MGL869 Lab Report

Building a Bug Prediction Model for the Apache Hive project

Authors

Alex Hoang-Cao
Henri Aïdasso
Samah Kansab
Cylia Airouche

Date: November 17, 2022

1 Introduction

Bug prediction is very important to improve the code quality and solve issues. For this reason this aspect has become a relevant research problematic. In this lab of the MGL69 course, we were asked to develop a solution to predict bugs for HIVE, and this will be using JIRA (*Hive JIRA — issues.apache.org* n.d.) and GITHUB (*GitHub - apache/hive: Apache Hive — github.com* n.d.) as data sources. In this report, we will detail the different steps followed to build the bug prediction models. First, we will present the global solution architecture. Then, we will explain the different steps of this solution. Finally, we will explore and analyse the obtained results for each prediction model.

2 Solution Architecture

The goal of this work is to predict the files that contain bugs before realising any new version of the Hive project's code. To achieve this, we followed the architecture shown in figure 1. We started by extracting the relevant data from the sources that were indicated in the lab which are Jira (*Hive JIRA — issues.apache.org* n.d.) and GitHub (*GitHub - apache/hive: Apache Hive — github.com* n.d.). From each source, we were able to obtain a set of information related to the HIVE project. At this point, we have applied an interesting set of filters and joins to build the dataset. This later suffered from various skews such as impertinent fields, the imbalance of data groups, etc. For these reasons, the dataset has been preprocessed so that it is ready to be used as input (features) to machine learning models. Then, we have continued with the construction of the Linear Regression and Random Forests models. For each model, we measured the prediction performances as well as analysing the obtained results. In the following, we will detail each step that we have followed in this work.

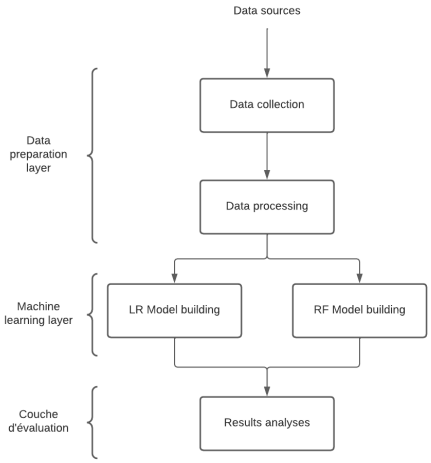

Figure 1: Solution Architecture Diagram

3 Solution steps

3.1 Data collection pipeline

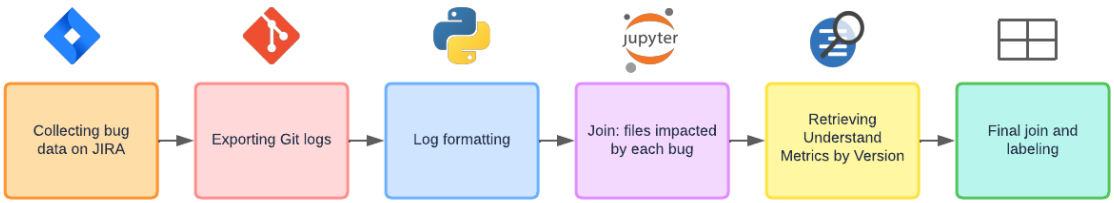

Figure 2: Data Collection Pipeline Diagram

In order to get the necessary data, we went through a set of steps represented by the pipeline in figure 2. For our implementation, we used multiples Jupyter notebooks and the Python programming languages. For the collection of metrics we used the Understand (*Understand by SciTools — scitools.com* n.d.) application.
Each step of our pipeline serves the following:

1. **Collecting bug data on Jira:** In this step, we have retrieved the list of the bugs from the Jira web application of the Hive project (*Hive JIRA — issues.apache.org* n.d.). The exported csv file contains all the bugs (identified by column *Issue key*) that impacted any final version from 2.0.0 to 3.1.3 (columns *Affects Version/s*). We deleted afterwards all the remaining data about the bugs (summary, reporter, etc.) that we considered non important for our work.

