Early Prediction of Test Case Verdict with Bag-of-Words vs. Word Embeddings

Khaled Walid Al-Sabbagh[1][0000−0003−2571−5099] , Miroslaw Staron[2][0000−0002−9052−0864], Miroslaw Ochodek[3][0000−0002−9103−717X], and Wilhelm Meding[4]

[1] Chalmers | University of Gothenburg, Computer Science and Engineering Department, Gothenburg, Sweden
{khaled.al-sabbagh, miroslaw.staron}@gu.se
[2] Institute of Computing Science, Poznan University of Technology
miroslaw.ochodek@cs.put.poznan.pl Ericsson AB
wilhelm.meding@ericsson.com

Abstract. Regression testing is an important testing activity in continuous integration (CI) since it provides confidence that modified parts of the system have not adversely affected its expected behavior. As test suites grow in size over the evolutionary cycle of software, executing large test suites becomes costly and precluding. Since CI provides a large volume of data, machine learning approaches can be used to allow test orchestrators make inferences about which subset of test cases to run at each CI cycle. MeBoTS is a machine-learning-based method that utilizes CI data to improve test case selection (TCS). In order to decide which extraction algorithm is more suitable, we designed and performed an experiment to investigate the effect of using one of two widely used feature extraction algorithms—Bag of Words (BoW) and Word Embeddings (WE)—on the predictive performance of the MeBoTS classifier. We used stratified cross-validation and precision and recall measures to evaluate the performance of two machine-learning models trained on the input generated by the feature-extraction algorithms. The results from this experiment show a significant difference between the models' performance scores with a higher mean precision and recall scores for the BoW based classifier. We conclude that the use of BoW allows training a more accurate MeBoTS classifier than WE.

Keywords: Machine Learning, Verdicts, Code Churn, Test Case Selection.

1 Introduction

Continuous integration is used increasingly often in software engineering projects. Both large and small software companies use this technique to increase the quality of their products, as continuous integration advocates small increments in software code and frequent testing. However, one of the challenges in continuous integration is the need for resources for testing, which needs to be done on [3] every single code commit. In our work, we addressed this problem by defining a method for predicting whether a given source code commit should be tested by a specific test case. Based on the observation that faults may occur in similar patterns of source code, our method abstracts these patterns by tokenizing lines of code and weighing them using their associated frequency. This means that the numbers of syntax tokens in source code are regarded as predictors to test case failures. Using these feature vectors (source code tokens) as predictors to test failures provides a basis for test orchestrators to prioritize and select test cases.

The prediction algorithms used in our method (MeBoTS, [2]) combines feature extraction from the source code and analysis of test verdicts. It is a modular method where we can select different algorithms for feature extraction and prediction. However, the selections have effect on the performance of the predictions (precision, recall and F1-score). Therefore, in this paper, we study the effects of two different techniques for feature extraction - Bag-of-words (BoW) and Word Embeddings (WE). The first technique is based on statistical analysis of the frequency of using software code statements. The second technique is a semantic program code analysis based on neural networks. Both techniques can be interchanged, but they have different complexity and work differently.

Henceforth, in this paper, we pose the following research question:

RQ: Is there a statistically significant difference between the performance of the test predictor based on the usage of BoW vs. WE?

In order to address this question, we design an experiment, where we use 15 different sets of code commits as experiment factors. We use the statistical performance measures of recall and precision as the dependent variables. The results of the experiment show that the prediction from the WE-based classifier have a statistically significant lower precision and recall scores than that produced by a BoW-based classifier. This means that the simpler method for making predictions is better in this context.

The paper is organized as follows: in Section 2 we introduce related work on code feature extraction techniques in text classifications; in Section 3 we introduce background information; in Section 4 we describe all the steps in the design of our experiment; in Section 5 we provide the results of our experiment; in Section 6 we introduce some threats to validity and limitations; in Section 7 we describe our recommendations for future implementation of MeBoTS; finally, in Section 8 we make our conclusions.

