Developing a JIT bug prediction classifier

A Machine Learning Project in Automated Software Engineering

Pertti Palokangas
Computer Science/Karlstad
University
Karlstad, Sweden
palokangas.pertti.r.e@gmail.com

ABSTRACT

This report is written by Pertti Palokangas as a result of the assignment in the university course *DVAD81: Project in Automated Software Engineering* at Karlstad University, 2024.

The purpose of this study is to construct, train and evaluate three different machine learning models and their learning algorithms. The dataset used in this study, was originally collected and partly prepared by McIntosh & Kamei [1]. In this study the dataset is pre-processed further, in order to achieve a good input to the models.

The study also uses some different libraries for developing, training, and evaluating the three models. For the evaluations a number of well-known and widely used metrics will be used.

KEYWORDS

empirical software engineering, software metrics, defect prediction, just-in-time prediction, software defect prediction

ACM Reference format:

Pertti Palokangas. 2024. Developing a JIT bug prediction classifier: A Machine Learning Project in Automated Software Engineering. Computer Science, Karlstad University, Karlstad, Sweden, *8 pages.*
https://github.com/PerttiP/ML-Project-in-Automated-Software-Engineering/blob/main/

1 Introduction

Developing really good, secure, robust and efficient software is hard work. It would be very valuable for a development project to have a solid way of determining if a specific commit is to be considered as safe (free from bugs) or not, and even better if this signal was presented “in real-time”, so to speak, before the commit is approved and merged into an integration or a delivery branch.
For this reason, there have been a lot of investigations done during the recent decades in the area of just-in-time (JIT) bug prediction by using machine learning techniques.
This ANN (Artificial Neural Network) project in my study is about supervised learning, since I include labeled data in the training data set.

2 Pre-processing the data

2.1 The dataset

The dataset consists of two .csv files with commit data from two open-source projects, OpenStack and Qt. Each file has values covering several features for each unique commit, explained in a paper by McIntosh & Kamei [1]. The features are meta-data and metrics about the actual commit, and they have been categorized into metrics families. See Table 2 (“*A taxonomy of the studied families of code and review properties.*”) in McIntosh & Kamei’s report [1].
I drop the columns 'commit_id' and 'author_date' since they have unique values and are irrelevant for the bug classification. Also ‘fixcount’ is dropped because I do not see how it could affect the bug prediction.
Here is a simplified table with mappings to 27 feature names used in the dataset and in the code (omitting irrelevant features):

Name	Definition	Metrics Family
LA	Lines added	Size
LD	Lines deleted	Size
NF	Number of modified files	Diffusion
ND	Number of modified directories	Diffusion
NS	Number of modified subsystems	Diffusion
ENT	Distribution of the modified code across each file (entropy)	Diffusion
REVD	Reviewed (Boolean)	Review
NREV	Number of reviewers that approved	Review
RTIME	Review time window	Review
TCMT	Comments	Review
HCMT	Comments	Review
SELF	Self-reviewed (Boolean)	Review
NDEV	Number of developers that changed the modified file	History
AGE	The average time interval between the last and the	History

	current change	
NUC	The number of unique changes to the modified file	History
SAEXP	Subsystem author experience	Experience
REXP	Recent developer experience	Experience
OEXP	Older developer experience	Experience
AREXP	A? reviewer experience	Experience
RREXP	Recent reviewer experience	Experience
OREXP	Older reviewer experience	Experience
ASEXP	A? subsystem experience	Experience
RSEXP	Recent subsystem experience	Experience
OSEXP	Older subsystem experience	Experience
ASAWR	Author subsystem awareness	Experience
RSAWR	Recent reviewer subsystem awareness	Experience
OSAWR	Older reviewer subsystem awareness	Experience

Table 1: Feature names and definitions.

2.2 The pre-processing steps

There are data pre-processing steps that are crucial to do, in order to get as representative data as possible.

- Data Cleaning
- Data Analysis
- Data Manipulation

Handling of missing data can be done by ignoring (removing) data or by replacing data. The replacing can be achieved by imputing new data according to some clever rule or algorithm that produces “true and relevant” data, with key information included. Data values to impute could also be generated by ML.

The data analysis should be a major part of the pre-processing step as it gives valuable insights to understanding the data and its distribution and dependencies: What the data represents, which data are most relevant and finding out outliers or noise in the data.

Data manipulation is about transforming the data to numerical values, so that it is more suitable for neural network learning algorithms.

