CHALMERS
UNIVERSITY OF TECHNOLOGY

Selective Regression Testing based on Big Data: Comparing Feature Extraction Techniques

Citation for the original published paper (version of record):
Al Sabbagh, K., Staron, M., Ochodek, M. et al (2020). Selective Regression Testing based on Big Data: Comparing Feature Extraction Techniques. IEEE Software: 322-329.
http://dx.doi.org/10.1109/ICSTW50294.2020.00058

(article starts on next page)

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)

Selective Regression Testing based on Big Data: Comparing Feature Extraction Techniques

1st Khaled Walid Al-Sabbagh
Computer Science and Engineering Department
University of Gothenburg
Gothenburg, Sweden
khaled.al-sabbagh@gu.se

2nd Miroslaw Staron
Computer Science and Engineering Department
University of Gothenburg
Gothenburg, Sweden
miroslaw.staron@gu.se

3rd Miroslaw Ochodek
Institute of Computing Science
Poznan University of Technology
Poznan, Poland
miroslaw.ochodek@cs.put.poznan.pl

4th Regina Hebig
Computer Science and Engineering Department
University of Gothenburg
Gothenburg, Sweden
regina.hebig@gu.se

5th Wilhelm Meding
Ericsson AB
Gothenburg, Sweden
wilhelm.meding@ericsson.se

***Abstract*—Regression testing is a necessary activity in continuous integration (CI) since it provides confidence that modified parts of the system are correct at each integration cycle. CI provides large volumes of data which can be used to support regression testing activities. By using machine learning, patterns about faulty changes in the modified program can be induced, allowing test orchestrators to make inferences about test cases that need to be executed at each CI cycle. However, one challenge in using learning models lies in finding a suitable way for characterizing source code changes and preserving important information. In this paper, we empirically evaluate the effect of three feature extraction algorithms on the performance of an existing ML-based selective regression testing technique. We designed and performed an experiment to empirically investigate the effect of Bag of Words (BoW), Word Embeddings (WE), and content-based feature extraction (CBF). We used stratified cross validation on the space of features generated by the three FE techniques and evaluated the performance of three machine learning models using the precision and recall metrics. The results from this experiment showed a significant difference between the models' precision and recall scores, suggesting that the BoW-fed model outperforms the other two models with respect to precision, whereas a CBF-fed model outperforms the rest with respect to recall.**

***Index Terms*—Regression Testing, Continuous Integration, Machine Learning, Feature Extraction**

I. Introduction

Continuous integration (CI) 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 an efficient way to perform regression testing, which is essential to ensure that the program under test has not been adversely affected by new changes. Such type of testing is central for achieving CI since it promotes for faster release of products and features.

Naturally, a safe but costly regression testing strategy is to run all tests available for reuse [22]. However, this approach demands an inordinate amount of hardware resources and long execution time, which impede the continuous delivery of features and products to the user community. To address this growing cost of regression testing, several test case selection (TCS) approaches have been proposed in the literature [18] [13] [1] [8]. While these approaches seek to reduce the size of a test suite, the majority of the selection techniques advocates for maximizing test coverage, which typically involves static analysis of the program code. In effect, trying to solve the regression testing problem through coverage-based algorithms leads to limited success, as emphasized by Antinyan and Staron [3]. This is because TCS should be determined by examining code changes that might influence test behavior rather than on the coverage criteria they fulfill. In addition, conventional selection techniques may not scale well to the growing complexity of software systems, since they involve static code analysis of the program, which is known for being costly and prone to produce many false warnings [20] [4] [6] [11]. On the other hand, over the recent years, there has been a growing interest among researchers and software companies in capitalizing on machine learning to automate testing activities [2] [9] [12]. Such approaches process available data and build models that embody patterns from which we can predict the effectiveness of a test case in revealing faults. A number of studies proposed combining large volume of defect data with source code analysis for building predictive models to identify defective software code. These defective source code fragments are in need for testing and we need to predict which test cases will trigger failures and thus lead to finding the defects. One of the challenges in applying these methods lies in selecting a suitable algorithm for extracting machine learning features of these code fragments, which allow to train models. The selection has an effect on the performance of the predictive model (precision and recall).

