2020

DEEP HYBRID FEATURES FOR CODE SMELLS DETECTION

Abeer Hamdy Dr.
The British University in Egypt, abeer.hamdy@bue.edu.eg

MOSTAFA TAZY
The British University in Egypt

Journal of Theoretical and Applied Information Technology
31st July 2020. Vol.98. No 14
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

DEEP HYBRID FEATURES FOR CODE SMELLS DETECTION

[1]ABEER HAMDY, [2]MOSTAFA TAZY
[1,2]British University in Egypt, Faculty of Informatics and Computer Science, Egypt
E-mail: [1]abeer.hamdy@bue.edu.eg, [2]mostafa.wagih@bue.edu.eg

ABSTRACT

Code smells are symptoms of poor software design and implementation choices. Previous empirical studies have underlined their negative effect on software comprehension, fault-proneness and maintainability. A number of approaches have been proposed to identify the existence of code smells in the source code; recent studies have shown the potential of machine learning models in this context. However, previous approaches did not exploit the lexical and syntactical features of the source code; they instead modelled the source code using software metrics only. This paper proposes an approach for detecting the occurrence of the God class smell which utilizes both, the source code textual features and metrics to train three deep learning networks (i) Long short term memory, (ii) Gated recurrent unit and (iii) Convolutional neural network. We proposed utilizing deep leaning networks as they are reported to outperform traditional machine learning models in several domains including software engineering. To assess the proposed approach, a dataset for the God class smell was built using source codes acquired from the "Qualitas Corpus". Experimental results demonstrated that, the three deep learning networks outperformed three traditional machine learning models: Naïve Bayes, Random forests and Decision trees. Additionally, of the three deep learning networks the Gated recurrent unit model is the superior in this context. Furthermore, combining both, the source code metrics and textual features enhanced the accuracy of detecting the God class smell.

Keywords: *code smells, deep learning, God class, software maintenance, CNN, LSTM, GRU, VSM, IR, text mining*

1. INTRODUCTION

During the software maintenance phase, the system keeps evolving and changing due to (i) the request for feature enhancement, (ii) arise of new requirements, or (iii) bug fixing [1]. Owing to the imposed time constraints and/or the lack of resources, the developers usually do not look for good design solutions before applying the required modifications. This may lead to the introduction of the so called technical debt [2].

The term "code smell" was coined by Fowler [3] as the code structure that requires ("screaming for") refactoring. Later, other researchers defined the code smells as indicators of design problems and/or poor coding practices [4]. For example, the "God class" smell refers to the case in which a class implements a great deal of the system functionalities [3]; it is a complex class that incorporates a high number of instance methods and variables. God class smell occurs as a consequence of violating the principle of "Single Responsibility". Flower [3] presented twenty two code smells, as well as their characteristics and effects. These smells are low-level design problems in the source code; some of them were defined on the class level (e.g. God class and Data class), while others were defined on the method levels (e.g. Long method). Generally speaking, the presence of many code smells in the software hinders its comprehension [5], as well as its evolution and maintainability [6, 7]. So it is important to identify the existence of the smells in order to get the source code refactored.

There has been an ongoing research in this area, on one hand research studies were conducted with the aim of understanding when the code smells are introduced and how do they evolve in the software systems [8-11]. On the other hand, several approaches have been proposed in the literature for detecting the code smells. The early approaches are rule-based approaches [12-17] that characterize the symptoms of the code smells in order to detect them, they include the following phases: Firstly, a collection of software metrics (e.g. Complexity, source line of codes (SLOC)) are identified and computed, then threshold values are determined and applied upon the metrics, finally a set of rules are defined to differentiate between smelly and non-smelly code components. These approaches differ from each other in: (i) the utilized metrics which depend on the code smell under investigation and (ii) the way of combining these metrics together (rules) to identify the smells; for example, such a combination can be implemented using simple AND/OR operators [12]. Although, these approaches showed reasonable accuracy, they suffer a number of limitations that might preclude their use in practice [18,19]. Most importantly, these approaches require the specification of the threshold values, taking into considerations that the thresholds greatly influence the accuracy [18]. Additionally, the agreement between these approaches is low [20]. Some research studies showed that using historical data can improve the accuracy of smell detection [21,22]. Some researchers proposed formulating the problem of the code smell detection as an optimization problem, then applied search algorithms, to solve it, such as Parallel Collaborative search algorithms [23], Competitive Co-evolutionary search [24] and Genetic Programming [25].

Recent approaches utilized machine learning (ML) supervised learning techniques for smell detection [26-34] to overcome the limitations of the rule-based approaches. ML-approaches are based on training a supervised model using data from the same software project or from another project. The source code components are modeled using metrics, same as the heuristic-based approaches, however ML-approaches do not require threshold specifications. They learn from the data to classify a given code component as smelly or non-smelly. However, previous ML-approaches did not benefit from the textual features of the source code. To our knowledge only two studies leveraged the textual features of the source code and the deep learning neural networks for code smell detection [35,36]. Liu et al. [35] trained a Convolutional neural network (CNN) for detecting the Feature envy smell; while, Fakhoury et al. [36] trained a CNN for detecting linguistic smells.

1.1. Aims and Contributions

This work aims at identifying the existence of the God class smell in the source code through extracting the textual features of the source code in addition to its characteristics in terms of software metrics. These hybrid features will be used to train three deep learning networks: (i) Long short term memory (LSTM) [37], (ii) Gated recurrent unit (GRU) [38] and (iii) Convolutional neural network (CNN) [39].