2. **Exporting git logs:** In order to have the files modification history to identify those modified to solve each bug, we needed to get the git logs. To do so, we cloned the public Hive repository from Github (*GitHub - apache/hive: Apache Hive — github.com* n.d.). We then executed the *git log* command with the options *–name-only –pretty=oneline* to get for each commit, its message which starts with the *Issue key* and the list of modified files for that commit in the Hive project. Finally, we saved the command's output to a file.

3. **Logs formatting:** The saved git logs were in a particular format: a line of commit message followed by multiple lines of impacted files. We had to parse this logs data to have a tabular csv format were for each row, we had *Issue key* and one impacted file. Through this operation, we were able to get in a suitable format, the files impacted by the issues (not only bugs) in the project.

4. **Join files impacted by each bug:** In this step, we joined the results obtained in the previous steps. We joined every bug to the files that its resolution impacted. Each row in the resultant dataset, represents an impacted file for a bug. We finally deleted bugs that did not impact any file or duplicated bugs.

5. **Retrieving metrics:** To get the important metrics that will allow us to build a prediction model, we have used the Understand tool (*Understand by SciTools — scitools.com* n.d.). We have retrieved all the metrics required in the lab. For each metrics we have added the related version.

6. **Final join and labelling:** Arriving to this stage, we assigned each bug to the appropriate set of metrics and by doing so we have labelled the dataset (file contains bug put 1 else put 0).

3.2 Models building pipeline

After completing the data collection, it is time to build the prediction models. First, we tried to explore the obtained dataset and ensure that it is ready for training then build LR and RF models.

1. **General dataset analysis:** Before starting the data processing, we have analysed the dataset. We noticed that the variables extracted directly using Understand are very correlated. Therefore, we replaced some variables with their ratios (feature normalization).

2. **Correlation analysis:** We computed and plotted the correlation matrix and the similarity dendrograms presented in the figures 3, 5 and 4 respectively. The correlation matrix shows that the dependency in the dataset is variant. All the correlations are almost less then 0.75. But we can noticed also that there are variables which are highly correlated. To have more precise analysis, we used a hierarchical clustering to group the variables by dissimilarities. We also added a limit line to see dissimilarities between variables when it is less then 0.3. A dendrogram helped us to visualise the results in the figure 5. In the case of the dissimilarities under the limit line, we can say that the variables are highly correlated. To cope with this, we have chosen one variable from each group of similar variables. Then we checked the dissimilarities of the new set of variables. The dendrogram in the figure 4 shows that all the dissimilarities are above 0.7. This means that the high correlation has been reduced in the dataset.

3. **Data splitting:** To build and validate the models, we needed to split our data into training and testing sets. We chose to split the dataset as follows: 80% for the training and 20% for the testing. To ensure that the train/test splits contain the same variation of data, we have calculated the bug to no bug ratios which were of 1.30% and 1.33% for train and test sets respectively. We noticed that the ratios are close, which means that the data splitting were good.

4. **Hyper-parameters tuning:** We have tuned the hyper-parameters for both the Logistic Regression and Random Forest models using 10-Fold Cross Validation method and selected best parameters based on the AUC performance. Then, we computed the models performance metrics (AUC, Precision, Recall) using the selected best parameters.

5. **Data re-balancing:** As shown in the figure 6, there is more than 70000 files which do not contain a bug and less than 5000 files which contain a bug. This unbalanced distribution of data reduce the quality of the model. We needed to re-balance the training data. We have used *SMOTE* and *RandomUnderSampler* algorithms from python *imblearn* package for re-balancing. The methods were chosen according to the results obtained by Tantithamthavorn, Hassan, and Matsumoto (2018).

