IMA Project 2

Zhizi Wen

June 3, 2022

0 Code Repository

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

Repo: https://github.com/usi-ima/2022-project-2-ZhiziWen
Commit: 2c7b7fe

1 Data Pre-Processing

I downloaded the Closure repository from the github repo, and used the code in the following subfolder for the this project "resources/defects4j-checkout-closure-1f/src/com/google/javascript/jscomp".

The resulting csv of extracted, labelled feature vectors can be found in the repository at the following path https://github.com/usi-ima/2022-project-2-ZhiziWen/blob/main/new_feature_vector_file.csv.

1.1 Feature Vector Extraction

I extracted 280 feature vectors. Table 1 shows aggregate metrics about the extracted feature vectors. As it is shown, except BCM and WRD, all the matrics have a minimum of 0, and metrics WRD has the largest maximum and average number.

In extract_feature_vectors.py, I first open all the files with .java in the name as well as have the same name as the file name, and parse then into a tree. When I find a class using ClassDeclaration, I start to look for metrics in this class node.

For getting class metrics, using node.methods and node.fields, I found the metrics "MTH" and "FLD". To find the metrics "RFC", I write a function "RFC_finder". This function takes a method as input, and count how many times are "public" in modifiers in a method and how many MethodInvocation there is in nodes of a method. To get metrics "INT", I count the implements in the class node.

For getting method metrics, I iterate over all the methods. I get the number of throws with method.throws. In all the nodes in a method, I count statement

metrics	**maximum**	**minimum**	**average**
MTH	209	0	11.500000
FLD	167	0	6.739286
RFC	759	0	72.803571
INT	3	0	0.675000
SZ	347	0	19.539286
CPX	96	0	5.796429
EX	2	0	0.117857
RET	86	0	3.635714
BCM	221	1	13.375000
NML	28	0	13.752893
WRD	2694	2	314.642857
DCM	475	0	17.597736

Table 1: aggregate metrics of extracted feature vectors

using javalang.tree.Statement and exclude block statement. I count all IfStatement, ForStatement, and WhileStatement for "CPX" and ReturnStatement for "RET". Every time when I iterate over a method, then I check whether the current number is larger than the current maximum number, and update the maximum to the current number if the current one is bigger. I also put all the names of methods into a list called methods_name.

For getting NLP metrics, I iterate over all the nodes in the class node. If the tree node has "documentation", then I check whether it is a string or a list. After that, I count the number of its documentation and the word in it using re.findall(r' w+', S), and update them to metrics "BCM" and "WRD". I count the average of length of names of methods for the metrics "NML". I get the metrics "DCM" by deviding "WRD" with number of total statements.

After getting all the metrics in a class, I store them in a dictionary. After iterating over all the classes, I get a nested dictionary where each dictionary inside is a feature vector. Then I turn the nested dictionary into a dataframe and store it in csv file.

1.2 Feature Vector Labelling

There is 75 buggy classes and 205 non-buggy classes.

2 Classifiers

2.1 Decision Tree (DT)

For Decision Tree, I first set criterion to "gini", iterate over different hyperparameter (max_depth, min_samples_split, min_samples_leaf), get the best f1 score. And then set criterion to "entropy", iterate over the same hyperparameter, get the best f1 score. And then I compare the score from "gini" and "entropy",

DT	**precision**	**recall**	**f1**
Without tuning	0.27778	0.55556	0.37037
after tuning	0.8	0.44444	0.57143

Table 2: Precision, recall and f1 before and after tuning hyperparameter in DT

and I know that the best f1 score from entropy is better. At the end, I change max_features and class_weight, and score didn't get better.