We proposed utilizing these three models as each of them proved its effectiveness in the context of natural language related tasks [40-45]; and the source code is a special type of text. Furthermore, each of these three deep learning architectures works in a different way. The CNN is able to learn the local features of the input text while both of the LSTM and GRU are able to learn dependencies among a sequence of terms and generates a vector representation; but GRU has simpler structure than the LSTM. There is no decisive conclusion in the literature which deep learning model is the best for the text classification task. This is the reason, in this paper we experimented with the three models.

We selected the God class smell to detect, as the study conducted by Palomba et al. [46] demonstrated that the smells characterized by complex and/or long code (e.g. God class) are highly dispersed. Additionally, the smelly classes have a higher probability to change and fault-proneness than non-smelly classes.

1.2. Research Questions

This paper aims at answering the following research questions:

RQ1: Is it possible for the source code textual features to be an alternative to the source code metrics in detecting the God class smell?

RQ2: How effective are the proposed three deep learning neural networks in detecting the God class smell in comparison to traditional machine learning techniques?

RQ3: Does the combination of source code textual features and metrics boost the performance of the deep learning networks in detecting the God class smell?

The rest of the paper is structured as follows: Section 2 introduces deep learning neural networks, Section 3 summarizes the previous work that employed machine learning techniques to identify code smells in the source code. Section 4 discusses the proposed methodology. While section 5 discusses the experiments and results. Finally, section 6 concludes the paper and presents further possible extensions.

2. DEEP LEARNING NEURAL NETWORKS

Deep learning neural networks [47] is a recent field in machine learning that has been reported to achieve a remarkable performance in the area of vision. Nevertheless, they have been used extensively for tackling various classification and prediction problems. The widely used deep learning architectures in the literature are CNNs and RNNs; which are introduced in the following subsections.

2.1. Convolutional Neural Networks (CNNs)

CNNs are inspired by the hierarchical organization of the visual cortex and have been proven effective for text classification same as they are effective for image analysis [40,41]. In text classification applications the text is supplied to the CNN using one of two forms. The first form is a 1D vector obtained from a model like the vector space model (VSM) [48]; in this case the CNN is called one-hot CNN. The other form is the 2D matrices obtained from one of the word embedding models [49]. Although the VSM representation does not represent the semantic relations efficiently but it is more robust to data sparsity and speeds up the training by having fewer parameters.

To explain how CNN is used for text classification, let's consider a sentence of length l words that is represented using a 1D vector as follows:

$$x_{1:l} = x_1, x_2 \dots\dots x_l \tag{1}$$

Where, $x_{i:j}$ is a window of words starts from word i to word j ; x_i is a value represents the weight/importance of the word i to the sentence $x_{1:l}$.The convolution operation involves applying a filter to windows of size n words to produce new features [40]. Assume a new feature c_i that is generated from applying a filter over a window of words $x_{i:i+n-1}$ will be calculated using equation 2:

$$c_i = f(w.x_{i:i+n-1} + b) \tag{2}$$

Where, f is a non-linear function and b is a bias term. The filter is applied to every possible n words window. Consequently, a feature map C of size $(l - n + 1)$ is produced, and defined as follows:

$$\text{C} = \{c_1, c_2, \dots\dots c_{l-n+1}\} \tag{3}$$

A max-pooling operation is then applied to the feature map C to capture the key features. So, the max- pooling layer reduces the dimensionality of the feature map; consequently, it is considered a feature selection operation. Usually, the CNN utilizes multiple filters with varying window sizes (kernels) to produce different features.

Finally, the output of the max-pooling layer is passed to a dense Softmax layer, whose output is probabilities of the class labels, Fig. 1 depicts a sample CNN network encompasses two convolutional layers.

2.2. Recurrent Neural Networks (RNNs)

RNN [47] is another deep learning neural network that has been proven to be effective in processing sequential data like text; due to its capability to dynamically "memorize" information provided in previous states and incorporate them to a current state. The RNN computational units are connected in a directed cycle such that at each time step t, each unit gets two inputs: (1) the current time step X_t , (2) the hidden state of the same unit from previous time step h_{t-1} , and returns the new hidden state h_t, as depicted by Fig. 2; h_t is calculated using Equation (4).

$$h_t = f(X_t\,, h_{t-1}) \tag{4}$$

Where, h_0 is usually initialized as a vector of *Zeros* and X_i could be a 1D or 2D vector. f is a recursive function; the simplest recursive function is implemented using equation 5 as follows:

$$h_t = tanh(W_x X_t\,, U_h h_{t-1}) \tag{5}$$

Where, W_x, U_h are weight matrices.

However, the simple recursive function given by equation 5 suffers from the problem of vanishing gradient, so more complicated recursive functions were recommended in the literature, e.g. LSTM [37] and GRU [38].