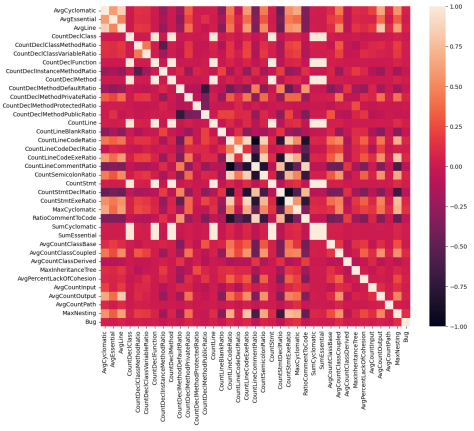

Figure 3: Correlation matrix

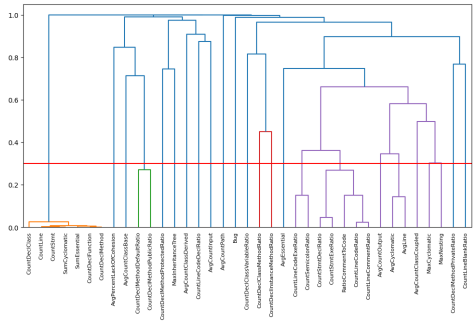

Figure 4: Dissimilarity dendrogram (subset A)

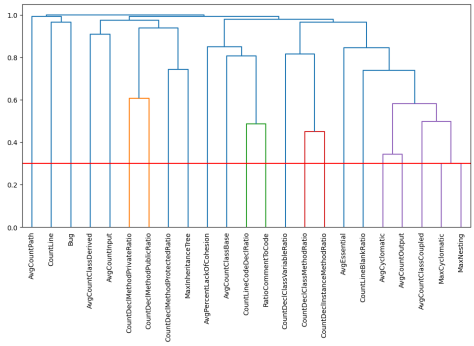

Figure 5: Dissimilarity dendrogram (subset B)

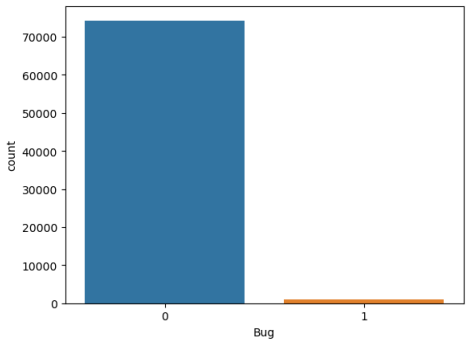

Figure 6: Bug repartition bar diagramm

	Model	ROC-AUC	Precision	Recall	TT (sec)
0	Logistic Regression (baseline)	0.4998	0.0000	0.0000	5.6156
1	Logistic Regression (SMOTE)	0.7729	0.0369	0.7898	3.3918
2	Logistic Regression (RandomUnderSampler)	0.7577	0.0348	0.7670	0.2151

Figure 7: Performances for LR model

	Model	ROC-AUC	Precision	Recall	TT (sec)
0	Random Forest (baseline)	0.5396	0.7000	0.0795	59.7577
1	Random Forest (SMOTE)	0.7759	0.0777	0.6420	236.2624
2	Random Forest (RandomUnderSampler)	0.8097	0.0446	0.8295	4.7211

Figure 8: Performances for RF model

3.3 Models Evaluation

The figures 7 and 8 represents the performances obtained for Logistic Regression (LR) and Random Forest (RF) models respectively. For the LR models, we noticed that the model with SMOTE re-

balancing performs better then the other models, with an AUC of 0.77 and a recall of 0.79. For the RF models, we notice that using RandomUnderSampler technique leads to better results than SMOTE and the baseline models. The AUC of this model is 0.81 and the recall is 0.83. We also noticed that the precision is generally really low (less then 0.05) for all the models leading us to the conclusion that the models commit a lot of false alarms predictions.