2 Related Work

Since the ultimate goal of this research is to improve TCS, we start by presenting an overview of some proposed TCS approaches and explore their drawbacks.

[3] Copyright 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Then we review the literature on studies that have examined the effectiveness of BoW and WE in text classification.

2.1 Test Case Selection Approaches

Rothermel and Harrold [11] presented an algorithm that employs control dependence graphs of two program revisions, and used these graphs to select test cases that may exhibit changed behavior on a modified revision of the program. The algorithm uses two control dependency graphs to compare changes made between two revisions where each node in the graph contains an actual program statement. Then it uses a list of test execution history that identifies regions in the original program that are reached by each test. If any two children nodes are different, then the algorithm computes and returns a subset of test cases that may have traversed the change in the modified version. A limitation in this approach is that it only selects tests that execute the modified statements but not the actual uses of variables, which leads to the inclusion of unnecessary tests for regression testing.

A TCS technique proposed by Volkolos and Frankl [13] uses textual difference between two versions of system source code and analyzes code modifications. The method uses a C program to remove stylistic differences such as comments and blank lines from the diffed output. Then, it analyzes modifications by checking which statement was modified and select all test cases that traversed through the modified statement. A limitation in this tool is that it considers code changes between versions without any semantic analysis, i.e. changes that do not affect the behavior of the system under test will trigger tests that traverse the modified statements to be selected.

Dynamic slicing based approaches for TCS use slice executions to determine which subset of test cases should be exercised. Agrawal et al [1] proposed an array of techniques that use execution slice (i.e., statements in the program that were executed by a test case) to decide on selective regression tests. The general idea can be summarized as follow: given a set of test cases `t` that were exercised against some execution slices in the original program execution, statements that were not reached in the control of set `t` will not affect the program's output for the same set `t` in future revisions. Based on this, they proposed a technique that required finding execution slices of the program under test given all test cases in a test suite. Then selecting test cases whose execution slices contain modified statements in the new revision.

2.2 Word Embeddings and BoW in Text Classification

In a study conducted by Chao et al. [3], the authors examined the difference between WE and traditional BoW in clinical text classification using an SVM model. The findings suggest that WE vectors outperformed BoW when using 1-gram features, whereas no similar conclusion could be drawn from the same model when using 2-grams features with BoW.

Similarly, Enriquez et al. [5] conducted an experiment to compare the effect of WE in document classification as compared with a BoW based approach. The experiment's data-set was a collection of texts from Amazon, covering 11 different domains. The classification results showed that the use of WE is not sufficient on its own to gain a performance improvement, but rather suggested an integrative approach of both WE and BoW. The evaluation of the performance was based on the accuracy of three versions of classifiers: a version based on BoW, a version based on WE, and a version based on both approaches. The results showed that the combined approach outperformed the classic BoW and WE in 9 out of 11 experimented domains.

3 Background

3.1 Method Using Bag of Words for Test Case Selection (MeBoTS)

MeBoTS is a machine learning based model that aims at predicting test case verdict using historical test execution results and code churns. The term code churns is used here to refer to source code changes made between two check-ins. The method is comprised of 3 steps, as shown in Fig 1. This section briefly describes these steps.

Code Churns Extractor (Step 1) The method uses a code churn extractor program that collects and compiles churns of source code from one or more repositories. The program expects one input parameter: a time ordered list of historical test case execution results queried from a database, where each element in the list is a metadata state representation of a previously run test case. Each state contains a hash reference that points to a specific location in Git's history for the tested check in. The program performs a file comparison utility (diff) across pairs of consecutive commit hashes in the list using the GitPython library [12]. The output is then arranged in a table-like format and written in a csv file, named as 'Lines of Code'.