Pre-processing steps done in this study:

1. Concatenate into one dataframe
2. Show number of missing values in each column
3. Analyze original dataframe
4. Analyze selected features using Correlation matrix
5. IMPUTATION STEP: Add column isbuggy = 1 (True) for all rows where nrev < 2 AND bugcount is NaN
6. Extend dataframe with additional column/feature called 'code_churn'
7. Encode “categorical” (actually Boolean) data in 'revd' and 'self' to numerical (int)

For data processing simplicity reasons and to get a large coherent dataframe, I concatenate the two datasets into one dataframe. Another possibility is to use data augmentation by simply appending multiple copies of the dataframe to make it larger. I chose to do both.

2.3 Data correlation analysis

From the first correlation matrix with a few of the features, it seems that ‘nrev’ (number of reviewers) has a rather high correlation (0.17) to the target class ‘bugcount’.

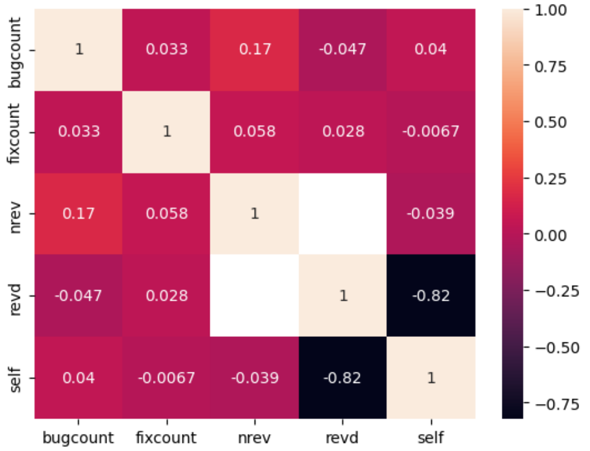


Figure 1: Correlation matrix for 5 features.

I chose to replace the ‘bugcount’ integer values with a Boolean ‘isbuggy’ that will be the target (predicted output) in a binary classification model.

For this reason and since ocular review shows that the number of reviewers seems to have an important impact on the probability of a bug, I handle the missing values for ‘bugcount’ by doing an imputation of values in ‘isbuggy’ column, like this:

Imputation step (pseudo code):

```
Add column isbuggy = 1 (True) for all rows where
    nrev < 2 AND bugcount is NaN
```

Important to note is that the imputation is done only for rows where ‘bugcount’ is NaN (Not a Number), so I do not overwrite any ‘isbuggy’ rows that already have either a 0 or a 1. Another reason for doing the imputation is that if I would have removed all rows with missing ’bugcount’ values, there would not be many rows left (only about 700), and I think a small dataset would have been bad for the training.

After the imputation all remaining rows where ‘isbuggy’ has not been updated with a number 0 or 1, are then removed. Also the columns 'bugcount' and ‘fixcount’ are removed completely.

2.4 Additional feature: code churn

The *absolute code churn measure* is a measure of the size of the change. It is derived simply by adding the **la** (lines added) and the **ld** (lines deleted) [2]:

code_churn = la + ld

From the correlation matrix it is seen that 'code_churn' has an extremely high correlation with la (lines added): 0.98. The correlation with ld (lines deleted) is not quite as high: 0.34. As might be expected 'self' and 'revd' has a very high negative correlation (-0.93).

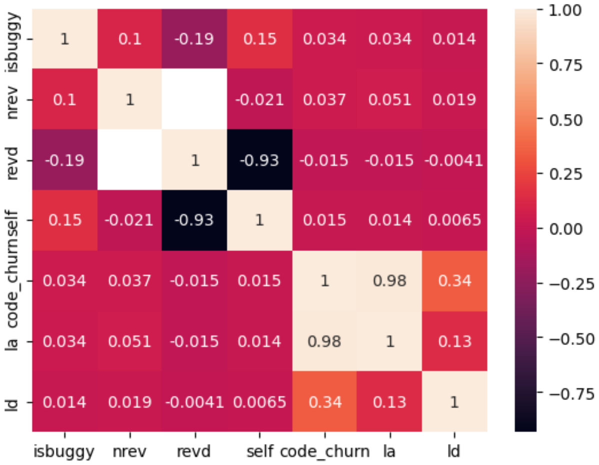


Figure 2: Correlation matrix for 7 features.

According to a study made by Nagappan and Ball [2], the relative code churn (la + ld / total lines of code) would have been a better measure, but in our case the data about total lines of code is missing in the dataset. I still chose to add the 'code_churn' as an additional feature.

2.5 Encoding data

I encode the boolean data values in columns 'revd' and 'self' to numerical integer values {0,1} using LabelEncoder from sklearn.preprocessing.