Therefore, in this paper, we study the effects of three different techniques for feature extraction — Bag-of-words

978-1-7281-1075-2/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSTW50294.2020.00058

(BoW), Word Embeddings (WE), and a new algorithm, called content-based feature extractor (CBF). 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. The third technique uses the source code content to build a features set of unique tokens i.e. each feature in the set corresponds to a unique token in the source code. 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, WE, and the CBF?*

In order to address this question, we design an experiment, where we use 15 different sets of code commits as experiment objects (one fragment for each experiment trial). We use the statistical performance measures of precision and recall as the dependent variables. The results of the experiment show that the prediction from the BoW-based classifier have a statistically significant higher precision than those reported by a WE-based and CBF-based classifiers, whereas the CBF-based classifier outperformed the other models with respect to recall. This means that the simpler method for making predictions is better in this context.

The paper is organized as follows: in section 2 we present an example that demonstrates the relationship between code modifications and test case failure; in section 3 we introduce background information; in section 4 we introduce related work on selective regression testing approaches, drawbacks of static code analysis, and feature extraction techniques in text classifications; in Section 5 we describe all the steps in the design of our experiment; in Section 6 we provide the results of our experiment; in Section 7 we introduce some threats to validity and limitations; in Section 8 we make our conclusions.

II. Example of the relationship between code change and test case failure

In this section, we introduce an example to illustrate the relationship between source code changes and test case failure. Figure 1 shows two revisions of an example program written in the C++ language. The modified revision consists of all code fragments in the original revision except for the framed lines of code. Here, the modified revision contains a new assignment to `pointers[2]`. In the context of C++ programming, any pointer set to zero is called a null pointer, and since there is no memory location zero, it is an invalid pointer assignment and will throw an error during run-time execution. An example test case is a CppUnit test `testTaskArrayDeclarations` that asserts whether all elements in the pointers' array equate to non-null values, as shown in Figure 1. By observing the test execution result of test case `testTaskArrayDeclarations` on both the original and modified revisions, we can induce that the new additions triggered test `testTaskArrayDeclarations` to fail. Thus, analyzing those code lines in the modified revision could help us build a model that derives coding style patterns that trigger test case failures.

III. Background

A. 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 a training sample of historical test execution results and the tested code churns. Our predictive method detects regressions by analyzing features derived from a collection of historical code churns and test execution results. This way, we can use previous test execution results and their relevant code changes for training a predictive model on classifying changes of new code lines into either defective or non-defective. The prediction of flawed lines in the new code is achieved by comparing the abstraction model of the derived feature-based set against similar or equivalent abstraction models used in the training. The method is comprised of 3 simple steps, as shown in Fig 2. This section briefly describes these steps.

a) 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 [19]. The output is then arranged in a table-like format and written in a csv file, named as 'Lines of Code'.

b) 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 the 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.

c) Training and Applying the Classifier Algorithm (Step 3): We exploit the set of extracted features provided by the