Textual Analysis and Features Extraction (Step 2) The second step in the method is to extract features from the collected code churns (output of step 1) and transform the source code into a numerical form. For thWe used an open source tool (ccflex) that utilizes BoW for modelling textual data. The input to ccflex is the output of the churn extractor in step 1. ccflex uses each line from the code churn and:

- creates a vocabulary for all lines (using the bag of words technique, with a specific cut-off parameter)
- creates a token for the words that are seldom used(i.e. fall outside of the frequency defined by the cut-off parameter of the bag of words)
- finds a set of predefined keywords in each line
- checks each word in the line to decide if it should be tokenized or if it is a predefined feature

This way of extracting information about the source code is new in our approach, compared to the most common approaches of analyzing code churns. In contrast with other approaches, MeBoTS recognizes what is written in the code, without understanding the syntax or semantics of the code. This means that we can analyze each line of code separately, without the need to compile the code and without the need to parse it.

Training and Applying the Classifier Algorithm (Step 3) We exploit the set of extracted features provided by the textual analyzer in step 2 as the independent variables and the verdict of the executed test cases as the dependant variable, which is a binary representation of the execution result (passed or failed). The MeBoTS method uses a second Python program that utilizes and trains an ML model to classify test case verdicts. The program reads the BoW vector space file in a sequence of chunks, merging the extracted feature vectors and the verdicts vector into a single data frame that gets split into a training and testing set before it is fed into the models for training.

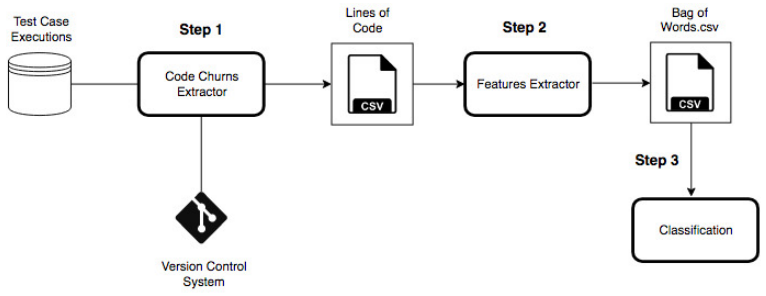

Fig. 1. The MeBoTS Procedure

3.2 Word Embeddings

Word embeddings is one of the approaches used to represent words in a machine-friendly way. It has been proposed as an alternative to the bag-of-words model and quickly become the state-of-the-art method used while training neural networks. In word embeddings, each word is represented as a dense, low-dimensional floating-point vector. Such vectors are learned from data. Another important benefit of using word embeddings is that the geometric relationships between word vectors should reflect the semantic relationships between these words. Using the word embeddings algorithm for source code classification could be beneficial for two main reasons. Firstly, it allows us to represent each of the tokens in a line of code and feed them to a neural network capable of processing sequences (e.g., a convolutional neural network). Secondly, it captures similarities between the roles of tokens in the code without the need for parsing the code.

4 Design of Experiment

4.1 Context of the experiment: Collaborating company

The study has been conducted at an organization, belonging to a large infrastructure provider company. The organization develops a mature software-intensive telecommunication network product. The organization consists of several hundred software developers, organized in several agile teams. The organization is mature with regard to measuring. For instance every agile team, as well as leading functions/roles, uses one or more monitors to display status and progress in various development and devops areas. A well-established and efficient measurement infrastructure, automatically collects and processes data, and then distributes the information needed by the organization.

4.2 Code Churns and Test Executions Data Collection

Our data-set comprised of historical test execution results and code churns for software that has lived and evolved for over a decade at the collaborating company. The analyzed software was written in the C language and contained a few million lines of code and a test pool size of over 10k test cases. In this experiment, our sample data-set comprised of 150k LOC belonging to 12 test cases, with 46% of the lines belonging to the 'passed' class and 54% to the 'failed' one.

4.3 Experiment Subjects

The subjects of our study are samples of the original data-set. The stratified cross-validation technique was used to partition the data-set into 15 different subsets (k=15), such that the representation of the binary strata have approximately an equal representation across the 15 samples. Each subset consisted of 9200k LOC for validation and approximately 140k LOC for training. The representation of the binary classes in each fold followed the same distribution of classes in the original base set with 46% of lines belonging to the 'failed' class and 54% to the 'passed' class.

