IMA Project 2: Bug prediction

Navdeep Singh Bedi

June 1, 2024

A bug prone class is a class that deserves intense verification and validation because its features point to a high chance that it may contain bugs:

1. it contains complex/long methods
2. it contains a high number of fields/methods
3. it is documented poorly

The main goal of this project is to predict bug prone classes from the code and NLP metrics using machine learning algorithms.

The project is divided into the following sections:

1. We download the code of Closure using defects4j, and extract the features vectors from the code based on Class, Method and NLP metrics.
2. Next, we label the feature vectors as buggy or non-buggy based on the list of buggy classes presented to us.
3. Then we train different classifiers on the labelled feature vectors.
4. Finally, we evaluate the classifiers based on their precision, recall and F1 values.

0 Code Repository

The code and result files, part of this submission, can be found at

Repo:
https://github.com/infoMA2023/project-02-bug-prediction-navdeeps350.git
Commit: 268d43a

1 Data Pre-Processing

I did the data-preprocessing in two steps:

1. I parsed the java files resources/defects4j-checkout-closure-1f/src/com/google/javascript/jscomp folder using javalang library to compute the code metrics. The code metrics computed are as follows:

   (a) **Class Metrics**

   For this metrics, I computed the following features for each class:

   i. number of Methods
   ii. number of Fields
   iii. number of Public methods + number of method invocations
   iv. number of Implemented interfaces

   (b) **Method Metrics**

   For this metrics, I computed the following features for each method in the class and then took the maximum value over all the methods for each class:

   i. number of statements
   ii. number of CONDITIONAL statements + number of LOOP statements
   iii. number of Exception in *throws* clause
   iv. number of Return statements

   (c) **NLP Metrics**

   For this metrics, I computed the following features for each class:

   i. number of Block comments
   ii. Average length of method names
   iii. number of Words (longest alphanumeric substrings) in block comments
   iv. number of Words in comments/number of Statements

   The resulting csv of extracted, labelled feature vectors can be found in the repository at the following path: results/feature_vectors.csv. The code for this step can be found in the following file: extract_feature_vectors.py.

2. The to create the labels for the buggy and non-buggy classes, I parsed the list of buggy classes provided to us in the resources/modified_classes path and labelled all the classes present in the folder as 1 and others as 0 in the dataframe created in the last step by adding a *buggy* column. The code for this step can be found in the following file: label_feature_vectors.py.

1.1 Feature Vector Extraction

I extracted feature vectors for **281** classes. Table 1 shows aggregate metrics about the extracted feature vectors.

1.2 Feature Vector Labelling

I labelled **281** classes, out of which **228** are buggy and **53** are non-buggy.

Stat	# MTH	#FLD	#RFC	#INT	#SZ	#CPX	#EX	#RET	#BCM	#NML	#WRD	#DCM
mean	12.01	6.76	110.37	0.14	19.62	6.04	0.4	3.52	13.83	13.76	324.67	38.84
min	0	0	0	0	0	0	0	0	1	0	2	0
max	209	167	872	2	347	99	9	86	221	28	3133	950

Table 1: Aggregate metrics about the extracted feature vectors.

2 Classifiers

Before doing hyperparameter tuning, I split the data into training and test sets using a 80-20 split. The trained set was used for hyperparameter tuning and the test set was used for evaluation. Next I did a hyperparameter tuning for each of the classifiers listed below using *GridSearchCV* with 5-fold cross-validation with the following list of hyperparameters:

1. Decision Tree (DT)

```
params_dt = {
    ’max_depth’: [2, 3, 5, 10, 20, None],
    ’min_samples_leaf’: [5, 10, 20, 50, 100],
    ’criterion’: ["gini", "entropy", "log_loss"]
}
```

2. Naive Bayes (NB)

```
params_NB = {’var_smoothing’: np.logspace(0,-9,
    num=100)}
```

3. Support Vector Machine (SVM)

```
params_svm = {’kernel’: [’rbf’, ’linear’, ’
    sigmoid’],
      ’C’: [0.1, 1, 10],
      ’gamma’: [’scale’, ’auto’]
}
```