The final output of the pre-processing is saved in a file called 'PrepDataframe24features.csv'. Exactly the same file covering 24 selected features, will be the input data to all 3 models, in order to be able to compare them adequately.

3 Learning algorithms

The learning algorithms I have chosen to evaluate are:

1. Perceptron.
2. Sequential ANN.
3. Random Forest.

Before training starts, the dataset needs to be split into the following parts:

– Training part (70%): I train the model on this part

– Test part (30%): I evaluate the performance on this part

– Validation part: I could have used this part for parameter fine-tuning

I chose to do a split into training and test data only, and not validation data, since parameter fine-tuning is not part of the scope in this project.

3.1 Dealing with imbalanced data

The dataset is heavily imbalanced, as can be seen from class distribution counters of the target class 'isbuggy' in trainY:

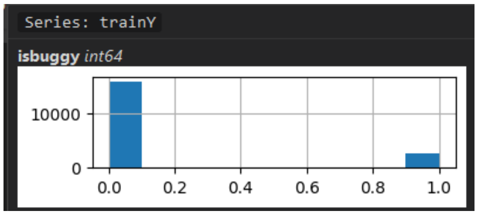


Figure 3: Plot of 'isbuggy' distribution in the training data.

An option to deal with this is by oversampling of training data using SMOTE (Synthetic Minority Oversampling Technique) [3]. The implementation has been done using the imbalanced-learn Python library [4]. Important that SMOTE is only applied to the training dataset. This should improve performance on the minority class, in this case the '1' (true) values of the 'isbuggy' feature.

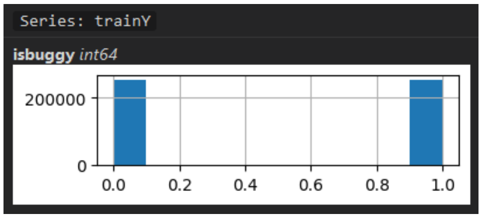


Figure 4: Plot of 'isbuggy' distribution after data augmentation and oversampling with SMOTE.

3.2 Data transforms

I normalize the data using *StandardScaler* from the Scikit-Learn library, so that it resembles standard normally distributed data, having zero mean and unit variance. The transformation is done on both training and test data sets, but fit is done only on the training data. The test data is previously unseen data by the model. Other possibilities are to use the z-score method or the max-min normalizing method.

3.3 The Perceptron model

A perceptron model is a linear model using a linear combination of input features and their weights. It is a good basic model when constructing an ANN for binary classification.

The perceptron model for this study is created with the following parameters:

- *max_iter* = 1000 : The maximum number of passes over the training data (aka epochs).
- *eta0* = 0.1 : Constant by which the updates are multiplied (aka learning rate).

The learning rate will affect the weights, so that a higher learning rate gives a faster way to overshoot the optimum, while a lower learning rate takes more time to converge to the optimum. [5]

Training is done by fitting the model. So, I pass in trainX an trainY as the training data to the *model.fit* function.

3.4 The Sequential ANN model

To construct a bit more complex ANN, I have used the Sequential model with a linear stack of layers. Each layer is built using the *Dense* core layer class from Keras API [6]. I also have added two *Dropout* layers, which is a regularization technique, to randomly ignore some neurons in the training. This should make the network less sensitive to specific weights and help in avoiding overfitting the training data [7].

The activation function will be "relu" (rectified linear unit) for all layers, except the last output layer, which will need to have a sigmoid function.

For the training I specify the optimizer as "*adam*", which is a popular optimization algorithm.

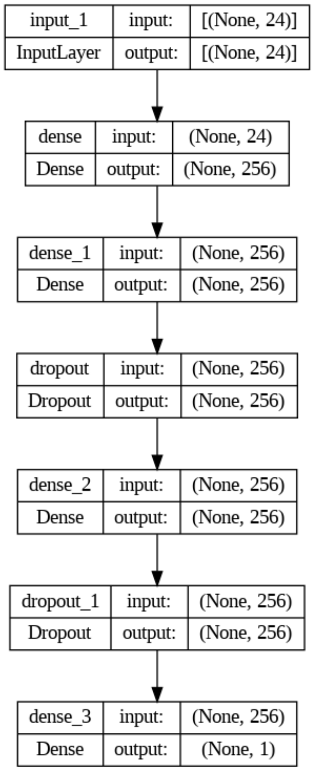

input_1 | input: | [(None, 24)]
InputLayer | output: | [(None, 24)]
dense | input: | (None, 24)
Dense | output: | (None, 256)
dense_1 | input: | (None, 256)
Dense | output: | (None, 256)
dropout | input: | (None, 256)
Dropout | output: | (None, 256)
dense_2 | input: | (None, 256)
Dense | output: | (None, 256)
dropout_1 | input: | (None, 256)
Dropout | output: | (None, 256)
dense_3 | input: | (None, 256)
Dense | output: | (None, 1)

Figure 5: The sequential ANN model with 7 layers.

