code tag in Stack Overflow, we also evaluated how relevant the use of our code detector is for pre-processing Stack Overflow data.

We found that although the code tag was originally designed to highlight code snippets, users also apply it to emphasise on aspects like library and variables names, commands, error traces and output texts. Moreover, as the use of the code tag is devolved on the users, in some cases it is omitted. These issues can affect the performance of tools that depend on the extraction of text tagged as code.

The code detector was experimentally shown to achieve an F-score of 0.866 regarding the classification of code and F-score of 0.839 for the classification of natural language. It can detect correctly multiple types of code, such as programming languages or markup, while it also identifies technical aspects, such as stack traces or logs.

The proposed code detector can be used in multiple applications. For example, it could be used by Stack Overflow to find exactly which text parts contain code and alert users to format them correctly. Another possible application in Stack Overflow is flagging badly formatted posts to be checked by other users. In general, the code detector can be used as a pre-processing method, to improve the performance of tools such as identifiers of duplicate questions and extractors of code snippets or to match code with natural language.

In the future, we expect to improve the code detector by including other programming languages such as Python and C++ in the training data, but also elements as stack traces or logs. We shall explore the use of other data sources to increase the variety of natural language in the training set. This last point will allow improving the detection accuracy regarding natural language blocks and in consequence to ameliorate the general performance of the code detector. Finally, we plan to manually annotate more data from Stack Overflow in order to give a clearer idea of how users apply the code tag.

References

[1] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web? (NIER track)," in *33rd International Conference on Software Engineering (ICSE)*. Waikiki, Honolulu, Hawaii: IEEE, 2011, pp. 804–807.

[2] T. D. LaToza and A. van der Hoek, "Crowdsourcing in Software Engineering: Models, Motivations, and Challenges," *IEEE Software*, vol. 33, no. 1, pp. 74–80, Jan. 2016.

[3] R. Abdalkareem, E. Shihab, and J. Rilling, "What do developers use the crowd for? a study using stack overflow," *IEEE Software*, vol. 34, no. 2, pp. 53–60, 2017.

[4] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 349–359.

[5] Y. Mizobuchi and K. Takayama, "Two improvements to detect duplicates in Stack Overflow," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 563–564.

[6] D. Yang, P. Martins, V. Saini, and C. Lopes, "Stack overflow in github: any snippets there?" in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 280–290.

[7] T. Ahmed and A. Srivastava, "Understanding and evaluating the behavior of technical users. A study of developer interaction at StackOverflow," *Human-centric Computing and Information Sciences*, vol. 7, no. 1, p. 8, 2017.

[8] L. Ponzanelli, A. Mocci, A. Bacchelli, M. Lanza, and D. Fullerton, "Improving Low Quality Stack Overflow Post Detection," in *2014 IEEE International Conference on Software Maintenance and Evolution*. Victoria, British Columbia, Canada: IEEE Computer Society, 2014, pp. 541–544.

[9] T. Merten, B. Mager, S. Bürsner, and B. Paech, "Classifying unstructured data into natural language text and technical information," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. Hyderabad, India: ACM, 2014, pp. 300–303.

[10] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *34th International Conference on Software Engineering (ICSE)*, M. Glinz, G. Murphy, and M. Pezzè, Eds. Zurich, Switzerland: IEEE, 2012, pp. 375–385.

[11] L. Cerulo, M. Di Penta, A. Bacchelli, M. Ceccarelli, and G. Canfora, "Irish: A Hidden Markov Model to detect coded information islands in free text," *Science of Computer Programming*, vol. 105, no. Supplement C, pp. 26 – 43, 2014.

[12] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of the 2013 International Conference on Software Engineering*, D. Notkin, B. H. C. Cheng, and K. Poh, Eds. San Francisco, CA, USA: IEEE Press, 2013, pp. 832–841.

[13] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of Tricks for Efficient Text Classification," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, vol. 2, Valencia, Spain, 2017, pp. 427–431.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[15] Y. Chen, B. Perozzi, R. Al-Rfou, and S. Skiena, "The Expressive Power of Word Embeddings," *CoRR*, vol. abs/1301.3226, 2013. [Online]. Available: http://arxiv.org/abs/1301.3226

[16] J. Guo, W. Che, H. Wang, and T. Liu, "Revisiting Embedding Features for Simple Semi-supervised Learning." in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 110–120.

[17] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts, "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, 2013, pp. 1631–1642. [Online]. Available: 978-1-937284-97-8

[18] W. Y. Zou, R. Socher, D. Cer, and C. D. Manning, "Bilingual Word Embeddings for Phrase-Based Machine Translation," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, 2013, pp. 1393–1398.

[19] M. Baroni, G. Dinu, and G. Kruszewski, "Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Baltimore, Maryland: Association for Computational Linguistics, Jun. 2014, pp. 238–247.

[20] S. C. Deerwester, S. T. Dumais, G. W. Furnas, R. A. Harshman, T. K. Landauer, K. E. Lochbaum, and L. A. Streeter, "Computer information retrieval using latent semantic structure," Patent US4 839 853 A, 1988.

[21] S. C. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. A. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, p. 391, 1990.

[22] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global Vectors for Word Representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*. Doha, Qatar: Association for Computational Linguistics (ACL), 2014, pp. 1532–1543.

[23] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[24] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching Word Vectors with Subword Information," *Transactions of the Association of Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[25] V. Zolotov and D. Kung, "Analysis and Optimization of fastText Linear Text Classifier," *arXiv preprint arXiv:1702.05531*, 2017.

[26] J. Močkus, V. Tiešis, and A. Žilinskas, "The application of Bayesian methods for seeking the extremum," in *Towards Global Optimisation*, G. P. Szegö and L. C. W. Dixon, Eds. North-Holland, 1978, vol. 2, pp. 117–128.

[27] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds., vol. 2. Curran Associates Inc., pp. 2951–2959.

[28] D. Goldhahn, T. Eckart, and U. Quasthoff, "Building Large Monolingual Dictionaries at the Leipzig Corpora Collection: From 100 to 200 Languages," in *Proceedings of 8th Language Resources and Evaluation Conference (LREC'12)*, Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Asunción Moreno, Jan Odijk, and Stelios Piperidis, Eds., Istanbul, Turkey, 2012, pp. 759–765.

[29] M. Allamanis and C. Sutton, "Mining Source Code Repositories at Massive Scale using Language Modeling," in *The 10th Working Conference on Mining Software Repositories*, T. Zimmermann, M. Di Penta, and S. Kim, Eds. San Francisco, CA, USA: IEEE, 2013, pp. 207–216.

[30] J. R. Landis and G. G. Koch, "The Measurement of Observer Agreement for Categorical Data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977.

Neurocomputing 315 (2018) 496–511

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

Detection of spam-posting accounts on Twitter

Isa Inuwa-Dutse*, Mark Liptrott, Ioannis Korkontzelos

Department of Computer Science, Edge Hill University, Ormskirk, Lancashire, UK

ARTICLE INFO

Article history:
Received 6 March 2018
Revised 13 June 2018
Accepted 27 July 2018
Available online 8 August 2018

Communicated by Dr Fenza Giuseppe

Keywords:
Social network
Twitter
Spam
Social media
Twitter microblog
Spam detection

ABSTRACT

Online Social Media platforms, such as Facebook and Twitter, enable all users, independently of their characteristics, to freely generate and consume huge amounts of data. While this data is being exploited by individuals and organisations to gain competitive advantage, a substantial amount of data is being generated by spam or fake users. One in every 200 social media messages and one in every 21 tweets is estimated to be spam. The rapid growth in the volume of global spam is expected to compromise research works that use social media data, thereby questioning data credibility. Motivated by the need to identify and filter out spam contents in social media data, this study presents a novel approach for distinguishing spam vs. non-spam social media posts and offers more insight into the behaviour of spam users on Twitter. The approach proposes an optimised set of features independent of historical tweets, which are only available for a short time on Twitter. We take into account features related to the *users* of Twitter, their *accounts* and their *pairwise engagement* with each other. We experimentally demonstrate the efficacy and robustness of our approach and compare it to a typical feature set for spam detection in the literature, achieving a significant improvement on performance. In contrast to prior research findings, we observe that an average automated spam account posted at least 12 tweets per day at well defined periods. Our method is suitable for real-time deployment in a social media data collection pipeline as an initial preprocessing strategy to improve the validity of research data.

1. Introduction

Online social media is one of the defining phenomena in this technology-driven era. Platforms, such as Facebook and Twitter, are instrumental in enabling global connectivity. 2.46 billion users are estimated to be now connected and by the year 2020 one-third of the global population will be connected [1]. Users of these platforms freely generate and consume information leading to unprecedented amounts of data. Several domains have already recognised the crucial role of social media analysis in improving productivity and gaining competitive advantage. Information derived from social media has been utilised in health-care to support effective service delivery [2,3], in sport to engage with fans [4], in the entertainment industry to complement intuition and experience in business decisions [5] and in politics to track election processes, promote wider engagement with supporters [6] and predict poll outcomes. However, alongside the benefits, the rapid increase in social media spam contents questions the credibility of research based on analysing this data. A report by Nexgate [7] estimates that on average one spam post occurs in every 200 social media posts and a more recent study reports that approximately 15% of active Twitter users are automated bots [8]. The growing volume of spam posts and the use of autonomous accounts (social bots) to generate posts raises many concerns about the credibility and representativeness of the data for research.

In this study, we focus on Twitter and we propose a novel, effective approach to detect and filter unwanted tweets, complementing earlier approaches in this direction [8–11]. Previous studies rely on historical features of tweets that are often unavailable on Twitter after a short period of time, hence not suitable for real-time use. Our approach utilises an optimised set of readily available features, independent of historical textual features on Twitter. The employed features are categorised as related to the *Twitter account*, the *user* or referring to the *pairwise engagement* between users. A number of machine learning models have been trained. Recursive feature elimination has been employed in order to ascertain the robustness and the discriminative power of each feature. In comparison to an earlier study [9], the proposed features exhibit stronger discriminative power with more consistent performance across the different learning models. Spam posting users exhibit some evasive tactics, such as posting on average of 4 tweets

* Corresponding author.
E-mail addresses: dutsei@edgehill.ac.uk (I. Inuwa-Dutse), Mark.Liptrott@edgehill.ac.uk (M. Liptrott), Yannis.Korkontzelos@edgehill.ac.uk (I. Korkontzelos).

https://doi.org/10.1016/j.neucom.2018.07.044

per day, and tricks to balance the follower–followee relationship [9]. Our analysis shows that an average automated spam posting account posts at least 12 tweets per day within well-defined activity periods. The activity pattern resembles the staircase function exhibiting surges of intermittent activities. Our study contributes (a) a new set of lightweight features suitable for real-time detection of spammers on Twitter and (b) an additional dataset source[1] offering an insight into the behaviour of spam users on Twitter to support further studies.

The paper is structured as follows: Section 2 offers a high-level overview of spamming on social media and Section 3 presents a survey of the relevant literature. The Dataset and the feature selection process are presented in Sections 4 and 5, respectively. Section 6 presents the experimental results and Section 7 discusses relevant findings. Finally, Section 8 concludes this work and suggests some directions for further future research.

2. Online social media spamming

Online spamming activities come in different forms such as malware dissemination, posting of commercial URLs, fake news or abusive contents, automated generation of large volume of contents [8] and following or mentioning random users [9]. Another form of online spamming is the growing use of machine learning models to generate fake reviews on products and services [12] and the use of social bots to influence the opinion of users [13]. The volume of global spam is growing tremendously, with an estimated rate of 355% in 2013 [7]. Specifically on Twitter, for every 21 tweets, one is spam and about 15% of active users are autonomous agents, i.e. social bots [8]. The growth rate of spam volume can be attributed to the lack of physical contact between the communicating parties. This makes it difficult to ascertain the actual identity of the user and the legitimacy of the contents being posted. Evidently, utilising data directly from social media platforms without effective filtering may mislead the analysis and lead to wrong conclusions due to unrepresentative data. Numerous sophisticated approaches have been developed in this direction and are reviewed in Section 3. However, at the same time, spammers evolve rapidly to evade detection systems. As a result, some approaches may be rendered obsolete and ineffective in responding to the new tricks introduced by the spammers.

3. Literature review

Spam entails any form of activity that causes harm or disrupts other online users. The increasing amount of spam tweets can be attributed to humans' inclination to spread misleading information, even if such information originated from unreliable sources, such as a social bot account. Recently, Vosoughi et al. [14] discover that both genuine and false news spread at equal rate. False news on Twitter spread rapidly. Social bots are deployed to accelerate the process and human users further amplify the content. To detect spam tweets, numerous detection systems have been proposed, using various techniques that are reviewed in this section.

The pioneering work of Wang [15] on spam detection utilised directed graph models to analyse *follower friend* relationships on Twitter and define feature sets for effective spam detection. In broad context, approaches for spam detection can usually be classified under the following categories: social graph analysis [16–18], text analysis and activity patterns [19], analysis of user profile meta-data, URL usage and the effect of URL obfuscation [20–22], analysis of interaction behaviour [8,9,23], and URL blacklisting and its effect [24].

Recently, in November 2017, Twitter increased the maximum number of characters in a tweet for most users, after just over a month of testing [25]. Up to that time, users were limited to 140 characters per tweet thereby making URLs and URL shortening services widespread. Thomas et al. [20] and Lee and Kim [21] analysed streams of URLs used by spam users and studied how spammers exploit URLs obfuscation to redirect users to malicious sites. Grier et al. [24] analysed a large number of distinct URLs pointing to blacklisted sites due to their involvement in scam, phishing and malware activities. Although the approach is effective, it is often slow and fails to detect URLs that point to malicious sites but have not been blacklisted previously. Gao et al. [19] also studied URL usage on Facebook to detect spamming activity and observed that this form of spamming is mostly associated with compromised accounts rather than accounts created solely for spam activity. Benevenuto et al. [22] studied the statistical properties of user accounts and how URL shortening services affect spam detection mechanisms. However, the universal use of URLs and URL shortening by the vast majority of Twitter users makes it difficult to directly identify potentially nefarious links on a large scale. In general, the use of URLs relies on historical information, limiting the possibilities for real-time detection.

Danezis and Mittal [18] utilised a social network model to infer legitimate user accounts that are being controlled by an adversary. Lee et al. [9] created *social honeypot accounts* mimicking naive Twitter users to entice spam posting users. Users who fall prey by engaging with these accounts are assumed to be in violation of usage policy. Users identified using this method were analysed to distinguish different user types focusing on link payloads and features that can capture the dynamics of *follower-following* networks of users. Varol et al. [8] employed many features related to users, content and the network to develop a system for social bot account detection.

Sedhai and Sun [26] analyse the distribution of spammy words, i.e. terms with higher probability of occurrence in spam than in non-spam tweets, in tweets to detect spam. Chen et al. [27] provides an in depth analysis of deceptive words used by spammers on Twitter. The work of Chen et al. [28] is motivated by *Twitter Spam Drift*, i.e. the property of statistical features of spam tweets to change over time. *Twitter Spam Drift* is caused because spammers continuously adopt and abolish various evasive tricks. Features related to this phenomenon were utilised in training machine learning classifiers. Li and Liu [29] analysed how the effect of unbalance datasets can be mitigated in detection tasks.

Standard machine learning methods are sometimes considered as inadequate in capturing the variability of spamming behaviour. Wu et al. [30] utilised a deep learning technique based on Word2Vec [31] to capture the variation of spam-related challenges. While it is essential to allow detection models to continuously learn features strong enough to distinguish spam from non-spam, methods that solely rely on textual information are be inadequate to draw the distinction between a habitual spam posting account and a non-spam posting account. Hand-crafted features related to the account and the user need to be considered. In this study, a set of hand-crafted features are leveraged in tandem with features learn by deep neural networks. Features studied by humans and encoded to classifiers can achieve better performance and low false positive rates [32].

The use of a large number of features introduces extra overheads to the detection system, some of which may be unavailable for real-time use. Subrahmanian et al. [13] offer insights into techniques utilised in identifying *influence bots*, i.e. autonomous entities determined to influence discussions on Twitter. Influence bots comprise a category of social bot accounts that seek to assert in-

[1] We are not able to provide the fully-hydrated tweets, i.e. accompanied with full details, due to sharing restrictions but we provide the relevant IDs and computed features.

Table 1
Summary of datasets: The size of original data refers to data collected before some preliminary preprocessing steps such as discarding non-English tweets and duplicates.

Dataset	Size of original data	Size of preprocessed datasets	Class	Collection	Verified?
Honeypot	19,297	19,276	Legitimate	Automated	No
Honeypot	23,869	22,223	Polluter	Automated	No
$SPD_{automated}$	10,318	8515	Legitimate	Automated	Yes
$SPD_{automated}$	25,568	9831	Spam	Automated	No
SPD_{manual}	2000	1300	Legitimate	Manual	Yes
SPD_{manual}	2000	700	Spam	Manual	No

fluence on topical or new discussions thereby generating unrepresentative or fake data.

The surveyed studies on spam detection largely rely on either historical tweets of a user to extract features which contribute to an extra overhead for the detection system [33] or limited features learnt by unsupervised techniques. Our proposed approach relies on readily available features in real-time for better performance and wider applicability.

4. Dataset

This section discusses the collection and validation of datasets utilised in our experiments: *Honeypot*, the automatically annotated *spam-posts detection* dataset *($SPD_{automated}$)* and the manually annotated *spam-posts detection* dataset *(SPD_{manual})*. Table 1 presents statistics about these datasets. The *Honeypot* dataset [9] is publicly available and useful for studying spam activity on Twitter. It was utilised both as a dataset per se and for collecting the *SPD* datasets using keywords. Keywords play a crucial role in retrieving specific documents from large corpora [34] and this study speculates that keywords extracted from the *Honeypot* dataset can be used in retrieving large quantities of similar data.

In Table 1, *Legitimate* refers to data from genuine users whose accounts have been verified by Twitter. A verified account is certified by Twitter to be genuine and such information is available from the meta-data section of the tweets. In contrast to the randomised approaches utilised in [9] to ascertain user legitimacy on Twitter, we used accounts verified by Twitter in building the legitimate part of the *SPD* datasets to avoid the potential risk of a high false positive rate.

SPD_{manual} is a manually annotated dataset created to supplement evaluation. It contains tweets randomly selected from the full set of tweets that have been downloaded between February, 2017 and June, 2017 via the traditional Twitter API[2] using relevant keywords as query terms. It consists of 1700 tweets of *legitimate users* and 300 tweets of *spam users*.

For our analysis, we took a sample of 2000 accounts for manual annotation resulting in the disproportionate ratio of 700:1300 (spam:non-spam). Unbalanced datasets often affect the performance of detection systems [29], including ours. To mitigate that, we applied the SMOTE resampling technique [36] to balance the data by upscaling the minority class. Additionally, we further query the accounts of spam users to retrieve more spam tweets. This technique was used before training Word2Vec. The cost and labour intensiveness of annotations as well as the general unbalanced ratio of spam/non-spam on Twitter contributes to this disproportionate ratio in SPD_{manual}.

$SPD_{automated}$ contains tweets that have been collected between February, 2017 and June, 2017, and have been automatically marked as legitimate or spam. Tweets posted by users whose accounts have been verified as *Legitimate* by Twitter were marked as legitimate. Tweets that contained at least two of the most representative keywords in the *Polluter* part of the *HoneyPot* dataset were marked as spam.

Keywords, both for querying Twitter and validating spam, are obtained by applying Latent Semantic Analysis (LSA) on the *Honeypot* dataset [37]. LSA is useful in capturing the semantics and relevance of terms to a document [38]. Prevalent keywords from LSA concepts include *free, new, lots, win, follow, trade, good, great, make, create, twitter, followers, check,gain, buy, account, get, making, online, want*. See Table A.2 for full list. A block diagram of the collection and validation process is shown in Fig. 1. Table 2 shows some example tweets that satisfy this criterion.

4.1. Validation of $SPD_{automated}$

Labelling data in $SPD_{automated}$ as spam is based on the hypothesis that spam users are more likely to use at least two of the terms obtained via LSA on the part of the *Honeypot* dataset that is known to be spam [9]. To validate this, we compute and compare in the legitimate and the spam part of $SPD_{automated}$:

- the distribution of the co-occurring keywords
- lexical richness and lexical density
- the distributions of user mentions and URLs

Table 4 shows the results.

4.1.1. Distribution of co-occurring keywords

Spammers heavily leverage certain deceptive words to lure users [27]. Words normally preceded by *free, follow* and *gain* have high probability of occurrence in spam tweets [26].

In this study, we aim to capture important *n-grams* used by spammers by leveraging a public spam dataset. To select the best *n-grams* as well as the number of co-occurring terms sufficient for identifying spam tweets, we first apply Latent Semantic Analysis (LSA) as a decomposition technique to discover the most representative keywords in the corpus and compared with a list of known spammy words[3]. Based on the list of spammy words, we compute the relative frequencies of various spammy *n-grams* (bigrams, trigrams and four-grams) in the corpus. Table 3 shows the relative frequencies of spammy *n-grams* in various datasets. Fig. A.3 shows an example of common *spammy n-grams*. In Table 3 we observe that bigrams have higher relative frequencies in spam datasets and the individual terms that they consist of occur in the spammy list, in Table A.1. Accordingly, a tweet is highly probable to be a spam if it contains at least bigrams of spammy words and has low lexical richness.

We observe in Table 4 that only 1.05% of the tweets in the legitimate part of $SPD_{automated}$ contain two or more keywords, extracted using *LSA* from the *Polluter* part of the *HoneyPot* dataset. In contrast, more than 89.5% of the tweets in the spam part of $SPD_{automated}$ contain keyword pairs. This distribution is a strong indicator of a probable spam tweet and also minimises the risk of

[2] The dedicated channel provided by Twitter to enable access to public datasets [35].

[3] Compiled by Sedhai and Sun [26] and shown in Table A.1, in the Appendix.

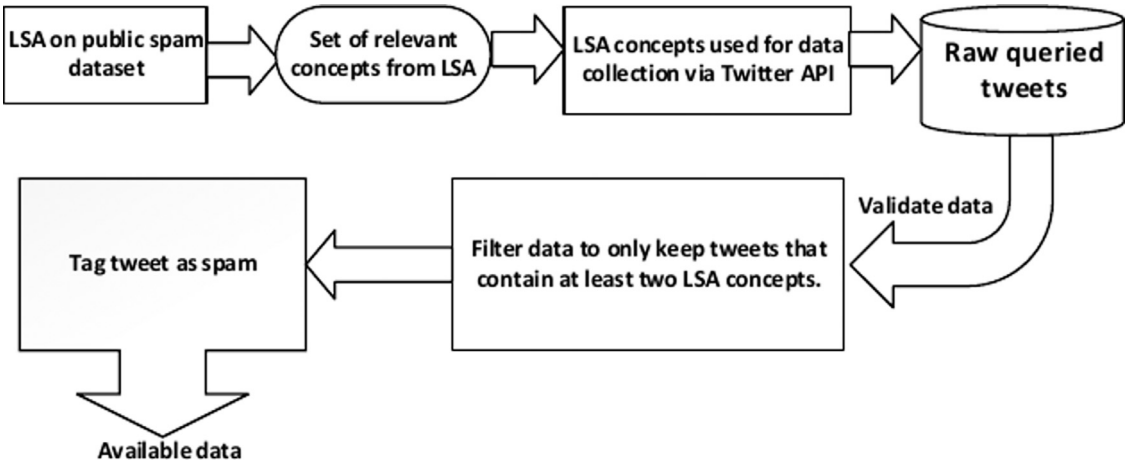

Fig. 1. Collection and validation of the spam part of the $SPD_{automated}$ dataset from Twitter.

Table 2
Examples of collocational bigrams from the spam part of $SPD_{automated}$. Keywords returned by *LSA* on the *Honeypot* dataset are shown in bold face. Actual users mention were replaced with the '*user*' placeholder to preserve anonymity.

Id	Tweet
T1	RT @user: Retweet to **win** up to 121+ **followers** must be **following** me 💍
T2	Retweet this for 81+ **free follows** ⚜ ♻
T3	Retweet for 125 **free follows** 🌈 '\n'Retweet and Fav 💕 this if you have my post notifications on! 🐠 For 125 **free followers** 💘😱
T4	Watch and like this video for **free** 80 **followers** 💕 url
T5	Retweet to win up to 130+ **free followers** 🔗 ' '@user
T6	RT @user: Retweet this to **gain followers** faster 😍 🔮 🔗
T7	**Follow everyone** who FAV this 🍎
T8	@user @user @user @user @user @user **follow everyone** who likes this 🌸 #SolarEclipse2017 \

Table 3
Relative frequencies of n-grams that consist of some spammy words in the dataset; in particular the n-grams B1, B2, T1, T2, F1 and F2, that are shown in bold face in table A.1.

Dataset	N-gram proportions		
	Bigrams	Trigrams	Four-grams
$Honeypot_{spam}$	B1: 1.26×10^{-3}	T1: 1.83×10^{-3}	F1: 4.40×10^{-4}
	B2: 3.51×10^{-4}	T2: 0.0	F2: 0.0
$Honeypot_{non\text{-}spam}$	B1: 4.07×10^{-4}	T1: 2.50×10^{-4}	F1: : 0.0
	B2: 3.3×10^{-3}	T2: 0.0	F2: 0.0
SPD_{spam}	B1: 6.04×10^{-2}	T1: 1.05×10^{-2}	F1: 6.42×10^{-3}
	B2: 2.21×10^{-2}	T2: 2.87×10^{-2}	F2: 4.74×10^{-3}
$SPD_{non\text{-}spam}$	B1: 2.34×10^{-7}	T1: 0.0	F1: 0.0
	B2: 0.0	T2: 0.0	F2: 0.0

labelling legitimate users as spammers. Table 2 shows examples of frequent co-occurring keywords sampled from $SPD_{automated}$.

4.1.2. Lexical richness and density

In quantitative linguistics, lexical richness measures the wealth of vocabulary in a given text [39]. Basic measures, such as *Type Token Ratio (TTR)* and *Mean Word Frequency*, are utilised to assess the quality of lexicons in spam and non-spam corpora. We hypothesise that spam users will have low lexical diversity and sophistication compared to genuine users. Legitimate users are expected to use rich and diverse lexicons in tweets depending on the discussion topic. In contrast, spam users focus on specific targets such as promoting a certain product or marketing to increase the number of their followers. Users engaging with this behaviour are highly likely to recycle specific sets of similar words.

Type-token ratio (TTR) measures the richness of a lexicon in a document [40]. It is useful in understanding how distinct words are utilised in the legitimate and the spam part of $SPD_{automated}$. For a dataset *D*, *TTR* can be computed as follows:

$$TTR = \frac{\text{unique tokens in D}}{\text{tokens in D}} \quad (1)$$

Table 4
Percentage distributions of relevant metrics computed in the two parts of $SPD_{automated}$, i.e. legitimate and spam.

Data	% name similarity	% digits in names	% containing spam bigrams	% LexRich unfiltered	% LexRich filtered
Legitimate	82.59	14.07	1.05	97.43	86.74
Spam	26.27	88.84	89.51	90.94	49.46

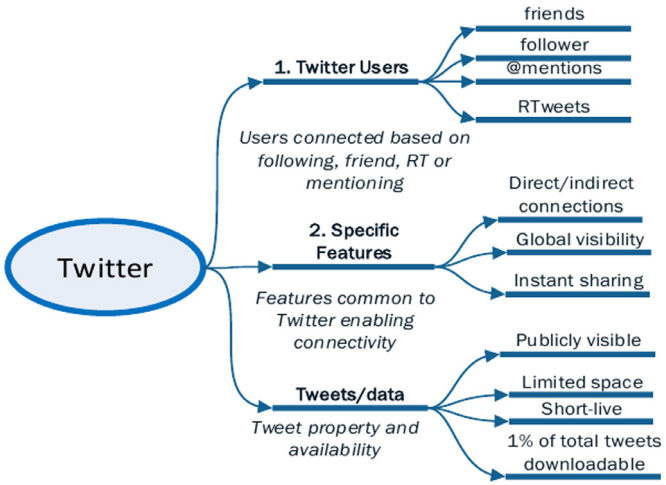

Fig. 2. An overview of Twitter: three different categories of attributes that support global interconnectivity of users are shown. The features utilised in this study are derived from these categories directly or indirectly.