4.4 Features Extraction with BoW

In this study, we used an open source measurement tool [8] for transforming the experimental subjects into feature vectors using BoW. The tool starts by tokenizing each line in a file using white and special characters: ()[]!@#$%&*-=;:' "~,<>|/?. Then it counts the frequency of occurrence of the tokens found in each line. Depending on whether the frequency count of a token exceeds a lower threshold value, the token gets either selected as a feature or discarded. In our experiment, we kept the frequency threshold value to its default - 25% and set the BoW n-gram to 2 to generate features of two adjacent tokens that are originally separated by white spaces. The resulting space of feature vectors for each subset comprised of a total of 2248 features.

4.5 Features Extraction with WE

In our study, we used the Continuous Bag-Of-Words (CBOW) variant of the Word2Vec word embedding algorithm proposed by Mikolov et al. [7]. We used the implementation available in the Gensim library [10]. IN CBOW, the word embeddings are obtained as a side-effect of training a single-layered neural network to predict a given word based on other words in its neighborhood, called window. In our study, we use the window size equal to 5 and generate embedding vectors of 70 numbers. We trained 15 Word2Vec models on the 15 generated subsets and used these models to preprocess lines of code in both the validation and training sets, for each subset respectively. The resulting vectors for each subset were saved locally so they can be fed as inputs to a neural network classifier. After training the Word2Vec models, tokens that share similar semantic orientation are closely placed in the vector space. Fig 2 illustrates an example of how the tokens in the original data-set (before partitioning) are placed. [4]. The figure was generated using the t-distributed stochastic neighbor embedding (t-SNE) visualization algorithm in Python. The representation of vectors are plotted in two dimension for a set of positive and negative words. As the words are spread across the entire diagram, we can expect that it is possible to find vectors that are unique and therefore are good predictors. The processing pipeline used in this study is presented in Figure 3. In the first step, a line of code is tokenized. Then, each token is replaced by its identifier in the vocabulary. In the following step, we pad each sequence with zeros so all of them contain 55 numbers. The generated sequences can be provided to a neural network as an input. In its first layer, the nn replaces token identifiers with their embedding vectors stored in the so-called embedding matrix. Therefore, an input sequence of 55 numbers is transformed into a 70x55 matrix.

4.6 Evaluation with Random Forest and Neural Network

All the vectors representing the 15 subsets of code churns were categorized into two groups, one group containing the BoW vector files and another for the Word2Vec generated outputs. To evaluate the effect of WE, we ran 15 trials of training a random forest model on the BoW vector representations in the BoW group and another 15 trials for training a CNN model on the files in the WE group. Our choice of training a random forest model on the BoW vectors is based on the fact that RF is known for performing well with high dimensional data, such as those generated by BoW transformations [6]. We used the implementation of Random Forest available in the scikit-learn library [9] and the implementation of CNN in the Keras library [4] to implement the CNN model. Since the WE model results in multidimensional array, we could not use the combination of WE and random forest classifier. Our experiments with the CNN architecture

[4] Each point in the figure represents a word used in the source code. As the figure represents the actual code, and due to a non-disclosure agreement with our industrial partner, words that are not language specific such as variable and class names are not visualized in the figure