In the first attempt I actually had to decrease the maximal number of epochs to 50, otherwise the training would have taken several hours on my poor performing machine.

Then I found a function called *Early Stopping*, which will stop training when the accuracy has stopped to improve. This is important to stop a possibly very long training process and keep the best weights found.

However, according to [8], relying on Early Stopping's *restore_best_weight* should be avoided. It can cause the model to not keep the best weight when Early Stopping doesn't reach the pre-defined amount of patience epochs. For this reason, I have used the *ModelCheckpoint* callback [9].

3.5 The Random Forest model

A *decision forest* is a generic term to describe models made of multiple decision trees, and where the prediction of a decision forest is the aggregation of the predictions of its decision trees [10].

Google developer docs state that in machine learning, an *ensemble* is a collection of models whose predictions are averaged (or aggregated in some way). If the ensemble models are different enough without being too bad individually, the quality of the ensemble is generally better than the quality of each of the individual models [11].

A Random Forest (RF) is an ensemble of decision trees in which each decision tree is trained with a specific random noise [12]. The main reason for me to investigate Random Forest is the fact that they are the most popular form of decision tree ensemble. I have also found out that RF is rather popular to use in other JIT bug prediction studies, like JITLine [13].

The Random Forest classifier is constructed with the parameters:

- *n_estimators* = 300 : The number of trees in the forest.
- *random_state* = 42 : Use randomization.
- *n_jobs* = -1 : number of jobs to run in parallel, -1 means using all processors.

I implemented the RF using *RandomForestClassifier* from Scikit-Learn [14].

After some experimenting, I ended up with using the same number of trees (300), as in JITLine [13].

It is possible to visualize the Decision Tree classifier model, using *export_graphviz* in Scikit-Learn library [15].

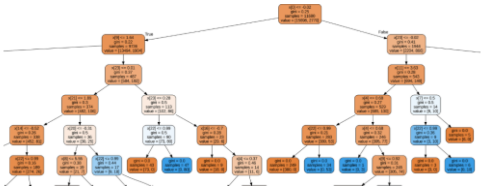


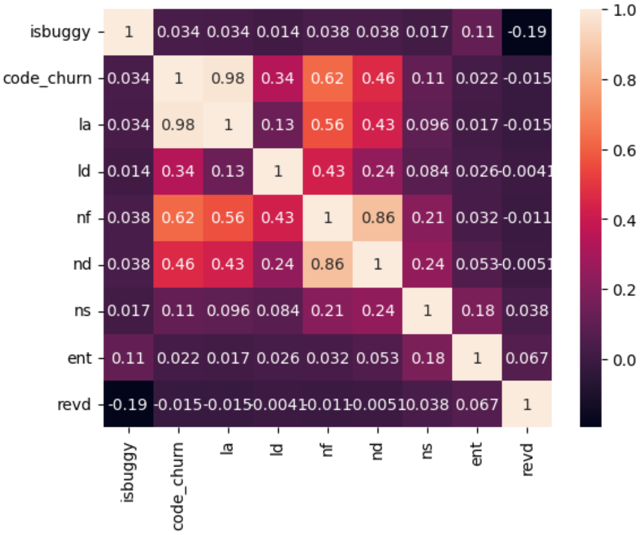
Figure 6: Visualized Decision Tree classifier model.

4 Classification

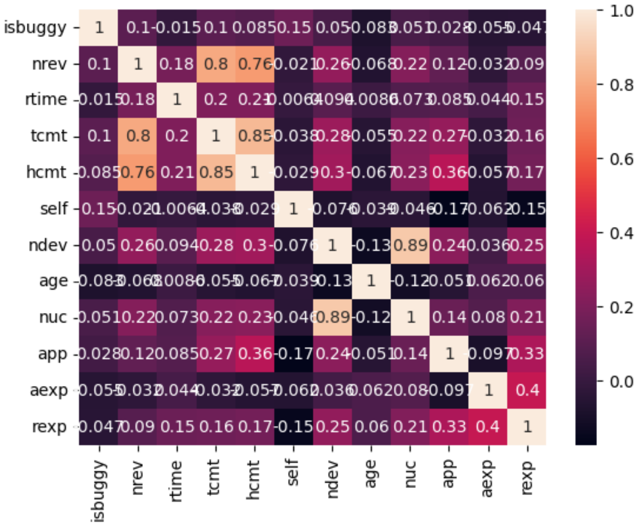
4.1 Feature selection

Feature redundancy, low correlation, and irrelevant data can all decrease the performance of a classification model.
Since highly correlated features lead to that the other features are irrelevant, I do a feature selection based on correlation matrix heat maps. Since the number of features is so large I divide the heat maps into 3 parts.
By studying these 3 heat maps and deciding on a correlation threshold of 0.88 (88%), for determining if a feature pair has a very high correlation.