2.3. Long Short Term Memory (LSTM)

LSTM is one of the recent variants of RNN, which is able to preserve long-term dependencies. Furthermore, LSTM has been found to perform reasonably well on various data sets within the context of applications that exhibit sequential patterns, such as: (i) text classification [43], (ii) language translation [42] and (iii) source code clone detection [44]. The LSTM computational unit comprises a memory cell and three gates: (i) an input gate, (ii) an output gate and (iii) a forget gate, as depicted by Fig. 3(a). The three gates control the flow of information into and out of the cell; where, each gate is composed of a sigmoid layer and a pointwise multiplication operation. This structure enables the LSTM to overcome the vanishing gradient problem. The inputs to the LSTM unit at time step t are: h_{t-1}, c_{t-1}, c_t, the outputs are: c_t, h_t and they are updated by equations 6-11 as follows:

$$i_t = \sigma(\,W_i X_t + U_i h_{t-1} + b_i) \tag{6}$$
$$o_t = \sigma(W_o\, X_t + U_o h_{t-1} + b_o) \tag{7}$$
$$f_t = \sigma\left(W_f\, X_t + U_f h_{t-1} + b_f\right) \tag{8}$$
$$\widetilde{c_t} = \tanh(\,W_c\, X_t + U_c h_{t-1} + b_c) \tag{9}$$

$$c_t = (i_t \odot \tilde{c}_t) + (f_t \odot c_{t-1}) \quad (10)$$

$$h_t = o_t \odot tanh(c_t) \quad (11)$$

Where, i_t, f_t o_t are the input, forget and output gates. $\tilde{c}_t$, c_t are the candidate and new memory cell content. h_t is the activation. σ is the logistic sigmoid function, ⨀ is the pointwise vector multiplication. h_0, c_0 are usually initialized to zeros.

2.4. Gated Recurrent Unit (GRU)

GRU is a variant to the LSTM which has simpler structure, yet is still able to preserve long term dependencies. Furthermore, GRU has achieved competitive performance with LSTM for many natural language tasks [38], [42].

GRU computational unit merges the LSTM forget and input gates into a single gate, and merges the memory cell state and hidden state, in addition to other changes, as depicted by Fig. 3(b). The GRU unit output at a time step t is updated using equations 12-15 as follows:

$$r_t = \sigma(W_r X_t + U_r h_{t-1}) \quad (12)$$
$$z_t = \sigma(W_z X_t + U_z h_{t-1}) \quad (13)$$
$$\tilde{h}_t = \tanh(W_h X_t + U_h (r_t \odot h_{t-1}) \quad (14)$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t) \quad (15)$$

Where, r_t, z_t are the update and the reset gates; h_t ,$\tilde{h}_t$: are the activation and the candidate activation.

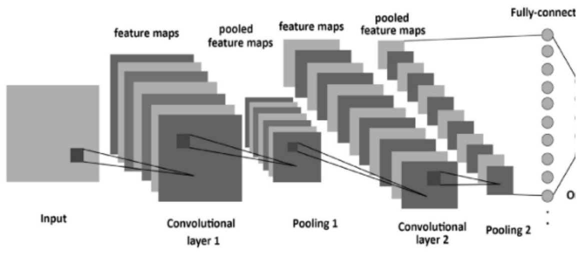


Fig. 1: A sample CNN network.

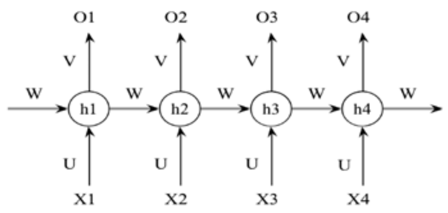


Fig. 2: An unrolled RNN neural network unit.

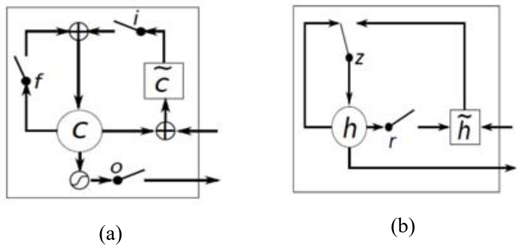


Fig.:. (a) LSTM unit, (b) GRU unit.

3. LITERATURE REVIEW

Artificial intelligence techniques drew the attention of software engineering researchers and were employed with a multitude of problems including: bug triage, effort estimation, next release problem and code smells detection [26-36], [50-57]. This section summarizes the early approaches for code smells detection that utilized machine learning techniques. The idea of using machine learning for code smell detection was originally suggested by Kreimer [27]; the author utilized the Decision trees in detecting the Blob and Long method code smells. Kreimer used metrics composed of the number of variables, number of methods, and number of sentences as decision criteria for the Blob detection. Later on, Amorim et al. [28] extended the work of Kreimer [27] to detect 12 anti-patterns. Khomh et al. [29] proposed the Bayesian detection expert (BDTEX) which is a metric based approach to build Bayesian Belief Networks using the definitions of the smells. This approach provides the probability of occurrence of a certain smell instead of classifying the code as smelly or non-smelly. The authors validated their approach on three smells which are: God class, Functional decomposition, and Spaghetti code smells. Maiga et al. [30] proposed using Support Vector Machines classifiers for detecting four smells: God class, Functional decomposition, Spaghetti code, and Swiss army knife. The authors used the PADL meta-model [31] to compute 60 structural metrics to identify the God class. Fontana et al. [32] built a massive dataset and conducted a large study to compare among 16 different machine learning algorithms for the detection of four smells: Data class, God class, Feature envy, and Long method. They conducted their experiments on 74 software systems from the Qualitas Corpus dataset [58]. They computed a set of independent metrics and used them as input features to the different machine learning techniques. Furthermore, their datasets were balanced using an under-sampling technique to avoid the poor classification performances commonly reported from machine learning models on imbalanced datasets. Their study showed that the J48 decision tree is the superior technique in detecting each of the God class and the Feature envy smells. Di Nucci et al. [33] replicated the study of [32] after merging their datasets together to make the datasets more realistic. Di Nucci et al. [33] showed empirically that the performance reported in [32] could be attributed to the unrealistic datasets (balanced and each data set has only one smell) not to the capabilities of the machine learning techniques.