We also compute *lexical density (LD)* [40] as follows:

$$LD = \frac{\text{words in D excluding stopwords}}{\text{tokens in D}} \quad (2)$$

Table 4 shows the result of computing these metrics in both datasets.

However, lexical richness is insufficient for the purpose, because it does not capture term semantics [41]. Some spammy words are not exclusive to spammers,as non-spam users may also use them in different context. To capture the semantics of words in spam and non-spam datasets, we experimented with word embeddings as classification features. Table 10 shows evaluation results of various classifiers trained on word embedding features and features without word embeddings and tested on $SPD_{automated}$. Table A.3 summarises the datasets used in training our Word2Vec model [31].

4.1.3. Users mention

Random mentioning of users [42] is a common tactic employed by spammers in an effort to expand the visibility or their network of followers [9,23]. In Table 4, lexical richness, i.e. *%LexRich (unfiltered)*, in the spam set is marginally higher than expected. Noting the high proportion of user mentions in spam data, lexical richness *(% LexRich (filtered)) or lexical density* is computed without considering the *user mentions* and *URLs* in both datasets. The computation in the spam dataset led to a very low score suggesting that the large number of *user mentions* and *URLs* are responsible for the relatively high TTR score in the spam dataset.

TTR in the legitimate dataset is not affected by filtering out *user mentions* and *URLs* and is indicative of the richness and diversity of the lexicon used by genuine users. The low TTR score in the spam dataset indicates that the same words are being used repetitively usually not really matching the discussion topic. Table 4 also shows metrics related to naming conventions by computing the degree of similarity between the username and the screenname of each user and the proportion of digits in their names. This topic is discussed further in Section 5.

5. Features

The Twitter platform facilitates global connections and interactions of diverse users [43]. Fig. 2 presents an overview of the platform and its relevant attributes that enable users to connect and form the basis of our feature extraction.

5.1. Accessibility, dynamism and categorisations of features

Tweets are available only for a short time, approximately seven days, after being posted. Many real time spam detection systems that rely on historical features from past tweets, are affected by this constraint and may be practically less effective. Readily available, dynamic features offer an enhanced opportunity to distinguish spam from non-spam tweets in real-time. To leverage this potential, features are categorised as follows:

- *User Profile Features (UPF)* include information about the user, such as their *user name, screen name, location* and *description*
- *Account Information Features (AIF)* consist of information such as *account creation time (account age)* and *account verification flag (verified or not verified)*
- *Pairwise engagement features* subcategorised into:
  - *Engage-with Features (EwF)* include features that describe user activities on Twitter and users can influence or choose how to alter their values. Features under this group include *friends count, statuses count, tweet type, tweet creation time, tweet creation frequency*, etc.
  - *Engaged-by Features (EbF)* are similar to features in the EwF group . The main difference is that features under this group cannot be influenced by users directly. For instance, a user relies on other users to increase their *favourites count* or to attract more *followers*. Features in this group include *followers count, favourites count, number of retweet (RT)*, etc.

Furthermore, features can be classified as *basic features* or *derived features*. The aforementioned features, i.e. under *UPF, AIF, EwF* and *EbF*, are basic features, whereas derived features are computed using two or more basic features or are based on further analysis, e.g. sentiment analysis or entropy computation on textual data. Features can also be characterised as *static or dynamic*. Static features cannot be changed once the account is created e.g. *user ID* and *account creation time*, whereas dynamic features keep changing depending on the user's level of engagements on Twitter e.g. *statuses count*. All features and their properties are shown in Table 5.

5.2. Feature selection

The early work of Qazvinian et al. [43] categorises features for Twitter-based study into *content-based, network-based* and *Twitter spec ific memes*. These categorisations are further expanded in Fig. 2 and were utilised directly or indirectly in previous related studies [8,15,20–22]. Statistical properties of tweet metadata in relation to user, accounts and URLs usage have been effective in spam detection systems [22]. Based on this categorisation, basic features have been analysed for various Twitter-related tasks. For instance, basic features on Twitter have been analysed to detect simple social bots accounts which lack or repeat basic account information such screen names, profile picture [23]. Retweets, user mentions and low reciprocity of friendship [8] or the dynamism of follower-following networks overtime [9] have also been investigated. The sophistication level of automated accounts on Twitter varies from random following and re-tweeting to advanced social bots that actually generate content. Studies that focus on the detection of such accounts rely on basic features on Twitter to define complex ones [13]. Varol et al. [8] developed a detection framework by leveraging numerous features. Our study takes a similar direction by defining a novel set of additional features derived from the basic ones, that have been discussed and exploited in many studies concerning Twitter. The choice of features for experimentation is informed by insights gained from a series of exploratory analysis to understand the distribution of textual features, the composition of data, and the dynamism of features, such as *statuses count, friends count, followers count, favourites count,*

Table 5
Features proposed and used in the current study, the corresponding groups and definitions. The *VerifiedAccount* feature, f_{22}, was excluded form our final feature set, because in preliminary experiments it was shown to cause overfitting.

Id	Features	Groups	Status	Description/Definition
f_1	*AccountAge*	AIF	Static	Days since account creation to date of collection
f_2	*FollowersCount*	EbF	Dynamic	In user profile meta-data
f_3	*FriendsCount*	EwF	Dynamic	In user profile meta-data
f_4	*StatusesCount*	EwF	Dynamic	In user profile meta-data
f_5	*DigitsCountInName*	UPF	Static	Number of digits in screen name
f_6	*TweetLen*	EwF	Dynamic	Number of characters in tweet
f_7	*UserNameLen*	UPF	Static	Number of characters in user name
f_8	*ScreenNameLen*	UPF	Static	Number of characters in screen name
$f_{9,10,11,12}$	*Metric entropy* for all textual features: tweet, user profile description, user name and screen name, respectively	UPF	Dynamic	To measure randomness in text. $\frac{H(x)}{\lvert x\rvert}$: where $\lvert x\rvert$ is the length of a string, x, and $H(x)$ is the Shannon entropy of text: $\frac{\sum_{i=1,k} p_i \log_2 p_i}{\lvert x\rvert}$
f_{13}	*URIsRatio*	EwF	Dynamic	$\frac{\lvert\text{characters in URLs}\rvert}{\lvert\text{tweet length}\rvert}$
f_{14}	*MentionsRatio*	EwF	Dynamic	$\frac{\lvert\text{characters in user mentions}\rvert}{\lvert\text{tweet length}\rvert}$
f_{15}	*NameSim*	UPF	Static	% proportion of similarity in User name and Screen name
f_{16}	*LexRichWithUU*	EwF	Dynamic	TTR in tweets: $\frac{\lvert\text{token types}\rvert}{\lvert\text{total tokens}\rvert} * 100$
f_{17}	*Friendship*	EwF	Dynamic	$\frac{FriendsCount}{FollowersCount}$
f_{18}	*Followership*	EbF	Dynamic	$\frac{FollowersCount}{FriendsCount}$
f_{19}	*Interestingness*	EbF	Dynamic	$\frac{FavouritesCount}{StatusesCount}$
f_{20}	*Activeness*	EwF	Dynamic	$\frac{StatusesCount}{AccountAge}$
f_{21}	*LexRichWithOutUU*	EwF	Dynamic	$\frac{\lvert\text{lexical worlds}\rvert}{\lvert\text{total number of words}\rvert} * 100$
f_{22}	*VerifiedAccount**	AIF	Static	In tweet metadata
f_{23}	*FavouritesCount*	EwF	Dynamic	In user profile meta-data
f_{24}	*NamesRatio*	UPF	Static	$\frac{\lvert\text{screenname length}\rvert}{\lvert\text{username length}\rvert}$

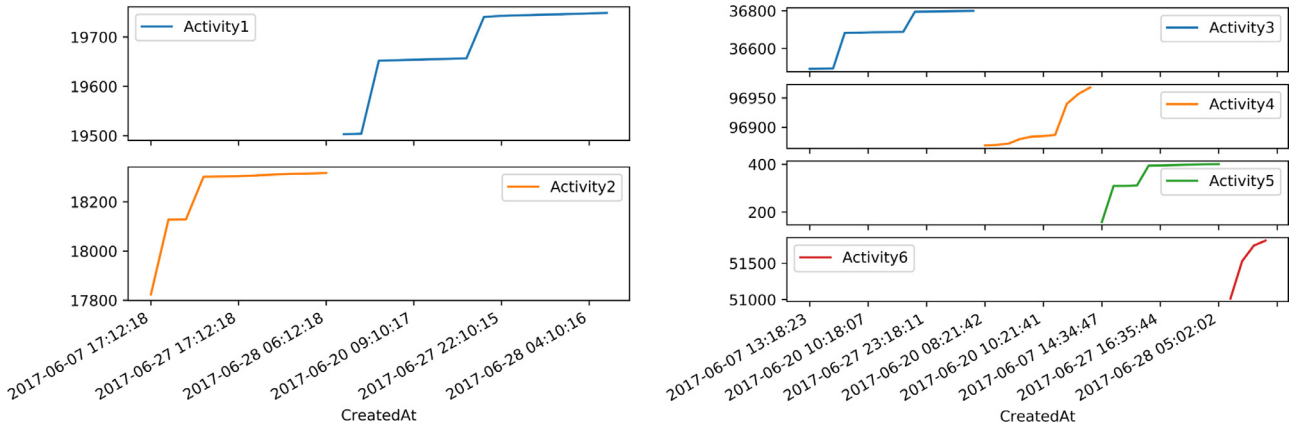

Fig. 3. Example of activity patterns of spam-posting social bot accounts. All sub-figures depict hyperactive automated users that generated very high traffic within a short period. The activity distribution over time for most users resembles the *staircase* function. Some users generate much higher traffic than other, e.g. *Activity4* and *Activity6* represent many times more tweets than *Activity5*.

naming conventions and *tweeting patterns*. Figs. A.1 and A.2 in the Appendix present further exploratory results.

Account age is useful in capturing the frequency of user activity. From our analysis, accounts with very high statuses and friends count but low favourites count and followers count at young age are likely to be automated spam posting accounts. For example, Fig. 3 shows huge amounts of content generated within short period. We utilised these observations in deriving features, such as *Activeness, Interestingness* and *Followership*, as shown in Table 5.

Naming conventions: The *Username* and *screenname* of a Twitter user usually exhibit a high degree of similarity. Normally, *screennames* of legitimate users contain segments of the *username*, are not very lengthy and rarely begin with a digit. In some cases, *usernames* of legitimate users contain a reasonably small number of digits in the middle or at the end. In spam accounts, the mix of letters, digits, special characters and unusual symbols is much more widespread. Often, names begin with digits or email addresses and, as shown in Table 4, there is high discrepancy between *usernames* and the corresponding *screenname*. Features, such as *NameSim* and *NamesRatio*, in Fig. 5, are inspired by this analysis. Other static features in the metadata of a user account on Twitter, such as the *Language* and *Location* fields, may be useful to some extent for identifying spam accounts, due to the fact that most of these fields are either vacant or populated with meaningless content for spam users. Genuine users often report a real location name, but spam posting accounts often return irrelevant content or lengthy and unintelligible sequences of characters or just email addresses.

Tweeting activity and posting behaviour: In an earlier study, spam posting users have been observed to post four tweets per day on average [9]. We observed that an automated spam posting account posts on average at least 12 tweets per day at well-defined

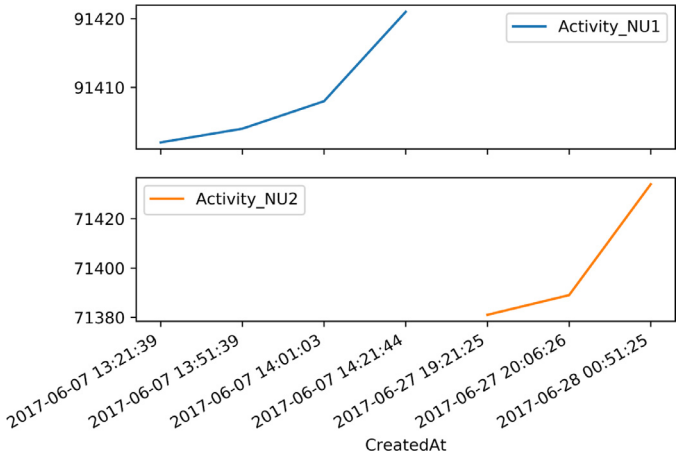

Fig. 4. Example of activity patterns of two legitimate users.

periods. Usually, activity levels remain constant within approximately four long-lasting periods. Figs. 3 and 4 show examples of spam and legitimate user activity patterns from our June 2017 collection, respectively.

In contrast to automated spam-posting users, a legitimate user of Twitter often follows random usage patterns and takes long breaks of inactivity. Fig. 4 represents the activity patterns of two different users with relatively low traffic generation within the same period as the users in Fig. 3. Table 5 shows the features proposed for prediction model training, the corresponding feature groups and definitions.

The *VerifiedAccount* feature, labelled as f_{22}, takes on binary values, '1' for verified accounts or '0' otherwise. These values reflect the target labels in the user profile meta-data. The feature was used in the feature set for training classification models during our early experiments. The resulting model overfitted the training data and, for this reason, the feature was later removed due to its role in leaking the correct prediction into the test data [44].

It is crucial for detection models to be able to continuously and automatically learn features strong enough to distinguish spam from non-spam, avoiding handcrafted features. Wu et al. [30] report good performance of a spam detection system that learns suitable features using Word2Vec. However, such methods rely on textual information, only. Social media, including Twitter, offer a wealth of information other than the textual content that are important to draw the distinction between a habitual spam posting account and a non-spam posting account. To improve the classification, we define and experiment with a set of handcrafted features, including features about the account and the user that posted each tweet.

Handcrafted features can be used in tandem with features learn by deep neural networks. Our study follows similar approaches to spam detection systems [26,28,30] by adopting the unsupervised paradigm. Unsupervised methods effectively counter the effect of *Twitter Spam Drift*, which affect detection systems [28,30], by capturing the variability of spammer behaviour effectively. Sedhai and Sun [26] used a semi-supervised framework for spam detection at tweet level, whereas Chen et al. [28] used both traditional machine learning on handcrafted features and deep learning to automatically learn some key features. We experimented with both handcrafted features and features learnt by deep learning models and compare their performance, as shown in Table 10. To account for full variability, the more handcrafted features are used, the better the classification performance and the lower the false positive rate [32]. Significant performance improvements were achieved at different levels in our study.

6. Experimentation and results

This section discusses the experimental procedure and the results obtained. All experiments are conducted using the Scikit-learn toolkit [45].

6.1. Parameter tuning and classification models

An effective classifier should be able to correctly classify previously unseen data by leveraging the experience gained from training on n labelled samples, i.e. data instances and the corresponding class. The target of the classification task at hand is to predict spam-posting users or normal legitimate users correctly, by accessing one of their tweets associated with user account meta-data. Effective hyper-parameter tuning is key for significantly improving the performance of machine learning models [46]. Thus, we tuned the hyper-parameter values of all classification models, used in experiments of our study, via grid search on standard 10-fold cross-validation.

6.2. Feature importance and correlation

During an initial analysis stage, a large number of features have been used for training and some features were discarded due to their relatively low contribution to the overall performance. Figs. A.1 and A.2 in the Appendix provide more evidence about the feature selection process. A recursive feature elimination approach was adopted to measure the contribution of each feature to the overall performance. The results of this process are graphically illustrated in Fig. 5.

Correlation analysis plays a crucial role in achieving optimum performance. Features that correlate perfectly introduce redundancy and do not add extra information into classification models [47]. We conducted univariate feature analysis to understand the relevance of each feature in predicting the target class. The results are shown in Fig. 6 formatted as a heat-map to visualise as colour intensity the correlation degree of each feature with the target class, i.e. *AccountClass*, and with other features. With the exception of *lexical richness, LexRichWithUU*, and *lexical density, LexRichWithOutUU*, which are derived from same root, there is no other pair of features with perfect correlation. Thus, the features shown in 6 comprise our feature set for all experiments in this paper.[4] The main diagonal of the heatmap matrix represents perfect correlation because each feature is correlated with itself. The column of the target *(AccountClass)* shows the intensity of the correlation of each feature with the target.

6.3. Performance metrics

For evaluation, different metrics are utilised in order to avoid any type of bias towards the majority class, especially when the dataset is imbalanced [48]. In particular, we use the following metrics to summarise experimental results: *F-score, Precision, Recall, Accuracy*, the *Receiver Operating Characteristics (ROC) curve* and the *area under the ROC curve (AUC). F-score*, the geometric mean of *Precision* and *Recall*, captures a model's prediction quality especially in sensitive areas, by requiring both *Precision* and *Recall* to be high. The *AUC* offers a more encompassing metric, insensitive to the imbalance between classes that sometimes provides better evaluation than accuracy [49]. Specifically, the higher the AUC score, the larger the area under the curve, well above the diagonal, e.g. Fig. 7.

[4] In the preliminary stages of this study, we experimented with many more features, mainly derived as combinations of the features in Fig. 6. Most of these features were discarded due to correlating almost perfectly with others and, thus, not contributing to the accuracy of the model.

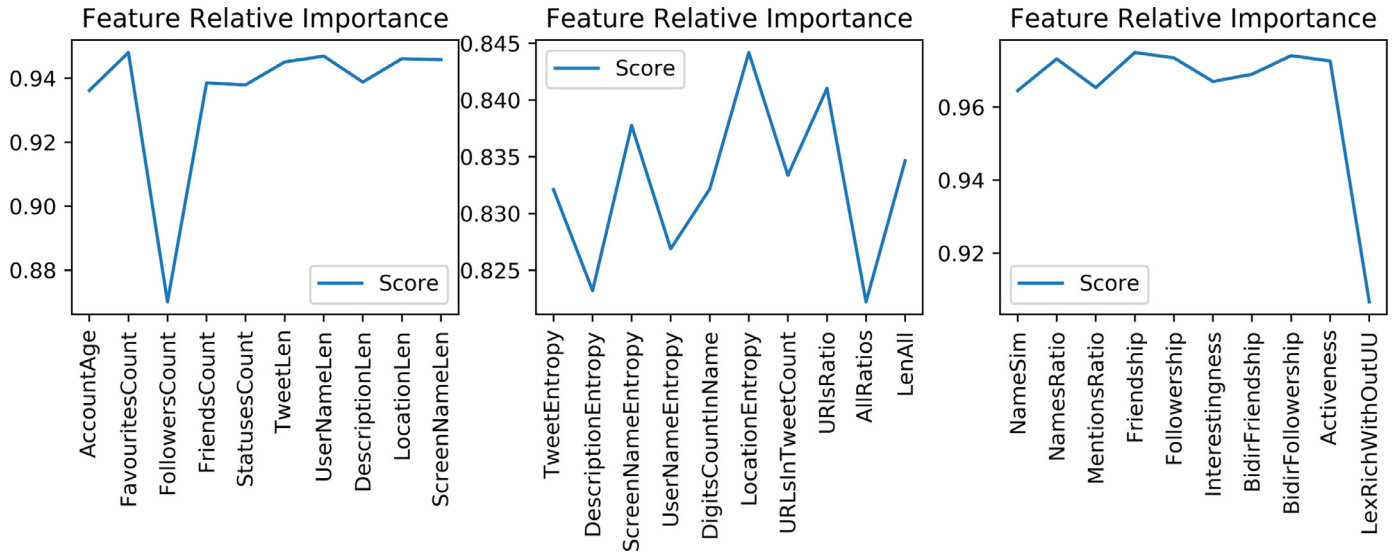

Fig. 5. This figure shows the performance of features measured using recursive feature elimination. The most informative feature is the lexical richness of tweets including user mentions and URLs *(LexRichWithUU)*. It contributed significantly to the overall performance, as evidenced from the sharp drop in the figure. The complete set of the features is provided in Figs. A.1 and A.2 in the Appendix.

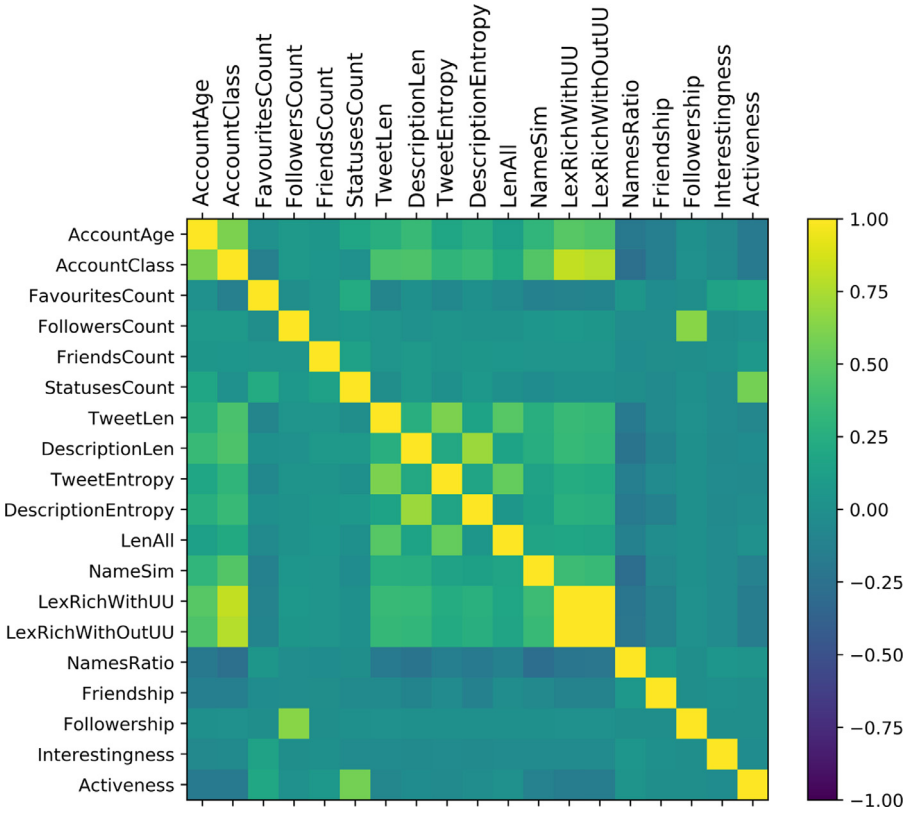

Fig. 6. Visual representation of the univariate analysis of correlation of each feature with the target, i.e. *AccountClass* and other features. Correlation magnitudes range from 1 to −1, with 1 denoting perfect positive correlation, 0 no correlation and −1 perfect negative correlation. Features highly correlated with the target constitute the optimum features set.

6.4. Experimental results

We conducted a series of experiments with different classification models and assessed them using various metrics, as discussed in Section 6.3. Our first experiment, aimed to investigate the effectiveness of the proposed features, which we called *Spam Post Detection (SPD)* features, and compared the suitability of different classification models for the task at hand. We trained six different classification models: Maximum-Entropy (MaxEnt), Random Forest, Extremely Randomized Trees (ExtraTrees), C-Support Vector Classification (SVC[5]), Gradient Boosting and Multi-layer Perceptron (MLP). We also included additional model i.e. *SVM + MLP* which utilises the features learnt by the MLP during training as input for training regime. Fig. 7 shows the learning curves and

[5] Which is based on Support Vector Machines (SVM). SVM and SVC used interchangeably in this study.

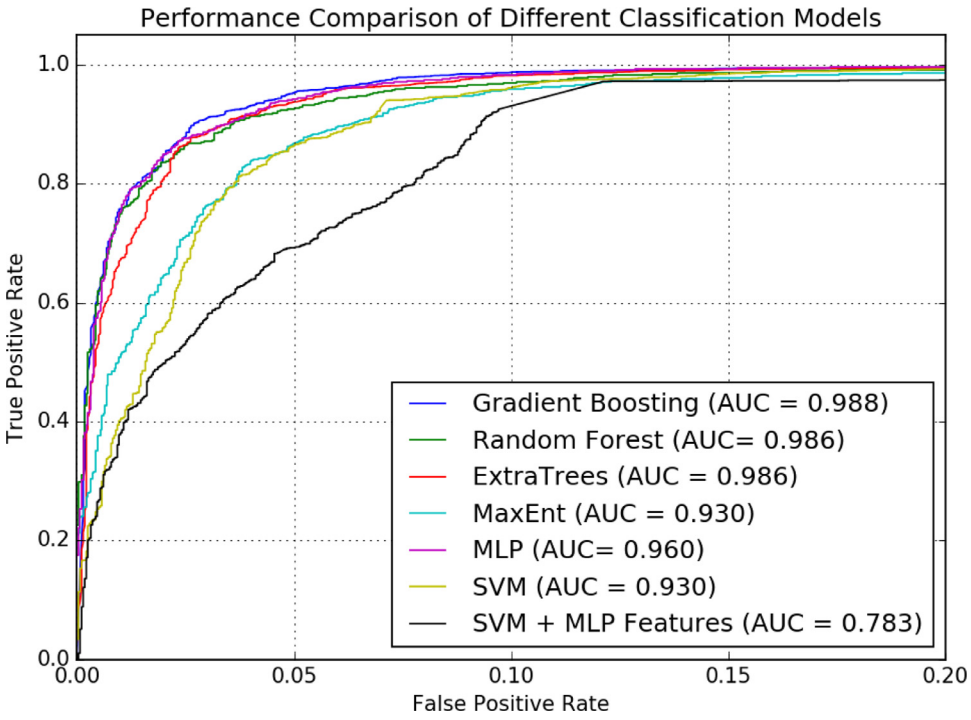

Fig. 7. Performance of different classification models evaluated on the $SPD_{automated}$ dataset using 10-fold cross-validation.

corresponding AUC scores achieved by each model on the best hyper-parameter values, as explained in Section 6.1. All models were trained and evaluated on the $SPD_{automated}$ dataset using 10-fold cross-validation. The chart shows relative consistency in terms of performance across the different classification models, which can be attributed to the effectiveness of the proposed *SPD* features. *Gradient Boosting* is chosen for subsequent use in our next experiments due to its higher performance.

Our second experiment compared the features proposed in this study, *SPD* features, with the *Honeypot* features, proposed in Lee et al. [9]. Since the study of Lee et al. [9] is our main baseline, we compared the two feature sets on the *Honeypot* dataset and the $SPD_{automated}$ dataset, using 10-fold cross validation. The associated learning curves are shown in Figs. 8 and 9, respectively. The figures show that *SPD* features perform better than the *Honeypot* features for both datasets. The improvement is small for the *Honeypot* dataset, whereas it is significant for the $SPD_{automated}$ dataset

It should be noted that the *Honeypot* dataset does not provide enough information for computing all *SPD* features. As a result, the *SPD* features line in Fig. 8 is based on some *SPD* features, only. Features such as *Interestingness, Activeness, NameSim* and *Lexical Richness* are not used in this experiment. The lack of these features explains why the improvement in performance is minimal.

In addition to the univariate correlation analysis of features, we investigated the importance of features groups. Table 6 shows the features grouped into three distinct groups: account, users and network features. In the additional experiment with Word2Vec, features learnt by the trained Word2Vec model and some handcrafted features from the study are utilised, in particular lexical richness, activeness and interestingness. Tables 7–9 present experiment results for for *Honeypot*, $SPD_{automated}$ *and* SPD_{manual}, respectively on various feature groups. Best performing features in each group constitute the optimum set of features i.e. $SPD_{selected}$) for improved effectiveness and efficiency.