The first hyperparameter I chose is max_depth. It is the maximum depth of the tree. Without tuning, nodes would be expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. I tune it because if max_depth is too large, it would cause overfitting. And if it is too small, the classifier would not predict well. The second and third hyperparameter I chose is min_samples_split and min_samples_leaf. Min_samples_split is the minimum number of samples required to split an internal node. Min_samples_leaf is the minimum number of samples required to be at a leaf node. Min_samples_leaf is always guaranteed no matter the Min_samples_split value. Without tuning, the classifier will split until it has only one 1 instance, which might cause overfitting problem. The last hyperparameter is criterion, it is the function to measure the quality of a split. I chose it, because difference criterion might be superior in some cases and inferior in others and I don't know which one is better in our case.

The reason why the classifier works better after tuning is that after tuning, there is a limit on maximum depth and min_samples_leaf is higher than default, which means that it can prevent the classifier from overfitting.

2.2 Naive Bayes (NB)

For Naive Bayes classifier, I choose to tune the hyperparameter var_smoothing. It adds a user-defined value to the distribution's variance. This can explain more samples that are further away from the distribution mean. Actually the score didn't change after tuning it, the three value I chose doesn't make a difference. The reason might be that there are not many samples far away from the distribution mean.

NB	**precision**	**recall**	**f1**
Without tuning	0.2	0.11111	0.14286
after tuning	0.2	0.11111	0.14286

Table 3: Precision, recall and f1 before and after tuning hyperparameters in NB

2.3 Support Vector Machine (SVP)

For Support Vector Machine, I first tried to change the kernel. It turns out the linear and poly kernal takes a long time for computing, so I chose rbf kernel.

SVP	**precision**	**recall**	**f1**
Without tuning	0.66667	0.22222	0.33333
after tuning	0.42857	0.33333	0.375

Table 4: Precision, recall and f1 before and after tuning hyperparameters in SVP

Next hyperparameter is C, which is the penalty parameter of the error term. It controls the trade-off between a smooth decision boundary and correctly classifying training points. When C is larger, it might cause overfitting problem. So it is important to find a fitting C. The last hyperparameter I chose is gamma. It is a parameter for kernal of non linear hyperplanes. The value of gamma define how the data is transformed.

The reason why the classifier works better after tuning is that after changing C, the classifier defines the boundary of each class good enough and not overfit. And the gamma I got after tunning is the best for tranforming data.

2.4 Multi-Layer Perceptron (MLP)

MLP	**precision**	**recall**	**f1**
Without tuning	0.42857	0.33333	0.375
after tuning	0.5	0.55556	0.52632

Table 5: Precision, recall and f1 before and after tuning hyperparameter in MLP

I chose these hyperparameter to tune: solver, hidden_layer_sizes and activation. "Solver" stands for the solver for weight optimization. According to the manual of MLP in sklearn, solver 'adam' works well with data with thousands of records and 'lbfgs' works better for smaller dataset. But our data has 280 records, which is neither too large or too small, so I decided to check which solver is better. Hidden_layer_sizes are the number of neurons in the hidden layers. It is the essential part of this algorithum and decides the feedback we get. Activation stands for activation function for the hidden layer. In order to learn complicated patterns, we need to introduce non-linearity to the network. A neural network with numerous hidden layers would become a large linear regression model without non-linear activation functions, making it useless for learning complicated patterns from our data. I choose to turn it, because the sort of activation function we use inside the hidden layers will have a substantial impact on the performance of a neural network model.

The reason why after tuning the parameter, the result is better, might be that there is more layers and the classifier can therefore classify better.

Noticeably, a warning "ConvergenceWarning: lbfgs failed to converge (status=1): STOP: TOTAL NO. of ITERATIONS REACHED LIMIT." raise. To solve this warning, it is possible to make the max_iter larger. However, after testing, it would take a longer time and the score didn't get better. With some higher max_iter, the score even got worst. So I leave the max_iter like this.