More recently, Liu et al. [35] trained a CNN model to detect the Feature envy. They fed the CNN with the name of the feature, the feature's enclosed class and the target class. The CNN is also fed with a numerical metric which is the distances between the feature and its enclosing class and the feature and its target class. Fakhoury et al. [36] trained CNN for detecting linguistic smells. Nevertheless, Fakhoury found out that some regular machine learning techniques like Support vector machine can achieve the same performance of the CNN in detecting the linguistic code smells.

3.1. Difference from previous work

The work in this paper is different from the work of [35] and [36] in the following:

(1) The smell under study in this work is the God class smell.
(2) We utilized hybrid features comprised of source code textual features and metrics.
(3) We compared among the performance of three deep learning neural networks: (i) LSTM, (ii) GRU and (iii) CNN in order to identify the most suitable network for detecting the occurrence of the God class smell.

4. METHODS AND TOOLS

4.1. Dataset Building

The dataset used to assess our approach was built on the dataset compiled by Fontana et al. [32]. This dataset is a fragment of the software systems encompassed in the Qualitas Corpus (QC) [58]. QC is widely used in empirical software engineering studies as it has a massive collection of software systems, written in Java code, which are developed in various domains and have different sizes. Fontana et al. [32] used QC to develop a massive dataset for the code smell detection problem by modelling a 74 systems (out of 111 systems) in terms of a set of object-oriented metrics. The software metrics utilized model six aspects of the software which are: (i) size, (ii) cohesion, (iii) coupling, (iv) inheritance, (vi) encapsulation and (vii) complexity. For each system a 61 source code metrics were computed for the class level code smells (God class and Data class) and 82 for the method level smells (Feature envy and Long Method). The metrics were computed using the DFMC4J tool. Then the authors labelled the data instances using a set of code smells detection tools (including: iPlasma, PMD and AntiPattern) in addition to a manual validation by three experts. Afterwards, a balanced dataset was generated for each of the four code smell types; such that each dataset contains 1/3 smelly samples and the rest 2/3 are non-smelly samples. Each dataset has a size equal to 420 instances.

We merged the God class dataset with the Data class one, as recommended by Di Nucci et al. [33], to make the God class dataset more realistic. As, the resulting dataset includes more than one smell and less proportion of smelly instances with God class.

To serve the aim of this paper we acquired the source code of the software systems encompassed by this dataset from the QC to extract their textual features. However, during the process of accessing the source codes some software systems were unreachable; consequently, we reconstructed a data set consisting of 684 instances (out of the original 840 instances) with 554 non-smelly instances and 130 smelly instances.

4.2. Proposed Approach

Fig. 4 shows the phases of the proposed methodology for God class detection. It starts with pre-processing the source code corpus using the natural language processing (NLP) techniques to extract the textual features. Also, the source code is characterized using suitable software metrics for detecting the occurrence of God class smell. However, this part is not implemented in this paper, as explained in the dataset collection section; we used Fontana et al. [32] metric features. The two sets of features are combined and utilized to train three deep learning neural networks. The following subsections discuss each of these phases in detail.

4.2.1. Source code pre-processing

This section discusses the pre-processing of the source code to extract its textual features, taking into consideration the language keywords. Then, convert these textual features into numerical form that is suitable to be passed to the machine learning models. Source code is not simply a plain text it contains multiple aspects of information such as tokens and control flows. In this work we consider only the tokens.

The pre-processing process compromises two consecutive tasks: (i) tokenization then (ii) vectorization. Javalang library [59] was used for the tokenization; it is a python library that provides a Lexer and a Parser to the Java code. Whereas, NLTK [60] was used for the vectorization.

Tokenization: Each class is parsed to generate its sequence of tokens. Tokens are considered the smallest form of data that can be interpreted by a compiler than a representation to the programming language elements such as reserved word. All the identifiers are parsed into a sequence of tokens according to the camelCase and underscore heuristics. For example, a method name “listFiles” or “list_files” will be parsed into the tokens “list” and “files”. Finally, all the tokens are normalized through transferring them into lowercase.

Vectorization: A vector space model [48], [51-55] was built, where each class is represented as a 1D vector. All the vectors have equal size equal to the number of unique tokens in the dataset. Each value in the vector represents the weight of the corresponding token for the class. The term frequency-inverse document frequency (TF-IDF) weighing scheme was used in this work [48] and is computed by equation `16

$$tf - idf_{i,j} = \frac{tf_{i,j}}{n} log \frac{D}{df_i} \qquad (16)$$

Where, $tf - idf_{i,j}$ is the TF-IDF of token i in class j, $tf_{i,j}$ is the number of times token i appears in class j, n is the total number of unique tokens in the source code dataset, D is the total number of classes in the dataset, df_i is the number of classes that include the token *i*.

4.2.2. *Proposed Deep Learning Architectures*

The deep learning architectures utilized in this paper were supplied with the one-hot encoded vectors generated by the pre-processing phase. Keras [61] was used in the implementation; the following subsections explain the configuration of each utilized architecture.

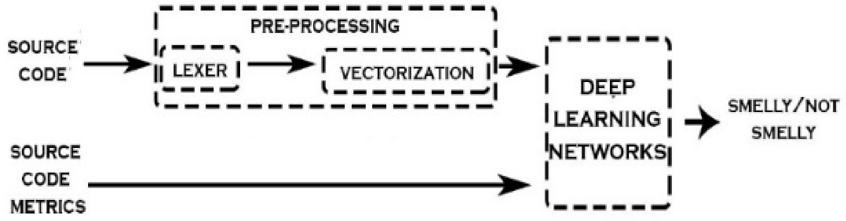


Fig. 4: Proposed methodology for God class detection.