Similarly, Table 10 shows the performance of various classifiers on including or excluding Word2Vec features tested on $SPD_{automated}$. Combining Word2Vec features and lexical richness features performs significantly better than the *Honeypot features* baseline. The combination performs slightly worse that the optimised feature set but uses a much smaller number of features.

To address the imbalance in the SPD_{manual} dataset, we utilised the *SMOTE* technique [36], which up-samples the minority class during training the classifier. We observe that the set of features proposed in this paper, *SPD*, performs better than the *Honeypot* [9] on all datasets. The lightweight version of *SPD* features, as computed by the feature selection process in Section 6.2, perform better than the *Honeypot* feature set when applied on $SPD_{automated}$ but worse than the *Honeypot* feature set when applied on *Honeypot* and SPD_{manual}. The lightweight version of *SPD* features consistently perform worse than the full *SPD* feature set, as expected.

6.5. Error analysis

Error analysis is carried-out to investigate cases that were not classified correctly by the classification model. In this section, we discuss the reasons that may have led to misclassification of some representative samples, shown in Fig. 11

In the study, dataset that was used to design the *SPD* features proposed in this study only tweets in English were considered. As a result, some tweets in the *SPD* dataset, such as tweet #1 in Table 11, were not in English and were misclassified. This can be attributed to the fact that although the original language field in the meta-section of some user profiles was set as English, the actual interaction language in the tweet is not English.

As shown in Fig. 5, lexical richness and density are important classification features. The occurrence of irrelevant tokens in a tweet, which were regarded as unique, leads to a richer lexicon, which in turn increases the chance of classifying the tweet as legitimate. Tweets #2-#6 in Table 11 contain some irrelevant symbols, which were counted as unique, increased the corresponding

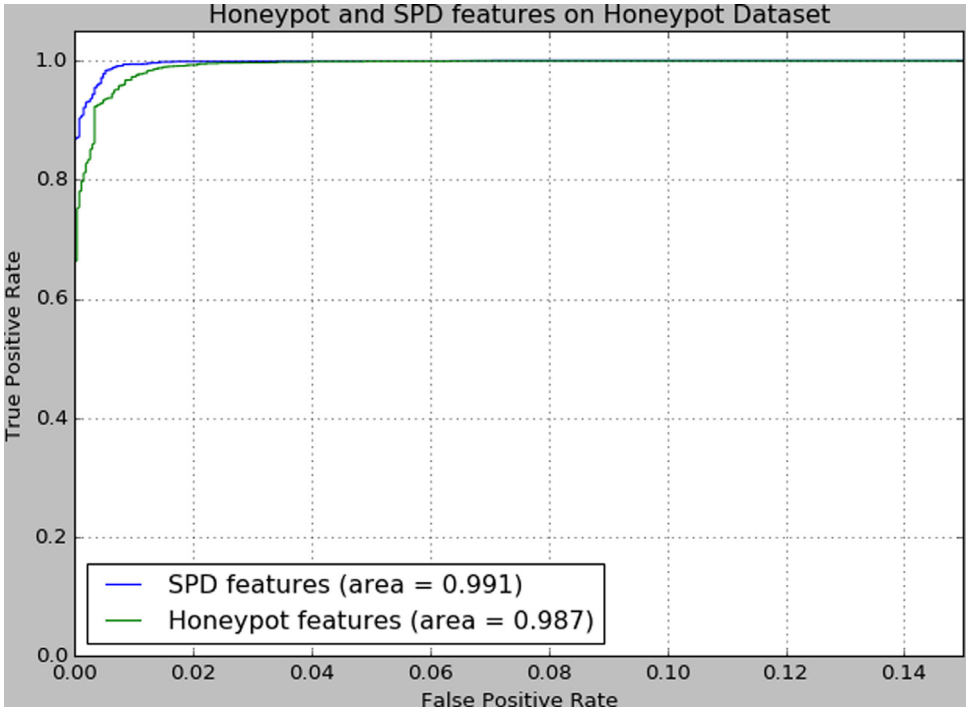

Fig. 8. Learning curves of the *SPD* features and the Honeypot features on the *Honeypot* dataset [9]. The *SPD* features achieve a slight improvement in performance.

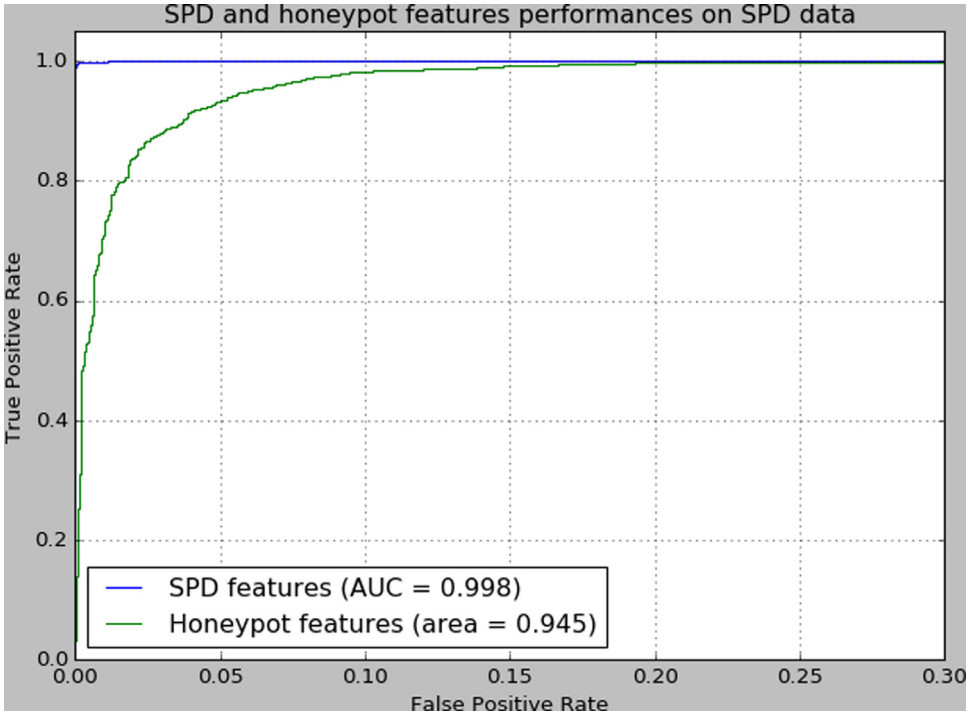

Fig. 9. Learning curves of the *SPD* features and the Honeypot features [9] on the $SPD_{automated}$ dataset. The *SPD* improve performance significantly.

lexical score and misled the classifier. Emoticons are also a source of confusion for the classifier, especially when computing the lexicon of unique tokens for a tweet and its similarity to lexicons of other tweets.

7. Discussion

This section presents an additional analysis of the manual annotations in the SPD_{manual} dataset, a description of the different user groups and discusses the distribution of relevant features in the dataset.

7.1. Characterising users

A thorough inspection of the tweets in the spam and legitimate parts of the SPD_{manual} dataset suggests that there are two kinds of users on Twitter: *human* users and *social bot (autonomous entity)* users. Each user type consists of a legitimate (non-spam) and a spam part, as depicted in Fig. 10, with the following characteristics:

Table 6
All Features and respective feature sets.

Feature group	Features			
Account	AccountAge	DescriptionLen	LocationLen	LocationEntropy
	DescriptionEntropy			
User	UserNameLen	ScreenNameLen	AllRatios	LexicalRichness
	TweetLen	URLsInTweetLen	Activeness	TweetEntropy
	LenAll	StatusesCount	URLsRatio	NamesRatio
	PosSentiment	NegSentiment	OverallSent	DescPosSent
	DescNegSent	DescSentiment	UserNameEntropy	
	SingleHashtagInTweetLen		ScreenNameEntropy	
Pairwise (Network)	*Engaged with:*		*Engaged by:*	
	FriendsCount	MentionsInTweetLen	FollowersCount	Interestingness
	MentionsRatio	HashtagsInTweetLen	Followership	BidirFriendship
	Friendship	HashtagsRatio	BirdirFollowership	
Optimised	AccountAge	FollowersCount	TweetLen	TweetEntropy
	Friendship	StatusesCount	LenAll	FriendsCount
	NamesRatio	Interestingness	NameSim	Followership
	Activeness	LexRichWithUU	DescriptionEntropy	
	BirdirFollowership			

Table 7
Evaluation results of all combinations of classifiers and feature sets applied on the *Honeypot* dataset. '(0, 1)' denotes performance on the spam part and the legitimate part of each dataset, respectively.

Classifier	Features	Accuracy %	AUC %	Precision % (0,1)	Recall % (0,1)	F-score % (0,1)
Random Forest	*Honeypot*	**94.70**	96.19	(90, 93)	**(91, 94)**	(92, 92)
	$SPD_{Selected}$	94.68	**96.38**	(93, 91)	**(92, 93)**	**(93, 92)**
ExtraTrees	*Honeypot*	93.74	**96.37**	**(91, 91)**	**(92, 89)**	**(91, 90)**
	$SPD_{Selected}$	**93.86**	95.32	(89, 90)	(91, 89)	(90, 90)
Gradient Boosting	*Honeypot*	98.53	98.55	(99, 98)	(98, 99)	(99, 98)
	$SPD_{Selected}$	**98.93**	**98.94**	**(99, 99)**	**(99, 99)**	**(99, 99)**
MaxEnt	*Honeypot*	83.57	83.62	(86, 81)	(83, 84)	(84, 83)
	$SPD_{Selected}$	**85.99**	**86.21**	**(90, 82)**	**(83, 89)**	**(86, 86)**
MLP	*Honeypot*	89.53	89.54	(91, 88)	(89, 90)	(90, 89)
	$SPD_{Selected}$	**93.70**	**90.28**	**(91, 90)**	**(92, 89)**	**(91, 90)**
SVM	*Honeypot*	86.26	86.29	(88, 84)	(86, 87)	(87, 85)
	$SPD_{Selected}$	**88.13**	**88.21**	**(90, 86)**	**(86, 90)**	**(88, 88)**
SVM + MLP Features	*Honeypot*	87.57	87.62	(90, 85)	(87, 88)	(88, 87)
	$SPD_{Selected}$	**89.08**	**89.09**	**(90, 88)**	**(89, 89)**	**(89, 89)**

7.1.1. Legitimate users

Legitimate users interact with moderate frequency, within the reasonable and acceptable Twitter usage policy. This user group also contains *genuine multiple users*, i.e. accounts managed by organisations or *useful social bots*. Users in this group tend to show a proportionate interaction level and *(activeness)*, i.e. their statuses count matches their account age and the tweets they post are of interest to followers, hence exhibit high *interestingness*. Followers of users in this group often outnumber friends, sometimes even by twice as much. This is expected, since most users subscribe or follow an account due to their interest in it.

The *username* and *screenname* of useful social bot accounts often contain the word *'bot'* as part of name, e.g. *AIBigDataCloudIoTBot* and *Troll Bot*. In some cases, groups of *screennames* share the same suffix separated by the underscore character from a description of the account. Accounts in this group achieve relatively high *interestingness* levels and an almost equal proportion of friends and followers. They also exhibit moderate similarity between their *username* and *screenname* and use a wide variety of words and expressions, i.e. diverse lexicons.

7.1.2. Spam-posting users

Spam-posting users are hyperactive and generate irrelevant content, potentially offensive to other users and in violation of Twitter's terms of use.[6] Accounts in this group exhibit very low *interestingness* and disproportionate *activeness* levels i.e. the statuses count does not match the account age indicating that they employ flooding techniques. Friends of users in this group usually outnumber followers. The interaction patterns of spam-posting social bot accounts are often randomised rather than well-defined, as shown in Fig. 3. There is also a high level of inconsistency in naming conventions and a high dissimilarity between *usernames* and corresponding *screennames*. The *screenname* of spam-posting social bot accounts is often unintelligible, mostly containing digits and special characters. Spam-posting users also exhibit low lexicon richness due to the high proportion of URLs, retweets, and user mentions. Spam users generally engage in subscribing to different conversations on Twitter (based on hashtags) and generate tweets not related to the topic of discussion. Fig. 11 shows a summary of user groups in Twitter, human and social bot, legitimate and spam-posting. The filtering mechanism developed in this study was succesfully applied in the work of [50] to detect and remove irrelevant posts from spam and automated accounts.

[6] Detailed in [42].

Table 8
Evaluation results of all combinations of classifiers and feature sets applied on the $SPD_{automated}$ dataset. '(0, 1)' denotes performance on the spam part and the legitimate part of each dataset, respectively.

Classifier	Features	Accuracy %	AUC %	Precision % (0,1)	Recall % (0,1)	F-score % (0,1)
Random Forest	*Honeypot*	90.71	96.28	(91, 91)	(92, 89)	(92, 90)
	$SPD_{Account}$	80.90	85.74	(86, 61)	(90, 51)	(88, 56)
	SPD_{User}	85.81	91.37	(91, 69)	(91, 69)	(91, 69)
	$SPD_{Network}$	92.06	96.70	(95, 83)	(95, 82)	(95, 82)
	$SPD_{Optimised}$	**98.46**	**98.87**	**(99, 99)**	**(99, 99)**	**(99, 99)**
	SPD all	94.41	98.13	(96, 88)	(96, 87)	(96, 88)
ExtraTrees	*Honeypot*	90.57	96.32	(91, 91)	(92, 89)	(91, 90)
	$SPD_{Account}$	80.53	85.85	(86, 60)	(89, 54)	(87, 57)
	SPD_{User}	86.22	91.48	(89, 73)	(94, 61)	(91, 67)
	$SPD_{Network}$	91.99	96.51	(94, 83)	(94, 81)	(94, 82)
	$SPD_{Optimised}$	**98.63**	**99.89**	**(100, 97)**	**(97,100)**	**(99, 98)**
	SPD_{all}	93.78	98.09	(96, 87)	(96, 87)	(96, 87)
Gradient Boosting	*Honeypot*	94.93	94.94	(96, 94)	(95, 95)	(95, 95)
	$SPD_{Account}$	82.17	87.13	(85, 66)	(93, 46)	(89, 54)
	SPD_{User}	85.74	91.82	(88, 76)	(94, 57)	(91, 65)
	$SPD_{Network}$	91.62	96.41	(93, 85)	(96, 77)	(95, 81)
	$SPD_{Optimised}$	**98.97**	**99.93**	**(99, 98)**	**(98, 99)**	**(98, 99)**
	SPD_{all}	93.60	97.96	(96, 88)	(97, 85)	(96, 87)
MaxEnt	*Honeypot*	84.59	84.65	(87, 82)	(84, 86)	(85, 84)
	$SPD_{Account}$	80.93	69.45	(86, 60)	(90, 48)	(88, 54)
	SPD_{User}	81.00	68.56	(85, 61)	(91, 46)	(88, 52)
	$SPD_{Network}$	85.37	72.35	(86, 84)	(97, 48)	(91, 61)
	$SPD_{Optimised}$	**97.12**	**97.13**	**(98, 96)**	**(97, 98)**	**(97, 97)**
	SPD_{all}	91.54	87.67	(94, 82)	(94, 81)	(94, 82)
MLP	*Honeypot*	89.34	89.40	(91, 87)	(89, 90)	(90, 89)
	$SPD_{Account}$	81.85	74.49	(87, 63)	(89, 60)	(88, 62)
	SPD_{User}	85.51	79.62	(91, 69)	(91, 69)	(91, 69)
	$SPD_{Network}$	91.40	86.84	(94, 83)	(95, 78)	(94, 81)
	$SPD_{Optimised}$	**98.42**	**98.43**	**(99, 98)**	**(98, 99)**	**(98, 98)**
	SPD_{all}	94.17	91.83	(96, 87)	(96, 88)	(96, 88)
SVM	*Honeypot*	86.38	86.39	(88, 85)	(86, 87)	(87, 86)
	$SPD_{Account}$	81.22	71.50	(87, 60)	(89, 54)	(88, 57)
	SPD_{User}	82.08	68.54	(85, 68)	(94, 43)	(89, 53)
	$SPD_{Network}$	87.33	75.12	(87, 89)	(98, 52)	(92, 62)
	$SPD_{Optimised}$	**97.35**	**97.38**	**(98, 97)**	**(97, 98)**	**(97, 97)**
	SPD_{all}	91.50	88.82	(95, 79)	(94, 83)	(94, 81)
SVM + MLP Features	*Honeypot*	88.21	88.23	(90, 87)	(88, 89)	(89, 88)
	$SPD_{Account}$	80.69	69.04	(85, 60)	(91, 47)	(88, 53)
	SPD_{User}	84.38	74.99	(88, 70)	(92, 58)	(90, 63)
	$SPD_{Network}$	90.24	83.43	(92, 85)	(96, 71)	(94, 77)
	$SPD_{Optimised}$	**97.71**	**97.74**	**(99, 97)**	**(97, 98)**	**(98, 98)**
	SPD_{all}	93.70	90.95	(96, 85)	(96, 85)	(96, 85)

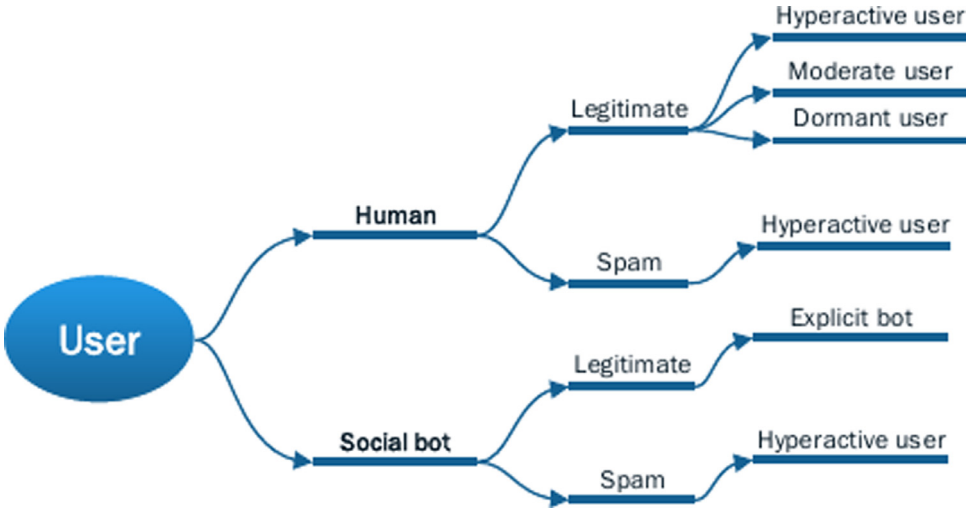

Fig. 10. User types in the SPD_{manual} dataset.

Table 9
Evaluation results of all combinations of classifiers and feature sets applied on the SPD_{manual} dataset. '(0, 1)' denotes performance on the spam part and the legitimate part of each dataset, respectively.

Classifier	Features	Accuracy %	AUC %	Precision % (0,1)	Recall % (0,1)	F-score % (0,1)
Random Forest	*Honeypot*	93.03	93.11	(91, 89)	(89, 90)	(91, 90)
	$SPD_{Account}$	77.16	79.98	(75, 77)	(74, 78)	(75, 76)
	SPD_{User}	84.29	92.89	(83, 84)	(84, 84)	(85, 85)
	$SPD_{Network}$	95.43	99.74	(92, 94)	(94, 92)	(93, 95)
	$SPD_{Optimised}$	**97.79**	**98.03**	**(94, 98)**	**(98, 94)**	**(97, 97)**
	SPD_{all}	96.29	97.97	(93, 99)	(99, 93)	(96, 96)
ExtraTrees	*Honeypot*	**99.26**	**99.24**	**(99,100)**	**(100, 98)**	**(99, 99)**
	$SPD_{Account}$	75.43	79.49	(73, 78)	(78, 72)	(71, 75)
	SPD_{User}	83.54	91.86	(85, 88)	(84, 90)	(82, 84)
	$SPD_{Network}$	95.80	97.97	(94, 96)	(97, 93)	(96, 96)
	$SPD_{Optimised}$	97.29	99.95	(94, 98)	(98, 93)	(97, 97)
	SPD_{all}	97.90	98.90	(96, 99)	(73, 78)	(98, 98)
Gradient Boosting	*Honeypot*	89.38	59.19	(35, 93)	(23, 96)	(27, 94)
	$SPD_{Account}$	78.13	79.40	(76, 78)	(78, 75)	(77, 76)
	SPD_{User}	87.39	93.45	(85, 90)	(91, 84)	(88, 87)
	$SPD_{Network}$	89.99	95.83	(87, 95)	(96, 83)	(91, 89)
	$SPD_{Optimised}$	**96.08**	**99.88**	**(97, 99)**	**(99, 96)**	**(98, 97)**
	SPD_{all}	93.20	98.22	(89, 98)	(99, 86)	(94, 93)
MaxEnt	*Honeypot*	72.93	73.02	(76, 70)	(69, 78)	(72, 74)
	$SPD_{Account}$	60.82	60.82	(60, 61)	(61, 61)	(61, 61)
	SPD_{User}	67.37	67.39	(77, 63)	(49, 86)	(60, 72)
	$SPD_{Network}$	55.01	56.09	(64, 52)	(32, 80)	(43, 63)
	$SPD_{Optimised}$	**75.40**	**75.51**	**(79, 72)**	**(70, 81)**	**(74, 76)**
	SPD_{all}	75.40	75.51	(79, 72)	(70, 81)	(74, 76)
MLP	*Honeypot*	82.58	82.43	(84, 82)	(81, 80)	(81, 80)
	$SPD_{Account}$	69.72	69.59	(70, 69)	(73, 66)	(71, 68)
	SPD_{User}	80.22	80.25	(82, 78)	(77, 83)	(80, 81)
	$SPD_{Network}$	62.42	62.29	(63, 62)	(57, 68)	(60, 65)
	$SPD_{Optimised}$	**82.94**	**82.95**	**(85, 81)**	**(83, 83)**	**(84, 82)**
	SPD_{all}	92.58	92.65	(89, 97)	(97, 88)	(93, 92)
SVM	*Honeypot*	73.81	73.79	(73, 74)	(73, 74)	(73, 74)
	$SPD_{Account}$	66.01	66.79	(60, 76)	(82, 51)	(70, 61)
	SPD_{User}	73.30	72.92	(80, 69)	(60, 86)	(68, 77)
	$SPD_{Network}$	58.84	58.36	(62, 57)	(77, 40)	(49, 66)
	$SPD_{Optimised}$	**75.65**	**75.58**	**(79, 73)**	**(69, 82)**	**(77, 74)**
	SPD_{all}	80.47	80.46	(81, 80)	(81, 80)	(81, 80)
SVM + MLP Features	*Honeypot*	73.98	74.13	(76, 77)	(77, 75)	(73, 74)
	$SPD_{Account}$	63.54	63.69	(61, 67)	(67, 61)	(63, 64)
	SPD_{User}	71.32	70.96	(77, 68)	(58, 84)	(66, 75)
	$SPD_{Network}$	59.46	59.04	(62, 58)	(43, 75)	(51, 66)
	$SPD_{Optimised}$	**76.64**	**76.58**	**(79, 75)**	**(72, 81)**	**(75, 78)**
	SPD_{all}	87.64	87.49	(85, 91)	(92, 83)	(89, 87)

Table 10
Evaluation results of Word2Vec features in comparison with our optimised set of features for all classifiers. The Word2Vec feature group contains features learnt by the Word2Vec model, and some handcrafted features *lexical richness, activeness* and *interestingness*. '(0, 1)' denotes performance on the spam part and the legitimate part of the dataset, respectively.

Classifier	Features	Accuracy %	AUC %	Precision % (0,1)	Recall % (0,1)	F-score % (0,1)
Random Forest	$SPD_{Word2Vec}$	94.95	**99.05**	(95, 95)	(95, 95)	(95, 95)
	$SPD_{Optimised}$	98.46	**99.87**	(99, 99)	(99, 99)	(99, 99)
ExtraTrees	$SPD_{Word2Vec}$	95.47	**99.34**	(96, 95)	(96, 95)	(96, 95)
	$SPD_{Optimised}$	98.63	**99.89**	(100, 97)	(97, 100)	(99, 98)
Gradient Boosting	$SPD_{Word2Vec}$	95.04	**99.09**	(95, 95)	(95 ,95)	(95, 95)
	$SPD_{Optimised}$	98.72	**99.93**	(99, 98)	(98, 99)	(98, 99)
MaxEnt	$SPD_{Word2Vec}$	89.03	**89.14**	(92, 86)	(87, 91)	(89, 89)
	$SPD_{Optimised}$	97.12	**97.13**	(98, 96)	(97, 98)	(97, 97)
MLP	$SPD_{Word2Vec}$	94.40	**94.43**	(96, 93)	(93, 96)	(94, 94)
	$SPD_{Optimised}$	98.42	**98.43**	(99, 98)	(98, 99)	(98, 98)
SVM	$SPD_{Word2Vec}$	89.91	**90.01**	(93, 87)	(87, 93)	(90, 90)
	$SPD_{Optimised}$	97.35	**97.38**	(98, 97)	(97, 98)	(97, 97)
SVM + MLP Features	$SPD_{Word2Vec}$	92.08	**92.24**	(96, 88)	(88, 96)	(92, 92)
	$SPD_{Optimised}$	97.71	**97.74**	(99, 97)	(97, 98)	(98, 98)

Table 11
Sample tokens from misclassified tweets

Id	Tweet
1	gain followers 👉@ .... 🍎🏕アメリカとカナダでまっています。\n地域社会に溶けめずにいた民が、ディナ会での出会.
2	retweet this 👆✅
3	like this 👆✅
4	follow like & retweet 👆✅
5	follow back follow you 👆✅
6	gain followers 👆...', 68), gain followers 👉...', this ••; retweet this 👆✅

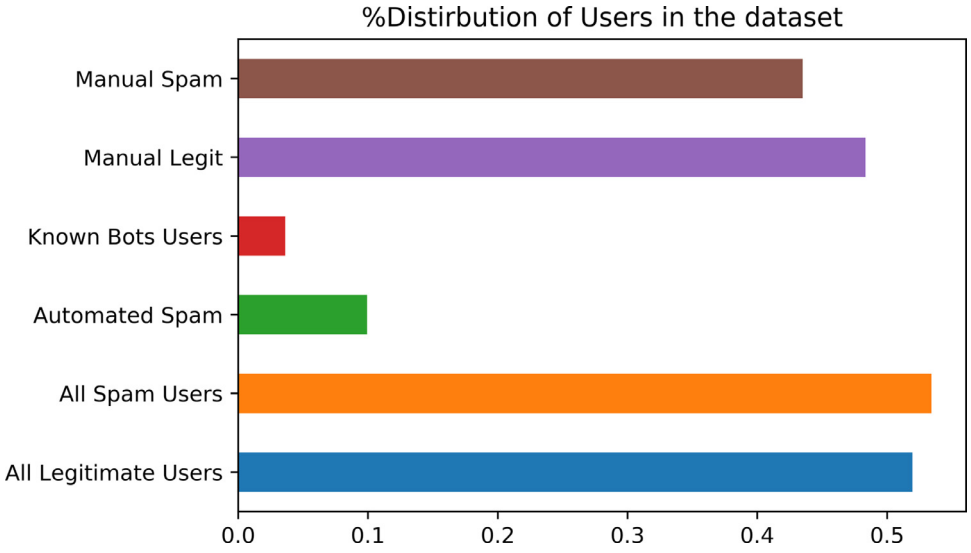

Fig. 11. Distribution of different users in the SPD_{manual} dataset. Known bots are accounts that mention the word 'bot' explicitly as part of their name and share some basic features similarities with normal users such as the level of name similarity *(NameSim)*. Known bots in the dataset account for less than 10% of all users.