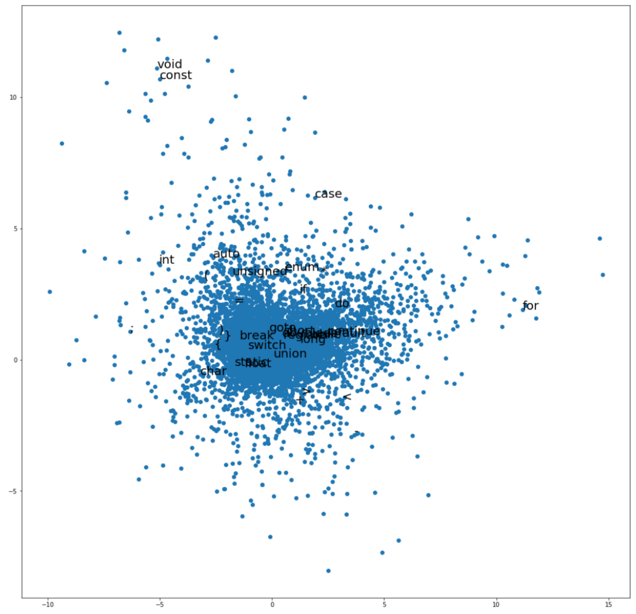

Fig. 2. Semantic orientation of similar tokens in the analyzed software

and the BoW feature extraction, on the other hand, provided results that were too poor to consider in the paper (BoW does not provide the feature set that is rich enough for CNN). Therefore, we selected two pairs which are best suited for each other, rather than forcing the algorithms to work with the feature extraction technique that is not suitable for them. For each training trial, we recorded two performance metrics: precision and recall. The architecture of the CNN is presented in Figure 4. It accepts input as a sequence of vectors, each containing 55 numbers representing identifiers of tokens in the Word2Vec vocabulary. In the first layer, these vectors are transformed into matrices (70x55) using word embeddings (see Figure 3 for details). We use two convolutional layers consisting of 20 and 16 filters, respectively. The output of each is subjected to maximum pooling (pooling size = 3) to reduce the dimensionality of the features maps. The output of the last maximum pooling layer is flattened to a vector of 96 numbers and processed in the dense layer to produce a 1x1 output with the use of the sigmoid activation function. The precision and recall scores of the two models across the 15 folds as shown in Table 2.

5 Results

To decide whether to use a parametric or non-parametric statistical test, we checked if the data sample was normally distributed. We plotted frequency histograms for the precision and recall scores of both models (RF and CNN) for

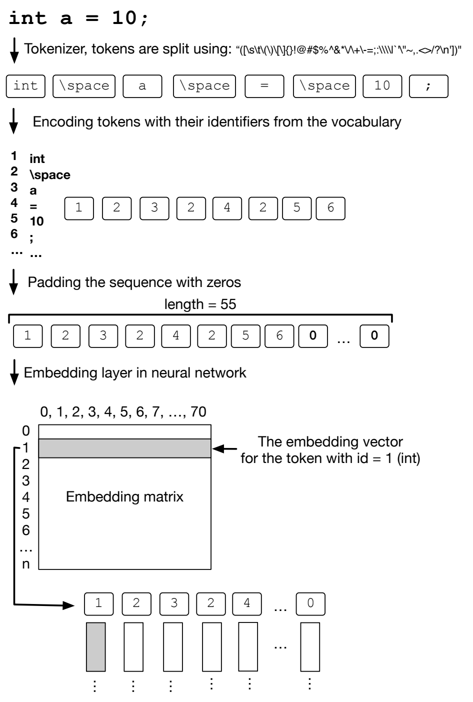
int a = 10;
Tokenizer, tokens are split using: “([\s\t\(\)\[\]{}!@#$%^&*\/\+\-=;:\\\\|`'\"~,.<>/?\n'])"
int
\space
a
\space
=
\space
10
;
Encoding tokens with their identifiers from the vocabulary
1 int
2 \space
3 a
4 =
5 10
6 ;
... ...
1 2 3 2 4 2 5 6
Padding the sequence with zeros
length = 55
1 2 3 2 4 2 5 6 0 ... 0
Embedding layer in neural network
0, 1, 2, 3, 4, 5, 6, 7, ..., 70
0 1 2 3 4 5 6 ... n
Embedding matrix
The embedding vector for the token with id = 1 (int)
1 2 3 2 4 ... 0