CNN ARCHITECTURE

Fig. 5(a) depicts the architecture of the CNN architecture utilized in this paper. It consists of a stack of convolution stages, for feature selection, followed by a stack of dense layers for classification. Each convolution stage starts with a convolution layer followed by a batch normalization layer then a max pooling layer. The convolution layer applies a set of filters on the input sequence and produces the feature map which represents an input to the max pooling layer. Max-pooling reduces the dimensionality of the feature map by half. We experimented with different depths of the convolution stack stages, different values to the filters and kernels; table I Summarizes the CNN parameters used in the experiments.

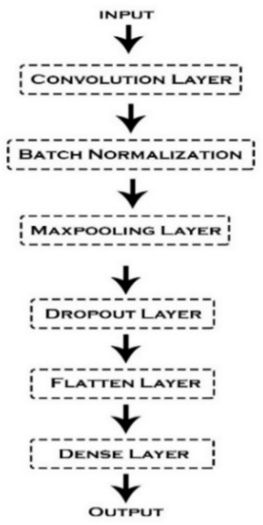
The output of the last max pooling layer is connected to a dropout layer. Dropout performs regularization by ignoring some random nodes during training to prevent over-fitting. In our experiments we set the dropout rate to be equal to 0.5 which means that the nodes to be ignored are randomly selected with probability equal to 0.5.

The output of the last dropout layer is fed into a dense layer which consists of a fully connected multi-perceptron neural network that works as a classifier.

We used a stack of two dense layers. The first dense layer with 32 units and Relu activation; followed by the second dense layer with number of outputs set to equal one, in order to make predictions on whether the God class smell occurs/does not occur in a given source code. This layer uses the sigmoid activation function in order to produce a probability within the range of 0 to 1. Cross entropy or log loss is the loss function we utilized as it is proved effectiveness with the binary classification problems. Finally, we set the maximum number of epochs to equal 100.

LSTM/GRU ARCHITECTURES

Fig. 5(b) depicts the architecture of the utilized LSTM/GRU models which consists of a stack of LSTM/GRU layers, dropout layer then a dense layer. The LSTM/GRU layer learns a representation for each source code class. We set the regular dropout (dropout layer) to 0.5, while the recurrent dropout parameters of the LSTM/GRU layer to 0.1. The recurrent dropouts drop the connections between the recurrent units along with dropping units at inputs and/or outputs. We experimented with different values to the LSTM/GRU units (32,64). The dense layer is same as the one used with the CNN architecture. Table II Summarizes the LSTM/GRU parameters used in the experiments.



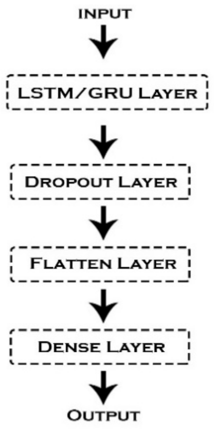


Fig. 5: (a) Proposed CNN architecture, (b) Proposed LSTM/GRU architecture.

Table I: CNN architecture parameters.

# of convolution stages	1,2,3
# of filters	32,64
Kernel sizes	2,3
# of Dense layers	2
Drop out rate	0.5
Loss fn.	Binary cross entropy
# of epochs	100

Table II: LSTM/GRU architecture parameters.

# of LSTM/GRU layers	1,2,3
# of units	32,64
Regular drop out rate	0.5
Recurrent drop out rate	0.1
# of Dense layers	2
Loss fn.	Binary cross Entropy
# of epochs	100

4.2.3. *Traditional Machine Learning Models*

For the purpose of comparison, we implemented three traditional machine learning models, that are reported in the related work of having high classification performance for smelly/non-smelly source code; these models are: (i) Naïve bayes, (ii) C4.5 decision trees and (iii) Random forests. Scikit-learn [62] was used for the implementation of these models.

Decision trees: is a supervised machine learning technique that creates a flow chart like model which is capable of classification, through learning some decision rules inferred from the data features.

Random Forests: is a classifier that builds several decision trees which forms a forest of random classifiers, each one uses a specific subset of the input features. Each tree gives a classification (vote). The class that is having the most votes among all the trees in the forest, is selected.

Naïve bayes: is a probabilistic classifier that is based on Bayes' theorem. However, it presumes that the input features are independent among each other".

4.3. Performance Metrics

The performance of our approach was assessed using the popular metrics in evaluating the binary classifiers, which are: Precision, Recall and F-measure metrics.

Precision is defined as the percentage of the correctly identified code smells to the total detected code smells; it is given by (17).

$$\text{Precision} = \frac{\text{TP}}{\text{TP+ FP}} \quad (17)$$

Where, TP (true positives) represents the number of smelly instances that are correctly detected, FP (false positives) represents the number of non-smelly instances that are incorrectly identified as smelly.

Recall is defined as the percentage of the correctly identified code smells to the total number of the actual code smells; it is given by (18).

$$\text{Recall} = \frac{\text{TP}}{\text{TP+ FN}} \quad (18)$$

Where, FN (false negatives) represents the set of smelly instances that are missed.