8. Conclusion and future work

This study offers an effective method for spam detection and new insights into the sophisticatedly evolving techniques for spamming on Twitter. The proposed spam detection method utilised an optimised set of readily available features. Being independent of historical tweets which are often unavailable on Twitter makes them suitable for real-time spam detection. The efficacy and robustness of the proposed features set is shown by testing a number of machine learning models and on dataset collected orthogonally from the study data. Performance is consistent across the different models and there is significant improvement over the baseline. It was also shown that automated spam accounts follow a well-defined pattern with surges of intermittent activities. The proposed spam tweet detection approach can be applied in any real-time filtering application. For example, it is applicable to data collection pipelines to filter out irrelevant content at an early pre-processing stage to ensure the quality and representativeness of research data. The combination of handcrafted features and features learnt in an unsupervised manner using word embeddings is shown to significantly improve baseline performance and to perform comparably to the best performing feature set using a smaller number of features.

During the analysis of the data, we observed that spam users tend to be selective in following other users thereby forming enclaves of spammers. This is a high-level observation that we aim to explore further in the future. Additionally, both the two broad user groups, i.e. human users and social bot (autonomous entity) users contain spammers, whose spamming behaviour tends to be similar. The distinction between legitimate human users vs. legitimate social bots as well as human spammers vs. social bot spammers needs to be investigated further. Another interesting dimension for future work is to study the effect of the recent increase in the maximum length of tweets [25] on spamming activity. Intuitively, automated spam accounts will face difficulties in generating lengthier tweets intelligently, thereby making these tweets easier to identif y.

Acknowledgments

The authors would like to thank Prof. Francesco Rizzuto for the fruitful discussions and exchange of ideas about a multitude of aspects related to social media, spam content and the motives of spammers. The third author has participated in this research work as part of the CROSSMINER Project, which has received funding from the European Unions Horizon 2020 Research and Innovation Programme under grant agreement No. 732223.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.neucom.2018.07.044.

References

[1] Social media statistics and facts, Online: http://www.statista.com/topics/1164/social-networks, Accessed: 18-02-2018.

[2] E. Rojas, J. Munoz-Gama, M. Sepúlveda, D. Capurro, Process mining in healthcare: a literature review, J. Biomed. Inf. 61 (2016) 224–236, doi:10.1016/j.jbi.2016.04.007. https://doi.org/10.1016/j.jbi.2016.04.007.
[3] K.C. Yee, E. Miils, C. Airey, Perfect match? Generation Y as change agents for information communication technology implementation in healthcare, Stud. Health Technol. Inf. 136 (2008) 496–501.
[4] T. Davenport, Analytics in Sports: The New Science of Winning, International Institute for Analytics, 2014 Technical report. White paper
[5] Deloitte, Social Analytics in Media Entertainment the Three-minute Guide, Deloitte Development LLC, 2014 Technical report.
[6] D. Contractor, B. Chawda, S. Mehta, L. Subramaniam, T.A. Faruquie, Tracking political elections on social media: applications and experience, in: Proceedings of the Twenty-Fourth International Conference on Artificial Intelligence, AAAI Press, 2015, pp. 2320–2326.
[7] NexGate, State of Social Media Spam Research Report, NexGate. 2013. Online, Accessed: 18-02-2018.
[8] O. Varol, E. Ferrara, C.A. Davis, F. Menczer, A. Flammini, Online human–bot interactions: detection, estimation, and characterization, in: Proceedings of the International AAAI Conference on Web and Social Media, AAAI Press, 2017, pp. 280–289.
[9] K. Lee, B.D. Eoff, J. Caverlee, Seven months with the devils: a long-term study of content polluters on twitter, in: Proceedings of the Fifth International Conference on Weblogs and Social Media, Barcelona, Catalonia, Spain, 2011, pp. 185–192.
[10] M. Alsaleh, A. Alarifi, F. Al-Quayed, A.S. Al-Salman, Combating comment spam with machine learning approaches, in: Proceedings of the Fourteenth IEEE International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA, 2015, pp. 295–300.
[11] C.A. Davis, O. Varol, E. Ferrara, A. Flammini, F. Menczer, Botornot: a system to evaluate social bots, in: Proceedings of the Twenty-Fifth International Conference on World Wide Web (WWW), Companion Volume, Montreal, Canada, 2016, pp. 273–274.
[12] Y. Yao, B. Viswanath, J. Cryan, H. Zheng, B.Y. Zhao, Automated crowdturfing attacks and defenses in online review systems, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, TX, USA, 2017, pp. 1143–1158.
[13] V.S. Subrahmanian, A. Azaria, S. Durst, V. Kagan, A. Galstyan, K. Lerman, L. Zhu, E. Ferrara, A. Flammini, F. Menczer, The DARPA twitter bot challenge, IEEE Comput. 49 (6) (2016) 38–46.
[14] S. Vosoughi, D. Roy, S. Aral, The spread of true and false news online, Science 359 (6380) (2018) 1146–1151.
[15] A.H. Wang, Don't follow me: spam detection in Twitter, in: Proceedings of the IEEE International Conference on Security and cryptography (SECRYPT), 2010, pp. 1–10.
[16] C. Yang, R. Harkreader, J. Zhang, S. Shin, G. Gu, Analyzing spammers' social networks for fun and profit: a case study of cyber criminal ecosystem on twitter, in: Proceedings of the Twenty-First International Conference on World Wide Web, WWW '12, ACM, New York, NY, USA, 2012, pp. 71–80.
[17] H. Yu, M. Kaminsky, P.B. Gibbons, A.D. Flaxman, Sybilguard: defending against Sybil attacks via social networks, IEEE/ACM Transa. Netw. 16 (3) (2008) 576–589.
[18] G. Danezis, P. Mittal, Sybilinfer: detecting Sybil nodes using social networks, in: Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 2009.
[19] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, B.Y. Zhao, Detecting and characterizing social spam campaigns, in: Proceedings of the tenth ACM SIGCOMM Internet Measurement Conference (IMC), Melbourne, Australia, 2010, pp. 35–47.
[20] K. Thomas, C. Grier, J. Ma, V. Paxson, D. Song, Design and evaluation of a real-time URL spam filtering service, in: Proceedings of the Thirty-Second IEEE Symposium on Security and Privacy (S&P), Berkeley, CA, USA, 2011, pp. 447–462.
[21] S. Lee, J. Kim, Warningbird: Detecting suspicious URLs in twitter stream, in: Proceedings of the Nineteenth Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 2012, pp. 183–195.
[22] F. Benevenuto, G. Magno, T. Rodrigues, V. Almeida, Detecting spammers on twitter, in: Proceedings of the Electronic Messaging, Anti-Abuse and Spam Conference, CEAS, 6, 2010.
[23] P.N. Howard, B. Kollanyi, Bots, #Strongerin, and #Brexit: Computational Propaganda During the UK-EU Referendum (June 20, 2016). Available at SSRN: https://ssrn.com/abstract=2798311 or doi:10.2139/ssrn.2798311.
[24] C. Grier, K. Thomas, V. Paxson, C.M. Zhang, @spam: the underground on 140 characters or less, in: Proceedings of the Seventeenth ACM Conference on Computer and Communications Security (CCS), Chicago, IL, USA, 2010, pp. 27–37.
[25] T. Blog, Giving you more characters to express yourself. Online: http://blog.twitter.com/official/en_us/topics/product/2017/Giving-you-more-characters-to-express-yourself.html, Accessed: 18-02-2018.
[26] S. Sedhai, A. Sun, Semi-supervised spam detection in Twitter stream, IEEE Trans. Comput. Soc. Syst. 5 (1) (2018) 169–175.
[27] C. Chen, S. Wen, J. Zhang, Y. Xiang, J. Oliver, A. Alelaiwi, M.M. Hassan, Investigating the deceptive information in Twitter spam, Fut. Gen. Comput. Syst. 72 (2017a) 319–326.
[28] C. Chen, Y. Wang, J. Zhang, Y. Xiang, W. Zhou, G. Min, Statistical features-based real-time detection of drifted Twitter spam, IEEE Trans. Inf. Foren. Secur. 12 (4) (2017b) 914–925.
[29] C. Li, S. Liu, A comparative study of the class imbalance problem in Twitter spam detection, Concurr. Comput. Pract. Exp. 30 (5) (2018) e4281.
[30] T. Wu, S. Liu, J. Zhang, Y. Xiang, Twitter spam detection based on deep learning, in: Proceedings of the Australasian Computer Science Week Multiconference, ACM, 2017, p. 3.
[31] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient Estimation of Word Representations in Vector Space, in: Proceedings of the International Conference on Learning Representations (ICLR 2013) , 2013.
[32] N. Chavoshi, H. Hamooni, A. Mueen, Temporal patterns in bot activities, in: Proceedings of the Twenty-Sixth International Conference on World Wide Web Companion, International World Wide Web Conferences Steering Committee, 2017, pp. 1601–1606.
[33] B. Wang, A. Zubiaga, M. Liakata, R. Procter, Making the most of tweet-inherent features for social spam detection on Twitter, in: Proceedings of the the Fifth Workshop on Making Sense of Microposts, co-located with the Twenty-Fourth International World Wide Web Conference (WWW), Florence, Italy, 2015, pp. 10–16.
[34] B. Lott, Survey of Keyword Extraction Techniques, UNM Education, 2012 Technical report.
[35] Twitter, Twitter Streaming APIs. Online: http://dev.twitter.com/streaming/overview, Accessed: 18-02-2018.
[36] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, SMOTE: synthetic minority over-sampling technique, J. Artif. Intell. Res. 16 (2002) 321–357.
[37] N.P. Halko, P.G. Martinsson, J.A. Tropp, Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions, Soc. Ind. Appl. Math. (SIAM) Rev. 53 (2) (2011) 217–288.
[38] P. Wiemer-Hastings, K. Wiemer-Hastings, A.C. Graesser, Latent semantic analysis, in: Proceedings of the Sixteenth international joint conference on Artificial intelligence, 2004, pp. 1–14.
[39] F.J. Tweedie, R.H. Baayen, How variable may a constant be? Measures of lexical richness in perspective, Comput. Human. 32 (5) (1998) 323–352.
[40] D. Biber, S. C., G. Leech, The Longman Student Grammar of Spoken and Written English, Longman, 2002.
[41] Z. Šišková, Lexical richness in EFL students' narratives, Lang. Stud. Work. Pap. 4 (2012) 26–36.
[42] Twitter, the Twitter Rules. Online: http://help.twitter.com/en/rules-and-policies/twitter-rules, Accessed: 18-02-2018.
[43] V. Qazvinian, E. Rosengren, D.R. Radev, Q. Mei, Rumor has it: identifying misinformation in microblogs, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, (EMNLP), A Meeting of SIGDAT, a Special Interest Group of the ACL, Edinburgh, UK, 2011, pp. 1589–1599.
[44] G. Forman, M. Scholz, Apples-to-apples in cross-validation studies: pitfalls in classifier performance measurement, ACM SIGKDD Explor. Newslett. 12 (1) (2010) 49–57.
[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.
[46] R.S. Olson, W.L. Cava, Z. Mustahsan, A. Varik, J.H. Moore, Data-driven advice for applying machine learning to bioinformatics problems, Proc. Pac. Symp. Biocomput. (PSB) 23 (2018) 192–203.
[47] I. Guyon, A. Elisseeff, An introduction to variable and feature selection, J. Mach. Learn. Res. 3 (2003) 1157–1182.
[48] T. Fawcett, An introduction to ROC analysis, Pattern Recognit. Lett. 27 (8) (2006) 861–874. ROC Analysis in Pattern Recognition.
[49] N. Japkowicz, The class imbalance problem: Significance and strategies, in: Proceedings of the International Conference on Artificial Intelligence (ICAI), 2000, pp. 111–117.
[50] I. Inuwa-Dutse, Modelling formation of online temporal communities, in: Proceedings of the Companion of the The Web Conference on The Web Conference, International World Wide Web Conferences Steering Committee, 2018, pp. 867–871.

Isa Inuwa-Dutse earned his B.Sc. and M.Sc. degrees from Bayero University, Kano, Nigeria and The University of Manchester, UK respectively. Before joining Edge Hill University in 2016 as a Ph.D. candidate and Graduate Teaching Assistant, Isa worked as full-time lecturer in the Department of Computer Science, Federal University Dutse. Previously, he worked as a lecturer in the department of Statistics and Computer Science, Kano State Polytechnic. He holds full membership of British Computer Society, Computer Professional Council of Nigeria, IEEE and a fellow of the HEA, UK. His research interests span areas of artificial intelligence, text mining, social network analysis and modelling. He is currently working on the problem of detection of online temporal communities.

Mark Liptrott is a Senior Lecturer in Computer Science and Director of Post Graduate Studies and Director of International Students within the Department of Computer Science at Edge Hill University, UK. He has a doctorate, a first class honours B.Sc. in Information Systems, a PGCE in Higher Education Teaching and Learning Support, and a PGCE in Research Degree Supervision. His doctorate analysed the relevance of diffusion theory on the voluntary adoption of technological innovations introduced through the public policy process. His current research has shifted to focus on the adoption of e-initiatives and text analysis. Mark has addressed conferences throughout Europe and has contributed to journals, including the UK Government Computing News Magazine, and readers on technology adoption. He is an associate editor for the Journal of Information Technology and Citizenship and reviews for a number of other journals.

Yannis Korkontzelos is a Reader in Computer Science (Data Analytics) in the Department of Computer Science at Edge Hill University, UK. His research focus on a variety of subfields of Natural Language Processing and Text Mining, such as compositional semantics, multiword expression and term extraction, document classification and sentiment analysis, applied to a number of domains, such as biomedicine, social sciences, social media, open source software and scientific publications. Yannis holds a Ph.D. in Computer Science from the University of York, an M.Phil. in Computer Text, Speech and Internet Technology from the University of Cambridge, a Postgraduate Certificate in Teaching in Higher Education (PGCTHE) and a first class honours Diploma in Electrical and Computer Engineering from the National Technical University of Athens, Greece.

LEXICAL ANALYSIS OF AUTOMATED ACCOUNTS ON TWITTER

Isa Inuwa-Dutse[1], Bello Shehu Bello[2] and Ioannis Korkontzelos[1]
[1]*Department of Computer Science , Edge Hill University, UK*
[2]*Department of Informatics*, *University of Leicester, UK*

ABSTRACT

In recent years, social bots have been using increasingly more sophisticated, challenging detection strategies. While many approaches and features have been proposed, social bots evade detection and interact much like humans making it difficult to distinguish real human accounts from bot accounts. For detection systems, various features under the broader categories of account profile, tweet content, network and temporal pattern have been utilised. The use of tweet content features is limited to analysis of basic terms such as URLs, hashtags, name entities and sentiment. Given a set of tweet contents with no obvious pattern can we distinguish contents produced by social bots from that of humans? We aim to answer this question by analysing the lexical richness of tweets produced by the respective accounts using large collections of different datasets. Our results show a clear margin between the two classes in lexical diversity, lexical sophistication and distribution of emoticons. We found that the proposed lexical features significantly improve the performance of classifying both account types. These features are useful for training a standard machine learning classifier for effective detection of social bot accounts. A new dataset is made freely available for further exploration.

KEYWORDS

Twitter Bots, Lexical Analysis, Bot Detection, Social Network Analysis

1. INTRODUCTION

As a social being, human's behaviour is largely influenced by close associates. In this era of modern civilisation, the Internet has been the precursor of the socialisation being witnessed today. Social networking sites are important avenues for instant information sharing and interaction on a large scale (Gilani et al., 2017; Wilson et al., 2012). Modern social media platforms, such as Twitter and Facebook, enable various forms of interactions among diverse users. This capability results in a huge amount of data waiting to be mined by researchers. However, the credibility of such content is being questioned by the growing activities of automated accounts otherwise known as social bots. *Social Bots* are automated accounts designed to follow a well-defined algorithm to interact with users either by amplifying or generating contents in social media platforms (Ferrara et al., 2016). Some social bot accounts are legitimate, with clear distinguishing features, such as *@congressedits*[1], whereas the majority are created to mislead in various ways, such as by creating superficial popularity (Varol et al., 2017) or influencing public opinion (Howard and Kollanyi, 2016). Some bots are obvious, because they use the word "bot" explicitly in their Twitter handle (Inuwa-Dutse et al., 2018). Autonomous accounts contribute a sizeable part of social media content. It was estimated that 9% - 15% of active Twitter accounts are social bots accounts (Varol et al., 2017) and require effective methods to be detected.

How could we detect if a given tweet was posted by a social bot or a human user? We investigate this question based on a collection of tweets sampled from Twitter. In particular, we investigate how effective lexical features are for the detection of social bot accounts. In contrast to previous work (Benevenuto et al., 2010; Cai t al., 2017; Dockerson et al., 2014; Lee and Kim, 2012; Thomas et al., 2011; Varol et al., 2017; Wang, 2010), our study focuses on comprehensive linguistic analysis to define lexical features effective enough for accurate detection of social bot accounts on Twitter. As shown in Table 1, the study explores a

[1] A bot that tweets anonymous Wikipedia edits made from IP addresses in the US Congress.

large number of tweets from social bots and humans to understand the difference between the two in terms of lexical richness and distribution of emoticons, further discussed in section 3. Our analysis highlights the distinguishing characteristics of automated accounts and how lexical features can improve detection of bots.

Contribution: The study contributes a powerful set of features useful in distinguishing between humans and social bots on Twitter. We provide the first comprehensive analysis of lexical richness of tweets computed on various accounts and investigate how their incorporation improves the performance of the detection system. Features based on *lexical diversity, type-token ratio, usage of contractions and emoticons* are powerful lexical signals in distinguishing between humans and social bots. The study provides a new set of distinctive features and a dataset to support the research community in identifying bot accounts.

The remaining of the paper is structured as follows. Section 2 and Section 3 present a review of related works and propose lexical features, respectively. Section 4 presents our experiments and Section 5 presents the results and a detailed discussion. Finally, Section 6 concludes the study and proposes some future work.

2. RELATED WORK

Social bot accounts are instrumental in spreading fake and malicious news on Twitter, employed to skew analysis results and opinion of users. The demand for effective detection systems has prompted a surge of various research approaches. Early social bot accounts have been reported to lack basic account information such as meaningful screen names or profile picture (Varol et al., 2017; Lee et al., 2011). This is no longer the case, as social bots grow in sophistication, making it difficult to identify distinguishing features from human accounts. Some approaches involve the analysis of accounts as far as their position in a *network*, their *temporal* metadata and the *content of their tweets*, to define a new set of features. The following review focuses on related studies that utilised aspects of these features to detect bot accounts.

Network and Temporal Features: The study of Chavoshi et al. (2017) analyses the behavioural patterns of accounts by focusing on features related to the network structure, such as local motifs, i.e. repeating behaviour, and discords, i.e. anomalous behaviour. The study also shows how temporal behaviour is useful as a means to distinguish bot from human user accounts. The work of Davis et al. (2016) and (Ferrara et al. (2016) developed a detection system that leverages features related to both network structure and tweeting behaviour exhibited by accounts. In [**Error! Reference source not found.**] researchers utilised many features related to network, temporal behaviour, tweet syntax, tweet semantics and user profile for the detection of *influence bots*. Influence bots are categories of social bot accounts that aim at influencing the opinion of other users. However, despite the wide spectrum of features considered in the study, analysis of the lexical richness was not covered.

Tweet Content: Many detection systems have been developed by leveraging the content of tweets posted by account holders on Twitter. This approach was adopted in Dockerson et al. (2014) to detect social bot accounts on Twitter based on sentiment features, such as topic sentiment and the transition frequency of tweet's sentiment, to train a machine learning classifier. Similarly, to the *tweet content* approach that relies on linguistic analysis, Inuwa-Dutse et al. (2018) utilised features based on lexical richness to detect spam accounts on Twitter. The study of (Cai t al., 2017) proposed a deep learning approach that incorporates content and behavioural information to detect social bots.

Related literature pays little attention to the analysis of lexical richness of various users' tweets and how they can inform the detection of social bots on Twitter. In contrast to previous work, this study is based on in-depth analysis of lexical richness as a basis for building a detection system. We present an approach solely based on lexical analysis of contents from both humans and social bots accounts to distinguish between them.

3. LEXICAL RICHNESS FEATURES

This section describes the lexical features utilised in the study, able to improve detection systems. Lexical richness is a broad concept, expressed in various forms and metrics to assess the quality of text. Metrics such as lexical diversity, lexical sophistication and lexical density are commonly used in linguistic analysis (Šišková, 2012; Templin, 1957). Our approach is based on lexical richness of tweets from various Twitter accounts to detect social bots.

3.1 Type-Token Ratio

Type-Token Ratio (TTR) is a simple, yet powerful metric used commonly in quantitative linguistics to measure the richness of vocabulary in a given context (Tweedie and Baayen, 1998). TTR can be expressed as $V(N)/N$, i.e. the size of vocabulary in N divided by the total size of N. In the context of this study, TTR is the ratio of unique tokens in a tweet to the total number of tokens in the tweet. It can be argued that computing TTR over all tweets of an account may lead to a better result. However, the small size of individual tweets may skew the result due to the sparsity of unique tokens relative to the total number of tokens.

3.2 Lexical Diversity

Lexical Diversity is an important metric in the analysis of lexical richness. It is useful in assessing the distribution of different content words[2] utilised in a textual corpus or in speech (Tweedie and Baayen, 1998). Lexical sophistication is similar to lexical diversity and focuses on understanding the distribution of advanced words. This study focuses on lexical diversity. The rationale behind using it as a feature is to assess its levels in bot and human accounts and its predictive power in detection of social bot accounts. While the contents produced by social bots were shown to be different across various social bots (Morstatter et al., 2016), they tend to exhibit similarities in terms of widespread use of URLs. In view of this, we computed lexical diversity as the total number of tokens in a tweet without URLs, user mentions and stopwords divided by the total number of tokens in the tweet.

3.3 Usage of Contractions

Text or speech in English can be shortened by ignoring some letters or phonetics. These kinds of contracted words are a form of lexical sophistication, useful to measure the fluency of users. Various forms of contracted words are widespread on Twitter. However, we focus on a predefined list of contractions[3] for our analysis. The expectation is for human users to use diverse contractions, while it would be difficult for a bot to use contraction unless generating its tweet from pre-existing sources, e.g. a book or a structured document.

3.4 Emoticons

Emoticons[4] are collections of pictorial representations of facial expressions or *emojis* in form of various characters (letters, punctuation, and numbers) to convey emotional mood. Emoticons are popular on social media, especially on Twitter, where tweets are of limited length, are useful indicators or users' sentiment. Common examples of *emoticons* are the smiley, *:-)*, and the sad face, *:-(*. Sentiment-related features have been shown to contribute in detection systems (Dockerson et al., 2014). We leverage this to understand the role of *emoticons* in detection of bot accounts using a comprehensive list of emoticons[5]. The goal is to understand how human and social bot users apply emoticons in tweets and utilise the insight to build classification models. We hypothesise human users will use emoticons in a larger proportion in comparison to social bots accounts.

4. EXPERIMENT

This section describes the datasets utilised in this study, including the collection procedure and pre-processing. This is followed by feature selection and building the classification framework.

4.1 Dataset

[2] Words with meaning in a text; not in stopwords.
[3] Wikipedia list of contractions: en.wikipedia.org/wiki/Wikipedia:List_of_English_contractions
[4] A portmanteau of *emotion* and *icon*.
[5] Available from: en.wikipedia.org/wiki/Emoticon

We utilized three different datasets. The first two are publicly available, *Dataset1* is obtained from (Morstatter et al., 2016) and *Dataset2* from (Gilani et al., 2017. The last dataset is collected for the purpose of this study. Table 1 summarises the datasets. *Dataset2* is classified under five different groups based on popularity and volume of contents generated by the accounts. We maintain the groupings in the study and analyse the lexical richness in each group to understand how the lexical richness will vary across the different groups. The two datasets contain some non-English accounts, which were removed to facilitate proper lexical analysis of tweets. *Dataset1* provides only account *ids* and *Dataset2* provides screen-names and account features. We used the Twitter API to crawl tweets from each account. Our analysis experience with *Dataset1* and *Dataset2* reveals that most of the bot accounts are suspended and many accounts are not in English. In order to ensure genuine representation both from humans and bots accounts, we collected an additional dataset as follows.

Human accounts: We collected human user accounts who directly engage with the Twitter handles of organisations such as university and have correspondences in terms of replies to the users' queries. This is a useful technique to discount for bot accounts since bots may find it difficult to engage in meaningful conversations. To the best of our knowledge, this is the first study to employ this approach to ensure the genuineness of human users. For the *social bots accounts*, we collected 500 bot accounts using a publicly available bot detection system known as *Botometer*[6] (Davis et al., 2016). The *Botometer* returns a probable bot account which may result in many false positives. To mitigate that, we manually annotated the results of *Botometer*. The annotators scanned through the accounts and labelled account as bot based on the following criteria: (1) the account should be active, not suspended or deleted and posting tweets in English only (2) if the account's screen name appears to be auto-generated e.g. *2jo120, 2jo24 and 37Hkyjdtytyhjgh* (3) if the profile picture of the account shows no obvious relationship with the account's posts e.g. account tweeting on Brexit but with *storm-trooper* profile picture (4) number of URLs or hashtags: if the tweets mostly consist of URLs or hashtags exceeding 70% of the content (5) activity interval: posting at least 15 tweets per minute.

The annotation process is quite laborious which explain why the small size in *Dataset3*. We are not particularly interested in collecting a very high number of accounts but a high number of tweets from real human and bot accounts. We used the Twitter API to collect a maximum of 1000 tweets from each account.

Table 1. Summary of datasets utilised in the study. Dataset2 is categorised in groups based on the number of the followers in each group (*k* and *m* denote thousand and million respectively)

Category	***Bot Accounts***	***Human Accounts***	***Bot Tweets***	***Human Tweets***
Dataset1	685	641	27,766	5,341
Dataset2 1k	75	76	22,432	116,576
Dataset2 100k	266	343	112,387	98,271
Dataset2 1m	137	184	45,605	53,700
Dataset2 10m	25	25	9,062	11,869
Dataset3	100	100	83,976	74,483

4.2 Classification of Account as Bot or Human Using Lexical Features

We use machine learning to measure the extent at which these features aid in identifying bot accounts. We train a number of classifiers, namely K-nearest neighbour (KNN), Naive Bayes, Support Vector machine for Classification (SVC) and Random Forest on the three datasets. As shown in Table 3, three experiments were conducted for *Dataset2*: (1) using our lexical features, denoted as *L,* (2) using the features in Table 2 from (Gilani et al., 2017), denoted as *F, and* (3) a combination of (1) and (2), denoted as *FL*. The proposed lexical features (L) in this study are *TTR, lexical diversity, average number contractions and emoticons.*

The classifiers were built and trained using *scikit-learn*[7] (Pedregosa et al., 2011), a machine learning toolkit supported by Google and *INRIA*[8]. Stratified 10-fold cross-validation was used to measure the overall

[6] botometer.iuni.iu.edu/#!/api
[7] scikit-learn.org/stable
[8] inria.fr/en

accuracy, precision, recall and roc-auc score of each classifier. The Random Forest classifier performs best as shown in Table 4.

Table 2. Features used in *Dataset2*

Features
Age of account, Favourites-to-tweets ratio, Lists per user, Followers-to-friends ratio, User favourites, Likes/favourites per tweet, Retweets per tweet, User replies, User tweets, User retweets, Tweet frequency, URLs count, Activity source type[S1= browser, S2= mobile apps, S3= OSN management, S4= automation, S5= marketing, S6=news content, S0= all other], Source count, CDN content size

Table 3. Datasets and respective description of features utilised in training the prediction model

Dataset	***Description***
Dataset2_F	features in Table 2
Dataset2_FL	features from in Table 2 and our proposed lexical features (L)
Dataset2_L	proposed lexical features (L)
Dataset3_L	proposed lexical features on Dataset3

5. RESULTS AND DISCUSSION

The following section presents the main findings of the study. Figure 1 shows empirical evidence that the proposed lexical features are among the important features for the identification of automated accounts. Similarly, Figures 2, 3 and 4 show how lexical features manifest in humans and bot accounts.