3.4 Model Interpretation

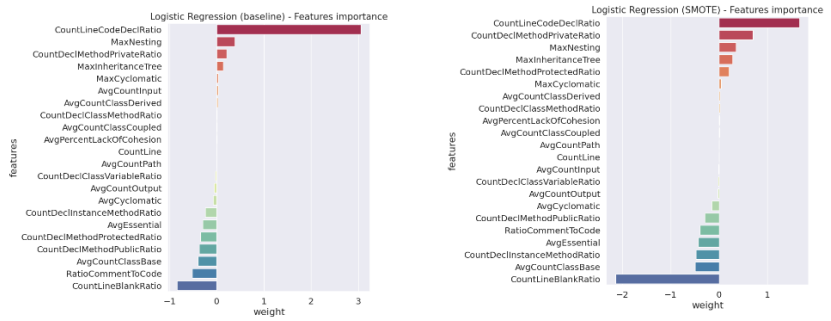

Figure 9: Feature importances for LR model

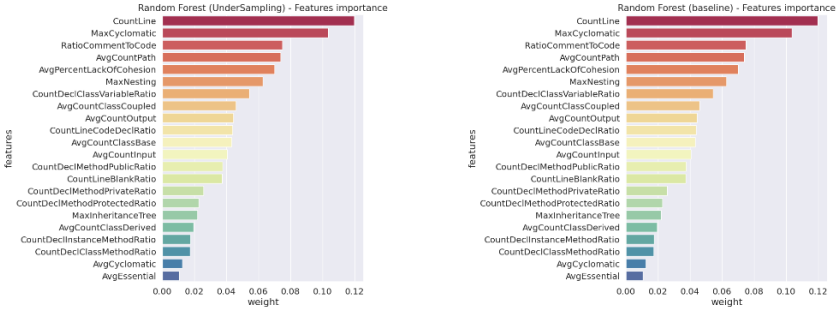

Figure 10: Feature importances for RF model

In order to interpret the obtained results, we used the feature importances diagrams and the nomogram (built using a R script) for Logistic Regression in figures 9, 10 and 11. Those plots helped us to find an explanation for the impact of the variables on the bug existence probability. Through our analysis we could find the following points:

1. **CountLineCodeDecl:** It represents the ratio of the declarative code in a class. We found that the augmentation of this value increases the probability of having a bug. To avoid this problem, developers must pay special attention to the declaration of dependencies and initialization of variables.

2. **CodeDeclMethodPrivate:** It represents the number of private methods in class. This metric impact positively the probability for a bug to exist in a file, contrary to the number of public and protected methods. This finding suggests that maybe developers do not put particularly effort in testing private methods while it is necessary. Therefore, we suggest to developers build robust tests for private methods and create them with caution.

3. **MaxNesting:** It represents the complexity of if/else blocs. It also has a positive impact for bug. We invite developers to manage complex choices with *switch case*, avoid overly complex nesting and handle exceptions with try/catch blocks.

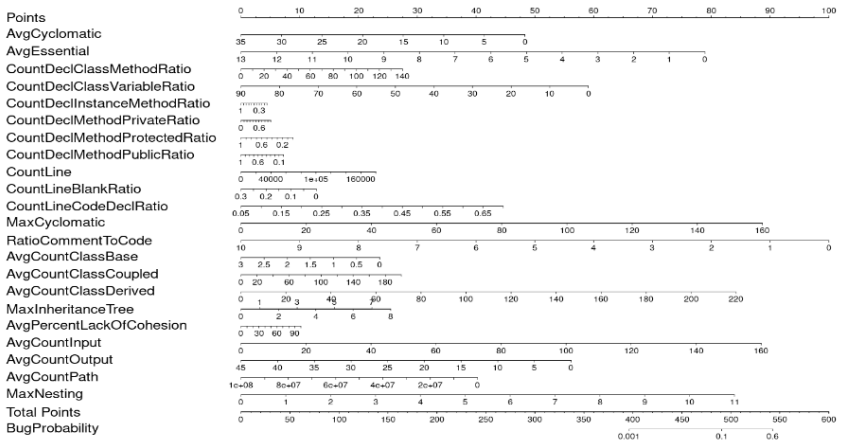

Figure 11: Logistic regression Nomogram

4. **MaxInheritanceTree:** This metric also impact positively the probability. So, we must avoid a large depth of the inheritance tree: the more dependency there is, the more the code is complex, and the more there is the possibility of having a bug. In other words, we need to apply the principle of segregation: separate responsibilities.