F-measure: the harmonic mean of the precision and recall metrics, it represents a balance between their values. It is given by (19).

$$\text{F} - \text{measure} = 2 * \frac{\text{Precision}*\text{recall}}{\text{Precision+Recall}} \quad (19)$$

5. EXPERIMENTS AND RESULTS

We designed the experiments to answer the previously mentioned research questions, as follows:

Answer to RQ1: Is it possible for textual features to be an alternative to the source code metrics for detecting the occurrence of God class smell?

The aim of this question is to evaluate the performance of the deep learning architectures and the traditional classifiers when they are trained using each of the two sets of features, metrics and textual, individually. So, we run two sets of experiments, in the first set we utilized the 61 software metrics recommended by Fontana et al. [32] as the input features to the classifiers, while in the second set we used the extracted textual features. Table III and Fig. 6 depict the results of all the experiments. As could be observed from Fig. 6, that CNN, NB, RF, C4.5 suffered drop in their performance when utilized the textual features; their F-measures dropped by 7%, 4%, 7% and 5% respectively. While the GRU and the LSTM could almost maintain the same performance across the two sets of metrics, their F-measures dropped only by 2% and 1% respectively.

Conclusion: The selection of the classifier is so important when utilizing the textual features instead of the metrics for detecting the God class smell. Some classifiers can maintain almost the same classification performance across the two sets of features while others suffer drop in the performance when replacing the metrics with the textual features. Maybe including more textual features of the source code (e.g. the parse tree) or using embedding vectors, instead of the VSM model, enhance the performance of the classifiers.

Answer to RQ2: How effective are the three deep learning neural networks in detecting the God class smell in comparison to traditional classifiers?

As observed from Figs. 7 and 8 that the three deep learning architectures are superior to the NB, RF and C4.5 regardless of the type of features used in the training. However, the CNN achieved the lowest performance across the three deep learning architectures. While The GRU is the superior model; GRU could achieve an improvement equal to 12% (using metrics), 14% (using textual) over the NB (the superior traditional classifiers).

Conclusion: Generally, deep learning architectures could achieve better results than traditional machine learning models in detecting the God class. Furthermore, the GRU is the superior model in this context.

Answer to RQ3: Does the combination of the source code textual features and metrics boost the performance of the deep learning networks in detecting the occurrence of the God class smell?

The aim of this question is to assess the performance of the proposed methodology, utilizing hybrid features (combination of the metrics and textual) in training the deep learning architectures. We trained each of the three proposed deep learning architectures using the hybrid features; Table IV and Fig. 9 depict the results of the experiments. As could be observed utilizing the hybrid features boosted the performance of the LSTM and GRU, while the performance of the CNN was declined. Moreover, GRU is the superior architecture.

Conclusion: Utilizing more features does not always boost the performance of the deep learning architectures. In our context, it boosted the performance of each of the LSTM and the GRU while harmed the performance of the CNN.

Table III: Performance comparison among deep learning architectures and three traditional machine learning techniques in detecting the God class smell.

Approach		P	R	F
LSTM	Metrics	90%	86%	88%
	Textual	89%	86%	87%
GRU	Metrics	**91%**	**88%**	**89%**
	Textual	89%	87%	87%
CNN	Metrics	84%	86%	87%
	Textual	80%	82%	80%
Naïve Bays	Metrics	82%	74%	77%
	Textual	79%	68%	73%
Random forest	Metrics	58%	62%	59%
	Textual	49%	56%	52%
C4.5	Metrics	60%	64%	61%
	Textual	61%	52%	56%

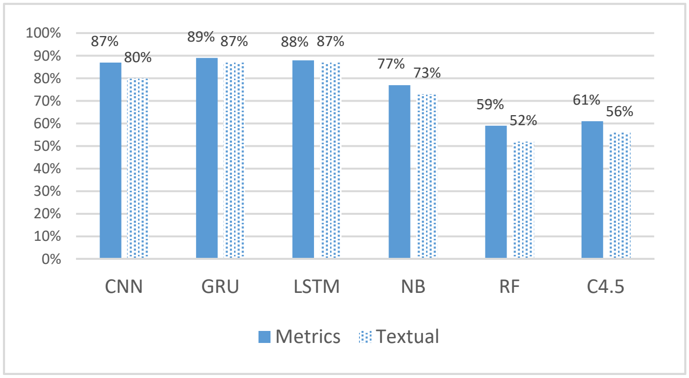


Fig 6: Comparison among the F-measure of the six classifiers when trained using metrics or textual features.

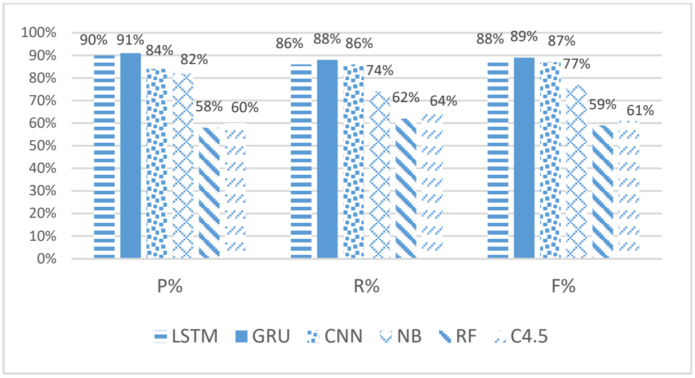


Fig. 7: Performance of the six classfiers trained using the source code metrics only.