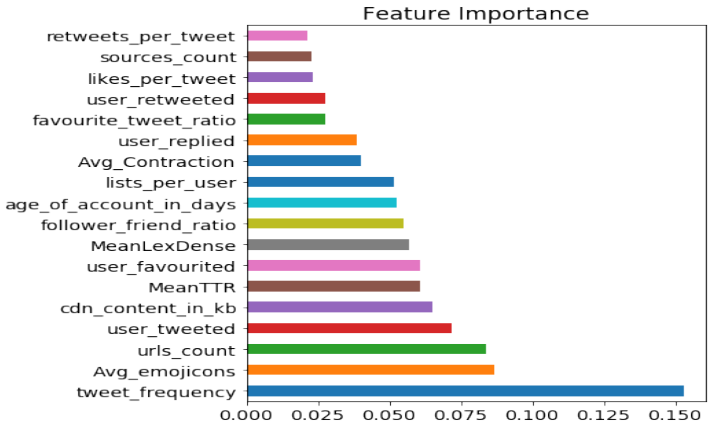

Figure 1. A comparison of importance of the proposed lexical features and features utilised in a related study

Lexical Diversity: Figure 2 shows the results of computing lexical diversity in human and bot accounts. *Lexical diversity* is expected to be higher in humans, since humans have been shown to generate better and novel content on Twitter (Gilani et al., 2017). However, this is not entirely true, especially in some automated accounts by prominent organisations, as shown in Figure 2. Accounts with a higher number of followers under the bot category are shown to have higher lexical diversities than the corresponding human counterparts. This is probably because such accounts are managed to update a large number of users on various topics on regular basis. Accounts in this category include organisational accounts, such as the BBC, politicians or popular celebrities. However, in *Dataset3*, humans have higher lexical densities which can be linked to the approach that was employed for the data collection. The dataset is a representative of an average user on Twitter. This confirms our earlier intuition that a typical human user account is expected to have a higher lexical diversity.

Usage of Contractions: Figure 3 shows how the usage of contraction varies across the datasets. With the exception of *Dataset3*, the difference in the use of contracted words between humans and bots is not very significant. This can be due to the fact that users with many followers on Twitter, such as organisational accounts or politicians, tend to use contracted words in tweets. With the exception of *Dataset3*, the difference in the use of contracted words between humans and bots is not very significant. This can be attributed to the

fact that users with a high number of followership on Twitter will tend to use contracted words, e.g. (*Dataset2 1M and Dataset2 10M*), which mainly contain tweets of organisations, celebrities or politicians.

Usage of Emoticons: We found that the usage of *emoticons* is higher in bot accounts than in human accounts across all the different datasets as shown in Figure 4. Surprisingly, the results suggest that bot accounts utilise more *emoticons* in their tweets than humans. This is contrary to our prior intuition that humans are more likely to use more emoticons.

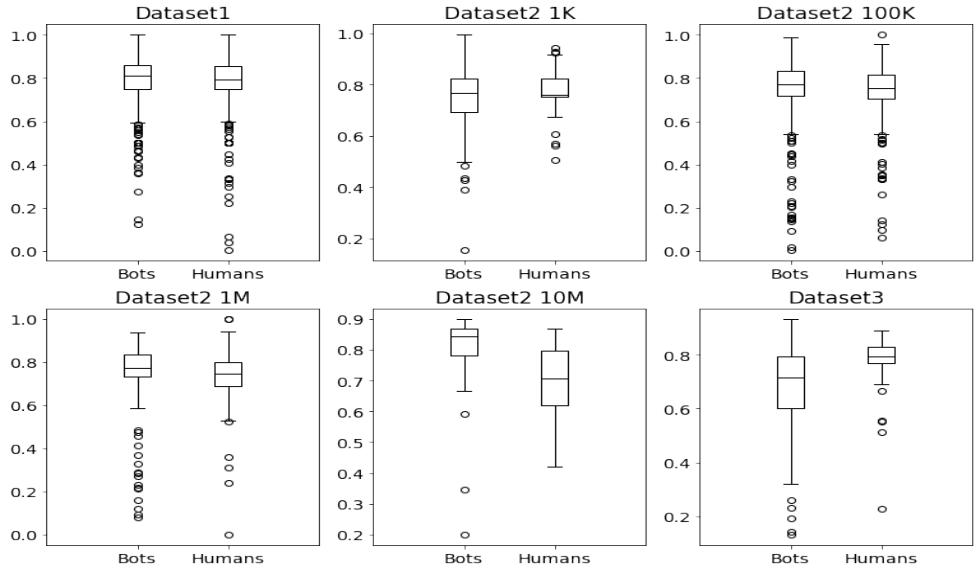

Figure 2. Average *lexical diversities* of *human* and *social bot* accounts

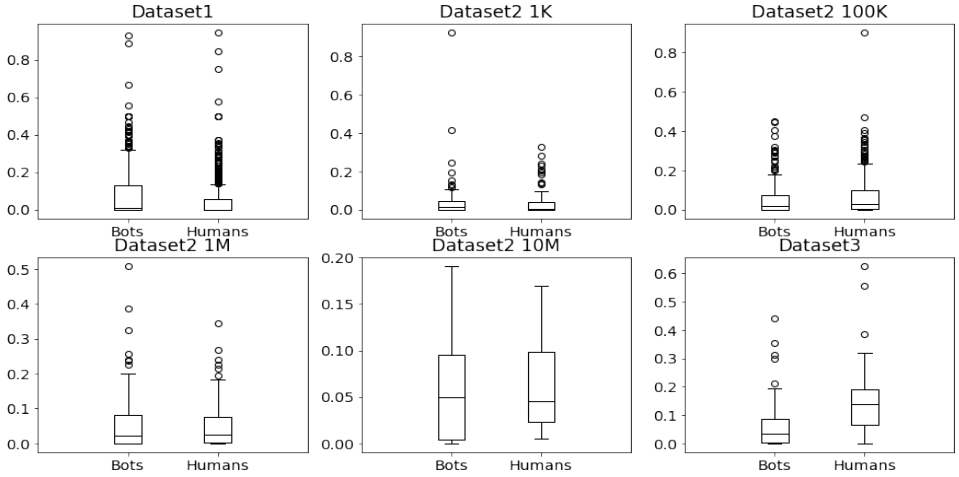

Figure 3. Average Contractions across datasets utilised in this study

Classification of bot and human accounts using lexical features: We use machine learning algorithms to measure the extent at which our proposed lexical features aid detection of bot accounts. Table 4 shows the results of a trained random forest classifier across the datasets. Using the lexical features only we achieve an accuracy of 86% and AUC score of 87% in Dataset_3. In Dataset2_FL we achieve an AUC score of 95% which is a significant improvement over 71% achieved using only the features utilised in (Gilani et al., 2017).

The primary goal is to improve detection of bot accounts by adding lexical features into the detection system. *Emoticons* happen to be the most distinctive features between humans and bots. In Figure 4 we observe an agreement in the usage of *emoticons* in all the datasets, i.e. bots use *emoticons* more frequently than humans. It is evident from the classification result (Table 4) that lexical features significantly improve the performance of the detection system. The distinguishing power of *lexical features* appears to be less

effective in *Dataset1* and *Datatset2* in relation to *Dataset3*. With extra measures during data collection and annotation, this effect can be mitigated. This is evident from *Dataset3* which was manually inspected, and emphasis should be placed in the collection of more robust and representative datasets for effective detection. Despite the variations, which we attribute to many false positives in *Dataset1* and *Datatset2*, *lexical features* prove to be strong indicators as shown in Figure 1. The figure shows that *emoticons (*captioned as *avg_emojicons)* is the second most important feature among the 18 features. The lowest performing of our proposed lexical features (*avg_contraction*) outperforms 6 features utilised in a related study.

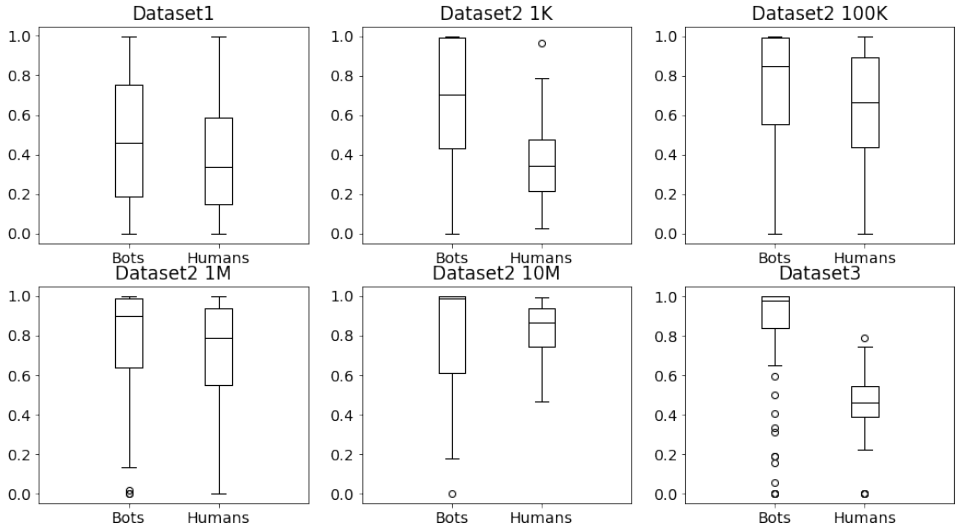

Figure 4. Average *emoticons* in the different datasets

Table 4. Datasets and respective prediction performances

Dataset	***Accuracy***	***Precision***	***Recall***	***AUC Score***
Dataset1_L	0.65	0.65	0.65	0.65
Dateset2_F	0.71	0.72	0.72	0.71
Dateset2_L	0.66	0.67	0.67	0.66
Dataset2_FL	**0.95**	**0.96**	**0.96**	**0.95**
Dataset3_L	**0.86**	**0.87**	**0.87**	**0.87**

6. CONCLUSION

Modern day social platforms have become part of our lives and effective social policing is required to ensure data credibility and civilised way of interaction. However, with the growing sophistication level of social bots, it is proving difficult to sanitise social platforms. The continuous increase in real-time streaming of tweets makes it practically ineffective to rely on many account features for detection. To effectively distinguish between a bots and a human user, an analysis of lexical richness of tweets posted by both users provides additional distinctive features. We train diverse classifiers to evaluate the role of lexical features in the detection of bot accounts. The newly proposed features significantly improve detection accuracy.

In this study, we have shown the difference between humans and bots in terms of lexical diversity, usage of contraction and emoticons, based on three different datasets. Lexical diversity and contractions vary across the different datasets. Contrary to our intuition, social bots accounts utilise a large number of emoticons in comparison to human accounts. We consider only English tweets as we do not focus on how lexical features are applied to other languages. Further investigation on this will improve the universality of our approach.

ACKNOWLEDGEMENT

The authors wish to thank Prof. Reiko Heckel of University of Leicester for his positive feedback. The third author has participated in this research work as part of the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732223.

REFERENCES

Benevenuto, F., Magno, G., Rodrigues, T., & Almeida, V. (2010, July). Detecting spammers on twitter. In *Collaboration, electronic messaging, anti-abuse and spam conference (CEAS)* (Vol. 6, No. 2010, p. 12).

Cai, C., Li, L., & Zengi, D. (2017, July). Behavior enhanced deep bot detection in social media. In *Intelligence and Security Informatics (ISI), 2017 IEEE International Conference on* (pp. 128-130). IEEE.

Chavoshi, N., Hamooni, H., & Mueen, A. (2017, April). Temporal patterns in bot activities. In *Proceedings of the 26th International Conference on World Wide Web Companion* (pp. 1601-1606). International World Wide Web Conferences Steering Committee.

Davis, C. A., Varol, O., Ferrara, E., Flammini, A., & Menczer, F. (2016, April). Botornot: A system to evaluate social bots. In *Proceedings of the 25th International Conference Companion on World Wide Web* (pp. 273-274). International World Wide Web Conferences Steering Committee.

Dickerson, J. P., Kagan, V., & Subrahmanian, V. S. (2014, August). Using sentiment to detect bots on twitter: Are humans more opinionated than bots?. In *Proceedings of the 2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining* (pp. 620-627). IEEE Press.

Ferrara, E., Varol, O., Davis, C., Menczer, F., & Flammini, A. (2016). The rise of social bots. *Communications of the ACM*, *59*(7), 96-104.

Gilani, Z., Farahbakhsh, R., Tyson, G., Wang, L., & Crowcroft, J. (2017, July). Of Bots and Humans (on Twitter). In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*(pp. 349-354). ACM.

Howard, P. N., & Kollanyi, B. (2016). Bots,# StrongerIn, and# Brexit: computational propaganda during the UK-EU referendum.

Inuwa-Dutse, I., Liptrott, M., & Korkontzelos, I. (2018). Detection of spam-posting accounts on Twitter. *Neurocomputing*.

Lee, K., Eoff, B. D., & Caverlee, J. (2011, July). Seven Months with the Devils: A Long-Term Study of Content Polluters on Twitter. In *ICWSM* (pp. 185-192).

Lee, S., & Kim, J. (2012, February). WarningBird: Detecting Suspicious URLs in Twitter Stream. In *NDSS* (Vol. 12, pp. 1-13).

Morstatter, F., Wu, L., Nazer, T. H., Carley, K. M., & Liu, H. (2016, August). A new approach to bot detection: striking the balance between precision and recall. In *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining* (pp. 533-540). IEEE Press.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. *Journal of machine learning research*, *12*(Oct), 2825-2830.

Šišková, Z. (2012). Lexical richness in EFL students' narratives. *Language Studies Working Papers*, *4*, 26-36.

Subrahmanian, V. S., Azaria, A., Durst, S., Kagan, V., Galstyan, A., Lerman, K., ... & Stevens, A. (2016). The DARPA Twitter bot challenge. *arXiv preprint arXiv:1601.05140*.

Templin, M. C. (1957). Certain language skills in children; their development and interrelationships.

Thomas, K., Grier, C., Ma, J., Paxson, V., & Song, D. (2011, May). Design and evaluation of a real-time url spam filtering service. In *Security and Privacy (SP), 2011 IEEE Symposium on* (pp. 447-462). IEEE.

Tweedie, F. J., & Baayen, R. H. (1998). How variable may a constant be? Measures of lexical richness in perspective. *Computers and the Humanities*, *32*(5), 323-352.

Varol, O., Ferrara, E., Davis, C. A., Menczer, F., & Flammini, A. (2017). Online human-bot interactions: Detection, estimation, and characterization. *arXiv preprint arXiv:1703.03107*.

Wang, A. H. (2010, July). Don't follow me: Spam detection in twitter. In *Security and cryptography (SECRYPT), proceedings of the 2010 international conference on* (pp. 1-10). IEEE.

Wilson, C., Sala, A., Puttaswamy, K. P., & Zhao, B. Y. (2012). Beyond social graphs: User interactions in online social networks and their implications. *ACM Transactions on the Web (TWEB)*, *6*(4), 17.

IADIS International Journal on WWW/Internet
Vol. 16, No. 2, pp. 50-65
ISSN: 1645-7641

THE EFFECT OF ENGAGEMENT INTENSITY AND LEXICAL RICHNESS IN IDENTIFYING *BOT* ACCOUNTS ON TWITTER

Isa Inuwa-Dutse[1], Bello Shehu Bello[2], and Ioannis Korkontzelos[1]
[1]*Department of Computer Science , Edge Hill University, UK*
[2]*Department of Informatics, University of Leicester, UK*

ABSTRACT

The rise in the number of automated or bot accounts on Twitter engaging in manipulative behaviour is of great concern to studies using social media as a primary data source. Many strategies have been proposed and implemented, however, the sophistication and rate of deployment of bot accounts is increasing rapidly. This impedes and limits the capabilities of detecting bot strategies. Various features broadly related to account profiles, tweet content, network and temporal patterns have been utilised in detection systems. Tweet content has been proven instrumental in this process, but limited to the terms and entities occurring. Given a set of tweets with no obvious pattern, can we distinguish contents produced by social bots from those of humans? What constitutes engagement on Twitter and how can we measure the intensity of engagement? Can we distinguish between *bot* and *human* accounts based on *engagement intensity?* These are important questions whose answer will improve how detection systems operate to combat malicious activities by effectively distinguishing between human and social bot accounts on Twitter. This study attempts to answer these questions by analysing the engagement intensity and lexical richness of tweets produced by human and social bot accounts using large, diverse datasets. Our results show a clear margin between the two classes in terms of engagement intensity and lexical richness. We found that it is extremely rare for a social bot to engage meaningfully with other users and that lexical features significantly improve the performance of classifying both account types. These are important dimensions to explore toward improving the effectiveness of detection systems in combating the menace of social bot accounts on Twitter.

KEYWORDS

Twitter bots, Automated Accounts, Engagement, Lexical Richness, Bot Detection, Social Network Analysis

1. INTRODUCTION

The behavior of humans, as social beings, is largely influenced by close associates. In this era of modern civilisation, the Internet has been the precursor of the socialisation being witnessed today. Social networking sites are important avenues for instant information sharing and interaction on a large scale (Gilani et al., 2017; Wilson et al., 2012). Modern social media platforms, such as Twitter and Facebook, enable various forms of interactions among diverse users. This capability results in a huge amount of data to be analysed by researchers. However, the credibility of content is questionable, due to the rising levels of activities of automated accounts, also known as social bots. *Social Bots* are automated accounts designed to follow well-defined algorithms to interact with users, either by amplifying content or generating new content in social media platforms. A more elaborate definition is provided by Akomoto (2011):

> "*Twitter bots are small software programs that are designed to mimic human tweets. Anyone can create bots, though it usually requires programming knowledge. Some bots reply to other users when they detect specific keywords. Others may randomly tweet pre-set phrases such as proverbs. Or if the bot is designed to emulate a popular person (celebrity, historic icon, anime character etc.) their popular phrases will be tweeted. Not all bots are fully machine generated, however, and interestingly the term "bot" has also come to refer to Twitter accounts that are simply "fake" accounts*"

Some social bot accounts, such as *@congressedits*[1], are legitimate, with clear distinguishing features whereas the majority are created to mislead in various ways, such as by creating superficial popularity (Varol et al., 2017) or influencing public opinion (Howard and Kollanyi, 2016). Some bots are obvious to identify, because they use the word "*bot*" explicitly in their Twitter handle (Dutse et al., 2018). Autonomous accounts contribute a sizeable part of social media content. It was estimated that 9% - 15% of active Twitter accounts are social bots accounts (Varol et al., 2017) and require effective methods to be detected.

How could we detect if a given tweet was posted by a social bot or a human user? We investigate this question based on a collection of tweets sampled from Twitter. In particular, we investigate how effective lexical features are for the detection of social bot accounts. In contrast to previous work (Benevenuto et al., 2010; Cai t al., 2017; Dockerson et al., 2014; Lee and Kim, 2012; Thomas et al., 2011; Varol et al., 2017; Wang, 2010), our study focuses on comprehensive linguistic analysis to define lexical features effective enough for accurate detection of social bot accounts on Twitter. As shown in Table 1, the study explores a large number of tweets from social bots and humans to understand the difference between the two in terms of lexical richness and distribution of emoticons, further discussed in section 3. Our analysis highlights the distinguishing characteristics of automated accounts and how lexical features can improve detection of bots. Furthermore, we analyse the level of engagement exhibited by social bot and human accounts and its role in building detection systems. Our analysis reveals that bots rarely engage into discussions meaningfully. Human users reply much more often, while social bots show retweet more often than humans in pursue of engagement. Further, we show how lexical features play a role in the detection task and how engagement and lexical features can be combined to develop a powerful detection system on Twitter. We evaluate our approach using a dataset that consists of both English and non-English tweets.

[1] A bot that tweets anonymous Wikipedia edits made from IP addresses in the US Congress.

Contribution: This is the first study to analyse the role of engagement intensity and lexical features for the detection of automated accounts. The study contributes a powerful set of features useful in distinguishing between humans and social bots on Twitter. We provide the first comprehensive analysis of both engagement intensity and lexical richness of tweets computed on various accounts and investigate how their incorporation improves the performance of the detection system. Specifically, engagement intensity and features based on *lexical diversity, type-token ratio, usage of contractions and emoticons* are powerful lexical signals in distinguishing between humans and social bots. The study provides a new set of distinctive features and a dataset to support the research community in identifying bot accounts.

The remaining of the paper is structured as follows. Section 2 and Section 3 present a review of related works and propose engagement & lexical features, respectively. Section 4 presents our experiments and Section 5 presents the results and a detailed discussion. Finally, Section 6 concludes the study and proposes some future work.

2. RELATED WORK

Social bot accounts are instrumental in spreading fake and malicious news on Twitter, employed to skew analysis results and opinion of users. The demand for effective detection systems has prompted a surge of various research approaches. The early work of Wang (2010) focuses on a machine learning approach for detection of a spam bot accounts on Twitter using features that range from graph-based to content-based features. Since the publication, *bots* gain sophistication to evade detection. Early social bot accounts have been reported to lack basic account information such as meaningful screen names or profile pictures (Varol et al., 2017; Lee et al., 2011). This is no longer the case, as social bots grow in sophistication, making it difficult to identify distinguishing features from human accounts. Some approaches involve the analysis of accounts as far as their position in a *network*, their *temporal* metadata and the *content of their tweets*, to define a new set of features. The following review focuses on related studies that utilised aspects of these features to detect bot accounts.

Network and Temporal Features: Motivated by the rise in automated accounts on Twitter, which often confuse users regarding their legitimacy, Chu et al. (2012) presents a method to detect fully automated accounts and partially automated accounts orchestrated by a human user (otherwise known as *Cyborg).* The study relies on account activities in terms of *posting behaviour, tweet content* and *account properties* to extract useful features for classification. *Tweet content features* should not be confused with *lexical features*, which have been shown to significantly improve the performance of detection systems (Dutse et al, 2018). This study improves on the effectiveness of *lexical features* by focusing on *engagement features.* The work of Davis et al. (2016) and (Ferrara et al. (2016) developed a detection system that leverages features related to both network structure and tweeting behaviour exhibited by accounts. Subrahmanian et al. (2016) reports how researchers utilised many features related to network, temporal behaviour, tweet syntax, tweet semantics and user profile for the detection of *influence bots*. Influence bots are categories of social bot accounts that aim at influencing the opinion of other users. However, despite the wide spectrum of features considered in the study, analysis of the lexical richness was not covered. The study of Chavoshi et al. (2017) analyses the behavioural patterns of accounts by focusing on features related to the network structure, such as local motifs, i.e. repeating behaviour, and discords, i.e. anomalous behaviour. The study also shows how temporal behaviour is useful as a means to distinguish bot from human user

accounts. Haustein et al. (2017) analyses the impact of a category of automated accounts on Twitter that focuses on distributing links to scientific articles to promote research impact leading to misuse due to superficial generation of huge contents. Such accounts were found to generate large number of tweets that could skew analysis results.

Tweet Content: Contents generated by users have been analyse to extract meaningful features such as *entropy* for detection. *Entropy* measures the complexity of a system in terms of randomness and *social bots* have been shown to exhibit less randomness (Sutton et al. 2008). However, malicious bots are increasingly becoming more sophisticated and disruptive. A typical bot account has been shown to generate many benign tweets pertaining to news and updates on feeds or employing a cunning way to balance *follower-followee* ratio (Chu et al., 2012). Many detection systems have been developed by leveraging the content of tweets posted by account holders on Twitter. This approach was adopted in Dockerson et al. (2014) to detect social bot accounts on Twitter based on sentiment features, such as topic sentiment and the transition frequency of tweet's sentiment, to train a machine learning classifier. Similarly, to the *tweet content* approach that relies on linguistic analysis, Inuwa-Dutse et al. (2018) utilised features based on lexical richness to detect spam accounts on Twitter. The study of (Cai et al., 2017) proposed a deep learning approach that incorporates content and behavioural information to detect social bots.

The large body of *bot detection* work exploring various detection strategies attests to the significance of the problem. Related literature pays little attention to the analysis of lexical richness of various users' tweets and how they can inform the detection of social bots on Twitter. In contrast to previous work, this study is based on in-depth analysis of engagement intensity and lexical richness as basis for building a detection system. This study contributes effective features and strategy to help in combating the menace of malicious automated accounts on Twitter.

3. FEATURES: LEXICAL AND ENGAGEMENT

This section describes the lexical and engagement features utilised in the study, able to improve detection systems.

3.1 Lexical Features

Lexical richness is a broad concept, expressed in various forms and metrics to assess the quality of text. Metrics such as lexical diversity, lexical sophistication and lexical density are commonly used in linguistic analysis (Šišková, 2012; Templin, 1957). Our approach is based on lexical richness of tweets from various Twitter accounts to detect social bots.

3.1.1 Type-Token Ratio

Type-Token Ratio (TTR) is a simple, yet powerful metric used commonly in quantitative linguistics to measure the richness of vocabulary in a given context (Tweedie and Baayen, 1998). TTR can be expressed as $V(N)/N$, i.e. the size of vocabulary in N divided by the total size of N. In the context of this study, TTR is the ratio of unique tokens in a tweet to the total number of tokens in the tweet. It can be argued that computing TTR over all tweets of an account may lead to a better result. However, the small size of individual tweets may skew the result due to the sparsity of unique tokens relative to the total number of tokens.

3.1.2 Lexical Diversity

Lexical Diversity is an important metric in the analysis of lexical richness. It is useful in assessing the distribution of different content words[2] utilised in a textual corpus or in speech (Tweedie and Baayen, 1998). Lexical sophistication is similar to lexical diversity and focuses on understanding the distribution of advanced words. This study focuses on lexical diversity. The rationale behind using it as a feature is to assess its levels in bot and human accounts and its predictive power in detection of social bot accounts. While the contents produced by social bots were shown to be different across various social bots (Morstatter et al., 2016), they tend to exhibit similarities in terms of widespread use of URLs. In view of this, we computed lexical diversity as the total number of tokens in a tweet without URLs, user mentions and stopwords divided by the total number of tokens in the tweet.

3.1.3 Usage of Contractions

Text or speech in English can be shortened by ignoring some letters or phonetics. These kinds of contracted words are a form of lexical sophistication, useful to measure the fluency of users. Various forms of contracted words are widespread on Twitter. However, we focus on a predefined list of contractions[3] for our analysis. The expectation is for human users to use diverse contractions, while it would be difficult for a bot to use contraction unless generating its tweet from pre-existing sources, e.g. a book or a structured document.

3.1.4 Emoticons

Emoticons[4] are collections of pictorial representations of facial expressions or *emojis* in form of various characters (letters, punctuation, and numbers) to convey emotional mood. Emoticons are popular on social media, especially on Twitter, where tweets are of limited length, are useful indicators or users' sentiment. Common examples of *emoticons* are the smiley, *:-)*, and the sad face, *:-(*. Sentiment-related features have been shown to contribute in detection systems (Dockerson et al., 2014). We leverage this to understand the role of *emoticons* in detection of bot accounts using a comprehensive list of emoticons[5]. The goal is to understand how human and social bot users apply emoticons in tweets and utilise the insight to build classification models. We hypothesise human users will use emoticons in a larger proportion in comparison to social bots accounts.

3.2 Engagement Intensity

Many forms of interactions happen on Twitter at various level of granularity. Interaction with other users comes in the form of simple retweet, likes, reply and direct messages. We defined the engagement of users on Twitter into four different levels and analyses the levels exhibit by both humans and automated accounts.