2.5 Random Forest (RF)

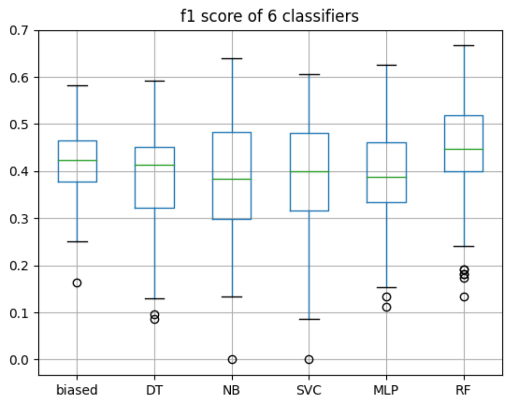
RF	**precision**	**recall**	**f1**
Without tuning	0.5	0.33333	0.4
after tuning	0.5	0.44444	0.47059

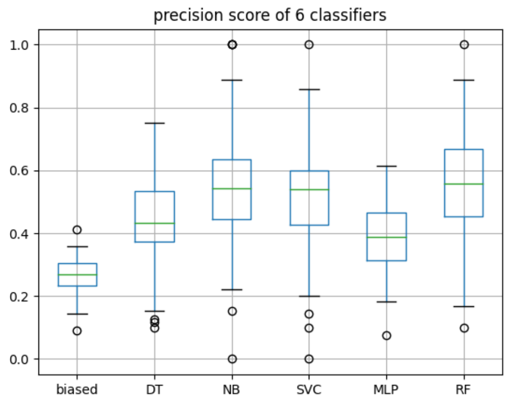
Table 6: Precision, recall and f1 before and after tuning hyperparameter in MLP

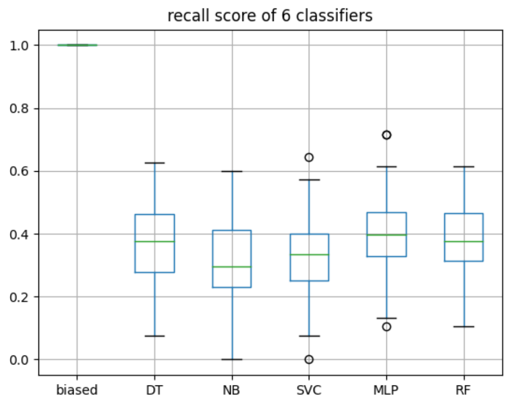
I choose to tune these hyperparameters: bootstrap, max_depth, max_features, min_samples_leaf, min_samples_split, n_estimators. According to sklearn, bootstrap stands for Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree. I choose it because I would like to see whether it is better to use bootstrap samples, which will cause less computational power. Max_features are the maximum number of features Random Forest is allowed to try in individual tree. Increasing max_features generally improves the performance of the model, but also decrease the speed. Similarly, N_estimators are the number of trees in the forest. Higher number of trees gives better performance but also makes the code runs slower. So I want to tune these two hyperparameters for the best fit. I tune max_depth, min_samples_leaf, min_samples_split for the same reason I gave in Decision Tree.

The reason why after tuning the parameter, the result is better, might be that I gave a limit to max_depth, which avoid overfitting; I also set bootstrap to true, so that the whole dataset is used to build each tree, and the result is more accurate.