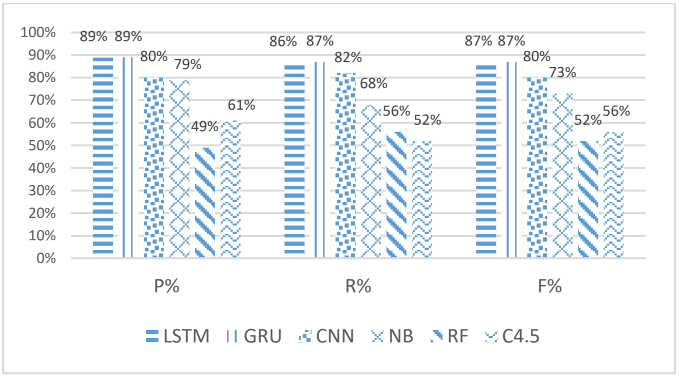


Fig. 8: Performance of the six classifiers trained using the source code textual features only.

Table IV: Performance of the proposed deep learning architectures trained using three set of features: metrics, textual and hybrid.

Approach		P	R	F
LSTM	Metrics	90%	86%	88%
	Textual	89%	86%	87%
	Hybrid	93%	90%	91%
GRU	Metrics	91%	88%	89%
	Textual	89%	87%	87%
	Hybrid	**94%**	**91%**	**92%**
CNN	Metrics	84%	86%	87%
	Textual	80%	82%	80%
	Hybrid	81%	83%	81%

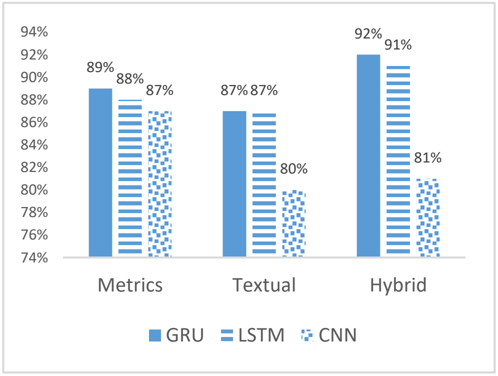


Fig. 9.: F-measure of the three deep learning architectures trained using three set of features: metrics, textual and hybrid.

6. CONCLUSION

The paper proposed a methodology for detecting the occurrence of the God class smell in the source code. The proposed methodology leveraged both of the characteristics (textual and metrics) of the source code and the capabilities of the deep learning architectures. To achieve our purpose, we built a dataset for the God class smell in source codes acquired from the "Qualitas Corpus" repository. The textual features of the source code were extracted using the natural language processing techniques, then integrated with the 61 metrics features computed by Fontana et al. [32]. Three deep learning architectures (CNN, LSTM and GRU) were trained using three different types of the source code features (metrics, textual features and hybrid metrics-textual features). Moreover, three traditional machine learning techniques (NB, RF, C4.5) from the literature were trained, to compare their performance with the proposed deep learning architectures. It was found that the performance of the three deep learning architectures is superior to the traditional machine learning techniques; and the GRU is the superior model. Furthermore, each of the GRU and the LSTM could maintain almost the same performance when trained using metrics or textual features only; while the CNN and the traditional machine learning techniques suffered a drop in their performance when trained using the textual features. It is noteworthy that the GRU and the LSTM achieved their best performance when they were trained using the hybrid features; while the best performance of the CNN was achieved when trained using the metrics features only.

As an extension to this work, the accuracy of detecting the God class could be boosted through utilizing other deep learning architectures like the hybrid CNN-RNN architecture; in addition to, representing the textual features of the source code using techniques other than the VSM (e.g. word embedding). Furthermore, datasets for other code smells could be built such that the proposed technique can be extended to detect the occurrences of other code smells.

REFERENCES

[1] M.M. Lehman, "Programs, life cycles, and laws of software evolution", in *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing technical debt in software engineering, Dagstuhl Reports, 6, Schloss Dagstuh- l-Leibniz-Zentrum fuer Informatik, 2016.

[3] M. Fowler, Refactoring: improving the design of existing Code, Addison-Wesley, 1999.

[4] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp.158 – 173, 2018.

[5] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two anti-patterns, blob and spaghetti code, on program comprehension", in *15th European conference on Software maintenance and reengineering (CSMR)* IEEE, 2011, pp.181–190.

[6] A. Yamashita and L. Moonen. "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, 2013, pp. 682–691.

[7] S. Olbrich, D.S.Cruzes, V. Basili, N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM09*, Washington, DC, USA: IEEE Computer Society, 2009, pp. 390–400.

[8] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshy- vanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063-1088, 2017.

[9] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Inf. Softw. Technol.* Vol. 99, pp. 1–10, 2018.

[10] A. Chatzigeorgiou, A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Quality of Information and Communications Tech- nology (QUATIC), 2010 Seventh International Conference on the*, IEEE, 2010, pp. 106–115.

[11] R. Peters, A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, IEEE*, 2012, pp. 411–416

[12] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2004, pp. 350–359.

[13] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proc. Int'l Software Metrics Symposium (METRICS), IEEE*, 2005, p. 9.

[14] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*, Berlin/Heidelberg: Springer-Verlag, 2006.

[15] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Gu´eh´eneuc, "Numerical signatures of antipatterns: An approach based on B-splines," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE, 2010, pp. 248–251.

[16] N. Moha, Y.-G. Gu´eh´eneuc, L. Duchien, and A.-F. L. Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[17] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[18] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, E. Figueiredo, "A review-based compar- ative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, in: EASE '16*, New York, NY, USA: ACM, 2016, pp. 18:1–18:12.