The following features (also detailed in Table 2) are used constitute the engagement levels: *retweet (RT) count, tweet favourite count, reply* and *user mention.* The *engagement levels* are defined as follows:

[2] Words with meaning in a text; not in stopwords.
[3] Wikipedia list of contractions: en.wikipedia.org/wiki/Wikipedia:List_of_English_contractions
[4] A portmanteau of *emotion* and *icon*.
[5] Available from: en.wikipedia.org/wiki/Emoticon

- *unengaged*: none of the engagement feature
- *low engagement*: only one engagement feature. For instance, *mentioning* many users (which may or may not imply engagement)
- *moderate engagement*: any two engagement features
- *high engagement*: all engagement features i.e. *retweet (RT) count, tweet favourite count, reply* and *user mention.*

4. EXPERIMENT

This section describes the datasets utilised in this study, including the collection procedure and pre-processing. This is followed by feature selection and building the classification framework.

4.1 Dataset

We utilized three different datasets. The first two are publicly available, *Dataset1* is obtained from (Morstatter et al., 2016) and *Dataset2* from (Gilani et al., 2017. The last dataset is collected for the purpose of this study. Table 1 summarises the datasets. *Dataset2* is classified under five different groups based on popularity and volume of contents generated by the accounts. We maintain the groupings in the study and analyse the lexical richness in each group to understand how the lexical richness will vary across the different groups. The two datasets contain some non-English accounts, which were removed to facilitate proper lexical analysis of tweets. *Dataset1* provides only account *ids* and *Dataset2* provides screen-names and account features. We used the Twitter API to crawl tweets from each account. Our analysis experience with *Dataset1* and *Dataset2* reveals that most of the bot accounts are suspended and many accounts are not in English. In order to ensure genuine representation both from humans and bots accounts, we collected an additional dataset as follows.

Human accounts: We collected human user accounts who directly engage with the Twitter handles of organisations such as university and have correspondences in terms of replies to the users' queries. This is a useful technique to discount for bot accounts since bots may find it difficult to engage in meaningful conversations. To the best of our knowledge, this is the first study to employ this approach to ensure the genuineness of human users. For the *social bots accounts*, we collected 500 bot accounts using a publicly available bot detection system known as *Botometer*[6] (Davis et al., 2016). The *Botometer* returns a probable bot account which may result in many false positives. To mitigate that, we manually annotated the results of *Botometer*. The annotators scanned through the accounts and labelled account as bot based on the following criteria: (1) the account should be active, not suspended or deleted and posting tweets in English only (2) if the account's screen name appears to be auto-generated e.g. *2jo120, 2jo24 and 37Hkyjdtytyhjgh* (3) if the profile picture of the account shows no obvious relationship with the account's posts e.g. account tweeting on Brexit but with *storm-trooper* profile picture (4) number of URLs or hashtags: if the tweets mostly consist of URLs or hashtags exceeding 70% of the content (5) activity interval: posting at least 15 tweets per minute.

[6] botometer.iuni.iu.edu/#!/api

The annotation process is quite laborious which explain why the small size in *Dataset3*. We are not particularly interested in collecting a very high number of accounts but a high number of tweets from real human and bot accounts. We used the Twitter API to collect a maximum of 1000 tweets from each account.

Table 1. Summary of datasets utilised in the study. Dataset2 is categorised in groups based on the number of the followers in each group (*k* and *m* denote thousand and million respectively)

Category	***Bot Accounts***	***Human Accounts***	***Bot Tweets***	***Human Tweets***
Dataset1	685	641	27,766	5,341
Dataset2 1k	75	76	22,432	116,576
Dataset2100k	266	343	112,387	98,271
Dataset2 1m	137	184	45,605	53,700
Dateset2 10m	25	25	9,062	11,869
Dataset3	100	100	83,976	74,483
Dataset4	1763	1832	693655	1337394

Table 2 describes the various features utilised in the study. The proposed engagement intensity features are valid across all datasets. Table A1 in appendix shows examples of common languages in the non-English dataset (*Dataset4).*

4.2 Use of Engagement and Lexical Features for Classification

For learning purposes, standard machine learning models have been applied to distinguish between *bot* and human based on the set of handcrafted features (Table 2). Deep learning approach have been applied in Cai et al. (2017) for *bot* detection and the method automatically extract features to use. Noting the growing sophistication level of *bot*, we view this approach as limiting the features space to explore. By carefully analysing contents from both human and bot, more effective detection mechanism can be achieved using handcrafted features. This is evidenced in Section 5. Various classification models have been trained to measure the extent at which these features aid in identifying bot accounts. Classifiers utilised for training include K-nearest neighbour (KNN), Naive Bayes, Support Vector machine for Classification (SVC) and Random Forest on the three datasets. The classifiers were built and trained using *scikit-learn*[7] (Pedregosa et al., 2011), a machine learning toolkit supported by Google and *INRIA*[8]. Stratified 10-fold cross-validation was used to measure the overall *accuracy, precision, recall and roc-auc score* of each classifier. The Random Forest classifier performs best as shown in Table 4. As shown in Table 3, three experiments were conducted for *Dataset2*: (1) using our lexical features, denoted as *L,* (2) using the features in Table 2 from (Gilani et al., 2017), denoted as *F, and* (3) a combination of (1) and (2), denoted as *FL*. The proposed lexical features (L) in this study are *TTR, lexical diversity, average number contractions and emoticons.*

[7] scikit-learn.org/stable
[8] inria.fr/en

Table 2. Description of features used in the study

Features	Description
Age of account, Favourites-to-tweets ratio, Lists per user, Followers-to-friends ratio, User favourites, Likes/favourites per tweet, Retweets per tweet, User replies, User tweets, User retweets, Tweet frequency, URLs count, Activity source type[S1= browser, S2= mobile apps, S3= OSN management, S4= automation, S5= marketing, S6=news content, S0= all other], Source count, CDN content size	This feature set was first used in Gilani et al, 2017 and is denoted as *F*
retweet (RT) count, tweet favourite count, reply and user mention. The engagement intensities/levels are defined as follows:	Engagement intensity features are denoted as *E* and are categorised based on intensity: low engagement, moderate engagement and high engagement.
lexical diversity, type-token ratio, usage of contractions and emoticons.	Lexical richness features are denoted as *L*. Thus, *FLE* refers to combination of all the three features sets.

Table 3. Datasets and respective description of features utilised in training the prediction model

Dataset	Description
Dataset2_F	Features in Table 2
Dataset2_FL	features from in Table 2 and our proposed lexical features (L)
Dataset2_L	Training Dataset2 on proposed lexical features (L)
Dataset3_L	Proposed lexical features on Dataset3
Dataset4_LE	Combination lexical and engagement features on Dataset4

5. RESULTS AND DISCUSSION

The following section presents the main findings of the study. Figure 1 shows empirical evidence that the proposed lexical features are among the important features for the identification of automated accounts. Similarly, Figures 2, 3 and 4 show how lexical features manifest in humans and bot accounts.

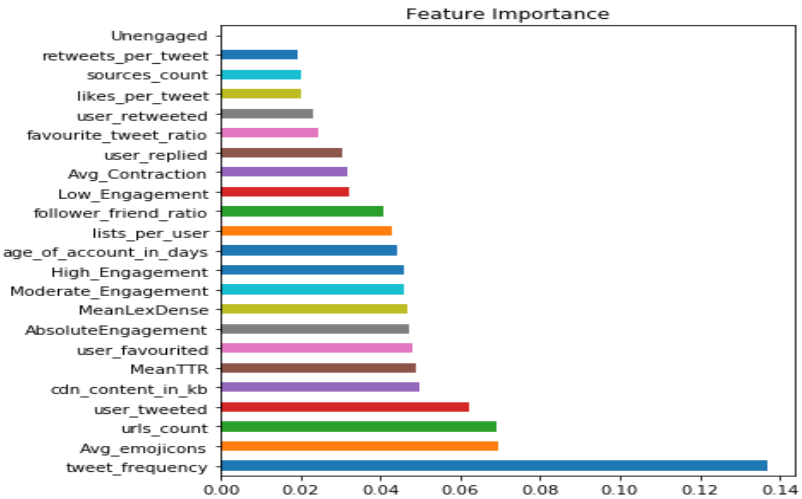

Figure 1. A comparison of importance of the proposed lexical features and features utilised in a related study

Lexical Diversity: Figure 2 shows the results of computing lexical diversity in human and bot accounts. *Lexical diversity* is expected to be higher in humans, since humans have been shown to generate better and novel content on Twitter (Gilani et al., 2017). However, this is not entirely true, especially in some automated accounts by prominent organisations, as shown in Figure 2. Accounts with a higher number of followers under the bot category are shown to have higher lexical diversities than the corresponding human counterparts. This is probably because such accounts are managed to update a large number of users on various topics on regular basis. Accounts in this category include organisational accounts, such as the BBC, politicians or popular celebrities. However, in *Dataset3*, humans have higher lexical densities which can be linked to the approach that was employed for the data collection. The dataset is a representative of an average user on Twitter. This confirms our earlier intuition that a typical human user account is expected to have a higher lexical diversity.

Usage of Contractions: Figure 3 shows how the usage of contraction varies across the datasets. With the exception of *Dataset3*, the difference in the use of contracted words between humans and bots is not very significant. This can be due to the fact that users with many followers on Twitter, such as organisational accounts or politicians, tend to use contracted words in tweets. With the exception of *Dataset3*, the difference in the use of contracted words between humans and bots is not very significant. This can be attributed to the fact that users with a high number of followership on Twitter will tend to use contracted words, e.g. (*Dataset2 1M and Dataset2 10M*), which mainly contain tweets of organisations, celebrities or politicians.

Usage of Emoticons: We found that the usage of *emoticons* is higher in bot accounts than in human accounts across all the different datasets as shown in Figure 4. Surprisingly, the results suggest that bot accounts utilise more *emoticons* in their tweets than humans. This is contrary to our prior intuition that humans are more likely to use more emoticons. The primary goal is to improve detection of bot accounts by adding lexical features into the detection system. *Emoticons* happen to be the most distinctive features between humans and bots. In Figure 4 we observe an agreement in the usage of *emoticons* in all the datasets, i.e. bots use *emoticons* more frequently than humans. It is evident from the classification result (Table 4) that lexical features significantly improve the performance of the detection system. The distinguishing power of *lexical features* appears to be less effective in *Dataset1* and *Datatset2* in relation to *Dataset3*. With extra measures during data collection and annotation, this effect can be mitigated. This is evident from *Dataset3* which was manually inspected, and emphasis should be placed in the collection of more robust and representative datasets for effective detection. Despite the variations, which we attribute to many false positives in *Dataset1* and Dataset2, *lexical features* prove to be strong indicators as shown in Figure 1. The figure shows that *emoticons (*captioned as *avg_emojicons)* is the second most important feature among the 18 features. The lowest performing of our proposed lexical features (*avg_contraction*) outperforms 6 features utilised in a related study.

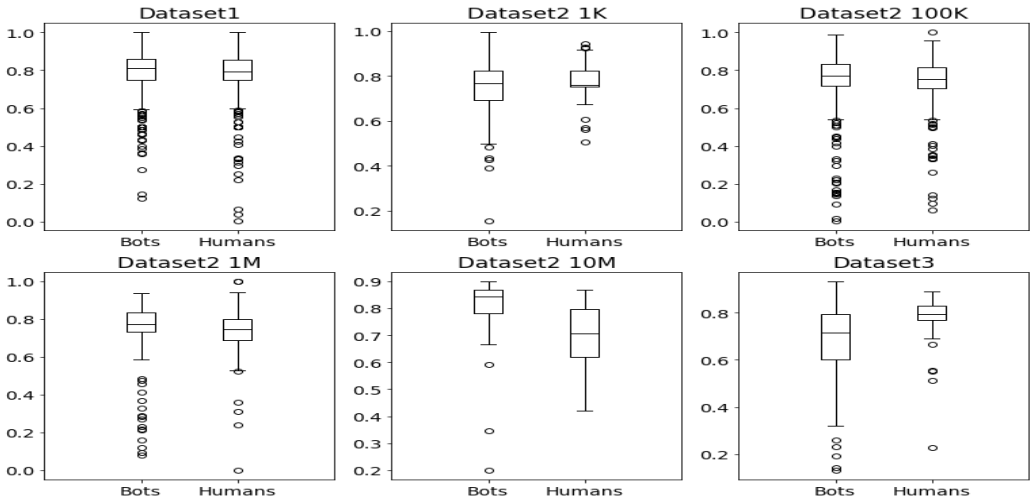

Figure 2. Average *lexical diversities* of *human* and *social bot* accounts

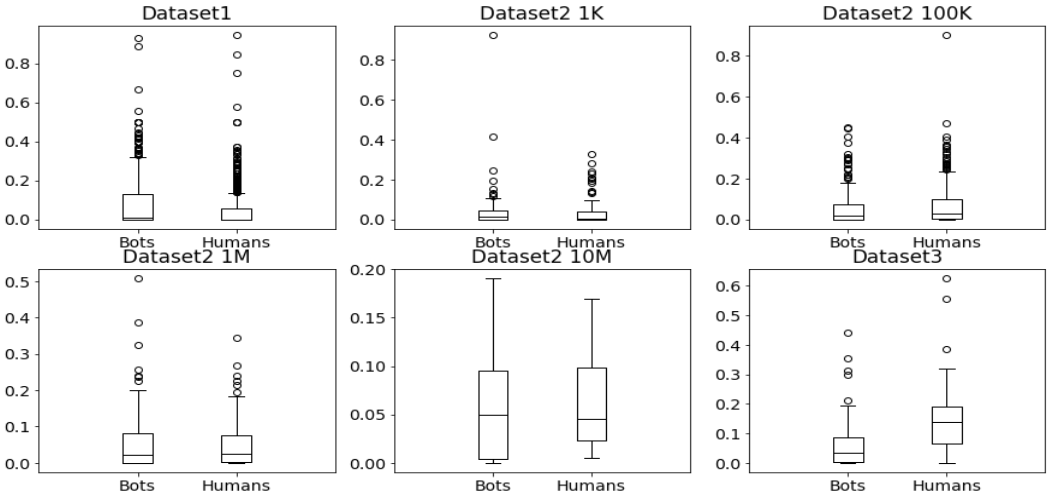

Figure 3. Average Contractions across datasets utilised in this study

Effect of engagement and lexical features on classification of bot and human accounts: We use machine learning algorithms to measure the extent at which our proposed lexical features aid detection of bot accounts. Table 4 shows the results of a trained random forest classifier across the datasets. Using the lexical features only we achieve an accuracy of 86% and AUC score of 87% in Dataset3. In Dataset2_FL we achieve an AUC score of 95% which is a significant improvement over 71% achieved using only the features utilised in (Gilani et al., 2017).

Table 4. Datasets and respective prediction performances

Dataset	***Accuracy***	***Precision***	***Recall***	***AUC Score***
Dataset1_L	0.65	0.65	0.65	0.65
Dataset2_F	0.71	0.72	0.72	0.71
Dataset2_L	0.66	0.67	0.67	0.66
Dataset2_FL	**0.95**	**0.96**	**0.96**	**0.95**
Dataset3_L	**0.86**	**0.87**	**0.87**	**0.87**
Dataset4_EL	**0.95**	**0.96**	**0.96**	**0.95**

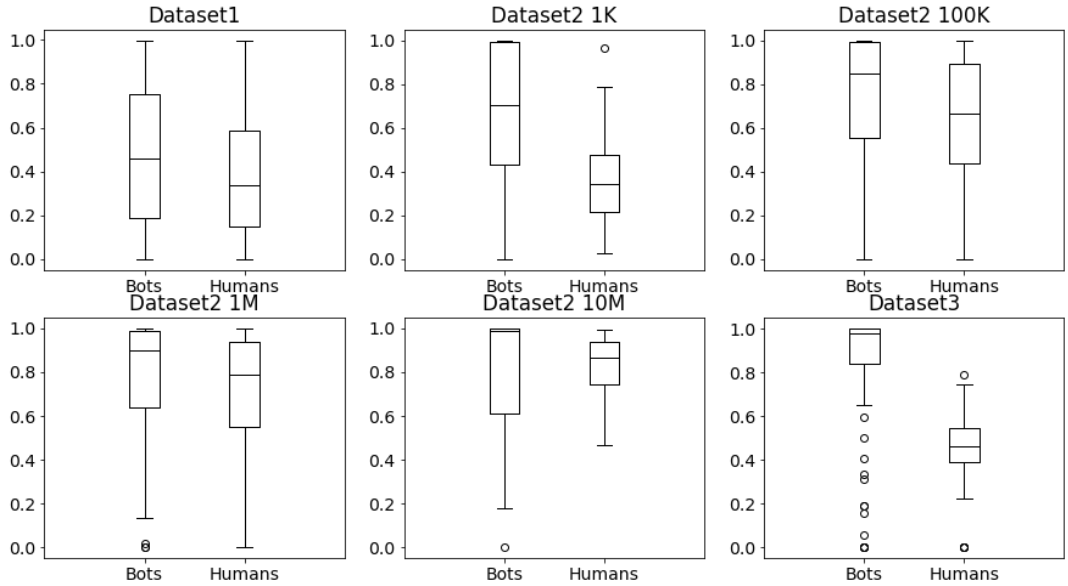

Figure 4. Average *emoticons* in the different datasets

The introduction of the engagement features can be seen to be influential. Table 4 shows the significant improvement in performance (see also Figure A1 and Figure A2 in appendix section). Figure 5 and Figure 6 shows the proportions of engagement features: low, moderate and high intensities from sampled accounts.

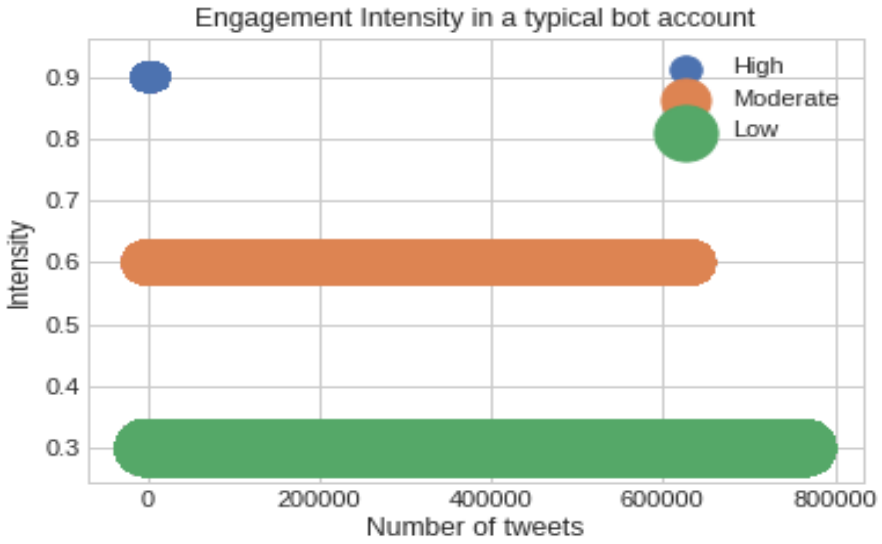

Figure 5. This figure shows the proportions per engagement intensity for bot accounts. There is a small number of bots that exhibit meaningful engagement. These accounts correspond to credible organisations that use automated accounts to respond to basic queries

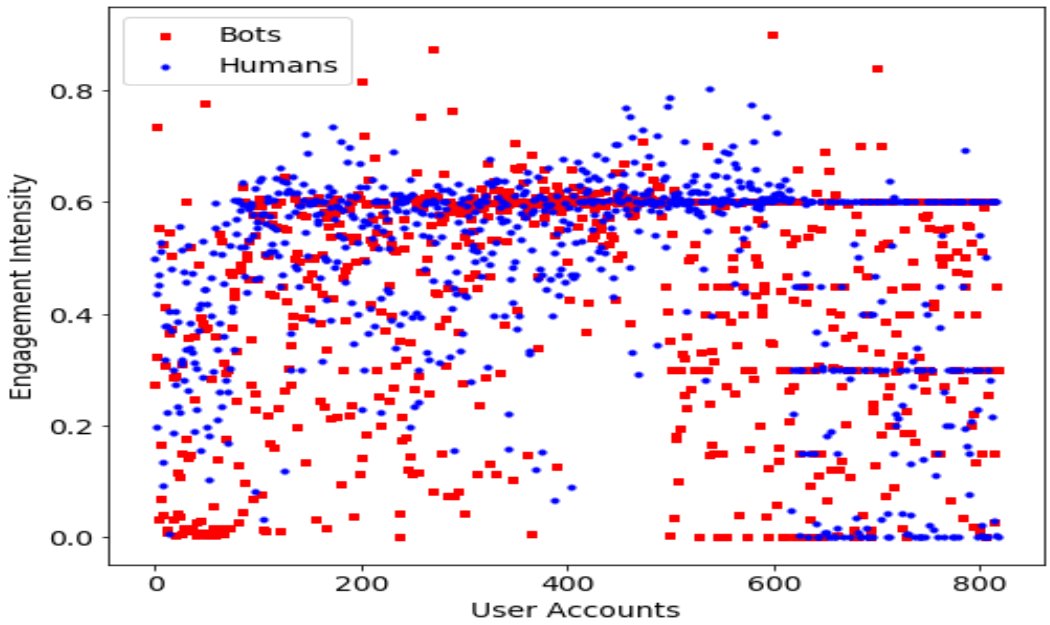

Figure 6. Proportions of engagement intensities in corresponding bot and human accounts. Both group of users show high number of low engagement intensity. The proportions of moderate and high engagements are more prominent in human users

6. CONCLUSION

Modern day social platforms have become part of our lives and effective social policing is required to ensure data credibility and civilised way of interaction. However, with the growing sophistication level of social bots, it is proving difficult to sanitise social platforms. The continuous increase in real-time streaming of tweets makes it practically ineffective to rely on many account features for detection. To effectively distinguish between a bots and a human user, an analysis of lexical richness of tweets posted by both users provides additional distinctive features. We train diverse classifiers to evaluate the role of lexical features in the detection of bot accounts. The newly proposed features significantly improve detection accuracy.

Our proposed an effective detection technique that utilises set of *lexical* and *engagement* features to distinguish between *human* and *social bot* accounts on Twitter. Our approach is motivated by the premise that meaningful engagement will be difficult for *bot* accounts to sustain. We have shown the difference between humans and bots in terms of lexical diversity, usage of contraction and emoticons, based on three different datasets. Furthermore, we show how various forms of *engagement* manifest in both *human* and *bot account*. Findings from the study suggest combination of *lexical* and *engagement features* are powerful tools to harness for improve detection systems. As the first study to introduce combination of *lexical* and *engagement features* for detection task, further research to investigate deeper interactions on Twitter will further widen the disparity between human and bot account independent of language usage.

ACKNOWLEDGEMENT

The authors wish to thank Prof. Reiko Heckel of University of Leicester for his positive feedback. The third author has participated in this research work as part of the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732223.

REFERENCES

Akimoto, A. (2011). *Japan the Twitter nation. The Japan Times. Available from https://www.japantimes.co.jp/life/2011/05/18/digital/japan-the-twitter-nation/#.XAHY4BCnzg5 [accessed 01/12/2018]*

Benevenuto, F., Magno, G., Rodrigues, T., & Almeida, V. (2010, July). Detecting spammers on twitter. In *Collaboration, electronic messaging, anti-abuse and spam conference (CEAS)* (Vol. 6, No. 2010, p. 12).

Cai, C., Li, L., & Zengi, D. (2017, July). Behavior enhanced deep bot detection in social media. In *Intelligence and Security Informatics (ISI), 2017 IEEE International Conference on* (pp. 128-130). IEEE.

Chavoshi, N., Hamooni, H., & Mueen, A. (2017, April). Temporal patterns in bot activities. In *Proceedings of the 26th International Conference on World Wide Web Companion* (pp. 1601-1606). International World Wide Web Conferences Steering Committee.

Chu, Z., Gianvecchio, S., Wang, H., & Jajodia, S. (2012). Detecting automation of twitter accounts: Are you a human, bot, or cyborg?. *IEEE Transactions on Dependable and Secure Computing*, *9*(6), 811-824.

Davis, C. A., Varol, O., Ferrara, E., Flammini, A., & Menczer, F. (2016, April). Botornot: A system to evaluate social bots. In *Proceedings of the 25th International Conference Companion on World Wide Web* (pp. 273-274). International World Wide Web Conferences Steering Committee.

Dickerson, J. P., Kagan, V., & Subrahmanian, V. S. (2014, August). Using sentiment to detect bots on twitter: Are humans more opinionated than bots?. In *Proceedings of the 2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining* (pp. 620-627). IEEE Press.

Dutse, I.I, Bello, B. S., & Korkontzelos, I. (2018). Lexical analysis of automated accounts on Twitter. *In Proceedings of 17th International Conference on WWW/Internet (pp. 75-82). IADIS.*

Ferrara, E., Varol, O., Davis, C., Menczer, F., & Flammini, A. (2016). The rise of social bots. *Communications of the ACM*, *59*(7), 96-104.

Gilani, Z., Farahbakhsh, R., Tyson, G., Wang, L., & Crowcroft, J. (2017, July). Of Bots and Humans (on Twitter). In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*(pp. 349-354). ACM.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, *11*(1), 10-18.

Haustein, S., Bowman, T. D., Holmberg, K., Tsou, A., Sugimoto, C. R., & Larivière, V. (2016). Tweets as impact indicators: Examining the implications of automated "bot" accounts on T witter. *Journal of the Association for Information Science and Technology*, *67*(1), 232-238.

Howard, P. N., & Kollanyi, B. (2016). Bots,# StrongerIn, and# Brexit: computational propaganda during the UK-EU referendum.

Inuwa-Dutse, I., Liptrott, M., & Korkontzelos, I. (2018). Detection of spam-posting accounts on Twitter. *Neurocomputing*.

Lee, K., Eoff, B. D., & Caverlee, J. (2011, July). Seven Months with the Devils: A Long-Term Study of Content Polluters on Twitter. In *ICWSM* (pp. 185-192).

Lee, S., & Kim, J. (2012, February). WarningBird: Detecting Suspicious URLs in Twitter Stream. In *NDSS* (Vol. 12, pp. 1-13).

Morstatter, F., Wu, L., Nazer, T. H., Carley, K. M., & Liu, H. (2016, August). A new approach to bot detection: striking the balance between precision and recall. In *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining* (pp. 533-540). IEEE Press.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. *Journal of machine learning research*, *12*(Oct), 2825-2830.

Sutton, J. N., Palen, L., & Shklovski, I. (2008). *Backchannels on the front lines: Emergency uses of social media in the 2007 Southern California Wildfires* (pp. 624-632). University of Colorado.

Šišková, Z. (2012). Lexical richness in EFL students' narratives. *Language Studies Working Papers*, *4*, 26-36.

Subrahmanian, V. S., Azaria, A., Durst, S., Kagan, V., Galstyan, A., Lerman, K., ... & Stevens, A. (2016). The DARPA Twitter bot challenge. *arXiv preprint arXiv:1601.05140*.

Templin, M. C. (1957). Certain language skills in children; their development and interrelationships.

Thomas, K., Grier, C., Ma, J., Paxson, V., & Song, D. (2011, May). Design and evaluation of a real-time url spam filtering service. In *Security and Privacy (SP), 2011 IEEE Symposium on* (pp. 447-462). IEEE.

Tweedie, F. J., & Baayen, R. H. (1998). How variable may a constant be? Measures of lexical richness in perspective. *Computers and the Humanities*, *32*(5), 323-352.

Varol, O., Ferrara, E., Davis, C. A., Menczer, F., & Flammini, A. (2017). Online human-bot interactions: Detection, estimation, and characterization. *arXiv preprint arXiv:1703.03107*.