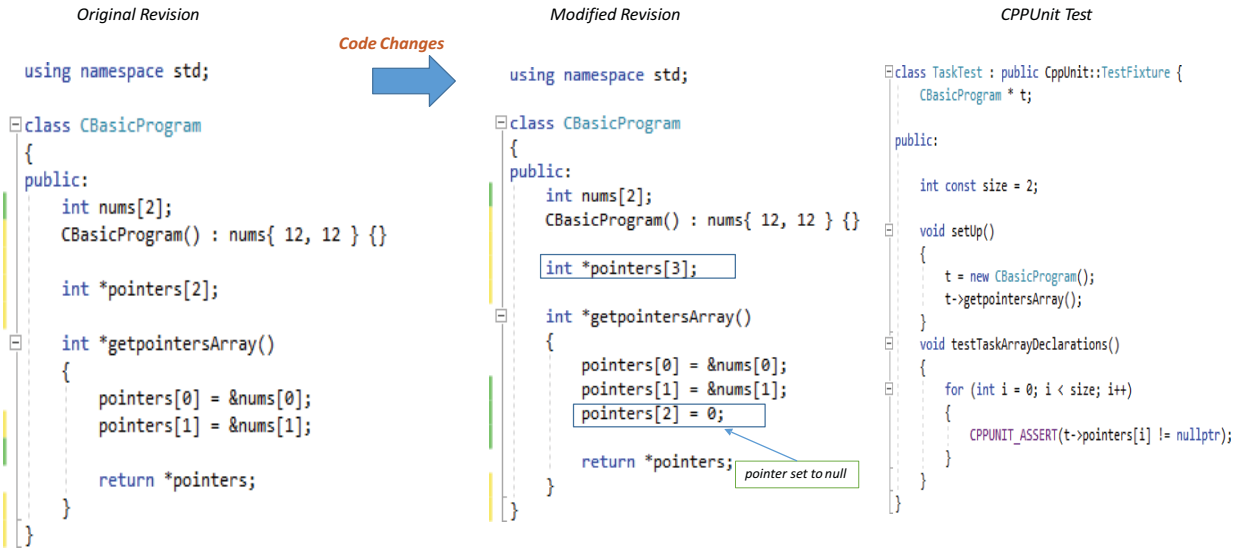

Fig. 1. Two Revisions of Program Under Test

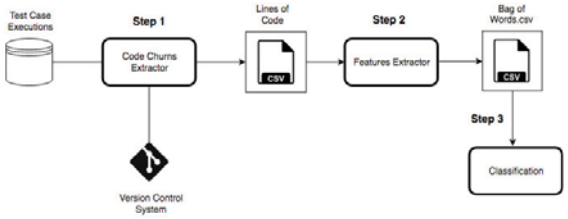

Fig. 2. The MeBoTS Method

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.

B. 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.

C. Content-based Features Extraction Algorithm

The current way of generating numerical vectors from textual input relies on statistical based approaches that either uses a predefined set of vocabulary, such as BoW, or a probability distribution function to capture similarities between keywords, such as Word Embeddings [21] [7]. However, the problem with such techniques is that they do not guarantee constructing distinct vectors for syntactically different code lines. In the context of MeBoTS, this distinction is necessary to maintain a set of consistent training examples for the classifier. At the core of all machine learning classification models, the purpose is to find a function that can separate training observations into their relevant classes and to converge into a model that can accurately classify new instances. Therefore, it is important to guarantee unique representations of similar observations to minimize the probability of classification errors in the predictive model.

Figure 3 presents an activity diagram that summarizes the procedure of the content-based feature extraction algorithm used in this study. The algorithm starts by initializing an empty list and reading one line at a time (steps 1 and 2) from the input source code. Each line is then analyzed (step 3) using an algorithm that creates a list of tokens based on the content of the input code. The tokens' list is created by splitting code lines on whitespace, comma, full stop, and bracket characters. In steps 4 and 5 the algorithm adds a new token from the list and if there is no new token, then it takes one more line and looks for new tokens there. In step 6, the algorithm uses the list of features to check for features in each of the lines analyzed so far. Then it checks whether there are feature vectors that are identical and whether the corresponding lines are identical (step 8). The exit condition in step 9 applies when there are no more lines to featurize in the input set (end of file) or when there are no more distinct tokens to be added. An example of a scenario where the second condition applies is when the algorithm reads lines that differ only in the number of whitespaces – e.g. "x = x + 1;" and "x = x+1" (spaces between x and 1)

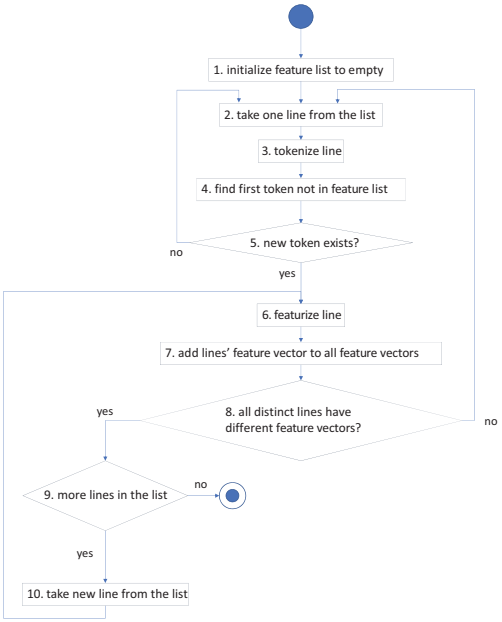

Fig. 3. Activity diagram presenting the algorithm for the content-based feature extraction.