Fig. 3. Preprocessing the lines of code to be used as input for the neural network.

the 15 folds and examined the distribution of the points. We decided to run a Shapiro-Wilk test to check if the distribution of points follow a Gaussian curve. The test results were statistically significant for the RF models when using BoW (Precision with BoW: Test statistic = 0.484, p-value = 0.000, Recall with BoW: Test statistic = 0.538, p-value = 0.000), which means that the assumption of normality in both samples can be rejected. Conversely, the results of precision and recall with WE suggest that the distribution of both samples were normal (Test statistic = 0.929, p-value = 0.262, Recall with WE: Test statistic = 0.893, p-value = 0.075). Since the statistical results of the Shapiro-Wilk test suggest that we have issues with normality in the precision and recall results, we decided to run a non-parametric test for comparing the difference between the precision and recall scores obtained by both models. The Mann whitney rank-based test was selected as an appropriate method since it can handle skewed data where the data does not follow a normal distribution.

We found a significant difference between the precision and recall scores for both the RF and CNN models. The results of the comparison for the precision scores showed a test statistics of 12.5 and a p-value below 0.001. Similarly, the comparison between the recall scores for the same models reported a test statistics of 32.5 and a p-value of less than 0.001, suggesting a significant difference in the recall metrics. Table 1 summarizes the mean scores of the 4 performance met-

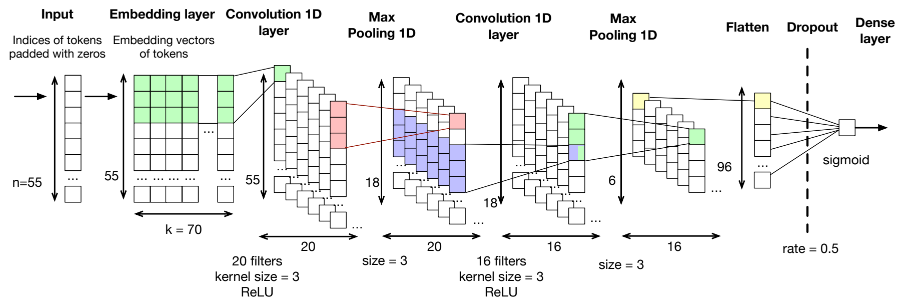

Fig. 4. The Architecture of Convolutional Neural Network.

rics (precision with BoW, precision with WE, recall with BoW, and recall with WE). The results show a higher mean precision and recall for the RF model than those obtained by the CNN model. This brings us to believe that using BoW for features extraction in the MeBoTS is more effective than WE in this context.

Table 1. Mean Values for the Precision and Recall

	precision with BoW	precision with WE	recall with BoW	recall with WE
mean	0.9	0.64	0.93	0.78

Table 2. The Precision and Recall Scores for RF and CNN Models

Fold	FE	Precision	Recall	Fold	FE	Precision	Recall	Fold	FE	Precision	Recall
k=1	BoW	0.6	0.61	**k=6**	BoW	0.9	0.92	**k=11**	BoW	0.944	0.994
	WE	0.6	0.706		WE	0.63	0.75		WE	0.69	0.95
k=2	BoW	0.91	0.96	**k=7**	BoW	0.949	0.952	**k=12**	BoW	0.928	0.959
	WE	0.68	0.885		WE	0.63	0.71		WE	0.65	0.81
k=3	BoW	0.908	0.919	**k=8**	BoW	0.937	0.959	**k=13**	BoW	0.9	0.94
	WE	0.64	0.756		WE	0.67	0.82		WE	0.6	0.74
k=4	BoW	0.902	0.927	**k=9**	BoW	0.907	0.939	**k=14**	BoW	0.909	0.976
	WE	0.61	0.73		WE	0.64	0.76		WE	0.59	0.68
k=5	BoW	0.9	0.94	**k=10**	BoW	0.956	0.991	**k=15**	BoW	0.9	0.92
	WE	0.64	0.81		WE	0.68	0.96		WE	0.6	0.68