Wang, A. H. (2010, July). Don't follow me: Spam detection in twitter. In *Security and cryptography (SECRYPT), proceedings of the 2010 international conference on* (pp. 1-10). IEEE.

Wang, A. H. (2010, June). Detecting spam bots in online social networking sites: a machine learning approach. In *IFIP Annual Conference on Data and Applications Security and Privacy* (pp. 335-342). Springer, Berlin, Heidelberg.

Wilson, C., Sala, A., Puttaswamy, K. P., & Zhao, B. Y. (2012). Beyond social graphs: User interactions in online social networks and their implications. *ACM Transactions on the Web (TWEB)*, *6*(4), 17.

APPENDIX

Table A1. Examples of most frequent languages in the non-English dataset (Dataset4)

Language	***size***	***Language***	***size***
Arabic (ar)	443949	Portuguese(pt)	837
Espaniols (es)	17263	Haitian (ht)	634
Turkish(tr)	12160	Persian(fa)	451
Japanese(ja)	3062	Estonian(et)	446
French (fr)	1877	Catalan(ca)	444
Tagalog(tl)	1107	German(de)	351

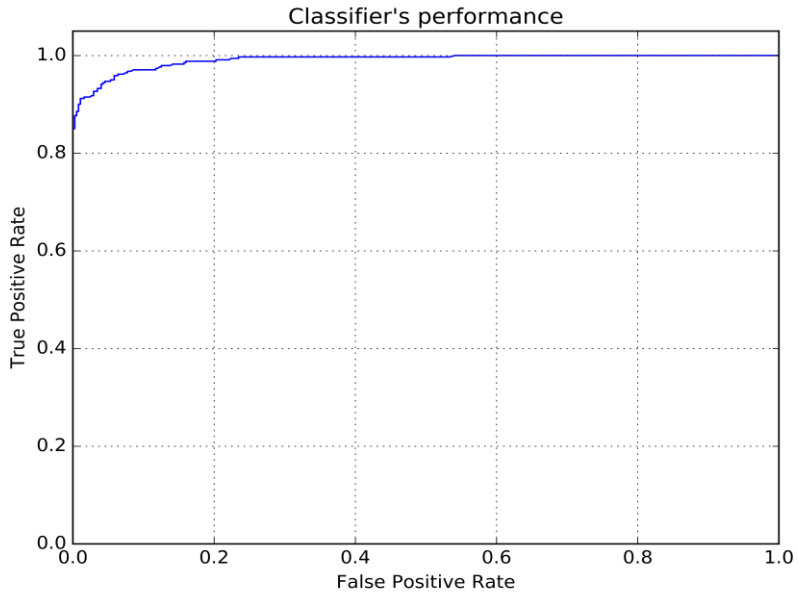

Figure A1. Performance of a classification model (random forest) on the non-English dataset (Dataset4)

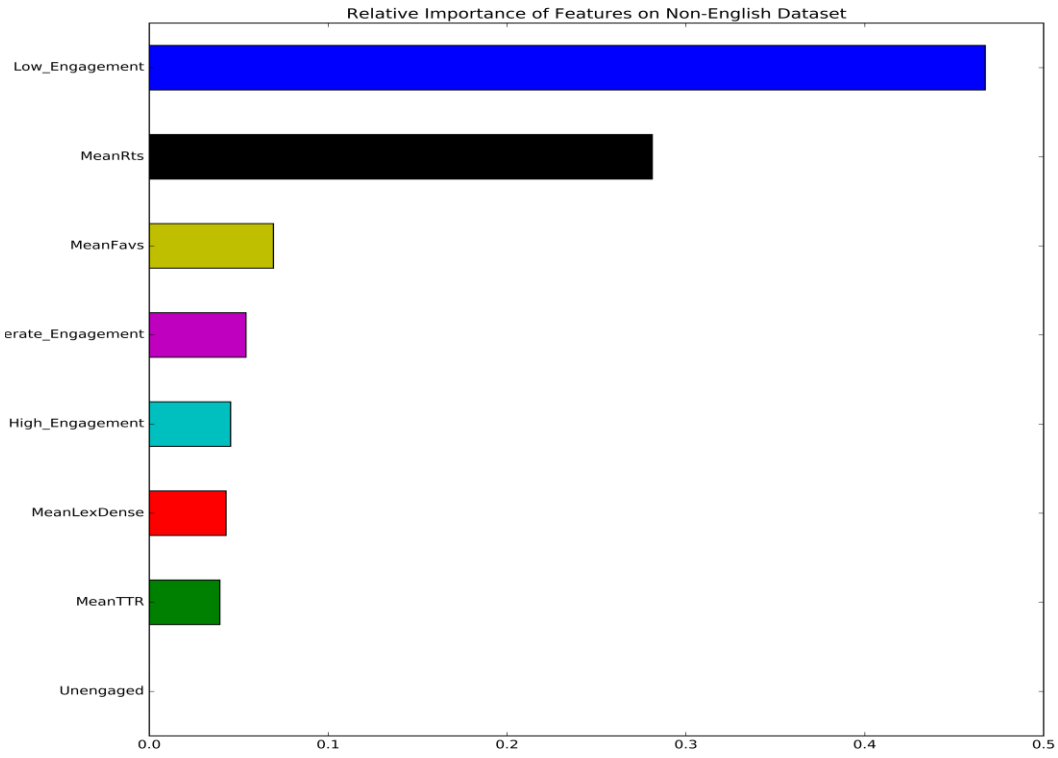

Figure A2. Feature importance of engagement and lexical features on Dataset4. Importance of each feature is assessed in the 0 – 1 continuum and the cumulative sum of all values for features sum to one. The low engagement feature is shown to be a strong distinguishing feature (about 0.45 value)

Code Detector

Adrián and Yannis

Abstract

1 Introduction

With the Internet expansion, new means and ways to communicate arose making it easier to be in contact with different kind of people but also to exchange knowledge with them. Thanks to this, websites based on Question and Answering (Q&A) or forums became popular among Internet users. Some of these knowledge exchange websites became specialised in a specific topics, from literature to programming languages. Nowadays, we can find, for example, websites specialised in software engineering like *Stack Overflow*[1] or *Code Project*[2]. In this kind of specialised websites, it is possible to publish questions, answers and in some cases, articles regarding a specific topic. As the communication between software developers and experts is specialised, it inherits a mixture of natural language text, e.g. English or Spanish, with portions of code, either from programming languages like Java or Perl or markup languages like HTML or XML.

Although many of these specialised websites or services propose the use of specific labels to format differently code from natural language, like in Stack Overflow, in many cases these labels can be misused or forgotten. Moreover, there are many other resources in which it is impossible to get formatting labels, like in emails, plain text, *Twitter*, among others. Therefore, it is necessary a code detector.

A code detector is a tool able to determine whether a document or a part of it, is code in some programming or markup language, e.g. C++ or XML, or whether it is a text in a natural language, e.g. English or French. Most code detectors of the literature aim at improving the communication among developers via electronic means, such as e-mail and forums. The reason behind is that knowing which parts of a document are natural language or code can improve the performance of other tools, such as search engines and indexes [14, 2, 6, 19].

The work here presented occurs in a very specific context, it is part of a pre-processing tool that will be used to analyse data retrieved on real time (or almost it). Thus, the code detector must work for multiple programming languages. And at the same time, the code detector must have a good performance, in terms of speed and accuracy, while small enough to be kept in memory without compromising the performance of other tools.

In the literature it is possible to find multiple approaches to differentiate text in natural language and text in programming code languages. Some methods are based on the use of heuristics ([14]) and grammar islands ([2]), while some other are based on more complex methods like Hidden Markov Models ([6]). Nevertheless, many of the already developed tools have been designed to detect specific programming languages, like Java, and have not explored the newest technologies based on neural networks for example.

Thus, to surpass the problems of the previous works and at the same time examine other technologies, in this article, we explore the use of word embeddings along to two different types of classifiers: a Support Vector Machine and a neural-network-based classifier. The reason to test methods based on word embeddings, is that word embeddings are capable of representing semantic properties and characteristics [15, 8], which in some cases, have been able to improve the performance of more complex tools [11, 21]. The use of word-embeddings-based-methods, might bring some benefits over methods used previously like grammar islands and heuristics.

2 Related Work

In the literature we can observe that state-of-the-art code detectors are based on a variety of methods. They range from basic techniques, such as heuristics, to more complex ones, such as Support Vector Machine (SVM). Below, we have classified the most relevant methods into 4 different groups: *heuristics*, *island parsers*, *machine learning* and *hybrid*.

[1] stackoverflow.com
[2] codeproject.com

Heuristics

In this group we find methods based on rules to determine whether a portion of text is natural language or programming code.

One of the most basic algorithms to detect code can be found in Stack Overflow. It is used to alert users that a question or answer cannot be posted because it has code that is not surrounded with the label $< code > \ldots < /code >$. This *Q&A* website makes use of regular expressions, keywords detection and space indentation counting.[3]

Another example, is the tool created by [3] for detecting Java code within e-mails. It is based on recognising brackets, keywords frequently used in Java (e.g. *public*, *static*) and end-of-lines characters (mainly the semicolon). It makes as well use of regular expressions to determine whether the code was split in multiple lines to apply differently the heuristics.

A more complex tool is presented in [14], where they use heuristics and an agglomerative hierarchical clustering in order to differentiate technical information and natural language. Technical information was defined as code, patches, stack traces and log extracts. The heuristics consist on the detection of keywords, repetitive lines, patch patterns and regular expressions to locate brackets, indentations and camel cases. As the heuristics are not infallible, the researchers added an agglomerative hierarchical clustering, which is used to find and group cases in which the heuristics could have made a mistake. For example, a string in a program could be classified as natural language text despite in its context occurs in code.

Island parsers

An island parser is a tool that splits text by detecting a specific language. More specifically, an island parser is a set of rules that defines the grammar of the language of interest (the "islands") but that ignore other languages constructions (the "water") [19].

In this group we can cite *InfoZilla* [4], a tool created to analyse bug reports in order to find technical information like patches, stack traces and source code. For the detection of code, InfoZilla offers an island parser designed for the programming language Java.

As well, it exists *Automatic Code Element Extractor (ACE)*[4] that detects and extracts Java code elements, e.g. classes, methods and variables, from digital communication means [19]. The objective of ACE is to improve aspects like queries by reducing ambiguity between terms that can be used both in natural language and in code. For instance, *execute* can be used in code as the name of a class or in natural language as a verb. ACE relies on an island grammar designed for Java and a database that stores code elements seen during parsing.

Machine learning approaches

The tools found in this group use machine learning methods to determine automatically the patterns that defines programming languages and natural ones.

One example is presented in [22], where the authors created a Support Vector Machine (SVM) that detects code within emails. To separate code and natural language, they start cleaning emails in a cascade way by filtering non-text elements and, finding paragraphs and sentences. Then they make use of a SVM trained specifically to detect whether a line represents the beginning or the end of a code snippet. The SVM was trained using features like, programming keywords (e.g. *double*, *string*, *#define*), patterns (e.g. $<=$, $a=b$), brackets, ending characters (*;*) or number of line breaks.

In [6] it is presented *InfoRmation ISlands Hmm (IRISH)*[5], a code detector based on two Hidden Markov Models (HMM); one dedicated for detecting natural language and another for discerning code. The tool proposes this method to avoid the problem of creating new island grammars when a new language needs to be analysed. The HMM are trained using an alphabet defined by words (alphanumeric characters chains separated by a white space or underscore), keywords, underscore, numbers and other characters.

Hybrid

In this group, the tools for detecting code can make use of more than one technology to achieve its goal. The best example is *E-mail Unified Content Classification Approach (MUCCA)*[6] [2]. MUCCA was created to classify email text lines automatically into natural language, code, patches or stack traces. To achieve this goal, MUCCA applies two different approaches. The first one consist in a Naïve-Bayes classifier that uses as features n-grams and punctuation marks. The second one is the utilisation of *island parsers*. Regarding the island parser for code detection, it was designed for the programming language Java, nonetheless, it offers the possibility to include island parsers for other languages.

As it can be observed, none of the previous methods have explored the use of the most recent technologies, like neural networks or word embeddings. This is why in this work we explore these new technologies in order to create a fast and general as possible code detector.

[3] meta.stackoverflow.com/questions/341763
[4] www.cs.mcgill.ca/~martin/software.html
[5] www.rcost.unisannio.it/cerulo/dataset-irish.tgz
[6] mucca.inf.usi.ch/index.html

3 Background Information

In this section we present the background information that will be necessary to understand the methodology used for the creation of the Code Detector.

3.1 FastText

FastText[7] is a tool developed by *Facebook Research* which offers the possibility of training only word embeddings [5] or a linear classifier based on specialised word embeddings [13]. FastText is based on improved versions of Word2Vec's [16, 15] neural networks *Continuous Bag-of-Words (CBOW)* and *Continuous Skip-grams*. FastText, according to [23], has shown to perform comparably to more complex deep learning algorithms while it takes less time to train and does not require a GPU.

3.1.1 Word Embeddings

Similar to its predecessor Word2Vec, FastText offers the possibility of training word embeddings that can be used in other NLP tasks. However, it offers some improvements like the capability of using character and word n-grams. While word n-grams give the possibility of creating embeddings of more complex word structures, character n-grams allow FastText to analyse words that were not seen during the training, but that share n-grams with known words. For example, in Spanish, the most frequent term for carbon dioxide is *dióxido de carbono*, however it has a less common variant, *bióxido de carbono*; if the second term was not seen during training, its vector can be inferred thanks to character n-grams.

For the construction of word embeddings, it is possible to use either modified versions of CBOW or the Continuous Skip-grams architectures, as well, increasing or decreasing the size of character and word n-grams.

FastText offers as well the possibility of printing document embeddings, i.e. dense vectors that represent documents. Unlike words embeddings, these dense vectors are not result from a training process, but from averaging the dense vectors of each word present in the documents. Put differently, FastText obtains in first place the word embedding for each word present in a document, and then it does an averaging, resulting a vector that represents the document.

3.1.2 Linear Classifier

FastText's linear classifier is based on a modified architecture of the CBOW, in which instead of training to predict the middle word of a context, it predicts the label of a certain text. To be more specific, FastText's CBOW consists of four layers: the input, the projection, the hidden and the output layer. Given the text of a document, FastText's CBOW trains the neural network to predict the classifier label at the same time it generates specialised word embeddings, i.e. focused on the classification task.

In Figure 1, we present a graphical representation of FastText's CBOW architecture. This graphical representation includes an example of how the input would be used during the training process. It should be noted that despite *getDate()* is code, in the context where it occurs is natural language, therefore, the label in the example is English.

3.2 Bayesian Optimisation

A Bayesian Optimisation [17] is a method which determines, by a Gaussian process, the set of parameters that will maximise the outcome of an objective function. In other words, the Bayesian Optimisation, is a method that varies a set of parameters, an projects them into a Gaussian process with their respective outcome from the objective function. Thanks to this projection, the Bayesian Optimisation is capable of predicting which parameters are the ones that will maximise the objective function.

In this work, the Bayesian Optimisation is applied to find the hyper-prameters that will provide the best performance for the tested classifiers. We made use of this method instead of a brute force one, like a grid, because, as stated in [20] it can be better suited in a machine learning scope. Moreover, the time necessary to find the parameters is lesser than using a grid method.

4 Methodology

We explore two classifiers that are easy to train , but also fast and that do not rely on special or high-end hardware. In other words, they are very quick and they work on a CPU basis instead on a GPU, unlike methods based on deep

[7] `fasttext.cc`

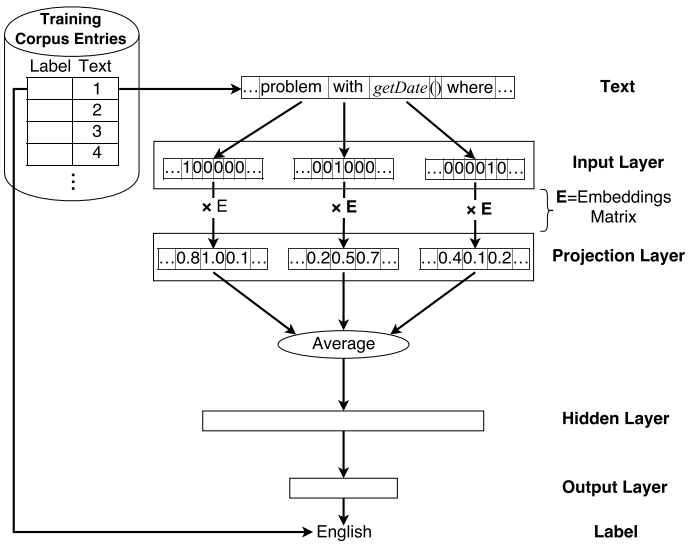

Figure 1: Architecture of FastText CBOW neural network including the inputs of a training example.

learning neural networks like a Long Short-Term Memory. These two characteristics are very important in the context of our project. The selected classifiers are the following ones:

- **Dense SVM:** This model consist in feeding a SVM[8] with documents embeddings coming from FastText. The difference from a sparse SVM[9], is that the input vectors, either for training or testing, are dense. In other words, the data matrices used have a lesser zeros and are generally smaller. Moreover, thanks to the dense representation it is possible to keep semantical aspects that sparse matrices cannot represent.
- **FastText's classifier:** This model, as explained in Section 3, it is based a neural network that uses specialised word embeddings.

5 Data

In this work we have decided to explore 4 different programming languages for the creation and testing of the code detector: *Java*, *C++*, *Python* and *JavaScript (JS)*. These programming languages were chosen as they are among the most used programming languages in 2017 at GitHub[10] and at Stack Overflow[11].

For each programming language, we generated 3 training corpora in which 50% of the entries corresponded to code text and 50% to English text. Being more specific, for a specific programming language, one of the training corpus is based on documents[12] (20,000 entries) while two are based on sentences[13], both of different sizes (80,000 and 400,000). The two last corpora have different sizes as we were not sure whether the quantity of information would be enough to get a performing classifier; the corpora based on 400,000 sentences is called heceforth *Large Set of Sentences (LSS)*. Similarly, we created 3 training corpora where the code text came from the 4 studied programming languages at a proportion of 12.5% (Code: $4 \times 12.5\%$, English: 50%); this set of training corpora is henceforth called *Mixed*.[14] In every training set, the English sentences and documents are always the same. We present in Table 1 the size of each training corpus used in this work in terms of types and tokens.

The size of training corpora were selected empirically, however, as it will be explained in Section 6, we carry out an optimization process in order to get the best performance of the classifiers with the current data.

Regarding the testing data set, we created in total 8 different corpora, 2 per programming language: Java, C++, Python and JavaScript. As we did for the training data sets, every corpus could represent either sentences or documents and 50% of its entries correspond to code text and 50% to English text. For every testing set, we make use of different

[8] The library used for the SVM was LibSVM [7].
[9] A sparse SVM is tested as well in order to compare the outcomes.
[10] octoverse.github.com
[11] insights.stackoverflow.com/survey/2017#technology
[12] In the case of programming languages, a document is defined as the code defined in one file.
[13] As it is impossible to split programming code in sentences, we used instead code lines.
[14] The entries were chosen randomly.

Table 1: Statistics regarding the training corpora based on Documents, Sentences and Large Set of Sentences (LSS). The figures were rounded into thousands and millions.

Tokens / Types		Documents		Sentences		Large Sentence Set	
		Code	English	Code	English	Code	English
Prog. Language	Java	6.9M / 403K	4.3M / 183K	233K / 34K	883K / 70K	1.1M / 122K	4.4M / 187K
	C++	27.7M / 1.1M		278K / 41		2.0M / 165K	
	Python	17.9M / 542K		269K / 33K		1.3M / 109K	
	JS	7.0M / 254K		290K / 31K		1.4M / 96K	
	Mixed	14.8M / 647K		269K / 40K		1.3M / 149K	

Table 2: Statistics regarding the test corpora based on Documents and Sentences. The figures were rounded into thousands and millions.

Tokens / Types		Documents		Sentences	
		Code	English	Code	English
Prog. Language	Java	1.7M / 121K	1.0M / 80K	58K / 10K	220K / 29K
	C++	7.2M / 441K	1.0M / 70K	76K / 12K	221K / 30K
	Python	4.7M / 202K	1.1M / 73K	67K / 11K	219K / 29K
	JS	1.7M / 92K	1.1M / 75K	71K / 10K	221K / 30K

collections of English sentences and documents. Each corpus consisting of sentences has 20,000 entries, while the one made of documents has 5,000 ones. In Table 2, we present a set of statistics that summarize every testing corpora.

The data used to build the training and test corpora were collected using different resources. For English, we made use of *Wikicorpus v. 1.0* [18] to get documents in English, while the *Leipzig Corpora Collection* of 1M sentences from Wikipedia for the years 2012 and 2016 [10] for collecting the English sentences. We made use of two different English sources in order to be certain that we would be analysing either documents or sentences uniquely.

For the code sources, the Java programs came from the *GitHub Java Corpus* [1]. While the code files for Python, C++ and JavaScript where crawled from GitHub, using the script offered in the project *HERMES*[15] developed by *Lab41*[16]. In order to run the crawling script, we indicated the projects' repositories to analyse and the files' extensions to download. The Python related projects were those proposed in Lab41's project, while for C++ and JavaScript we chose the most important projects that use those languages in GitHub.

To generate the corpora based on lines of code, we used a Perl Script that split every program file into lines using the newline character. Then, the lines of code were shuffled, to get a random distribution of them. Finally we select the number of lines indicated for each corpora, either training or testing. It should be noted that due to the nature of code lines, training and test corpora might contain some identical or very similar lines. This is expected because in many cases a same code line is used either by convention, need or coincidence. Examples of common code lines are: *"{"*, *"import java.util.*;"* and *"Date date = new Date();"*.

All the data, either for training or testing, are pre-processed in the same manner. We verify that all the files are in a codification UTF-8, otherwise they are converted using the linux command line *iconv*. Characters like brackets, question marks, quotes and dots[17] are separated from words by spaces For example, the following texts *"How is it done? Using the best wheat."* and *"public static void main(String[] args)"* were converted to *"How is it done ? Using the best wheat ."* and *"public static void main (String [ ] args)"* respectively. Empty lines are deleted and spacing is normalised, in order to have an uniform text. Following [2], text is lowercased and we do not apply any kind of stemming, lemmatisation or stop-word filtering.

6 Experimental Setting

As both classifiers, the Dense SVM and FastText's one, require a certain number of hyper-parameters to be set, we use a Bayesian Optimisation [17] to find the most appropriate ones. The method consists in varying the hyper-parameters and model the outcome of a objective function in a Gaussian process.

The target of Bayesian Optimization is to maximise the value of the objective function. In our case, the objective

[15] github.com/Lab41/hermes

[16] www.lab41.org

[17] This rule is only valid when a dot is followed any type of spacing or end-of-file character. This was done to differentiate dots in code and full stops in natural language, as it was done in MUCCA [2]. For example, a sequences such as *System.out.printl* is kept as one unit, whereas it splits *"This is an example."* as *"This is an example ."*.

Table 3: ALTERNATIVE FORM 1: Hyper-parameters used for training the models.

Method	Code	Text type	Parameters									Median F-score	Model size (MB)
			Dim	*Epoch*	*LR*	*Min*	*N-grams*	*N-min*	*N-max*	*Model*	C		
FastText	Java	Documents	300	30	0.0514	5	2	-	-	-	-	0.9990	671
		Sentences	500	5	0.4227	4	1	-	-	-	-	0.9904	33.2
		LSS	100	30	0.4588	5	1	-	-	-	-	0.9948	16.5
	C++	Documents	100	5	0.2062	3	1	-	-	-	-	0.9995	152
		Sentences	100	5	0.0717	1	1	-	-	-	-	0.9915	76
		LSS	100	30	0.2118	5	1	-	-	-	-	0.9915	19
	Python	Documents	100	30	0.4461	4	1	-	-	-	-	0.9992	68
		Sentences	400	5	0.2308	5	1	-	-	-	-	0.9600	71
		LSS	100	5	0.5000	5	1	-	-	-	-	0.9777	85
	JS	Documents	100	5	0.2575	2	1	-	-	-	-	0.9995	80
		Sentences	100	5	0.2547	1	1	-	-	-	-	0.9925	71
		LSS	500	30	0.1188	5	1	-	-	-	-	0.9837	85
	Mixed	Documents	100	10	0.1118	3	1	-	-	-	-	0.9990	98
		Sentences	100	5	0.1655	4	1	-	-	-	-	0.9845	49
		LSS	300	30	0.0952	5	1	-	-	-	-	0.9877	7.1
Dense SVM	Java	Documents	100	5	0.1000	5	1	1	6	Skip-gram	30	1.0000	257
		Sentences	100	30	0.02182	4	2	2	3	Skip-gram	-3	0.9904	210
	C++	Documents	100	5	0.0715	1	2	1	6	CBOW	18	1.0000	1170
		Sentences	200	5	0.0100	5	3	1	6	Skip-gram	5	0.9937	415
	Python	Documents	100	5	0.0880	2	3	1	5	Skip-gram	16	0.9995	458
		Sentences	300	25	0.0154	5	1	2	6	Skip-gram	2	0.9818	623
	JS	Documents	100	5	0.0500	5	3	3	5	Skip-gram	27	1.0000	250
		Sentences	300	30	0.0432	2	2	1	3	Skip-gram	3	0.9932	660
	Mixed	Documents	100	20	0.0739	5	2	1	2	Skip-gram	-1	1.0000	299
		Sentences	100	20	0.0764	3	2	3	5	Skip-gram	5	0.9863	212

Dim: Number of dimensions of the dense vectors
Epoch: Number of times that the neural network observes the complete training data set
LR: It is the learning rate, i.e. how the quickly the neural network should refresh its parameters
Min: Minimum occurrence number of a word/n-gram
N-grams: Maximal length of word n-grams
N-min/max: Indicates the minimal and maximal length of character n-grams
Model: Neural network architecture for training the embeddings, either (Continuous) Skip-gram and CBOW.
C: Value that will with used in 2^C and that will control the accepted misclassification rate in a SVM.

function is defined as the median F-score value of a 10-fold cross validation. In other words, our goal is to determine the hyper-parameters that would produce the classifier with the best F-score; the cross validation is done uniquely over the training corpus (see Section 5). We opted for Bayesian Optimization instead of a brute force method, e.g. grid search, due to its reduced processing time requirements for finding the best hyper-parameters and it usefulness in machine learning [20]. In the case that either a Dense SVM or a FastText classifier has multiple hyper-parameters sets that gives a similar F-score, we selected the set that that generate the smallest model.

More specifically, we optimised FastText's Classifier hyper-parameters on each training corpora. In Table **??** we show the hyper-parameters determined by the Bayesian Optimisation, as those that provide the best F-score in a 10-fold cross-validation. In Table 3 we present as well, the median F-score and the size of the resulting model trained with those hyper-parameters.

Regarding, the Dense SVM, we optimised the hyper-parameters for the creation of word embeddings with FastText and the parameter C related to the linear kernel for the SVM. It must be indicated that for the Dense SVM, we do not create models using the LSS corpora; the SVM takes several days to finish one fold of the cross validation. Therefore, Dense SVMs are trained uniquely over Documents and Sentences corpora. In Table 3 we present the best hyper-parameters found through the Bayesian Optimisation for each of the training corpora; it presents as well, the reached median F-score and the size of the model. In all the cases, every Dense SVM made use of a linear kernel, because the number of features, i.e. dimension of the vector, are much larger than the number of classes [12].

Table 4: Hyper-parameters used for training Sparse SVM models.