IV. Related Work

Over the recent years, the problem of regression testing has received a lot of attention from the research community. Several studies on TCS were conducted, and a number of TCS approaches were proposed [18] [13] [1] [8]. At the crux, most of these approaches rely on white box static code analysis techniques to automate selective regression testing [22]. Since the ultimate goal of our study is to improve TCS, we start by presenting an overview of existing approaches in this area and explore their drawbacks. Then we review the literature on static code analysis and report results from studies that discern their usage.

A. Test Case Selection

Rothermel and Harrold [18] 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 (one for the original program and one for the modified program) to compare changes made between the two revisions – 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 (from the execution history list) that may have traversed the change in the modified version. A weakness in this approach, as identified by Yoo and Harmen [22], is the lack of data dependence, the technique will select tests that execute the modified statements but not the actual uses of variables, which leads to the inclusion of unnecessary tests for regression testing.

Lee and He [13] proposed an approach that uses Integer programming for finding a minimum subset of tests that maximize test coverage at a minimal cost. A test case matrix is maintained to map relevant test cases to one or more test requirements, such that when new changes in the codebase are introduced, a subset of relevant tests are selected. Then a coverge algorithm is used to select another subset that guarantees full test coverage at a minimal cost. One drawback of this approach lies in the cost overhead for maintaining dependency links between each test requirement and test case identifiers. Additionally, the test case matrix depends on the control-flow structure of the program, which means that the approach does not handle control-flow modifications in the program. So if the control-flow structure changes, the test case matrix should be updated by re-executing the whole set of test cases.

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.

An alternative to approaches based on source code analysis are methods based on data-flow analysis. Here, variable assignments are tested by selecting test cases that execute

sub-paths from the point of definitions to the point of use (definition-pair use). Gupta et al. [8] proposed a technique that uses a slicing algorithm to detect definition-pair use in the modified program. Unlike other approaches that require data flow history, the proposed method uses two slicing type algorithms to detect affected and changed definition-pair uses. After all pairs have been identified, test cases that exercises these pairs are selected for execution. One limitation of using data-flow based approaches is that they do not detect modifications unrelated to data-flow changes.

B. Usage of static code analysis

In the recent years, the usage of continuous integration led to the popularization of the use of automated static analysis. In CI flows, static analysis tools provide a quick measure of internal software quality. However, these tools are known for their inevitability, reporting large amounts of false positives, as pointed out in [20] [11] [6]. In this subsection, we highlight some of the work conducted in this area and report their evaluation results.

In an empirical study conducted by Wedyan et al. [20] to evaluate coding concerns, the authors used three SCA tools on two open source code projects to analyze the precision of these tools. The evaluation results showed that 96% of the coding concerns were false positive, whereas less than 3% of the detected faults corresponded to real coding issues.

Kim et al. [11] observed the prioritization capabilities of three SCA tools on three programs and measured their precision scores. The precision of the three tools in producing real warning prioritization were only 3%, 12%, and 8%.

Another study by Couto et al. [6] examined the relationship between warnings issued by a bug reporting tool and defects to analyze the gain of using SCA tools for early detection of faults. The results showed no static relationship between the two variables, although a moderate correlation between them was detected.

The overall conclusion drawn from these studies suggest that the use of static analysis tools as a standalone solution is insufficient, since they are expensive to implement and generate a lot of false positives.

V. Design of Experiment

A. 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. It consists of several hundred software developers, organized in several agile teams, spread over a number of continents. Given that they have been early adopters of lean and agile software development methodologies, they have become mature in these areas of work.

B. 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.

C. 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 a validation file of 9200k LOC and a training file of approximately 140k LOC. 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.

D. Features Extraction with BoW

We used an open source tool, ccflex, to carry out the BoW vectors transformation across the experimental subjects. For each subset (fold), we ran the BoW vector transformation with ccflex and saved the resulting vectors locally, in files, so that we can supply them as inputs to the ML classifier. ccflex relies on a threshold-based criterion for extracting features from textual input, such that tokens whose frequency counts exceed a lower threshold value are selected as candidate features. In this experiment, we kept the frequency threshold to its default value (25%) and set the BoW n-gram to 2 to generate features of two adjacent tokens that are originally separated by whitespaces. The resulting space of vectors generated by ccflex at each round of transformation consisted of a total of 2249 feature vectors.