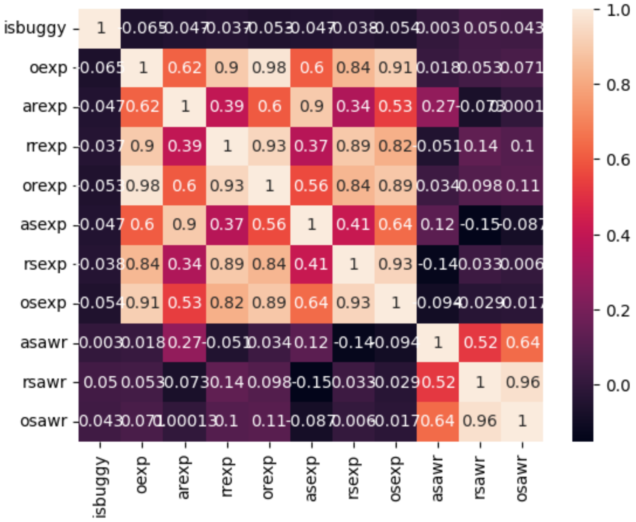
Heat Map 1:



Heat Map 2:



Heat Map 3:



The following feature-pairs have a very high correlation, above 0.88:
code_churn – la : 0.98
ndev – nuc : 0.89
Osexp – Orexp : 0.89
NOTE: Remove both because I keep "Oexp" only.
Osexp – Oexp : 0.91
Oexp – Orexp : 0.98
Oexp – Rrexp : 0.9
NOTE: Keep both since I chose to remove "Osexp" and "Orexp" already.
Asexp – Arexp : 0.9
Rrexp – Orexp : 0.93
Rrexp – Rsexp : 0.89
NOTE: Keep both since I chose to remove "Osexp" and "Orexp" already.
Osexp – Rsexp : 0.93
Osawr - Rsawr : 0.96

This analysis boils down to that I remove the following 6 redundant features: la, nuc, Osexp, Orexp, Arexp, Osawr.

4.2 Evaluation of the models

A confusion matrix is one performance evaluation tool in machine learning, representing the accuracy of a classification model. It displays the number of true positives, true negatives, false positives, and false negatives, like this:

- False negatives and false positives are samples that were incorrectly classified.
- True negatives and true positives are samples that were correctly classified.

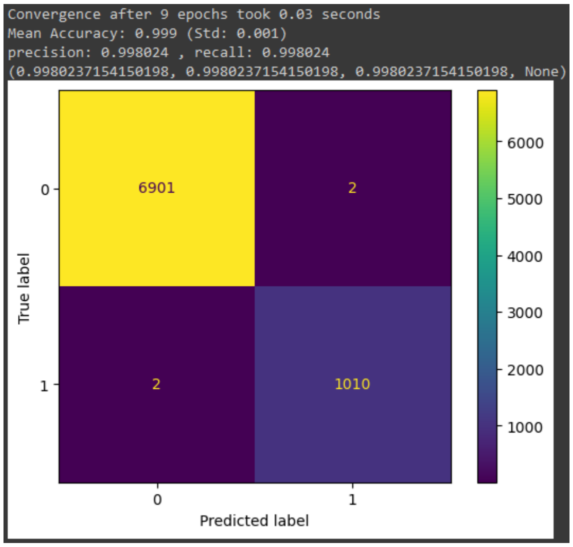
For the evaluation of the models, besides using confusion matrix, I have also used some other evaluation metrics commonly used for deterministic binary predictions:

- **Accuracy** is the percentage of examples correctly classified.

- **Precision** is the percentage of predicted positives that were correctly classified.
- **Recall** is the percentage of actual positives that were correctly classified.