3 Evaluation







3.1 Comparison and Significance

3.1.1 F1 Values

- Mean F1 for **biased**: 0.420217, Mean F1 for **DT**: 0.389791 $\Rightarrow$ biased is better then DT (p-value = 0.026338271887560962)
- Mean F1 for **biased**: 0.420217, Mean F1 for **NB**: 0.386395 $\Rightarrow$ biased is better then NB (p-value = 0.008237388786507756)
- Mean F1 for **biased**: 0.420217, Mean F1 for **SVC**: 0.390643 $\Rightarrow$ biased is not significantly better then SVC (p-value = 0.06668053379466528)
- Mean F1 for **biased**: 0.420217, Mean F1 for **MLP**: 0.387991 $\Rightarrow$ biased is better then MLP (p-value = 0.004117338902543791)
- Mean F1 for **biased**: 0.420217, Mean F1 for **RF**: 0.441794 $\Rightarrow$ RF is better then biased (p-value = 0.021439921745131617)
- Mean F1 for **DT**: 0.389791, Mean F1 for **NB**: 0.386395 $\Rightarrow$ DT is not significantly better then NB (p-value = 0.8765706153153421)
- Mean F1 for **DT**: 0.389791, Mean F1 for **SVC**: 0.390643 $\Rightarrow$ SVC is not significantly better then DT (p-value = 0.9268926049843437)
- Mean F1 for **DT**: 0.389791, Mean F1 for **MLP**: 0.387991 $\Rightarrow$ DT is not significantly better then MLP (p-value = 0.7270675547903731)
- Mean F1 for **DT**: 0.389791, Mean F1 for **RF**: 0.441794 $\Rightarrow$ RF is better then DT (p-value = 2.1666128599611302e-05)
- Mean F1 for **NB**: 0.386395, Mean F1 for **SVC**: 0.390643 $\Rightarrow$ SVC is not significantly better then NB (p-value = 0.5300741288371795)
- Mean F1 for **NB**: 0.386395, Mean F1 for **MLP**: 0.387991 $\Rightarrow$ MLP is better then NB (p-value = 0.8513556615531508)
- Mean F1 for **NB**: 0.386395, Mean F1 for **RF**: 0.441794 $\Rightarrow$ RF is better then NB (p-value = 4.2377615440064014e-05)
- Mean F1 for **SVC**: 0.390643, Mean F1 for **MLP**: 0.387991 $\Rightarrow$ SVC is not significantly better then MLP (p-value = 0.590508132596941)
- Mean F1 for **SVC**: 0.390643, Mean F1 for **RF**: 0.441794 $\Rightarrow$ RF is better then SVC (p-value = 9.799562355792362e-05)
- Mean F1 for **MLP**: 0.387991, Mean F1 for **RF**: 0.441794 $\Rightarrow$ RF is better then MLP (p-value = 1.6203393318660424e-05)

3.1.2 Precision

- Mean precision for **biased**: 0.267857, Mean precision for **DT**: 0.441348 ⇒ DT is better then biased (p-value = 6.157031936703586e-17)
- Mean precision for **biased**: 0.267857, Mean precision for **NB**: 0.539933 ⇒ NB is better then biased (p-value = 9.296413870794677e-18)
- Mean precision for **biased**: 0.267857, Mean precision for **SVC**: 0.519537 ⇒ SVC is better then biased (p-value = 4.0853742171444245e-17)
- Mean precision for **biased**: 0.267857, Mean precision for **MLP**: 0.386682 ⇒ MLP is better then biased (p-value = 4.285631019798359e-15)
- Mean precision for **biased**: 0.267857, Mean precision for **RF**: 0.551599 ⇒ RF is better then biased (p-value = 5.2603637164219604e-18)
- Mean precision for **DT**: 0.441348, Mean precision for **NB**: 0.539933 ⇒ NB is better then DT (p-value = 1.3537691046512196e-07)
- Mean precision for **DT**: 0.441348, Mean precision for **SVC**: 0.519537 ⇒ SVC is better then DT (p-value = 2.9993972042591306e-07)
- Mean precision for **DT**: 0.441348 Mean precision for **MLP**: 0.386682 ⇒ DT is better then MLP (p-value = 5.973127313428073e-05)
- Mean precision for **DT**: 0.441348, Mean precision for **RF**: 0.551599 ⇒ RF is better then DT (p-value = 5.1373734661568446e-11)
- Mean precision for **NB**: 0.539933, Mean precision for **SVC**: 0.519537 ⇒ NB is not significantly better then SVC (p-value = 0.26307751153002257)
- Mean precision for **NB**: 0.539933, Mean precision for **MLP**: 0.386682 ⇒ NB is better then MLP (p-value = 2.914182499085914e-11)
- Mean precision for **NB**: 0.539933, Mean precision for **RF**: 0.551599 ⇒ RF is not significantly better then NB (p-value = 0.5371472683468925)
- Mean precision for **SVC**: 0.519537, Mean precision for **MLP**: 0.386682 ⇒ SVC is better then MLP (p-value = 3.0933560229485024e-11)
- Mean precision for **SVC**: 0.519537, Mean precision for **RF**: 0.551599 ⇒ RF is better then SVC (p-value = 0.021586324413793722)
- Mean precision for **MLP**: 0.386682, Mean precision for **RF**: 0.551599 ⇒ RF is better then MLP (p-value = 8.064709328740633e-14)