5. **CountLineBlankRatio:** We have noticed that the blank in the code has a negative impact on the bug existence probability. We invite developers to avoid writing compact blocks of code and produce a readable code.

6. **RatioCommentToCode:** Through this study we can say that the comments are very important to help others developers to understand the code that they did not write. So, we invite developers to always provide a good documentation to their code.

4 Model limits

From figure 12, we noticed that the models' performances vary from a version to another. The most remarkable versions models are 2.3.0 where the model has at the same time the highests AUC and Recall but the lowest precision, and the 2.1.0 which has the best precision and really good AUC and Recall. We can also note that all the performances degraded from 2.1.0 to 2.2.0 (to 2.3.0 for the Precision) and from 3.0.0 to 3.1.0. On the other hand, there were improvements from 2.0.0 to 2.1.0. Finally, while Recall and AUC are overall high (more than 0.7), Precision is really low for all the models (less than 0.3) suggesting that all models on that dataset generally commit the False Positive error (i.e. predict that there is a bug while there is not).
Moving to features importance between 2.0.0 and 3.0.0 versions, we noticed (figure 13) that the feature importances are not the same between the versions. We therefore calculated the ratio of the differences between versions of each feature. We have noticed that some complexity features become more important in the newer versions. We then assumed that from a version to the other, the code become bigger and more complex. For the other features, the ratios are almost stable. We have also noticed that the dataset examples do not vary sufficiently: the same (non buggy) files do not change much from a version to another.

5 Conclusion

In this work, we built a model for bug prediction at files level on the Apache Hive project (*ahenrij/ets-mgl869-hive: Bugs prediction in Apache Hive project* n.d.). We have obtained satisfactory results for both LR and RF models. We have also analysed those results and cited some actions to take in to consideration when coding. Then, we have studied the limit of our work, we can say that the prediction

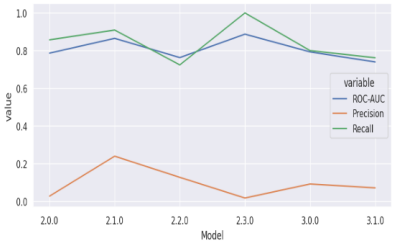

Figure 12: Performance variation between the different versions

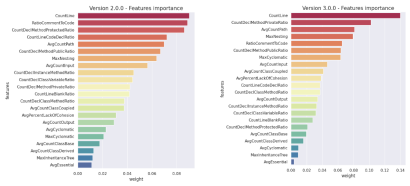

Figure 13: Feature importance comparison between 2.0.0 and 3.0.0 versions

model cannot be the same for all the future versions of hive.
In the future, we study for interpretability of the RF model using SHAP values, to have more general results. Also, we can test more re-balancing methods and improve the hyper parameters selection process.

References

ahenrij/ets-mgl869-hive: Bugs prediction in Apache Hive project (n.d.). https://github.com/ahenrij/ets-mgl869-hive. [Accessed 17-Nov-2022].

GitHub - apache/hive: Apache Hive — github.com (n.d.). https://github.com/apache/hive. [Accessed 16-Nov-2022].

Hive JIRA — issues.apache.org (n.d.). https://issues.apache.org/jira/projects/HIVE/issues/HIVE-13282?filter=allopenissues. [Accessed 05-Nov-2022].

Tantithamthavorn, Chakkrit, Ahmed E Hassan, and Kenichi Matsumoto (2018). "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models". In: *IEEE Transactions on Software Engineering* 46.11, pp. 1200–1219.

Understand by SciTools — scitools.com (n.d.). https://www.scitools.com/student. [Accessed 16-Nov-2022].