4.2.1 Evaluation of the Perceptron

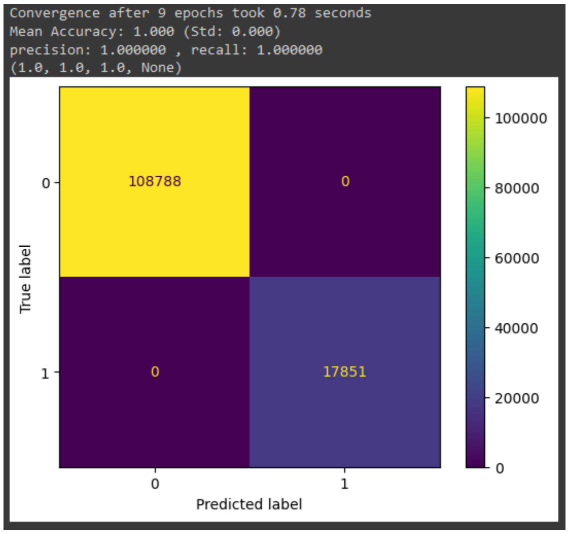
Test Run 1.1 - Without doing any data augmentation nor any oversampling: We get two false positives and two false negatives. Precision and recall are very close to 1.0 (100%).



```
Convergence after 9 epochs took 0.03 seconds
Mean Accuracy: 0.999 (Std: 0.001)
precision: 0.998024 , recall: 0.998024
(0.9980237154150198, 0.9980237154150198, 0.9980237154150198, None)
```

Figure 7: Confusion matrix for Test Run 1.1

Test Run 1.2 - With doing both data augmentation and oversampling with SMOTE: A perfect model with 100% accuracy is constructed!



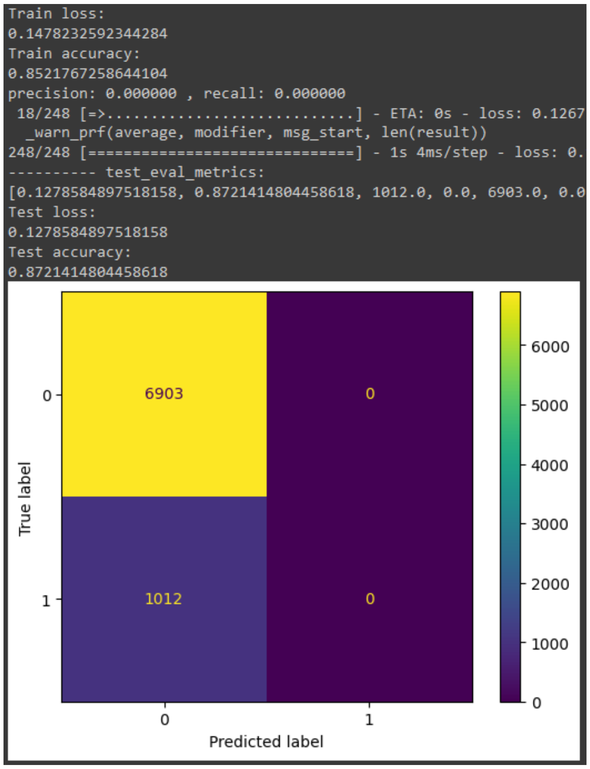
```
Convergence after 9 epochs took 0.78 seconds
Mean Accuracy: 1.000 (Std: 0.000)
precision: 1.000000 , recall: 1.000000
(1.0, 1.0, 1.0, None)
```

Figure 8: Confusion matrix for Test Run 1.2

The studies I made shows that in order to get a perfect model with the Perceptron, I need to do both data augmentation and oversampling. The data augmentation is done by extending the dataset with 3 copies of itself.