Code	Text type	C	Median F-score	Model size
Java	Documents	15	0.9997	16.6MB
	Sentences	5	0.9906	2.68MB
C++	Documents	15	0.9996	41.1MB
	Sentences	1	0.9939	3.31MB
Python	Documents	15	0.9988	18.7MB
	Sentences	5	0.9693	3.28MB
JS	Documents	5	0.9986	11.4MB
	Sentences	15	0.9879	2.53MB
Mixed	Documents	15	0.9991	22.5MB
	Sentences	1	0.9846	3.68MB

Value that will with used in 2^C and that will control the accepted misclassification rate in a SVM.

In the following sub-sections, we present the baselines to which we compared the results obtained with our methods, and we introduce the evaluation used to assess the methods here explained.

6.1 Baselines

Besides the methods described in 4, we experimented with a different number of baselines to determine how difficult is in fact the task. The group of baselines consists in using heuristics designed to detect one of the following brackets: curly *{}*, square *[]* and parentheses *()*; as well, one baseline involves the detection of any of the previous brackets. With a single occurrence of a bracket, either opening or closing, a piece of text is considered as code. All the baselines consist on a regular expression that looks for the specified bracket or set of them; the baselines are programmed in Perl.

6.2 Comparison with other Methods

In order to have a comparison point different from the baselines, we decided to train as well some Sparse SVM. For these classifiers the input is a Vector Space Model, where each feature is a token, i.e. a text unit defined between white spaces, weighted by relative frequency. The number of Sparse SVM is the same that Dense SVM, in other words, we train on documents and sentences uniquely; these SVM make use of a linear kernel and the parameter C is optimized doing a grid search. In Table 4 we present the values of parameter C that was used in each respective Sparse SVM.

Although in the literature, we can find different tools for doing a similar test, we could not arrive to run any from the state-of-the-art. The reasons were either because the system or training data is not available any more or could not be run.

6.3 Evaluation

All the methods are evaluated with F-score. More specifically, two F-score are calculated for each method, one regarding the prediction of sentences and one other of documents. To calculate these values of F-score we follow the next steps. In first place, the methods predict the classes for all entries in each testing corpus. Then, the predictions are concatenated by method and whether the testing corpora is based on sentences or documents. Finally, it is calculated a unique value F-score for each method.[18]

Besides the evaluation regarding the F-score, we evaluate the outcomes using significance tests to determine whether performance difference between methods is statistically significant or not. In particular, we use *Cochran's Q test*, a non-parametric test, that determines whether different methods produce statistically identical results on the same population. In cases where Cochran's Q test identify statistically significant differences (p value < 0.05), we apply as *post hoc* test a pairwise *McNemar's test*. McNemar's test is a non-parametric test designed for paired data, in which only two methods are compared. Furthermore, to determine how strong is the difference, for each method that McNemar's test identified a significant difference in its performance (p value < 0.05), we calculate its respective effect

[18]It must be noted, that an average F-score can be calculated. In other words, for each testing corpus, it is possible to calculate an F-score. Then, to average the F-score by the text type, either documents or sentences, of the testing corpora. We did not follow this path because the statistical test used to validate the results. Other statistical test, like a Repeated Measures Analysis of Variance, could have been applied if we would have had more testing corpora (not in size, but in quantity).

Table 5: We present the best and the worst method, in terms of F-score, trained in each code, Java, C++, Python, Js and Mixed, for analysing documents. Besides, we show similarly for the baselines.

Training		**Method**	**F-score**
Code	**Set**		
Java	LSS	FastText	0.989
	Documents	FastText	0.935
C++	Documents	Dense SVM	0.998
	Sentences	Sparse SVM	0.969
Python	Sentences	Dense SVM	0.994
	Documents	FastText	0.853
Js	Sentences	Dense SVM	0.998
	Sentences	FastText	0.992
Mixed	Documents	Dense SVM	0.999
	Sentences	Sparse SVM	0.987
		Baseline	**Fscore**
		Curly	0.933
		Parentheses	0.716

size using *Cramér's V*.[19] The thumb rule indicates that an Cramér's V Effect size is small if $V = 0.1$, medium $V = 0.3$ and large $V = 0.5$ [9].

Although Cochran's Q and McNemar's tests are non-parametric, data must be dichotomous. In other words, the results must be binary, e.g. either 0 or 1. In our case, the predictions done by the classifiers were evaluated with *YES* or *NO* in accordance whether the classifier gave the correct label or not.

Due to the nature of our project, in which performance and response time are highly relevant, we compare the models in terms of CPU time[20] during prediction.

7 Results

As the number of experiments, 78 in total, is large, in the following subsections we present only the most relevant results, the ones that can contribute the most to the state-of-the-art. The results are divided in 3 parts. The first part presents the outcomes from the testing corpora based on documents, while the second part does the same but for the testing corpora grounded on sentences. The third part shows the results regarding the CPU time analysis.

7.1 Tests over Documents' Corpora

In Table 5 we present, for each programming language used for training, their best and worst method according to F-scores. As it can be observed in Table 5, the task of identifying documents is not hard to do. In general all the methods here developed arrive to classify correctly the documents. The best methods, taking into account uniquely the F-score, are the Sparse and Dense SVM trained over Mixed documents with 0.999, while the worse methods are FastText trained over Python documents (0.853) and the baseline using normal brackets (0.716). Moreover, we can support the fact that the task is easy by observing that the baseline using the detection of curly brackets achieved an F-score of 0.933.

Regarding Cochran's Q test, we obtained a p value $< 1.0 \times 10^{-15}$, which means that at least in one pair of methods, the difference between the F-score is significant. Therefore, we applied the post hoc test, which showed us that the F-score obtained by multiple pairs of methods are significantly different, while some others are not significant. As expected, the post hoc test indicated that the difference between Sparse and Dense SVM trained over Mixed documents, F-scores of 0.999, is not statistically significant p value $= 0.186$. However, for instance, the F-score of the Dense SVM trained over Mixed documents (0.999) is significantly different to the F-score of FastText trained on Mixed LSS (0.995). Because the difference between methods is in several cases very small, we analyzed the results in terms of effect size. Therefore, in Table 6 we show the methods that should be considered with an equivalent performance to the most performing methods, i.e. Dense SVN trained over Mixed Documents; these methods were selected regarding its p value and effect size.

[19] Effect sizes are useful specially when large data sets are analysed with a statistical test. This is because the odds of having results showing differences statistically significant increase, making the statistical test's results less illustrative. With the effect sizes is possible to determine whether the difference in real life would be noticeable.

[20] In Linux, CPU time is approximated as the values *user* + *sys* obtained using the command *time*.

Table 6: We present other methods that have an equivalent performance to the best code detectors, Dense and Sparse SVM trained on Mixed documents, for the classification of documents. These methods were chosen in accordance to the statistical test and effect sizes.

	Training		**Method**	**F-score**	p **value**	**Cramér's V**
	Code	**Set**				
Dense SVM Mixed documents	Mixed	Documents	Sparse SVM	0.999	0.186	-
	Mixed	Documents	FastText	0.998	1.7×10^{-2}	0.017
	Js	Documents	Dense SVM	0.998	1.2×10^{-2}	0.018
	C++	Documents	Dense SVM	0.998	4.0×10^{-4}	0.025
	Mixed	LSS	FastText	0.995	4.3×10^{-15}	0.055
Sparse SVM Mixed documents	Js	Documents	Dense SVM	0.998	0.408	-
	Mixed	Documents	FastText	0.998	0.141	-
	C++	Documents	Dense SVM	0.998	2.4×10^{-2}	0.016
	Mixed	LSS	FastText	0.995	2.0×10^{-12}	0.050
	Python	Sentences	Dense SVM	0.994	6.5×10^{-16}	0.057

Table 7: We present the best and the worst method, in terms of F-score, trained in each code, Java, C++, Python, Js and Mixed, for analysing sentences. Besides, we show similarly for the baselines.

Training		**Method**	**F-score**
Code	**Set**		
Java	Sentences	Dense SVM	0.961
	Documents	FastText	0.764
C++	Sentences	Sparse SVM	0.974
	Documents	FastText	0.858
Python	Sentences	Dense SVM	0.967
	Documents	Sparse SVM	0.809
Js	Sentences	Dense SVM	0.982
	Documents	Sparse SVM	0.887
Mixed	Sentences	Dense SVM	0.987
	Documents	Sparse SVM	0.929
		Baseline	**Fscore**
		All	0.684
		Curly	0.391

As it can be noted in Table 6, in the cases where p value < 0.05, the effect size is very small, i.e. less than 0.06. This means that despite the significant difference between these pairs of methods, the difference would be hardly noticeable. Therefore, they should be considered with an equivalent performance.[21]

7.2 Test over Sentences' Corpora

We show in Table 7 the best and worst method for detecting code sentences in accordance to each programming language used for training. Unlike the classification of documents, the identification of sentences is a much harder task, as it can be observed in Table 7. While for detecting documents, 30 of 39 methods arrived to have an F-score greater than 0.950, in the case of sentences classification just 16 of them reached an F-score equal to 0.950 or greater.

As presented in Table 7, the best method to classify correctly sentences is a Dense SVM trained on Mixed Sentences (0.987); this assumption is taking into account uniquely the F-score. The worse methods are FastText trained over Java documents (0.764) and the baseline using square brackets (0.123). In general, it can be seen that all the methods have difficulties to classify correctly code and text when the input are sentences.

Concerning the statistical test, Cochran's Q test, it indicated us that for at least in one pair of methods, the difference between the F-score is significant (p value $< 1.0 \times 10^{-15}$). The application of the post hoc test showed us that just for a few pairs of methods, their difference between F-scores is not statically significant. In Table 8, we present the results that according to the statistical test are statistically equivalent to Dense SVN trained over Mixed Sentences.

As it can be observed in Table 8, there are 5 different methods for which Cochran's Q test indicated a difference

[21] This does not mean that the methods give the exact same results, but that the results are comparable.

Table 8: We present other methods that have an equivalent performance to the best code detector, Dense SVM trained on Mixed sentences, for the classification of sentences. These methods were chosen in accordance to the statistical test and effect sizes.

	Training		Method	F-score	p value	Cramér's V
	Code	Set				
Dense SVM Mixed sentences	Mixed	LSS	FastText	0.986	2.9×10^{-3}	0.010
	Mixed	Sentences	Sparse SVM	0.985	1.8×10^{-5}	0.015
	Mixed	Sentences	FastText	0.984	3.1×10^{-9}	0.021
	Js	Sentences	Dense SVM	0.982	1.4×10^{-22}	0.034
	Mixed	Documents	Dense SVM	0.979	1.0×10^{-61}	0.058

statistically significant (p value < 0.05) but where the effect size is very small, i.e. less than 0.06. As it happend with the in the experiments for the classification of documents, the difference between the methods presented in Table 8, despite being significant, in the real life the difference is minimal and might not be observable.

7.3 CPU Time

In Table 9 we show the average time that each prediction method needed to do the predictions in each testing corpora, based either on documents or sentences. Each average time includes its 95% confidence intervals, in order to give a clearer idea that what could be the real performance.

Table 9: Average CPU time, in seconds, with 95% confidence intervals concerning prediction on the test corpora.

Prediction Method		CPU Time [s]	
		Documents	Sentences
Baselines	Mix	0.528 ± 0.295	0.092 ± 0.003
	Curly	0.296 ± 0.143	0.086 ± 0.002
	Normal	0.434 ± 0.230	0.098 ± 0.015
	Square	0.256 ± 0.107	0.086 ± 0.002
FastText	Documents	2.006 ± 0.636	1.084 ± 0.458
	LSS	1.424 ± 0.254	0.533 ± 0.027
	Sentences	1.452 ± 0.250	0.555 ± 0.031
Dense SVM	Documents	13.878 ± 2.591	10.405 ± 0.695
	Sentences	27.391 ± 6.802	44.371 ± 11.024
Sparse SVM	Documents	10.163 ± 1.415	4.933 ± 0.765
	Sentences	15.468 ± 1.733	13.079 ± 1.893

As it can be observed in Table 9, the baselines tend to be the fastest methods for predicting the corpora, while the methods based on Dense SVM tend to be the slowest. Furthermore, in Table 9 we can see that FastText is a very fast classifier, sometimes 5 times faster than the best time of SVM-based methods.

8 Discussion

As it could be observed in Section 7, the performance, in terms of F-score, of the methods here tested depends on the length of the text analysed. In other words, as it can be seen in Table 5 and in Table 7, the F-score obtained by the methods changes greatly whether the input is a document or a sentence. While for documents, the methods, even the baselines, in general carry out the task with great precision and recall, the analysis of sentences show us that not all the methods or training corpus are the best suited for the task.

To understand better the difficulty of the problem, we present in Table 10 and Table 11 some examples of documents and sentences, respectively, that come from the testing corpora and were misclassified by the code detectors presented in Table 6 and Table 8.

Regarding the grade of difficulty between classifying documents and sentences, this fall on mostly on the size of the context. Documents, due to their length, will contain a greater number of tokens that represent the correct class, Code or English, than the number of ambiguous tokens. Sentences, at least in the way they were defined in this article, can be of length one, especially if they come from Code, the best example would be the closing curly bracket of a block statement in a Java, JavaScript or C++ program. Thus, while in some cases it is possible to find representative

Table 10: Misclassification examples from the documents testing corpora. In case a portion of the document was taken, horizontal or vertical ellipsis are used.

Actual Type		Documents
English	10.1	kanem may refer to: kanem-bornu empire; kanem prefecture; kanem region; kanem (department);
	10.2	births. deaths. ibn duraid; events.
	10.3	andrew mcintosh may be: andrew mcintosh baron mcintosh of haringey; andrew mcintosh (professor); andrew mcintosh (victorian politician);
	10.4	... trichiana borders the following municipalities: cison di valmarino limana mel revine lago sedico. demographic evolution. `colors = id : lightgrey value : gray ( 0.9 ) ...`
	10.5	the ji n district (okres ji n in czech) is a district (okres) within the hradec krlov region ...

Code	10.6	`"""`
		`the hacks module encapsulates all the horrible things that play with django internals in one evil place. this top file will automagically expose the correct hacks class.`
		`"""`
		`#currently these work for 1.0 and 1.1.`
		`from south.hacks.django_1_0 import hacks hacks = hacks ()  \\`

	10.7	`/* the contents of this file are subject to the common public attribution license version 1.0 (the "license");`
		⋮
		`logo it must direct them back to http://www.projity.com. */`
		`package com.projity.pm.graphic.model;`
		`import java.util.eventlistener;`
		`/** * */`
		`public interface nodeselectionlistener extends eventlistener{`
		`  void valuechanged(listselectionevent e);`
		`}`

	10.8	`# flake8 : noqa`
		`"""`
		`expose public exceptions & warnings`
		`"""`
		`from pandas._libs.tslib import outofboundsdatetime`
		`class performancewarning (warning):`
		`  """`
		`  warning raised when there is a possible performance impact.`
		`  """`
		`class unsupportedfunctioncall (valueerror):`
		`  """`
		`  exception raised when attempting to call a numpy function on a pandas object but that function is not supported by the object e.g. np.cumsum(groupby_object).`
		`  """`

Table 11: Misclassification examples from the sentences testing corpora. In case a portion of the document was taken, horizontal or vertical ellipsis are used.

Actual Type		Sentences
English	11.1	when the function's argument is 0 (zero) it will return the integer 1 (one).
	11.2	this lack of data restricts the questions we will be able to answer.
	11.3	; inspection * size: return the number of items in the heap.
	11.4	"* david copperfield -" arthur lowe's micawber is better than anything.
	11.5	steelman defends patent monopolies writing "consider prescription drugs for instance.
	11.6	*[suffix in Kwak'wala]* "oh if he would come!
	11.7	the sizeof operator is an exception: sizeof array yields the size of the entire array (that is 100 times the size of an int).
Code	11.8	`"d.c."`
	11.9	`"s\\u00e1b."`
	11.10	`"longform package description is too long. meteor uses the section of " +`
	11.11	`model.add(new tripleimpl(textannotation properties.enhancer_selection_context`
	11.12	`public backgroundcassandrahostservice(hconnectionmanager connectionmanager`
	11.13	`for subnet in subnet_list:`
	11.14	`the variables that are to be changed in the search for a minimum and`
	11.15	`// their prototype/group as they depend on the group for their layout.`
	11.16	`* expected to be in encoded form which represents the various rule modes`

characters like "{", in some other cases it might be harder or ambiguous like in Example 11.7, where the multiple tokens frequently-used in Code are found in an English Context. However, despite the differences between the analysis of documents and sentences, it was still possible to train methods, like Dense and Sparse SVM Mixed Sentences or FastText Mixed LSS, that are able to have a great performance in both kind of text.

From Table 10 and Table 11, we can observe a series of patterns that may help to explain why the code detectors misclassified of documents and sentences. Documents and sentences tend to be misclassified when they contain a mixture of English and Code, but the text majority belong to the opposite type of the text class. For instance, in Example 10.4 half of the document was English text, while the rest was a kind of CSS code; in Example 10.6, Example 10.7 and Example 10.8, the number and length of English comments surpass in the document the number of actual code. In Example 11.7 we can see a similar behaviour in a sentence explaining the functioning of *sizeOf*; Example 11.1 is another case, although the proportion of code-related terms is more balanced.

Similar to the previous case, but still different, are Example 11.10, Example 11.15 and Example 11.16, where the context given is not enough for the classifiers to determine that they are a String and two comments respectively. Example 11.14 represents a quite particular case that we find recurrently in sentences from Python programs. At first sight it is an English sentence and, in consequence, it should not appear in Table 11 and being considered as misclassified. However, the fact is that Example 11.14 although it has an English nature, it occurred within a Python program, more specifically, it happened in a multi-line comment block generated with a three double quotes (`"""`). This is another example that in occasions the size of the context is not large enough to determine the nature of the text.

Another reason of document misclassification is the proportion of semi-colon (;) in texts. See Example 10.1, Example 10.2 or Example 10.3. These errors give us an idea that the Code Detector associated the semi-colon as a characteristic feature of code instead of English. Example 11.3 is an example of how semi-colon and words frequently used in code, make the Code Detector to misclassify texts. On the other side, it looks like that quotes can shift the Code Detector to classify really short text as English, like in Example 11.8 and Example 11.9.

Furthermore, it seems that the number of unknown words affects the correct classification of documents, see Example 10.5, and sentences, see Example 11.6. In these examples, the text contain words that are not from English.

Despite having found some interesting patters, there are some examples that are hard to explain the reason for being misclassified. Some of these cases can be seen in Example 11.2, Example 11.4 and Example 11.5, where they are clearly portion of text from English. Or like in Example 11.11, Example 11.12 and Example 11.13, cases where a clearly portion of code has been incorrectly classified as English without an apparent reason. These cases must be related to some patters that we cannot see and, at least, in the case of the Dense SVM and FastText's classifiers, due to their nature, it is impossible to know which dimensions of the dense vectors are the responsible ones.

Regarding the baselines, as it happened with more complex methods, the size of the context affect severely their performance. While in documents the simple detection of one curly bracket could achieve an F-score of 0.933, this same method just arrived to have 0.391 in the prediction of sentences. Besides the reason related to context sizes, another aspect that affected is the fact that not all the code lines have necessarily a curly bracket and in some languages, like Python, the use of them are less frequent than in Java or C++. This is why, when analysing sentences, it most adequate to use a baseline settle on the detection of all kinds of brackets than on just one type; however, the F-score will be still low (0.684). It might be possible to increase the F-score with a baseline founded on brackets different than parenthesis as these ones are frequently used in English.

Baselines are really fast, as shown in Table 9 because they are based on a regular expression which only looks for the first occurrence of the specified bracket(s) and they were programmed in Perl, a well-known language regarding the speed of regular expressions. Regarding the SVM, they tend to be slower than any other method because it is necessary to transform the tokens of the input into numerical vectors. In the case of a Dense SVM, we need to call FastText's embeddings to return the dense vector of a document, while for a Sparse SVM, it is necessary to use a dictionary that transforms tokens into a position in the vector space. To have a dictionary or a word embeddings conversion process, implies to have an extra task to do in order to predict a text, while in methods based in FastText's classifier, this conversion process is already incorporated in the neural network and in consequence in the classifier.

Although the results presented in Table 6 and in Table 8, give us an idea of which are the most adequate methods to create a Code Detector, in the scope of our project, that information is not enough to determine which is the most appropriate one. Therefore, it is necessary to include the analysis regarding the size of the model, presented in Table 3 and Table 4, and the average CPU time presented in Table 9. From these tables, we can observe that Sparse SVM tend to generate the smallest models (from 2.5 MB to 41.1 MB) however they take at least 5 times more seconds to predict the testing corpora. Regarding FastText's classifiers, in this article, we find that they generate small models as 7.1 MB but also up to 671 MB. Moreover, they are very fast despite the fact that they only run over CPUs; actually, code detectors using FastText's classifier are the fastest apart from the baselines. Concerning Dense SVM, we determined that the classifier based on these method are the slowest to predict the testing corpora and they generate the largest

files (from 212 MB up to 1170MB). Therefore, taking into consideration all the previous elements, the most adequate Code Detector is the one based on FastText Mixed LSS, because it has a good performance in predicting documents (0.995) and sentences (0.984), it is fast predicting documents (1.568 ± 0.560s) and sentences (0.551 ± 0.008s), and it created a model of just 7.1MB.

We observed that the use of a Bayesian Optimisation is a method that should be used to train classifiers where multiple parameters are needed to fit. In fact, thanks to the Bayesian Optimisation, we arrived to have a performance that in many cases are similar no matter if the model was trained on documents or sentences, and on larger set, like in the LSS. Because of this, we are not sure whether a the increment of the data will produce better results. Nevertheless, a more representative set of English documents and sentences, might improve the general performance of the code classifier, because the text from Wikipedia do not represent the English usage that could be used in forums, for example. In other words, it might be better to use a set of English text that is formed of sentences or documents coming from sources like Wikipedia, forums and Q&A websites.

Besides, a way to improve the general performance of the code detector and avoid the mistakes seen previously, would be to use a most powerful neural network. For example, a recurrent neural network might be of help in the cases where the structure of the text is the one that will disambiguate whether a portion of text is Code or English. However, it is clear that the use of these kind of neural network, although viable in other kind of projects, in ours may not be suitable because they are slower and in many cases they need a GPU to speed-up the process.

9 Conclusion and Future Work

Thanks to the Internet, communication and knowledge exchange improved greatly. Nowadays, there are websites were we can find forums and Q&A were we can post messages and asking for help in a general or specialised subject. As well, there are means to communicate like emails and Twitter. All these means are not unknown for software developers and are frequently used by them by interchanging text mixed with natural language, e.g. English, and code like HTML or Python. Although several of these means offer the possibility of separating code and natural language, it is not a characteristic available always. This explain the relevance of the use of code detector in tools that are used for analysing the communication exchanged, like code searchers, duplicate entry verification, among others.

In this article we presented a Code detector developed for a project in which the speed of analysis and size of the model are relevant. Moreover, unlike the state-of-the-art, we explored the newest technologies related to the use of word embeddings and neural networks. The technologies were used in order to create a general, fast, reliable and small Code detector that could be used in our project. Being more specific, we explored how a Dense SVM and a FastText Classifier could be used to create a Code Detector. The experiments were done over a testing data set composed either of documents or sentences, which allow us to see how the size of the context affects the performance. A comparison with naïve baselines were done, but also against a Sparse SVM.

The results showed us, that the most adequate Code Detector is the one trained with a FastText Classifier and trained on a corpus were half of it is a mix different programming languages (Python, C++, Java and JavaScript) and the rest a set of sentences from Wikipedia. We achieve to get an F-score of 0.995 in the prediction of documents, while 0.984 in the prediction of sentences. Moreover, the model is just 7.1 MB. In terms of speed, it was the fastest method, apart from the baselines.

In the future, we will explore the creation of a neural network, similar to the one used in FastText's classifer, but that could have information related to the texts' structure. Maybe not as in recurrent neural networks, but as a part of the token used to train the models. The structure could be obtained using a shallow or even a full parser.

References

[1] Miltiadis Allamanis and Charles Sutton. Mining Source Code Repositories at Massive Scale using Language Modeling. In Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim, editors, *The 10th Working Conference on Mining Software Repositories*, pages 207–216, San Francisco, CA, USA, 2013. IEEE.

[2] Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, and Michele Lanza. Content classification of development emails. In Martin Glinz, Gail Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering (ICSE)*, pages 375–385, Zurich, Switzerland, 2012. IEEE.

[3] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. Extracting source code from e-mails. In Giuliano Antoniol, Keith Gallagher, and Pedro Rangel Henriques, editors, *2010 IEEE 18th International Conference on Program Comprehension (ICPC)*, pages 24–33, Braga, Portugal, 2010. IEEE.

[4] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting Structural Information from Bug Reports. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 27–30, Leipzig, Germany, 2008. ACM.

[5] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching Word Vectors with Subword Information. *Transactions of the Association of Computational Linguistics*, 5:135–146, 2017.

[6] Luigi Cerulo, Massimiliano Di Penta, Alberto Bacchelli, Michele Ceccarelli, and Gerardo Canfora. Irish: A Hidden Markov Model to detect coded information islands in free text. *Science of Computer Programming*, 105(Supplement C):26 – 43, 2014.

[7] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27, 2011.

[8] Yanqing Chen, Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. The Expressive Power of Word Embeddings. *CoRR*, abs/1301.3226, 2013.

[9] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Earlbaum Associates, 2 edition.

[10] Dirk Goldhahn, Thomas Eckart, and Uwe Quasthoff. Building Large Monolingual Dictionaries at the Leipzig Corpora Collection: From 100 to 200 Languages. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Asunción Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of 8th Language Resources and Evaluation Conference (LREC'12)*, pages 759–765, Istanbul, Turkey, 2012.

[11] Jiang Guo, Wanxiang Che, Haifeng Wang, and Ting Liu. Revisiting Embedding Features for Simple Semi-supervised Learning. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 110–120, Doha, Qatar, 2014. Association for Computational Linguistics.

[12] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. Technical report, Department of Computer Science, National Taiwan University, 2003.

[13] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of Tricks for Efficient Text Classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, volume 2, pages 427–431, Valencia, Spain, 2017.

[14] Thorsten Merten, Bastian Mager, Simone Bürsner, and Barbara Paech. Classifying unstructured data into natural language text and technical information. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 300–303, Hyderabad, India, 2014. ACM.

[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[16] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[17] Jonas Močkus, Vytautas Tiešis, and Antanas Žilinskas. The application of Bayesian methods for seeking the extremum. In George Philip Szegö and Laurence Charles Ward Dixon, editors, *Towards Global Optimisation*, volume 2, pages 117–128. North-Holland, 1978.

[18] Samuel Reese, Gemma Boleda, Montse Cuadros, Padró, Lluís, and German Rigau. Wikicorpus: A word-sense disambiguated multilingual wikipedia corpus. In Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of 7th Language Resources and Evaluation Conference (LREC'10)*, pages 1418–1421, La Valleta, Malta, 2010.

[19] Peter C. Rigby and Martin P. Robillard. Discovering essential code elements in informal documentation. In David Notkin, Betty H. C. Cheng, and Klaus Poh, editors, *Proceedings of the 2013 International Conference on Software Engineering*, pages 832–841, San Francisco, CA, USA, 2013. IEEE Press.

[20] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando Pereira, and Kilian Q. Weinberger, editors, *Proceedings of the 25th International Conference on Neural Information Processing Systems*, volume 2, pages 2951–2959, Lake Tahoe, Nevada, USA, 2012. Curran Associates Inc.

[21] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, 2013. Association for Computational Linguistics.

[22] Jie Tang, Hang Li, Yunbo Cao, and Zhaohui Tang. Email data cleaning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 489–498, Chicago, Illinois, USA, 2005. ACM.

[23] Vladimir Zolotov and David Kung. Analysis and Optimization of fastText Linear Text Classifier. *arXiv preprint arXiv:1702.05531*, 2017.