6 Validity analysis

In this paper we have only used a single industrial data-set that belongs to software, while other industrial software written in different languages and domains might reveal considerably different results. This was a design choice as we wanted to understand the dynamics of test execution and be able to use statistical methods alongside the machine learning algorithms. However, we are aware that the generalization of the results for different types of systems require further investigations using tests and churns from different systems. Another limitation comes from the randomness in selecting code churns and test cases without being ascertained about the nature of their failures. For example, there is a chance of encountering one or more tests that had failed due to non-functional related issues, for instance, a machinery failure at execution time. Likewise, the possibility of having tests that failed due to defects in the test script code and not the base source code exists. To minimize this threat, we collected data for multiple tests, thus minimizing the probability of identifying tests which are not representative.

7 Recommendations

This section provides our recommendations to practitioners who would like to use the MeBoTS method:

– We recommend practitioners to try a variation of classification models with different hyper-parameter tuning to assess models' effectiveness when using the MeBoTS method.
– Using a BoW based classifier in the MeBoTS method surpasses that used with WE, and therefore, we recommend the use of the simple BoW modelling technique for extracting features from code churns for training a classifier.

8 Conclusion and Future Work

In this study, we experimented with a set of industrial code churns and test execution results the effectiveness of using WE as an alternative feature extraction approach to BoW in the MeBoTS method. By conducting a total of 30 trials of training and validating two prediction models on the BoW and WE vector representations, we empirically compared the difference between the models' precision and recall scores. The results confirm with statistical significance (p-value less than 0.001) that modelling code churns with BoW results in higher prediction performance as compared with a WE-based model. In terms of future work, more empirical studies with larger industrial data are needed to validate the effectiveness of both techniques in the context of MeBoTS. Moreover, training a series of word embeddings using a different variation of parameters such as the vector and window sizes is needed to draw more conclusive results about the effectiveness of WE in this context.

References

1. Agrawal, H., Horgan, J.R., Krauser, E.W., London, S.A.: Incremental regression testing. In: 1993 Conference on Software Maintenance. pp. 348–357. IEEE (1993)

2. Al-Sabbagh, K., Staron, M., Hebig, R., Meding, W.: Predicting test case verdicts using textual analysis of commited code churns. EasyChair Preprint no. 1177 (EasyChair, 2019)
3. Chen, Y.W., Lin, C.J.: Combining svms with various feature selection strategies. In: Feature extraction, pp. 315–324. Springer (2006)
4. Chollet, F., et al.: Keras. https://github.com/fchollet/keras (2015)
5. Enríquez, F., Troyano, J.A., López-Solaz, T.: An approach to the use of word embeddings in an opinion classification task. Expert Systems with Applications **66**, 1–6 (2016)
6. Kho, J.: Why random forest is my favorite machine learning model. https://towardsdatascience.com/why-random-forest-is-my-favorite-machine-learning-model-b97651fa3706, accessed: 2019-06-28
7. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
8. Ochodek, M., Staron, M., Bargowski, D., Meding, W., Hebig, R.: Using machine learning to design a flexible loc counter. In: Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), IEEE Workshop on. pp. 14–20. IEEE (2017)
9. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
10. Řehůřek, R., Sojka, P.: Software Framework for Topic Modelling with Large Corpora. In: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks. pp. 45–50. ELRA, Valletta, Malta (May 2010), http://is.muni.cz/publication/884893/en
11. Rothermel, G., Harrold, M.J.: A safe, efficient algorithm for regression test selection. In: 1993 Conference on Software Maintenance. pp. 358–367. IEEE (1993)
12. Thie, S.: Gitpython documentation. https://gitpython.readthedocs.io/en/stable/, accessed: 2019-07-30
13. Vokolos, F.I., Frankl, P.G.: Pythia: a regression test selection tool based on textual differencing. In: Reliability, quality and safety of software-intensive systems, pp. 3–21. Springer (1997)