3.1.3 Recall

- Mean recall for **biased**: 1.000000, Mean recall for **DT**: 0.365292 ⇒ biased is better then DT (p-value = 3.824441982627038e-18)
- Mean recall for **biased**: 1.000000, Mean recall for **NB**: 0.314134 ⇒ biased is better then NB (p-value = 3.84002073447656e-18)
- Mean recall for **biased**: 1.000000, Mean recall for **SVC**: 0.330923 ⇒ biased is better then SVC (p-value = 3.841972302851667e-18)
- Mean recall for **biased**: 1.000000, Mean recall for **MLP**: 0.401048 ⇒ biased is better then MLP (p-value = 3.812581931165565e-18)
- Mean recall for **biased**: 1.000000, Mean recall for **RF**: 0.385033 ⇒ biased is better then RF (p-value = 3.807632619176109e-188)
- Mean recall for **DT**: 0.365292, Mean recall for **NB**: 0.314134 ⇒ DT is better then NB (p-value = 0.00037642804130508353)
- Mean recall for **DT**: 0.365292, Mean recall for **SVC**: 0.330923 ⇒ DT is better then SVC (p-value = 0.009458642219143557)
- Mean recall for **DT**: 0.365292, Mean recall for **MLP**: 0.401048 ⇒ MLP is better then DT (p-value = 0.027089081080596603)
- Mean recall for **DT**: 0.365292, Mean recall for **RF**: 0.385033 ⇒ RF is not significantly better then DT (p-value = 0.19604223214895067)
- Mean recall for **NB**: 0.314134, Mean recall for **SVC**: 0.330923 ⇒ SVC is not significantly better then NB (p-value = 0.19601688899770986)
- Mean recall for **NB**: 0.314134, Mean recall for **MLP**: 0.401048 ⇒ MLP is better then NB (p-value = 7.664348317787021e-08)
- Mean recall for **NB**: 0.314134, Mean recall for **RF**: 0.385033 ⇒ RF is better then NB (p-value = 9.695091870363237e-07)
- Mean recall for **SVC**: 0.330923, Mean recall for **MLP**: 0.401048 ⇒ MLP is better then SVC (p-value = 7.777110658524853e-07)
- Mean recall for **SVC**: 0.330923, Mean recall for **RF**: 0.385033 ⇒ RF is better then SVC (p-value = 7.301403530077352e-05)
- Mean recall for **MLP**: 0.401048, Mean recall for **RF**: 0.385033 ⇒ MLP is not significantly better then RF (p-value = 0.2907469586821052)

In general, Random Forest Classifier is the best classifier among the six classifiers. Its f1 score is significantly better than every other classifier, and its precision is significantly better than every other classifier except for NB, where its score is better but not significant. Surprisingly, biased classifier have a recall of 1.0, which means that it does not have false negative result. Also, except for Random Forest Classifier, biased classifier is not significantly better than SVC, and significantly better than three other classifiers with its f1 score. If the goal is to have a high recall, then biased classifier should be considered.

3.2 Practical Usefulness

In summary, this code can be useful for the first attempt of bug prediction, ans programmers should not blindly trust it. I predict bugs in the code using five classifiers. In general, we can see that the Random Forest Classifier is the best one due to its significantly better f1 score. When it is important to have a high recall, biased classifier should be considered. In reality, programmers should not blindly use Random Forest Classifier, because its F1, precision and recall are still far from 1.

Furthermore, I excluded some cases that took too long while tuning the hyperparameters. These removed cases might outperform my estimated parameters. Programmers should consider this restriction when using my code.