E. 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. [14]. We used the implementation available in the Gensim library [17]. 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 4 illustrates an example of how the tokens in the original data-set (before partitioning) are placed. [1] The

[1]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

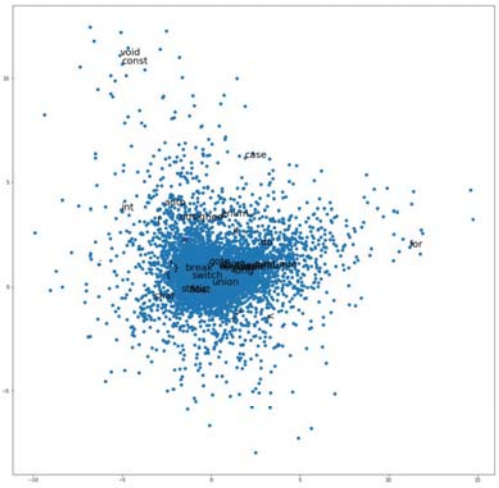

Fig. 4. Semantic orientation of similar tokens in the analyzed software

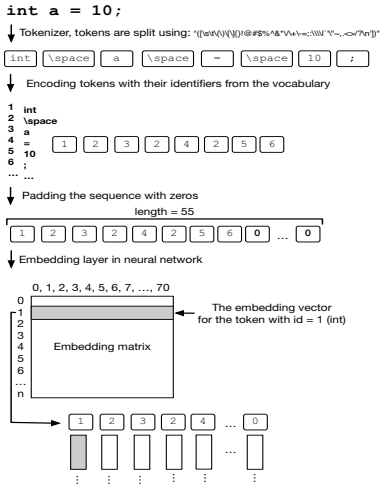

Fig. 5. Preprocessing the lines of code to be used as input for the neural network.

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 5. 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.

F. Features Extraction with the new featurizer

Applying the new features extraction algorithm on each of the experimental subjects (fold) resulted in a sparse high dimensional feature vectors of size 13,700, where each vector corresponds to a distinct token in the input source code. To reduce the high computational cost and error of parameter estimation, we used a dimensionality reduction technique (PCA) for projecting the entire dataset (13700 dimensions) onto a new subspace that is equivalent in size to that produced by the BoW transformation (2249 dimensions). By definition, PCA is a statistical technique that aims at finding patterns in data of high dimension and highlighting important information by expressing similarities and differences in the data [16]. However, since the technique uses linear projection approaches, we designed the experiment with the assumption that our experimental subjects are linearly separable.

G. Evaluation with Random Forest and Neural Network

The vector files representing the 15 subsets of source code and their corresponding classes were categorized into three different groups, one for each FE algorithm with 30 vector files in each group (15 for training and 15 for validation). We compared, on 45 subsets, the effectiveness of the three FE algorithms with classical state-of-the-art measures (precision and recall) using three predictive models: 2 RF and 1 CNN models. Since RF is known for performing well with high dimensional vectors [10], we decided to evaluate the effect of both BoW and CBF on the performance of a random forest model. To implement the models, we used the Random Forest utility available in the scikit-learn library [15]. For the WE group, we trained and validated a CNN model on each vectors file using the the implementation of CNN in the Keras library [5]. 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 and both the BoW and CBF feature extraction, on the other hand, provided results that were poor to consider in the paper (BoW and CBF do not provide the feature set that is rich enough for CNN). Therefore, we selected a CNN model to evaluate the effect of WE, rather than forcing the algorithms to work with the feature extraction technique that is not suitable for them.

The architecture of the CNN is presented in Figure 6. 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 5 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 II.

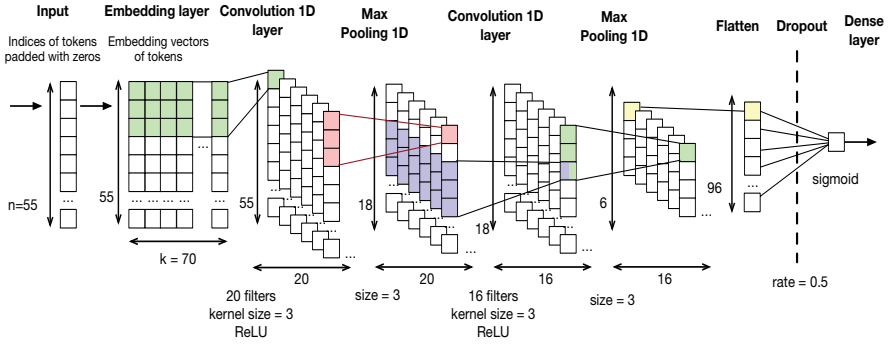

Fig. 6. The Architecture of Convolutional Neural Network.

TABLE I
DESCRIPTIVE STATISTICS FOR THE PRECISION AND RECALL SCORES

Featurizers	BoW	WE	CBF
Mean Precision	0.9	0.64	0.76
Mean Recall	0.93	0.78	0.94
Median Precision	0.91	0.64	0.77
Median Recall	0.94	0.76	0.95

VI. 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 the three models (the two RF and the CNN) for the 15 folds in each group (BoW, WE, CBF) and examined the distribution of the points. We decided to run a Shapiro-Wilk test to further validate the ocular inspection for normality. The test results were statistically significant for the performance measures of the RF model 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 the results of the BoW fed model can be rejected. Conversely, the normality test for the precision and recall samples for the WE based model suggests a normal distribution for both measures (Precision with WE: Test statistic = 0.929, p-value = 0.262, Recall with WE: Test statistic = 0.893, p-value = 0.075). Finally, the normality test results for the precision and recall samples of the CBF based model suggest that the samples are not normal (Precision with CBF: Test statistic= 0.881, p-value = 0.048, Recall with CBF: Test statistic = 0.870, p-value = 0.034). Since the statistical results of the Shapiro-Wilk test suggest that we have issues with normality in the precision and recall samples, we decided to run a non-parametric test for comparing the difference between the precision and recall scores across the three models. The Kruskal–Wallis H test was selected as an appropriate method since it can handle skewed data in more than two samples without normality presumption.

The results from the Kruskal–Wallis H test showed a significant difference between the precision and recall scores of the three models. The comparison results 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 both the precision and recall scores. Table I summarizes the mean and median scores of the 6 performance metrics (precision with BoW, precision with WE, precision with CBF, recall with BoW, recall with WE, and recall with CBF). The results suggest that the BoW-fed RF model outperformed the other two models with respect to precision (mean=0.9 and median=0.91), whereas the the CBF-fed RF model reported the highest recall score (mean= 0.94 and median= 0.95).

While recall indicates the model's ability to find all relevant tests that require no execution, precision expresses the proportion of tests that were predicted as passing and had actually passed. As with most concepts in data science, there is a trade-off in terms of which of the two metrics to maximize (precision and recall). In the context of TCS, the decision point should be based on minimizing the risk of missing tests that will actually fail. Therefore, we need to weigh more importance to the model's precision since we can accept some false alarms in the prediction of failed tests but not missing tests that could potentially fail. Based on this, our empirical evaluation suggests that the BoW based model is better suited for solving the TCS problem with 0.9 precision. This means that the model could correctly identify 9 out of 10 test execution results (recall= 0.93) with a precision of 9 out of 10 cases.

VII. 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 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 certainty in the nature of their failures. For example, chances that one or more tests had failed due to non-functional related issues can not be rolled out, for instance, a machinery failure

TABLE II
THE PRECISION AND RECALL SCORES FOR THE RF AND CNN MODELS USING THE THREE FEATURIZERS

Fold	CBF	Precision	Recall
k=1	BoW	0.6	0.61
	WE	0.6	0.706
	CBF	0.64	0.82
k=2	BoW	0.91	0.96
	WE	0.68	0.885
	CBF	0.76	0.95
k=3	BoW	0.908	0.919
	WE	0.64	0.756
	CBF	0.77	0.96
k=4	BoW	0.902	0.927
	WE	0.61	0.73
	CBF	0.77	0.93
k=5	BoW	0.9	0.94
	WE	0.64	0.81
	CBF	0.76	0.96

Fold	CBF	Precision	Recall
k=6	BoW	0.9	0.92
	WE	0.63	0.75
	CBF	0.78	0.92
k=7	BoW	0.949	0.952
	WE	0.63	0.71
	CBF	0.76	0.922
k=8	BoW	0.937	0.959
	WE	0.67	0.82
	CBF	0.77	0.96
k=9	BoW	0.907	0.939
	WE	0.64	0.76
	CBF	0.75	0.94
k=10	BoW	0.956	0.991
	WE	0.68	0.96
	CBF	0.80	0.93

Fold	CBF	Precision	Recall
k=11	BoW	0.944	0.994
	WE	0.69	0.95
	CBF	0.75	0.95
k=12	BoW	0.928	0.959
	WE	0.65	0.81
	CBF	0.79	0.91
k=13	BoW	0.9	0.94
	WE	0.6	0.74
	CBF	0.76	0.93
k=14	BoW	0.909	0.976
	WE	0.59	0.68
	CBF	0.77	0.96
k=15	BoW	0.9	0.92
	WE	0.6	0.68
	CBF	0.79	0.95

during 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.

VIII. CONCLUSION AND FUTURE WORK

In this study, we experimented with a set of industrial source code and test execution results the effectiveness of three feature extraction algorithms on the predictive performance of three ML models in predicting test case verdicts. As per the basis of the empirical evaluation, using the BoW technique is better suited for solving the TCS problem than WE and CBF, allowing test orchestrators to predict test execution results (pass or fail) with 0.9 precision rate. Those results can be used to reduce the size of test suite at each CI build by excluding tests that are predicted to pass.

In terms of future work, more empirical studies with larger industrial data are needed to validate the effectiveness of the three algorithms in the context of TCS. Additionally, training a series of word embeddings using different variation of parameters such as the vector and window sizes is needed to draw more conclusive results about the usage 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. Tech. rep., EasyChair (2019)

[3] Antinyan, V., Derehag, J., Sandberg, A., Staron, M.: Mythical unit test coverage. IEEE Software **35**(3), 73–79 (2018)

[4] Bolduc, C.: Lessons learned: Using a static analysis tool within a continuous integration system. In: 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). pp. 37–40. IEEE (2016)

[5] Chollet, F., et al.: Keras. https://github.com/fchollet/keras (2015)

[6] Couto, C., Montandon, J.E., Silva, C., Valente, M.T.: Static correspondence and correlation between field defects and warnings reported by a bug finding tool. Software Quality Journal **21**(2), 241–257 (2013)

[7] 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)

[8] Gupta, R., Harrold, M.J., Soffa, M.L.: An approach to regression testing using slicing. In: Proceedings Conference on Software Maintenance 1992. pp. 299–308. IEEE (1992)

[9] Hata, H., Mizuno, O., Kikuno, T.: Fault-prone module detection using large-scale text features based on spam filtering. Empirical Software Engineering **15**(2), 147–165 (2010)

[10] Kho, J.: Why random forest is my favorite machine learning model. https://towardsdatascience.com/why-random-forest-is-my-favo accessed: 2019-06-28

[11] Kim, S., Ernst, M.D.: Which warnings should i fix first? In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 45–54. ACM (2007)

[12] Kim, S., Whitehead Jr, E.J., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering **34**(2), 181–196 (2008)

[13] Lee, J.A., He, X.: A methodology for test selection. Journal of Systems and Software **13**(3), 177–185 (1990)

[14] Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)

[15] 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)

[16] Raschka, S., Linear, P., Gaussian, R., LLE, L.L.E.: Kernel tricks and nonlinear dimensionality reduction via rbf kernel pca. Blog, September (2014)

[17] 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

[18] Rothermel, G., Harrold, M.J.: A safe, efficient algorithm for regression test selection. In: 1993 Conference on Software Maintenance. pp. 358–367. IEEE (1993)

[19] Thie, S.: Gitpython documentation. https://gitpython.readthedocs.io/en/stable/, accessed: 2019-07-30

[20] Wedyan, F., Alrmuny, D., Bieman, J.M.: The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In: 2009 International Conference on Software Testing Verification and Validation. pp. 141–150. IEEE (2009)

[21] Wu, L., Hoi, S.C., Yu, N.: Semantics-preserving bag-of-words models and applications. IEEE Transactions on Image Processing **19**(7), 1908–1920 (2010)

[22] Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability **22**(2), 67–120 (2012)