[19] M. Zhang, T. Hall, N. Baddoo, "Code bad smells: a review of current knowledge," *J. Softw. Maint. Evol.*, Vol. 23, no. 3, pp. 179–202, 2011.

[20] F.A. Fontana, P. Braione, M. Zanoni, "Automatic detection of bad smells in code: an experimental assessment," *J. Object Technol.* Vol. 11, no. 2, pp. 1–5, 2012.

[21] F. Palomba, G. Bavota, M. Di Penat, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Trans. on Software*

Engineering, vol. 41, no. 5, pp. 462–489, May 2015.
[22] D. Ratiu, S. Ducasse, T. Gˆırba, and R. Marinescu, "Using history information to improve design flaws detection," in *European Conf. on Software Maintenance and Reengineering (CSMR)*, IEEE, 2004, pp. 223–232.
[23] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, Sept 2014.
[24] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Ben Chikha, "Competitive coevolutionary code-smells detection," in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science, Berlin/Heidelberg: Springer, 2013, vol. 8084, pp. 50–65.
[25] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Trans. Software Engineering Methodology*, vol. 24, no. 1, pp. 6:1–6:44, Oct. 2014.
[26] M.I. Azeem, F. Palomba, L. Shi, Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115-138, 2019.
[27] J. Kreimer. "Adaptive detection of design flaws," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117–136, 2005.
[28] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro. "Experience report: Evaluating the effectiveness of Decision trees for detecting code smells," in *IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 261–269.
[29] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.
[30] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2012, pp. 278–281.
[31] Y.-G. Guéhéneuc, "Ptidej: Promoting patterns with patterns," in *Proceedings of the 1st ECOOP workshop on Building a System using Patterns*, Berlin/Heidelberg: Springer-Verlag, 2005.
[32] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.
[33] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" *In IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 612–621.
[34] M. Hadj-Kacem and N. Bouassida, "A Hybrid Approach to Detect Code Smells using Deep Learning," in *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, 2018, pp. 137-146.
[35] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ACM, 2018, pp. 385–396.
[36] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso*, 2018, pp. 602-611.
[37] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[38] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modelling," arXiv preprint arXiv:1412.3555, 2014.
[39] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp.1-9.
[40] Y. Kim, "Convolutional Neural Networks for Sentence Classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Qatar, Doha*, Oct. 2014, pp. 1746–1751.
[41] R. Johnson and T. Zhang, "Semi-supervised convolutional neural networks for text categorization via region embedding," in *Advances in Neural Information Processing Systems*, 2015, pp. 919-927.

[42] K. Cho, B. Van Merri¨enboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," arXiv preprint arXiv:1409.1259, 2014.
[43] C. Baziotis, N. Pelekis, and C. 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)*, pp. 747–754.
[44] H. Wei and M. Li, "Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, pp. 3034–3040, 2017.
[45] T. Wen, M. Gasic, N. Mrkˇsi´c, P. Su, D. Vandyke, and S. Young, "Semantically Conditioned LSTM-based Natural Language Generation for Spoken Dialogue Systems," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1711–1721.
[46] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.
[47] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*,Vol. 1, Cambridge: MIT press, 2016.
[48] A. Hotho, A. Nurnberger, G. Paas, "A brief survey of text mining," *Journal for Computational Linguistics and Language Technology*, 2005, pp. 19–62.
[49] 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*, pp. 3111–3119, 2013.
[50] A. Hamdy, A. El-laithy, "Using smote and feature reduction for more effective bug severity prediction," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 6, pp. 897-919, 2019.
[51] A. Hamdy, A. El-Laithy, "Semantic Categorization of Software Bug Repositories for Severity Assignment Automation, Studies in Computational Intelligence," *Integrating Research and Practice in Software Engineering*, Vol. 851, pp 15-30, January 2020.
[52]A. Hamdy, M. Elsayed, "Towards more accurate automatic recommendation of software design patterns," in *Journal of Theoretical and Applied Information Technology*, 2018, vol. 96, no. 15, pp. 5069-5079.
[53]A. Hamdy, M. Elsayed, "Topic modelling for automatic selection of software design patterns," in *proceedings of the International Conference on Geoinformatics and Data Analysis ICGDA '18*, Prague, Czech Republic, April 20th - 22nd, 2018, pp. 41-46.
[54] A. Hamdy, M. Elsayed, "Automatic Recommendation of Software Design Patterns: Text Retrieval Approach", *Journal of Software*, Vol. 13, No. 4, pp. 260-268, April 2018.
[55] A. Hamdy and A. Mohamed, "Greedy Binary Particle Swarm Optimization for multi-Objective Constrained Next Release Problem," *International Journal of Machine Learning and Computing*, vol. 9, no. 5, pp. 561-568, October 2019.
[56] A. Hamdy, "Genetic fuzzy system for enhancing effort estimation models", *International Journal of Modeling and Optimization (IJMO)*, vol. 4, no.3, June 2014.
[57] A. Hamdy, "Fuzzy Logic for enhancing the sensitivity of COCOMO cost model", *Journal of Emerging Trends in Computing and Information Sciences (CIS)*, vol. 3, no. 9, September 2012.
[58] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas corpus: A curated collection of java code for empirical studies," in *17th Asia Pacific Software Engineering Conference (APSEC)*, 2010, pp. 336–345.
[59] JavaLang, Available: https://pypi.org/project/javalang/.
[60] NLTK, Available: https://www.nltk.org/.
[61] Keras, Available: https://keras.io/.
[62] Scikit, Available: https://scikit-learn.org/.