Which source files are most likely to require change?

Ashiqur Rahman
Queen's University

***Abstract*—Change in software is almost inevitable due to various issues such as error, bug, security flaw etc. Prediction of change prone files can help the developers to be more focused on the files that may require change. Besides, it can help software project managers to make better decisions regarding software development and maintenance.**
Selection of language features, and their combinations can have a significant effect on the quality of the developed code. Poorly selected features can degrade the code quality, and accelerate the need of change. In this project, we explore whether a file's language feature use can be used together with deep neural network to predict the file's change proneness.
We encode feature usage profile of source files using feature appearance and density vectors, and use these profiles to train deep neural network (DNN) for predicting the source files may require change. Besides, we find which vector, among feature appearance vector, density vector, and their combination, is the most useful vector, and whether feature data needs to be normalized.
Experiments are performed using two open source php software phpbb and WordPress. The results are evaluated against four other machine learning classifiers K-NN Classifier, Support Vector Classifier, Decision Tree Classifier, and Gaussian Naive Bayes Classifier. Results show the DNN model trained with feature use profile can successfully predict change proneness of source files. The ROC-AUC of the developed DNN is 89.53% . It achieves F-measure 93.27% and accuracy 88.26%. No other model was successful to achieve that performance. The findings of this project clearly shows language feature can be used for training deep neural network and predicting issues related to source code.

I. Introduction

Source files of a software may need to be changed to resolve various issues. An issue can be anything that indicates potential problems in existing code, such as a bug, inefficiency, or vulnerability. Prediction of change prone source files can help project managers to decide which files may pose a risk or require maintenance, and developers to understand where more focus should be given to do possible improvements.

The aim of this project is to explore the possibility of using language features together with deep learning to learn and predict which source files in a software base may change in future due to some issues.

Software code is written using the features provided by a programming language. Selection of language features is dependent on coding style, solution patterns, individual preference, and the problem that needs to be solved. The selected language features and combinations can have a significant effect on the quality of the developed code. Poorly selected features have the possibility of lowering code quality, creating potential issues in the code. Therefore, the language feature usage attributes of source code can be explored as data from which to use machine learning techniques to learn and reveal the potentiality of change of a source file due to some issues. Because source code is the main software artifact, we are motivated to use it directly to generate the code profile of files. The code profile of a file consists of feature appearance and density vectors extracted from its code. We are not interested in any traditional project or product metrics used to make such predictions [1] [2] [3] [4] [5] [6].

Our hypothesis is, files with certain language feature profiles are more likely to change frequently, and we want the ML system to learn to recognize those profiles.

So, our research questions are:

RQ1. Can language feature use be used to learn and predict the source files that will require change in future?

RQ2. If RQ1 is positive, can we ignore either feature appearance or density vector when developing the code profile?

RQ3. If RQ1 is positive, does normalization of feature data have any effect on the results?

In this project, we develop code profile of source files using feature appearance and density vectors. Then we train deep neural network (DNN) model using those code profiles. To the best of our knowledge, no study exists that explores language features and deep neural network for predicting change prone files in a code base.

Experiments are performed using two php projects phpbb [7] and WordPress [8]. AUC, F-measure, and accuracy of our model are 89.53%, 93.27% and 88.26% respectively. No other machine learning model shows that high performance. From the results it is clear that language feature use of source files can be used to learn and predict the source files that will require change in future. We also find feature appearance and density vectors together gives the best performance, and normalization has high positive effect on the performance.

In future, this project can be extended to predict the number of releases within which a file may change. Besides, the insights obtained can be used to predict bug, error, malware, software vulnerability etc. in source code.

Kagdi and Maletic [9] categorizes software change prediction approaches into Impact Analysis (IA), and Mining Software Repositories (MSR). While IA analyzes only the current version of a software, MSR analyzes multiple past versions to predict changes in a software. Based on this classification, our project falls in the MSR category.

Section II presents related work. It is followed by our approach description in Section III. Section IV presents results.

A discussion of the results is given in Section V. Finally, Section VI concludes the paper.

II. Releated Work

Any research that predicts some particular issue such as bug, error, security flaw etc. in source code can also be adopted to predict which source files need to change. But their scope of change proneness prediction is limited by the particular issue they predict and very narrow. Our approach aims to predict change proneness in general irrespective of the issue. For that reason, our focus is only those research that aims change proneness prediction, not any particular issue. Here, we present few papers which are very closely related to our work.

Catolino et al. [1] investigate the effect of developer-related metrics on predicting change proneness. They select three existing fault prediction models Basic Code Change Model (BCCM) [10], Developer Changes Based Model (DCBM) [11], and Developer Model (DM) [12], and adopt them for change prediction. All these three models capture the complexity of the development process. The authors find developer-based models perform better than existing state-of-the-art models [13] [14] based on evolution and code metrics. Among the three developer-based models the DCBM model performs the best with F-measure 66%, and accuracy 78%. However, after analyzing complementarity between developer metrics-based models, and evolution and code metrics-based models, the authors conclude that combination of the metrics may help to achieve improved performance.

Malhotra et al. [2] predict change proneness of classes using change history of the classes, and object oriented metrics. Change history is calculated using evolutionary metrics. Four different logistic regression models are developed using only evolutionary metrics, using only Object Oriented metrics, using both evolutionary and Object Oriented metrics, and using internal class probability. Among these four the model developed using evolutionary and Object-Oriented metrics gives the best performance.

Kumar et al. [3] study sixty-two source code metrics of four types size, cohesion, coupling, and inheritance for predicting change proneness of Object-Oriented software. They employ four feature selection techniques, and develop eight different machine learning models and two ensemble-based models. Their machine learning models include Logistic Regression, Naive Bayes Classifier, Extreme Learning Machine with linear, polynomial and RBF kernels, Support Vector Machine with linear, RBF and Sigmoid kernel. Two ensemble techniques include Best-in-Training, and Majority Voting. The authors find only a small subset of source code metrics really help in predicting change proneness, and coupling metrics are the most predictive metrics for predicting change proneness. Among the models, Extreme Learning with polynomial kernel function gives the best performance.

Sharafat and Tahvildari [4] propose a probabilistic approach to predict change proneness of classes of Object-Oriented systems. Their approach uses dependencies from UML diagram, code metrics together with change log, and expected time of next release to predict whether a class will change in the next release. They evaluate their approach experimenting on open source project JFlex [15].

Askari and Holt [5] mine software repositories, and analyze the obtained information to develop three probabilistic models in order to predict the files that will have changes or bugs. The models are Maximum Likelihood Estimation (MLE), Reflexive Exponential Decay (RED), and RED Co-Changes (REDCC). Experiments are performed on six open source projects OpenBSD, FreeBSD, KDE, Koffice, NetBSD and Postgres. The authors analyze the bug and change messages in the CVS log of the projects to develop the prediction models.

Amoui et al. [6] propose a Neural Network-based Temporal Change Prediction framework to predict which entities of a software may change and when. This framework uses software change history from software repository to measure applicable development metrics. It has three main processes locating hot spots, generating training data, and building prediction model. Part of this network, locating hot spots is closely related to our target. This process finds the change-prone software entities using Maximum Likelihood Estimation (MLE) model and development metrics. However, product metrics can also be used.

Though several research can be found in the literature that attempts software change prediction, they use project and product metrics. Our project is different from them in a number of ways. The main differences are the use of language features, and deep neural network to predict the files that may require change.

III. Approach

Our change prediction study aims to predict change proneness of source files training a deep neural network using only language feature vectors as predictor variables.

Figure 1 shows the overall approach we followed. In the first step, we collected releases of the source projects from GitHub. Then a comparator was used to get the file change information from the releases, and a language feature extractor for extracting the feature vectors of the files. The class (label) of each file was generated from the obtained file change information, and they were merged with their respective feature values. Finally, the generated data was fed to a deep learning network model developed using the Tensorflow library available for Python.

A. Language feature vectors

We use two feature vectors language feature appearance vector, and density vector, both individually and together, to encode a files language feature usage characteristics. We call this the code profile of a file.

Language feature appearance vector tells which features are used in a code. On the other hand, language feature density vector tells the usage frequency of each feature.

Figure 2 presents a sample php code. Suppose, we are interested in the following six features: if_cond, for_loop, while_loop, eval(), echo, date(). Then feature appearance and density vectors are as given below.

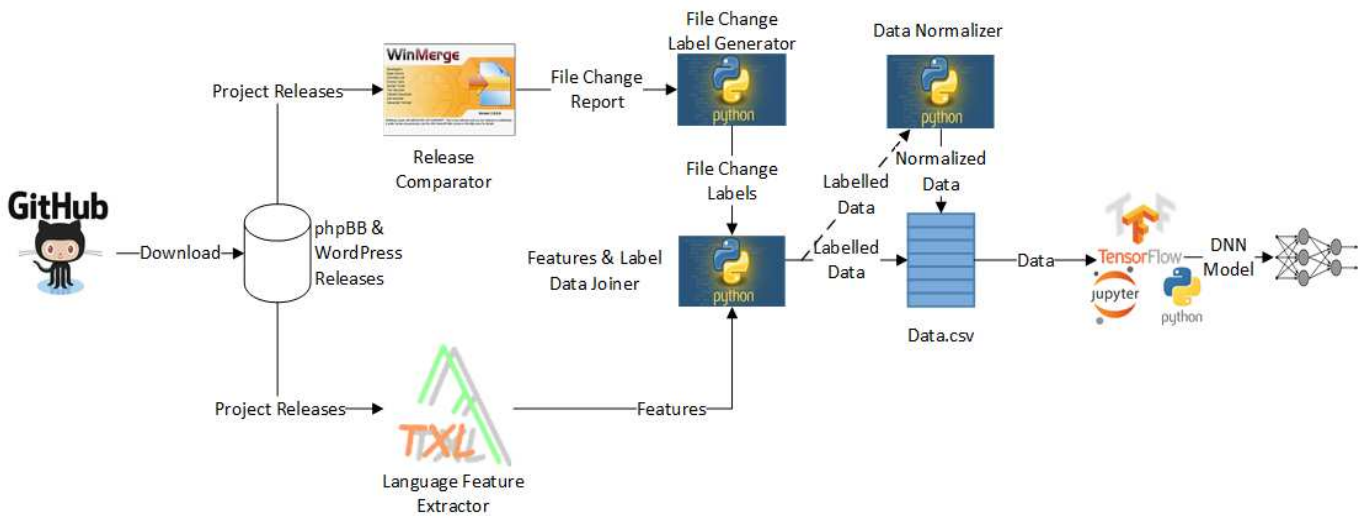


Fig. 1: Overall Approach

```
<?php
$t = date("H");
if ($t < "20") {
    echo "Have a good day!";
    }
for ($x = 0; $x <= 10; $x++) {
    echo "The number is: $x <br>";
    }
for ($y = 10; $y > 0; $y--) {
    echo "The number is: $y <br>";
    }
?>
```

Fig. 2: php sample code

Appearance vector: [if_cond, for_loop, while_loop, eval(), echo, date()] = [1,1,0,0,1,1]

Density vector: [if_cond, for_loop, while_loop, eval(), echo, date()] = [1,2,0,0,3,1]

Patrick [16] in his research, categorizes php language features into six categories as given in Table I. He extracted total 1,021 php features. Our aim is to build the feature vectors using all those features that are listed by Patrick.

B. Data Collection

Data was collected from two open source php projects in GitHub *phpbb*, and *WordPress*.

When collecting the releases only the main releases that comes linearly in increasing order were considered. For example, if the releases were in the following order 1.1.5, 1.2.0, 1.1.6, 1.2.1, 1.1.7 then among those releases only 1.1.5, 1.2.0, 1.2.1 were taken.

phpbb [7] is an open source online bulletin board. At the time of this project it had total 142 releases. We collected total 49 main releases that comes linearly in increasing order.

WordPress [8] is an open source content management system developed in php. It had total 307 releases at the time of this project. Among those 115 main releases were taken for this project.

We selected these two projects because they have a good number of releases, and their source code are publicly available.

Table II gives an overview of the collected data. Both .php and .inc extensions were considered in this experiment. In total there were more than 72,000 php files. Among them the parser failed to parse 259 files. In the end, total 14,700 file change records were obtained from the releases of the two projects.

Five datasets were prepared from the collected data with the aim of understanding the importance of the feature vectors, and the effect of normalization to predict change proneness. Each dataset contains the same number of samples and the same file change labels, but their feature vectors differ. Below is a description of each dataset.

- **Dataset#1:** This dataset contains only feature appearance vectors of the files.
- **Dataset#2:** It contains only feature density vectors of the files.
- **Dataset#3:** This dataset contains normalized density vectors. Each vector was normalized by dividing the value of each feature by the value of the vector.
- **Dataset#4:** This one is basically a join of Dataset#1 and Dataset#2, and contains both feature appearance and density vectors.
- **Dataset#5:** This one is similar to Dataset#4, but the density vector is normalized.

C. Model Development

1) Determining the number of hidden layers and nodes: In order to determine the number of hidden layers, and nodes per layer four types of tests were performed. During each test the

TABLE I: An Overview of the features in PHP [16]

Feature Category	Features
Expression	parenthesized expression, anonymous function creation expression, clone expression, object creation expression, anonymous class object creation expression, array creation expression, subscript expression, function call expression, member reference expression, postfix increment expression, postfix decrement expression, scoped reference expression, exponentiation expression, prefix increment expression, prefix decrement expression, unary op expression, error control expression, shell command expression, cast expression, instanceof expression, multiplicative expression, additive expression, concatenation expression, shift expression, relational expression, spaceship expression, equality expression, identity expression, bitwise expression, conditional expression, coalesce expression, simple assignment, byref assignment, exponentiation assignment, multiplicative assignment, additive assignment, concatenate assignment, shift assignment, bitwise assignment, logical expression, yield expression, include expression, include once expression, require expression, require once expression
Statement	compound statement, named label statement, expression statement, if statement, switch statement, case statement, default statement, while statement, do statement, for statement, foreach statement, goto statement, continue statement, break statement, return statement, throw statement, try statement, declare statement
Declaration	function static declaration, function static initializer, global declaration, function definition, function prototype, parameter declaration, type declaration, class declaration, const declaration, class const declaration, property declaration, method declaration, constructor declaration, destructor declaration, interface declaration, trait declaration, namespace definition, namespace use declaration
Intrinsic Function	echo intrinsic, empty intrinsic, eval intrinsic, exit intrinsic, die intrinsic, isset intrinsic, list intrinsic, print intrinsic, unset intrinsic
Core Function	arrays function calls, classes or objects function calls, date or time function calls, directories function calls, error handling function calls, program execution function calls, filesystem function calls, filter function calls, function handling function calls, hash function calls, php options or info function calls, mail function calls, math function calls, misc functions function calls, network function calls, output control function calls, password hashing function calls, phar function calls, reflection function calls, posix regex function calls, sessions function calls, spl function calls, streams function calls, strings function calls, tokenizer function calls, urls function calls, variable handling function calls
Others	heredoc or nowdoc string literal, inline html, unkeyed list, keyed list, anonymous function use clause, array initializer, variadic unpacking, compound member, elseif clause, else clause, catch clause, finally clause, variadic parameter, return type, nullable, default argument specifier, class extends clause, class interface clause, abstract modifier, final modifier, var modifier, public modifier, protected modifier, private modifier, static modifier, property initializer, interface extends clause, trait use clause, trait select insteadof clause, trait alias as clause, namespace aliasing clause

TABLE II: Data overview

sl.	**Project Name**	**#Release**	**#Files**	**#.php & .inc**	**Failed to parse**	**Avg #.php.inc files/release**	**Total #changed records**
1.	WordPress	115	111,848	43,388	199	421	9,196
2.	phpBB	49	60,295	23,980	60	490	5,531
	Total	164	172,143	**72,368**	259	442	**14,700**

number of hidden layers, and the number of nodes per layer were changed keeping the sample data, and all other settings the same. Then from the hidden layer settings we picked the one that had the best AUC. Table III shows the AUC of a deep neural network model under different hidden layer settings. Below is a description of the tests that were performed using Dataset#4 that contains both feature appearance and density vectors.

- **Type#1 Tests:** This test type contains five tests T1-T5 as shown in Table III. During these tests the number of nodes in the first hidden layer was set to one-third of the sum of #input_feature and #output_class. In case of fraction we took the ceiling value. All other layers had one-third of the total number of nodes in its previous layer. .
- **Type#2 Tests:** The number of nodes in the first hidden layer was half of the sum of #input_feature and #output_class, and all other layers had half of the total number of nodes in its previous layer. Eight tests from T6-T13 in Table III fall in this group.
- **Type#3 Tests:** The number of nodes in the first hidden layer was equal to the sum of #input_feature and #output_class, and all other layers had one-third of the total number of nodes in its previous layer. T14-T15 fall in Type#3 category.
- **Type#4 Tests:** Like Type#3 the number of nodes in the first hidden layer was set to total #input_feature and #output_class, but all other layers had half of the total number of nodes in its previous layer. T20-T28 fall in this group.

Comparing the AUC of Table III, we find the hidden layer setting [1022, 511, 255, 128] of test T8 is the best.

2) Algorithm: In addition to deep neural network classifier, four more machine learning models were developed to evaluate the performance of the change prediction model. Figure 3

TABLE III: Effect of hidden layers on the performance of a deep neural model

	Test No.	Hidden layer settings	AUC (%)
Type#1 Tests	T1	[682, 228]	83.61
	T2	[682, 228, 76]	84.01
	T3	[682, 228, 76, 26]	82.56
	T4	[682, 228, 76, 26, 9]	47.66
	T5	[682, 228, 76, 26, 9, 3]	75.50
Type#2 Tests	T6	[1022, 511]	82.39
	T7	[1022, 511, 255]	84.56
	T8	**[1022, 511, 255, 128]**	**85.90**
	T9	[1022, 511, 255, 128, 64]	85.54
	T10	[1022, 511, 255, 128, 64, 32]	84.83
	T11	[1022, 511, 255, 128, 64, 32, 16]	83.92
	T12	[1022, 511, 255, 128, 64, 32, 16, 8]	83.31
	T13	[1022, 511, 255, 128, 64, 32, 16, 8, 4]	50.00
Type#3 Tests	T14	[2044, 682]	84.61
	T15	[2044, 682, 228]	83.65
	T16	[2044, 682, 228, 76]	83.00
	T17	[2044, 682, 228, 76, 26]	82.89
	T18	[2044, 682, 228, 76, 26, 9]	50.00
	T19	[2044, 682, 228, 76, 26, 9, 3]	50.00
Type#4 Tests	T20	[2044, 1021]	68.92
	T21	[2044, 1021, 511]	83.25
	T22	[2044, 1021, 511, 255]	82.99
	T23	[2044, 1021, 511, 255, 128]	84.27
	T24	[2044, 1021, 511, 255, 128, 64]	74.08
	T25	[2044, 1021, 511, 255, 128, 64, 32]	82.78
	T26	[2044, 1021, 511, 255, 128, 64, 32, 16]	83.12
	T27	[2044, 1021, 511, 255, 128, 64, 32, 16, 8]	50.00
	T28	[2044, 1021, 511, 255, 128, 64, 32, 16, 8, 4]	50.00

shows the general steps followed to develop each model.

3) Architecture of the Developed Deep Neural Network Classifier: A four-hidden-layers perceptron model was developed to predict the files that may require change as shown in Figure 4. It's first, second, third and fourth hidden layers contain 1022, 511, 255, and 128 perceptrons respectively.

4) Model Parameters: Only the deep neural network classifier was developed using Tensorflow [17]. All other models were developed using the functionalities provided by scikit [18]. Table IV shows the model parameters we used.

D. Metrics

Five standard metrics are used in this report to compare the performance of the classifiers.

- ROC-AUC or simply AUC
- F-measure
- Accuracy
- Precision
- Recall

E. Tools

Below is a brief description of the main tools that were used in this experiment.

- **Winmerge:** It's a free tool for data comparison, and can be used to determine what has changed between versions of source code. We used Winmerge [19] version 2.14.0.0.
- **Feature extractor:** A php language feature extractor developed by Patrick [16] was used for extracting feature usage of source files. It is written in TXL [20].
- **TensorFlow library:** The TensorFlow-1.3.0 [17] library for Python was used for developing the deep neural network classifier.
- **scikit:** It is another machine learning library for python. scikit [18] was used for developing the baseline machine learning models in order to compare the performance of the developed DNN model.
- **Python:** Anaconda [21] version 3 distribution was used.
- **Editor:** Jupyter [22] notebook provides a web based editor interface. All coding was done in Jupyter notebook.

IV. RESULTS

In this section, we present the performance of our DNN model, and also present the evaluation results against four baseline models.

A. Performance of the DNN model in normalized confusion matrix

For each of the dataset, Figure 5 shows normalized confusion matrices to illustrate the ratio of correctly classified and misclassified samples of each class by the DNN model . For example, Figure 5(e) shows the normalized confusion matrix of Dataset#5. As illustrated, among the positive samples , the model successfully predicted the labels of 97% samples, and misclassified 3% of them. On the other hand, among the negative samples, it correctly predicted the labels of 43% samples, and misclassified 57%.

From the normalized confusion matrices we find that the model was most successful in classifying positive samples using Dataset#2, which contains density vectors without normalization. On the other hand, Using Dataset#1 the model was most successful in predicting the labels of negative samples. Dataset#1 contains only feature appearance vector.

B. Error rates of k-fold cross validation

Using each dataset, 10-fold cross validation was performed to train the DNN model, and the best model was picked to do the final test. Figure 6 presents k-fold training error rates, and the test error rate of the model for each dataset. In each subfigure of Figure 6, the last column shows the final test error rate of the model.

As can be seen from Figure 6, for each dataset the final test error rate was nearly average of all other error rates. Standard deviation of error of k-fold cross validation using Dataset#1 - Dataset#5 are 0.007, 0.01 ,0.007 ,0.006 , and 0.006 respectively.

C. Performance metrics of the DNN

From Table V - Table IX, we find that the performance of the DNN model varies between 79.03%-89.53% in terms of AUC.

The model using Dataset#5 has the highest AUC, F-measure and accuracy respectively 89.53%, 88.26% and 88.26%.

```
1. Split the dataset, and keep 80% for training and 20% for testing
2. Make 10-folds of the training data
    I. for each fold:
        a. Train a classification model using the remaining folds
        b. Evaluate the model using the evaluation fold
        c. Save the best performing model
3. Use the test data, and evaluate the performance of the best performing model
```

Fig. 3: The algorithm followed to develop each model

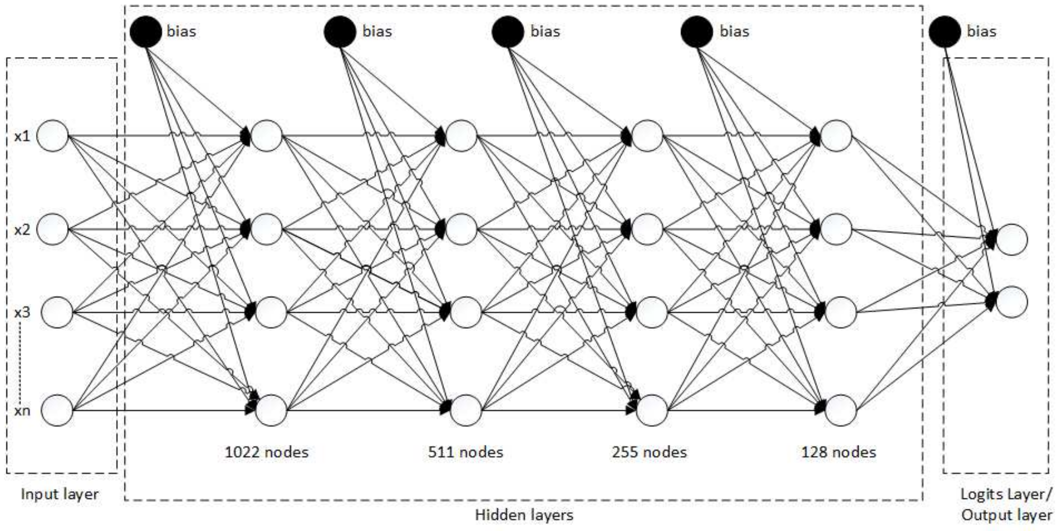


Fig. 4: DNN network architecture

TABLE IV: Model Parameters

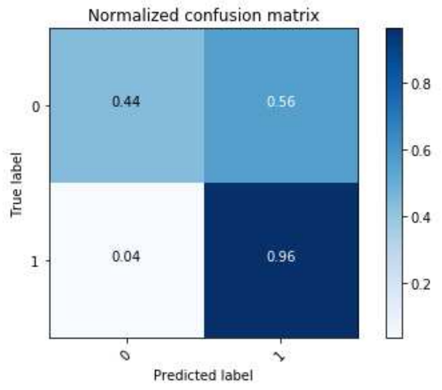
DNN	**K-NN**	**SVC**	**DT**	**Gaussian Naive Bayes**
[1022, 511, 255, 128], Activation function: relu, Optimizer: Adagrad, Epochs: 3,000, Steps: 3,000,	n_neighbors were changed between: 1, 3, 5, 7, 11, algorithm='auto', leaf_size=30, metric='minkowski', metric_params=None, n_jobs=1, n_neighbors=1, p=2, weights='uniform'	C=1.0, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf', max_iter=-1, probability=True, random_state=None, shrinking=True, tol=0.001, verbose=False	class_weight=None, criterion='gini', max_depth=None, max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, presort=False, random_state=None, splitter='best'	priors=None

TABLE V: Performance of the models based on dataset#1 (Feature appearance vector only)

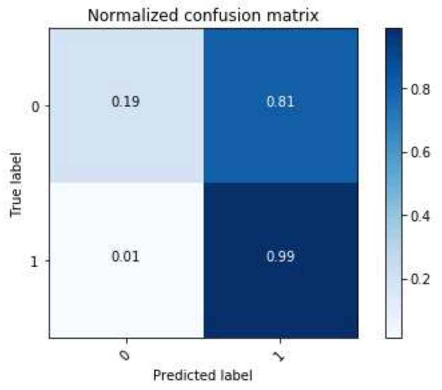
	AUC (%)	**F-measure** (%)	**Accuracy** (%)	**Recall** (%)	**Precision** (%)
DNN	88.46	93.07	87.95	96.39	89.97
SVC	85.26	92.23	85.88	**99.83**	85.70
K-NN	81.14	92.76	87.34	96.63	89.19
Decision Tree	73.50	91.56	85.68	92.54	90.59
Gaussian Naive Bayes	78.60	54.58	46.73	38.12	96.02

TABLE VI: Performance of the models based on dataset#2 (Feature density vector (without normalization) only)

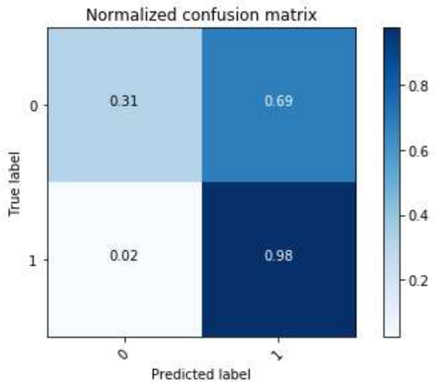
	AUC (%)	**F-measure** (%)	**Accuracy** (%)	**Recall** (%)	**Precision** (%)
DNN	79.03	92.21	85.98	98.82	86.42
SVC	72.97	91.33	84.72	95.90	87.18
K-NN	77.24	91.98	85.88	96.43	87.92
Decision Tree	72.43	90.74	84.48	90.55	90.92
Gaussian Naive Bayes	78.20	55.05	46.80	38.81	94.66



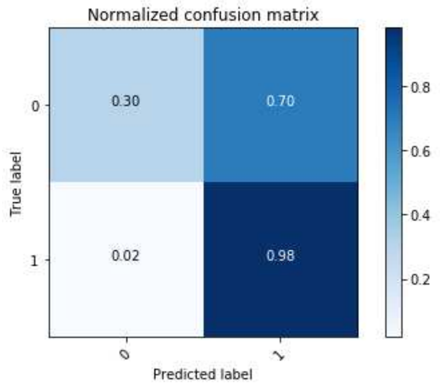
(a) Using Dataset#1



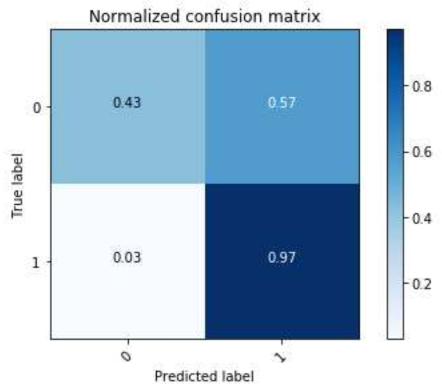
(b) Using Dataset#2



(c) Using Dataset#3



(d) Using Dataset#4



(e) Using Dataset#5

Fig. 5: Normalized confusion matrices of the DNN models

TABLE VII: Performance of the models based on dataset#3 (Feature density vector (normalized) only)

	AUC (%)	**F-measure (%)**	**Accuracy (%)**	**Recall (%)**	**Precision (%)**
DNN	82.93	92.65	87.00	97.64	88.14
SVC	76.60	91.27	83.94	1.00	83.94
K-NN	80.66	92.23	86.29	97.04	87.88
Decision Tree	69.72	90.11	83.36	90.31	89.91
Gaussian Naive Bayes	78.06	68.74	58.67	54.13	94.15

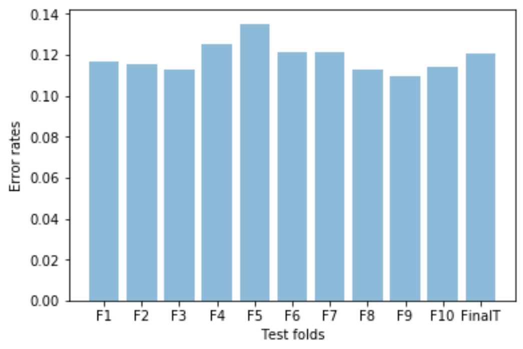
Though its recall and precision are a bit lower than the best recall and precision, it is the best model among all.

The model developed using Dataset#2 gives only the best recall, and the model developed using Dataset#1 gives only the best precision.

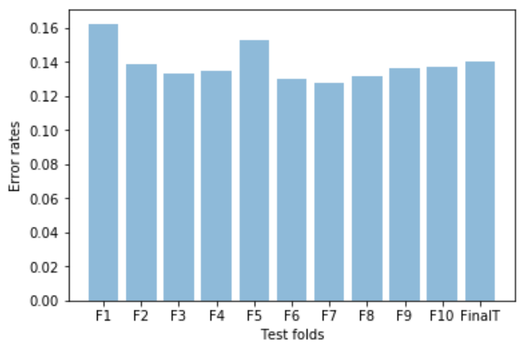
D. *Evaluation of the DNN model against the baseline models*

The DNN model was evaluated against four baseline machine learning classifiers K-NN Classifier, Support Vector Classifier, Decision Tree Classifier, and Gaussian Naive Bayes Classifier.

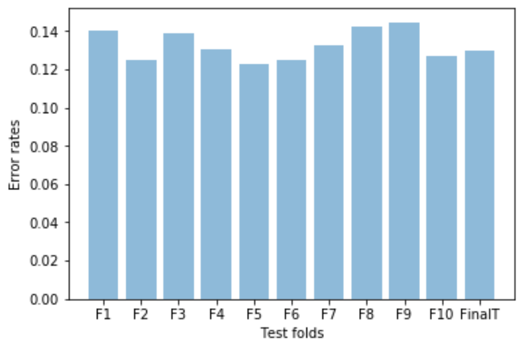
Table V - Table IX show evaluation results using each dataset. Among the four models SVC and K-NN shows overall



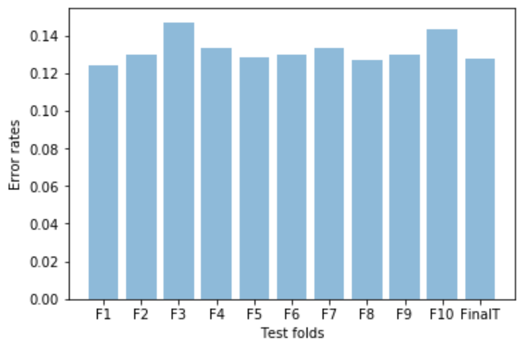
(a) Using Dataset#1



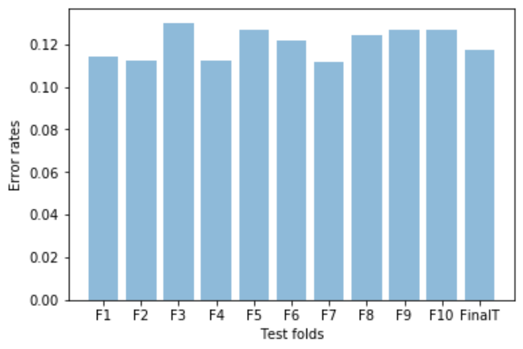
(b) Using Dataset#2



(c) Using Dataset#3



(d) Using Dataset#4



(e) Using Dataset#5

Fig. 6: Error rates of k-fold cross validation

TABLE VIII: Performance of the models based on dataset#4 (Feature appearance and density vector (without normalization))

	AUC (%)	**F-measure (%)**	**Accuracy (%)**	**Recall (%)**	**Precision (%)**
DNN	83.75	92.79	87.21	98.09	88.03
SVC	74.20	91.63	85.23	96.35	87.36
K-NN	78.25	92.14	86.12	96.96	87.78
Decision Tree	70.67	89.74	82.89	89.14	90.34
Gaussian Naive Bayes	78.24	58.83	49.9 6	42.58	95.11

TABLE IX: Performance of the models based on dataset#5 (Feature appearance and density vector (normalized))

	AUC (%)	**F-measure (%)**	**Accuracy (%)**	**Recall (%)**	**Precision (%)**
DNN	**89.53**	**93.27**	**88.26**	96.96	89.86
SVC	83.72	91.27	83.94	1.00	83.94
K-NN	82.37	93.00	87.72	97.28	89.09
Decision Tree	67.59	89.77	82.75	90.15	89.39
Gaussian Naive Bayes	80.55	64.15	54.62	48.37	95.21

good performance, which is sometimes very close to the DNN model.

As can be seen from Table V using Dataset#1 AUC, F-measure and accuracy of the DNN model are 88.46%, 93.07% and 87.95%, and the same metrics of SVC are 85.26%, 92.23%, and 85.88%. Table VII shows an example where the AUC, F-measure and accuracy of the K-NN are very close to that of DNN.

However, AUC of the best DNN model using Dataset#5 was far above than any of the baseline models.

V. Discussion

In this section we discuss our research questions, and limitations of our project.

A. RQ1: Usefulness of language feature vectors to predict change proneness

From the normalized confusion matrices of Figure 5 we see the developed DNN model using language feature vectors is able to differentiate between both positive and negative classes. Though performance varies depending on the input feature vectors, it doesn't fail completely in any cases.

Performance evaluation against the baseline models, as illustrated in Table V - Table IX, clearly shows the superiority of our language feature vector based model over all other models. It achieves AUC 89.53%, and the one closest to it achieves AUC only 83.72%.

All the results of our experiments clearly establishes that code profiles generated using language feature appearance and density vectors of source files can be used to train deep learning model to predict change proneness of the files.

Summary for RQ1. Language feature use can be used to learn and predict the source files that will require change in future.

B. RQ2: Importance of feature vectors

Because RQ1 is positive we explore RQ2.

Feature appearance and normalized density vectors together produces the best AUC, F-measure, and accuracy as can be seen in Table IX.

Comparing between feature appearance and density vectors, feature appearance vector is more effective than the density vector when considered individually. It is clear from Table V, VI, and VII. While feature appearance vector alone gives 88.46% AUC, density vector alone without normalization and normalized give only 79.03% and 82.93% respectively. Even feature appearance vector alone can produce better results than when it is used together with density vector without normalization, as can be seen from Table V, and Table VIII.

However, to get the best results both feature appearance and density vectors are needed to be used together.

Summary for RQ2. Both feature appearance and density vectors are needed for good results.

C. RQ3: Effect of data normalization

Because RQ1 is positive we explore RQ2.

Normalized density vector can help to achieve a good performance. The AUC obtained using Dataset#1 in Table IX is 5.78% higher than the AUC obtained using Dataset#4 in Table VIII due to normalization of the density vector. F-measure, accuracy and precision also increased in Table IX, but recall dropped a little.

When only density vector was used as the predictor variable, the normalized vector increased AUC nearly 4% compared to the density vector without normalization, as evident from Table VI and VII.

Summary for RQ3. Density vector needs to be normalized.

D. Limitations

Below we list some limitations of our project.

Limited data: Only 14,700 file change records were obtained from the selected projects. Machine learning models are usually trained with much larger datasets to get effective performance.

Imbalanced data: Our dataset is not balanced. We have more positive instances than negative instances. It may have negative effect on the model.

Model tuning: Due to the short duration of this project, it was not possible to tune the developed model a lot changing the parameters.

However, in future, we plan to collect more samples from other software projects. Besides, we would use oversampling and undersampling techniques for balancing the training data if needed. More tuning of the models will also be done changing the model parameters.

VI. Conclusions

Prediction of which source files may require change in a code base can help both the developers and software project managers to improve software quality. Besides, it can reduce the effort needed during software development and maintenance. In this project, code profiles developed from feature use of source files have been used to train deep neural network for predicting change proneness. The developed model uses four hidden layers with 1022, 511, 255 and 128 neurons respectively in its layers. Experiments were performed using phpbb and WordPress projects. Results show deep neural network model trained with code profile of source files is able to predict change proneness successfully. Performance of the developed model was compared with four other machine learning models, but no model achieved the performance of the developed DNN in terms of AUC, F-measure, and accuracy.

The contributions made of this project are:

1) **Code profile development of source files:** Language feature appearance and density vectors of each source file was extracted to develop the feature usage code profile of the files.
2) **DNN model development:** Several experiments were performed to find the DNN architecture good for predicting software change proneness. We find a model with four hidden layers work best for our settings. We train that model using five different datasets.
3) **Performance comparison:** Performance of the developed model was compared with four baseline machine learning models SVC, K-NN, Decision Tree, and Gaussian Naive Bayes.

The results obtained from this project provide several findings:

- **Language feature vectors as the predictor:** Language feature use profile can be used to train deep neural network for predicting change proneness of software. Our mode shows AUC 89.53%, F-measure 93.27%, and accuracy 88.26%. No other model achieves that performance.
- **Importance of feature appearance and density vectors:** Both feature appearance and density vectors are important when considering code profile to train deep neural network. We notice upto around 11% low performance in the developed DNN models in terms of AUC compared to the best performing DNN model that uses both feature appearance and normalized density vectors.
- **Need of normalization** Normalization of density vector is needed to get a good performance. In our experiment we find the effect of normalization on AUC is nearly 6%.

To the best of our knowledge, our project is the first attempt that investigates the capability of language features to predict change proneness. The insights obtained from this project can be used as a guideline for several other research. Below is a short list where we can utilize the observations obtained from this project.

- predicting the number of releases within which a source file may change.
- predicting change proneness of projects written in other languages
- predicting any kind of issue related to source file such as bug, error, security flaw etc.
- developing prediction model using abstract language features to make the model independent of any language

References

[1] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman, "Developer-related factors in change prediction: an empirical assessment," in *Proceedings of the 25th International Conference on Program Comprehension*, pp. 186–195, IEEE Press, 2017.

[2] R. Malhotra, A. Gupta, *et al.*, "Change prediction model using evolutionary history of a softwarean automated tool," in *Electrical, Computer and Electronics (UPCON), 2017 4th IEEE Uttar Pradesh Section International Conference on*, pp. 178–183, IEEE, 2017.

[3] L. Kumar, S. K. Rath, and A. Sureka, "Empirical analysis on effectiveness of source code metrics for predicting change-proneness," in *Proceedings of the 10th Innovations in Software Engineering Conference*, pp. 4–14, ACM, 2017.

[4] A. R. Sharafat and L. Tahvildari, "Change prediction in object-oriented software systems: A probabilistic approach," *Journal of Software*, vol. 3, no. 5, pp. 26–39, 2008.

[5] M. Askari and R. Holt, "Information theoretic evaluation of change prediction models for large-scale software," in *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 126–132, ACM, 2006.

[6] M. Amoui, M. Salehie, and L. Tahvildari, "Temporal software change prediction using neural networks," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 07, pp. 995–1014, 2009.

[7] "phpbb." https://github.com/phpbb/phpbb. Accessed: 2018-03-15.

[8] "WordPress." https://github.com/WordPress/WordPress. Accessed: 2018-03-15.

[9] H. Kagdi and J. I. Maletic, "Software-change prediction: Estimated+ actual," in *Software Evolvability, 2006. SE'06. Second International IEEE Workshop on*, pp. 38–43, IEEE, 2006.

[10] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pp. 78–88, IEEE, 2009.

[11] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 5–24, 2018.

[12] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Software Engineering*, vol. 18, no. 3, pp. 478–505, 2013.

[13] M. O. Elish and M. Al-Rahman Al-Khiaty, "A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software," *Journal of Software: Evolution and Process*, vol. 25, no. 5, pp. 407–437, 2013.

[14] Y. Zhou, H. Leung, and B. Xu, "Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 607–623, 2009.

[15] "Jflex." http://www.jflex.de/. Accessed: 2018-04-20.

[16] P. W.-h. Luk, "An empirical analysis of php in open source applications," Master's thesis, Queen's University (Canada), 2017.

[17] "TensorFlow." https://www.tensorflow.org/. Accessed: 2018-03-10.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] "WinMerge." http://winmerge.org/. Accessed: 2018-03-10.

[20] "The TXL Programming Language." https://www.txl.ca/. Accessed: 2018-03-10.

[21] "Anaconda." https://anaconda.org/anaconda/python. Accessed: 2018-03-10.

[22] "Jupyter." http://jupyter.org/. Accessed: 2018-03-10.