4.2.2 Evaluation of the Sequential ANN

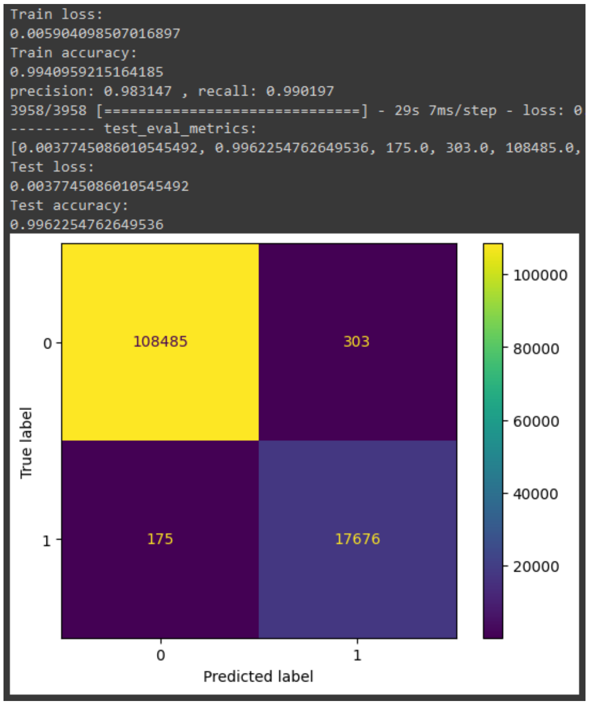
Test Run 2.1 - Without doing any data augmentation nor any oversampling: We get over 1000 false negatives, and no true positives at all. Precision and recall are therefore both 0.



```
Train loss:
0.1478232592344284
Train accuracy:
0.8521767258644104
precision: 0.000000 , recall: 0.000000
 18/248 [=>...........................] - ETA: 0s - loss: 0.1267
  _warn_prf(average, modifier, msg_start, len(result))
248/248 [==============================] - 1s 4ms/step - loss: 0.
---------- test_eval_metrics:
[0.1278584897518158, 0.8721414804458618, 1012.0, 0.0, 6903.0, 0.0
Test loss:
0.1278584897518158
Test accuracy:
0.8721414804458618
```

Figure 9: Confusion matrix for Test Run 2.1

Test Run 2.2 - With doing both data augmentation and oversampling with SMOTE: We get 175 false negatives and 303 false positives. Precision and recall are improved a lot. The test accuracy is 99.62%, which is really good, it is actually a bit higher than the training accuracy even.



```
Train loss:
0.005904098507016897
Train accuracy:
0.9940959215164185
precision: 0.983147 , recall: 0.990197
3958/3958 [==============================] - 29s 7ms/step - loss: 0
---------- test_eval_metrics:
[0.0037745086010545492, 0.9962254762649536, 175.0, 303.0, 108485.0,
Test loss:
0.0037745086010545492
Test accuracy:
0.9962254762649536
```

Figure 10: Confusion matrix for Test Run 2.2

To get even better accuracy with this model, I think that the data needs to be augmented even further. And possibly also need to impute better input data values by using either some reliable algorithm or by utilizing ML techniques also in the data pre-processing step.

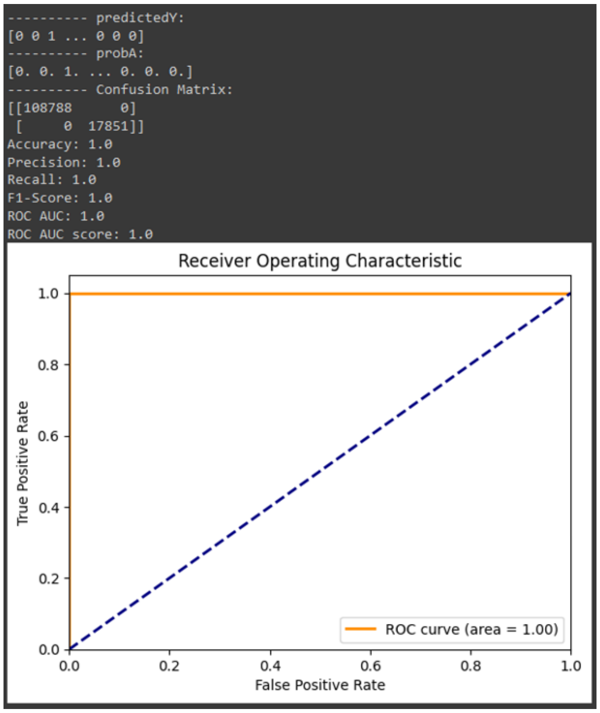
4.2.3 Evaluation of the Random Forest

With RF I get a perfect model with 100% accuracy. And I get the same result in all my test runs, whether I do or don’t include any data augmentation and whether I do or don’t do any oversampling using SMOTE.

For binary classifiers, there are also many other evaluation metrics, like:

- AUC refers to the Area Under the Curve of a Receiver Operating Characteristic curve (ROC-AUC). This metric is equal to the probability that a classifier will rank a random positive sample higher than a random negative sample.
- AUPRC refers to Area Under the Curve of the Precision-Recall Curve. This metric compute precision-recall pairs for different probability thresholds.