4. Multi-Layer Perceptron (MLP)

```
params_mlp = {’hidden_layer_sizes’: [(100,),
    (50,), (10,), (5,), (100, 50), (50, 10),
    (10, 5)],
      ’activation’: [’identity’, ’logistic’, ’
          tanh’, ’relu’],
      ’solver’: [’sgd’, ’adam’],
      ’alpha’: [0.0001, 0.001],
      ’learning_rate’: [’constant’, ’invscaling’
          , ’adaptive’]
      }
```

5. Random Forest (RF)

```
        param_rf = {’n_estimators’: [10, 50, 100, 200],
            ’criterion’: [’gini’, ’entropy’, ’log_loss’
                ],
            ’max_depth’: [2, 5, 10, 20, None],
            ’min_samples_leaf’: [1, 2, 5, 10],
            ’max_features’: [None, ’sqrt’, ’log2’]}
```

In the following sections, I report the results of the hyperparameter tuning for each of the classifiers.

2.1 Decision Tree (DT)

```
    best_hyperparameters: {’criterion’: ’gini’, ’max_depth’:
         2, ’min_samples_leaf’: 20}
    accuracy: 0.75
    precision: 0.50
    recall: 0.35
    f1: 0.41
```

2.2 Naive Bayes (NB)

```
    best_hyperparameters: {’var_smoothing’:
        2.848035868435799e-06}
    accuracy: 0.78
    precision: 0.62
    recall: 0.35
    f1: 0.45
```

2.3 Support Vector Machine (SVP)

```
    best_hyperparameters: {’C’: 10, ’gamma’: ’scale’, ’
        kernel’: ’rbf’}
    accuracy: 0.71
    precision: 0.37
    recall: 0.21
    f1: 0.27
```

2.4 Multi-Layer Perceptron (MLP)

```
    best_hyperparameters: {’activation’: ’relu’, ’alpha’:
        0.001, ’hidden_layer_sizes’: (100,), ’learning_rate’:
         ’adaptive’, ’solver’: ’sgd’}
```

```
accuracy: 0.75
precision: 0.50
recall: 0.28
f1: 0.36
```

2.5 Random Forest (RF)

```
best_hyperparameters: {'criterion': 'log_loss', '
    max_depth': None, 'max_features': 'log2', '
    min_samples_leaf': 5, 'n_estimators': 10}
accuracy: 0.77
precision: 0.55
recall: 0.35
f1: 0.43
```

The code for this step can be found in the following file: train_classifiers.py.

3 Evaluation

In the last step, I used the best models for each of the classifiers saved in the previous part and using those models performed 5 fold cross-validation on the whole dataset repeated 20 times. The results of the evaluation are as follows:

3.1 Output Distributions

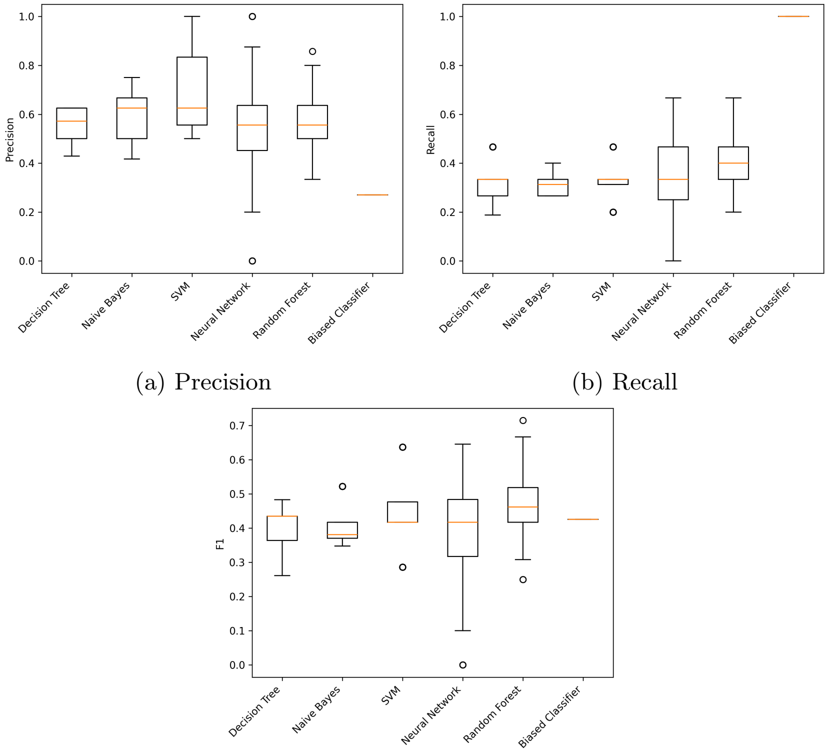
The boxplot showing mean and standard deviation for **Precision**, **Recall** and **f1** values on all 6 classifiers (5 trained + 1 biased) is shown in Figure 1.

3.2 Comparison and Significance

For every combination of two classifiers and every performance metric (precision, recall, f1) I compared which algorithm performs better, by how much, and reported the corresponding p-value. The results are as follows:

3.2.1 F1 Values

- Mean F1 for **DT**: 0.395, Mean F1 for **NB**: 0.407 ⇒ NB is better then DT (p-value = 0.0163).
- Mean F1 for **DT**: 0.395, Mean F1 for **SVM**: 0.446 ⇒ SVM is better then DT (p-value = 0.0001).
- Mean F1 for **DT**: 0.395, Mean F1 for **MLP**: 0.385 ⇒ DT is better then MLP (p-value = 0.4280).
- Mean F1 for **DT**: 0.395, Mean F1 for **RF**: 0.469 ⇒ RF is better then DT (p-value = 1.1628e-07).



(a) Precision

(b) Recall

(c) F1

Figure 1: Boxplot showing mean and standard deviation for Precision, Recall and F1 values on all 6 classifiers (5 trained + 1 biased).

- Mean F1 for **DT**: 0.395, Mean F1 for **Biased**: 0.425 ⇒ Biased is better then DT (p-value = 0.0156).
- Mean F1 for **NB**: 0.407, Mean F1 for **SVM**: 0.446 ⇒ SVM is better then NB (p-value = 0.0001).
- Mean F1 for **NB**: 0.407, Mean F1 for **MLP**: 0.385 ⇒ NB is better then MLP (p-value = 0.3585).
- Mean F1 for **NB**: 0.407, Mean F1 for **RF**: 0.469 ⇒ RF is better then NB (p-value = 2.7888e-07).
- Mean F1 for **NB**: 0.407, Mean F1 for **Biased**: 0.425 ⇒ Biased is better then NB (p-value = 0.0134).
- Mean F1 for **SVM**: 0.446, Mean F1 for **MLP**: 0.385 ⇒ SVM is better then MLP (p-value = 0.0018).
- Mean F1 for **SVM**: 0.446, Mean F1 for **RF**: 0.469 ⇒ RF is better then SVM (p-value = 0.0160).
- Mean F1 for **SVM**: 0.446, Mean F1 for **Biased**: 0.425 ⇒ SVM is better then Biased (p-value = 0.3051).
- Mean F1 for **MLP**: 0.385, Mean F1 for **RF**: 0.469 ⇒ RF is better then MLP (p-value = 1.8143e-06).
- Mean F1 for **MLP**: 0.385, Mean F1 for **Biased**: 0.425 ⇒ Biased is better then MLP (p-value = 0.0880).
- Mean F1 for **RF**: 0.469, Mean F1 for **Biased**: 0.425 ⇒ RF is better then Biased (p-value = 2.9928e-06).

3.2.2 Precision

- Mean Precision for **DT**: 0.550, Mean Precision for **NB**: 0.591 ⇒ NB is better then DT (p-value = 0.0017).
- Mean Precision for **DT**: 0.550, Mean Precision for **SVM**: 0.702 ⇒ SVM is better then DT (p-value = 9.4202e-12).
- Mean Precision for **DT**: 0.550, Mean Precision for **MLP**: 0.544 ⇒ DT is better then MLP (p-value = 0.7997).
- Mean Precision for **DT**: 0.550, Mean Precision for **RF**: 0.574 ⇒ RF is better then DT (p-value = 0.0289).
- Mean Precision for **DT**: 0.550, Mean Precision for **Biased**: 0.270 ⇒ DT is better then Biased (p-value = 1.6744e-18).
- Mean Precision for **NB**: 0.591, Mean Precision for **SVM**: 0.702 ⇒ SVM is better then NB (p-value = 7.8212e-07).

- Mean Precision for **NB**: 0.591, Mean Precision for **MLP**: 0.544 ⇒ NB is better then MLP (p-value = 0.1320).
- Mean Precision for **NB**: 0.591, Mean Precision for **RF**: 0.574 ⇒ NB is better then RF (p-value = 0.3069).
- Mean Precision for **NB**: 0.591, Mean Precision for **Biased**: 0.270 ⇒ NB is better then Biased (p-value = 2.6677e-18).
- Mean Precision for **SVM**: 0.702, Mean Precision for **MLP**: 0.544 ⇒ SVM is better then MLP (p-value = 4.7533e-12).
- Mean Precision for **SVM**: 0.702, Mean Precision for **RF**: 0.574 ⇒ SVM is better then RF (p-value = 7.1454e-12).
- Mean Precision for **SVM**: 0.702, Mean Precision for **Biased**: 0.270 ⇒ SVM is better then Biased (p-value = 2.6677e-18).
- Mean Precision for **MLP**: 0.544, Mean Precision for **RF**: 0.574 ⇒ RF is better then MLP (p-value = 0.1932).
- Mean Precision for **MLP**: 0.544, Mean Precision for **Biased**: 0.270 ⇒ MLP is better then Biased (p-value = 2.3726e-15).
- Mean Precision for **RF**: 0.574, Mean Precision for **Biased**: 0.270 ⇒ RF is better then Biased (p-value = 3.6462e-18).

3.2.3 Recall

- Mean Recall for **DT**: 0.317, Mean Recall for **NB**: 0.315 ⇒ DT is better then NB (p-value = 0.7397).
- Mean Recall for **DT**: 0.317, Mean Recall for **SVM**: 0.329 ⇒ SVM is better then DT (p-value = 0.3100).
- Mean Recall for **DT**: 0.317, Mean Recall for **MLP**: 0.325 ⇒ MLP is better then DT (p-value = 0.5862).
- Mean Recall for **DT**: 0.317, Mean Recall for **RF**: 0.403 ⇒ RF is better then DT (p-value = 1.6645e-07).
- Mean Recall for **DT**: 0.317, Mean Recall for **Biased**: 1.000 ⇒ Biased is better then DT (p-value = 1.6744e-18).
- Mean Recall for **NB**: 0.315, Mean Recall for **SVM**: 0.329 ⇒ SVM is better then NB (p-value = 1.0000).
- Mean Recall for **NB**: 0.315, Mean Recall for **MLP**: 0.325 ⇒ MLP is better then NB (p-value = 0.4060).
- Mean Recall for **NB**: 0.315, Mean Recall for **RF**: 0.403 ⇒ RF is better then NB (p-value = 2.4980e-11).

- Mean Recall for **NB**: 0.315, Mean Recall for **Biased**: 1.000 ⇒ Biased is better then NB (p-value = 1.6744e-18).
- Mean Recall for **SVM**: 0.329, Mean Recall for **MLP**: 0.325 ⇒ SVM is better then MLP (p-value = 0.6559).
- Mean Recall for **SVM**: 0.329, Mean Recall for **RF**: 0.403 ⇒ RF is better then SVM (p-value = 7.1186e-08).
- Mean Recall for **SVM**: 0.329, Mean Recall for **Biased**: 1.000 ⇒ Biased is better then SVM (p-value = 1.6744e-18).
- Mean Recall for **MLP**: 0.325, Mean Recall for **RF**: 0.403 ⇒ RF is better then MLP (p-value = 1.9083).
- Mean Recall for **MLP**: 0.325, Mean Recall for **Biased**: 1.000 ⇒ Biased is better then MLP (p-value = 3.5552e-18).
- Mean Recall for **RF**: 0.403, Mean Recall for **Biased**: 1.000 ⇒ Biased is better then RF (p-value = 2.8180e-18).

The code for this step can be found in the following file: evaluate_classifiers.py.

3.3 Practical Usefulness

The obtained classifiers can be used in a realistic bug prediction scenario to predict the bug prone classes in the codebase. The classifiers can be used to identify the classes that are more likely to contain bugs and hence can be used to prioritize the testing and verification of those classes. This can help in reducing the time and effort required for testing and verification of the codebase and can help in improving the quality of the software by identifying the bug prone classes early in the development process.