For the RF I also demonstrate how a ROC curve can be plotted. In this particular case it will be an ideal “curve”, more looking like an edge of a triangle (the orange-colored line).



```
---------- predictedY:
[0 0 1 ... 0 0 0]
---------- probA:
[0. 0. 1. ... 0. 0. 0.]
---------- Confusion Matrix:
[[108788      0]
 [     0  17851]]
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1-Score: 1.0
ROC AUC: 1.0
ROC AUC score: 1.0
```

Figure 11: ROC curve for perfect model

5 Discussion

For the Perceptron model:
Using a kind of crude data augmentation of the input dataframe, by appending several copies of it, constructs a perfect model! Obviously size matters, so if the data is big enough (and has good quality) the model will have a good chance of providing good predictions. It was a bit surprising to me that it was possible to achieve a perfect model with a rather simple Perceptron model.

In the imputing data step, I could have used a more elaborate method to construct the new values for missing ‘isbuggy’ rows. Perhaps I could have looked at the ‘self’ and ‘revd’ column values to make a safer decision whether the ‘isbuggy’ should be set to a 0 or a 1. At least for two of the models in this study, it seems though that the current chosen imputation strategy works well.

Could the feature selection be further improved?
Most likely, it would have been possible to drop even more features, and the model would have worked at least as good, and probably even better. I could have set the correlation threshold to a lower value (like 0.75) to find even more redundant features to drop from the training input data. This would make the training process faster.

6. Conclusion

When making a model using ensembling method Random Forest classifier, it is not necessary to do any data augmentation and not necessary to do any oversampling (using SMOTE).

It seems that Random Forest (RF) is the most accurate and robust deep neural network, out of the 3 models that I have investigated. For well-selected features and cleaned data, it shows a perfect fit (no under- nor overfitting) and seems to also handle outliers pretty well.

REFERENCES

[1] McIntosh & Kamei, "Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction", New York, 2017: https://posl.ait.kyushu-u.ac.jp/~kamei/publications/McIntosh_TSE2017.pdf

[2] Nagappan & Ball, "Use of relative code churn measures to predict system defect density. Proceedings - 27th International Conference on Software Engineering", 2005: https://doi.org/10.1145/1062455.1062514

[3] Brownlee Jason, "SMOTE for Imbalanced Classification with Python", 2021: https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/

[4] The imbalanced-learn library: https://github.com/scikit-learn-contrib/imbalanced-learn

[5] Nielsen Michael A., "Neural Networks and Deep Learning", Determination Press, 2015: http://neuralnetworksanddeeplearning.com/chap3.html

[6] The Keras API documentation, "Dense layer": https://keras.io/api/layers/core_layers/dense/

[7] Brownlee Jason, "Dropout Regularization in Deep Learning Models with Keras", 2022 : https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/

[8] Luiz Doleron, "Never use restore_best_weights=True with EarlyStopping", AI Mind, 2023: https://pub.aimind.so/never-use-restore-best-weights-true-with-earlystopping-754ba5f9b0c6

[9] The Keras API documentation, "ModelCheckpoint": https://keras.io/api/callbacks/model_checkpoint/

[10] Google for Developers, "Decision Forests", 2022: https://developers.google.com/machine-learning/decision-forests/intro-to-decision-forests-real, licensed under the Creative Commons Attribution 4.0 License (https://creativecommons.org/licenses/by/4.0/)

[11] Google for Developers, "Intro to Decision Forests", 2022: https://developers.google.com/machine-learning/decision-forests/intro-to-decision-forests, licensed under the Creative Commons Attribution 4.0 License (https://creativecommons.org/licenses/by/4.0/)

[12] Google for Developers, "Random Forests", 2024: https://developers.google.com/machine-learning/decision-forests/random-forests, licensed under the Creative Commons Attribution 4.0 License (https://creativecommons.org/licenses/by/4.0/)

[13] Chanathip Pornprasit & Chakkrit Tantithamthavorn, "JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction", Monash University, Melbourne, Australia, 2021: https://arxiv.org/pdf/2103.07068.pdf

[14] Scikit-Learn API Reference, "RandomForestClassifier", https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[15] Koehrsen Will, "How to Visualize a Decision Tree from a Random Forest in Python using Scikit-Learn", Cortex Sustainability Intelligence, 2018: https://towardsdatascience.com/how-to-visualize-a-decision-tree-from-a-random-forest-in-python-using-scikit-learn-38ad2d75f21c

Source Code Link References

The interactive Python notebook files (*.ipynb) have been implemented and run in Google Colab. Each notebook file should have a link to Google Colab, where it can be run in a Python 3 environment in the cloud.

[GITHUB] Link to GitHub repository: https://github.com/PerttiP/ML-Project-in-Automated-Software-Engineering

[COLAB] The Google Colab cloud blob storage is available from the link: https://colab.research.google.com/github/PerttiP/ML-Project-in-Automated-Software-Engineering/blob/main

Copyright notice: