Machine learning pipeline for flaky tests classification

A. Trovato[a], A. Alfetra[a], F. M. Rastelli[a]

[a] *CS department, Univesity of Salerno.*

May, 2022

Abstract

Our work is based on the identification of flaky tests, following what is described in the paper ”Flake-Flagger: Predicting Flakiness Without Rerunning Tests” by Alshmari et al. What we achieved is the construction of a feature engineering pipeline that could best support the classification algorithms for the problem under consideration.

1. Context of the project

Regression testing is widely used in quality assurance to determine if recent changes to a codebase introduce errors. When a test fails, developers typically expect that this failure represents a bug recently introduced. However, a growing and concerning trend is regression tests failures which are not in fact due to recent changes, but are instead flaky tests.

1.1. Problem Description

Flaky tests are non-deterministic tests which pass and fail when run on the exact same version of a codebase. When tests alternate between passing and failing without any code changes, developers can be frustrated: flaky tests are challenging to debug, and a single failing test can halt release cycles. Hence, automatically determining if a test is flaky or not has become a significant topic in recent software testing research. Existing techniques for detecting flaky tests rely on rerunning tests: if we can witness two different outcomes (passing and failing) from the same test on the same version of the codebase, then surely that test is flaky. However, rerunning tests can be quite expensive. Developers have reported re-running suspected flaky tests up to 1,000 times to increase the likelihood of determining if a test is flaky.

Analternative approach to detect flaky tests is to create a machine learning classifier that can distinguish between flaky and non-flaky tests. Following this approach, developers could use this classifier to determine which tests are more likely to be flaky, and focus computational resources on rerunning those tests first.

2. Goals of the project

This project aims to achieve two objectives: the development of an artificial intelligence system capable of distinguish between flaky and non-flaky tests and the insertion of a such intelligent system inside of an automation pipeline.

Starting from dataset extracted from Alshammari’s et al. work on flaky tests classification [3], we will try to build a classifier whose performance are better, or at least comparable to, the Alshammari’s et al one.

3. Methodological steps conducted to address the goals

Our work, as a data-driven approach to solve a problem, starts with understanding of the data made available by Alshammari et al.

3.1. Data undestanding and preprocessing

Each row of the dataset contains information of a specific test-case (eg. the file name, total line of code, whether the test case is flacky or not and etc). The dataset cosists of 12921 test case which the majority are non flaky (1229/623).

Among the rows it’s possible to encounter *set-up* or *tear-down* tests (eg. test connection to DB). A test case belong respectively to one of these categories if its `TestCase` property contains the substring ’setUp’ or ’tearDown’. Since *set-up* and *tear-down* methods are unlikely to contain flaky code sections, we took the chance to delete the rows associated to those tests.

The dataset contains no `null` value, then it was not involved in data cleaning operations beside the deletion of those columns which a machine learner should not infer any conclusion from. The following columns were removed:

column	corresponding to
`TestCase`	test file name
`Id`	test id
`NameProject`	project name

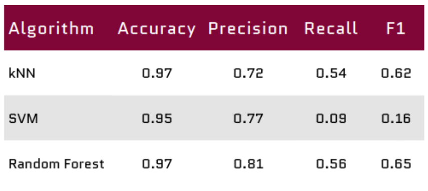
Table 1: Removed features and their semantic values.

The following sub-sections describe the algorithm selection process and the performed experiments regarding the feature engeneering pipeline.

3.2. Algorithm selection

The dataset cleaned in the previous step, was partitioned in train and test set following the 80/20 criteria with a stratified sampling (in order to preserve the flaky/non flaky ratio between the two sets).

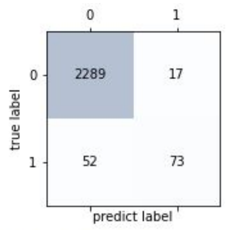
The train set was used to compare different classification algorithms: *kNN*, *SVM* and *RF*. For each algorithm we performed a *stratified k-fold cross-validation* (10x3) in order to obtain the real performance and the best configuration of the hyperparameters. For each fold, each algorithm has been evaluated in terms of metrics extracted from the confusion matrix: *accuracy*, *precision*, *recall* and *F1*.



Algorithm	Accuracy	Precision	Recall	F1
kNN	0.97	0.72	0.54	0.62
SVM	0.95	0.77	0.09	0.16
Random Forest	0.97	0.81	0.56	0.65

Figure 1: Averaged results from testing algorithms with k-fold.

Once established the most performing algorithm for our data (Random Forest [criterion=entropy, n_estimators=150]), its performance was validated against the initial test set.



```
Accuracy: 0.972
Precision: 0.811
Recall: 0.584
F1: 0.679
```

Figure 2: Validation Random Forest.

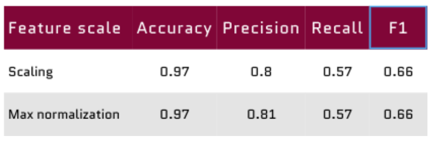
Set the classification algorithm, we reached a starting point for our experiments on feature engineering: feature scaling, feature construct, feature selection and data balancing. These are discussed in the next sub-section.

3.3. Feature engineering pipeline

For our experiments on feature engineering, we followed this strategy: after applying a modification on a copy of the dataset (partitioned exactly as before in train and test set), we trained and validated the performances using a *stratified k-fold cross-validation.*

If the application of one these modifications turned out to improve the algorithm performances, then the modification at issue would have been inserted in the pipeline.

For what concerns the **feature scaling** step, we tried two different approaches: z-score normalization (scaling) and min-max normalization.



Feature scale	Accuracy	Precision	Recall	F1
Scaling	0.97	0.8	0.57	0.66
Max normalization	0.97	0.81	0.57	0.66

Figure 3: Feature scaling results validated

Proved that min-max normalization lead to an improvement, it was selected as feature scaling method inside the pipeline.

For what concerns the **feature construct** step, we tested the Principal Components Analyses. However, to perform PCA, it was necessary to know how many components we wanted to obtain, so a graph was generated showing us the cumulative variance : number component ratio.

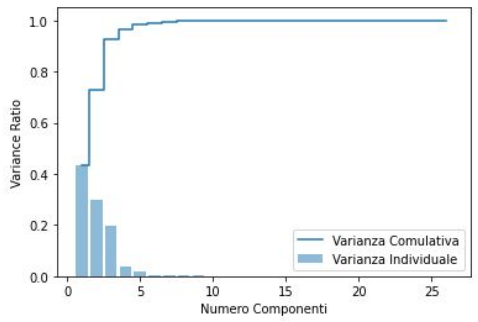


Figure 4: PCA Component Analysis

The graph showed that by using only 5 components we were able to maintain 98% cumulative variance. However, the use of PCA for our problem caused a decrease in performance for the model used, so although we were able to reduce the complexity of the problem, we felt it appropriate to exclude it from the pre-processing phase.

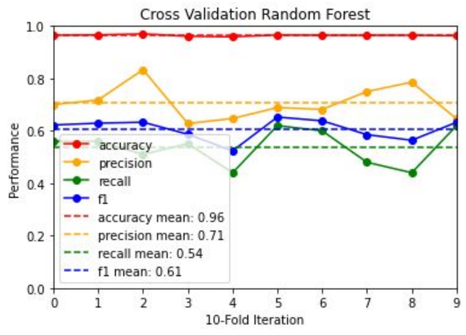
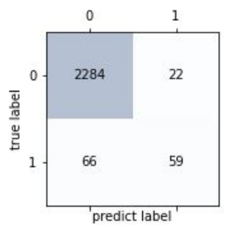


Figure 5: Cross Validation Random Forest on train set



```
Accuracy: 0.964
Precision: 0.728
Recall: 0.472
F1: 0.573
```

Figure 6: Validation Random Forest on test set

For what concerns the **feature selection** step, we experimented with the random forest algorithm. Using this algorithm we were able to check the importance of each feature for our problem, so that we could filter out the less relevant features.

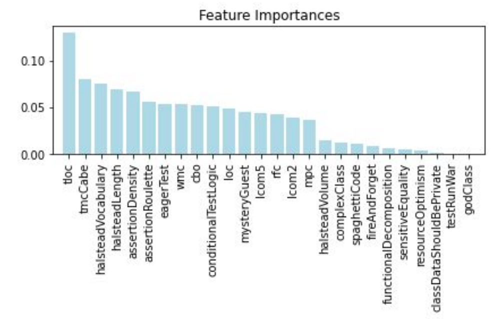


Figure 7: Feature importance

ased on the results generated by the random forest, it was decided to filter out all features whose Gini index was greater than 0.02. The removal of this feature generated a slight increase in model recall, so it is considered appropriate to include it in the pre-processing phase.

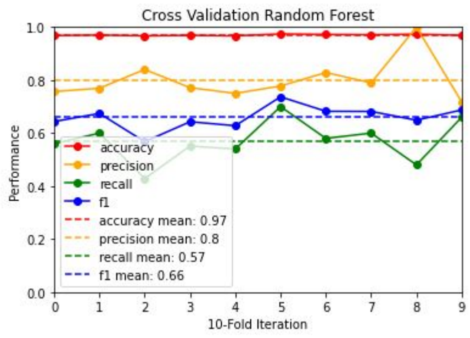
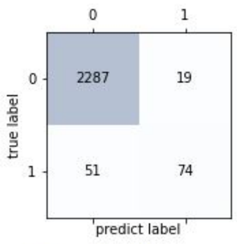


Figure 8: CV Random Forest on train set



```
Accuracy: 0.971
Precision: 0.796
Recall: 0.592
F1: 0.679
```

Figure 9: Validation Random Forest on train set

Regarding the **data balancing** phase, we experimented with SMOTE with the following configuration [sampling_strategy=auto, k_neighbors=3, random_state=42].

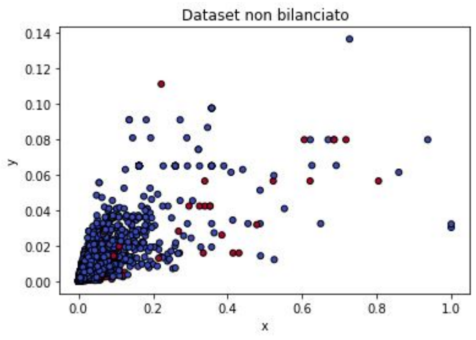


Figure 10: Before SMOTE

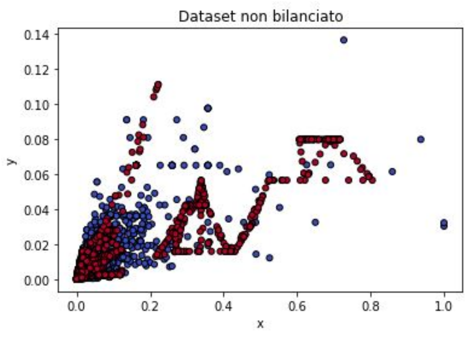


Figure 11: After SMOTE

When validating the model we discovered a degradation of the precision (-10%) followed by an increase of the recall (+5%). Since an increase of the recall can be translated as an increase of the number of flaky tests correctly detected, SMOTE data balancing was inserted in the pipeline.

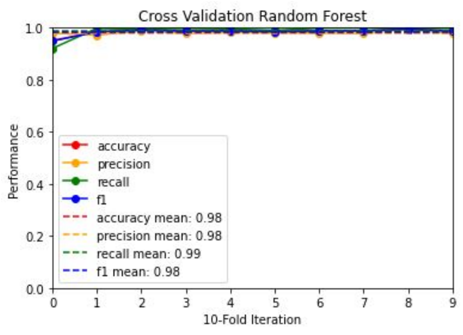
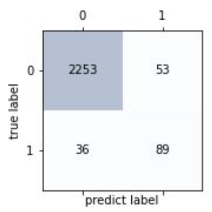


Figure 12: CV Random Forest on train set

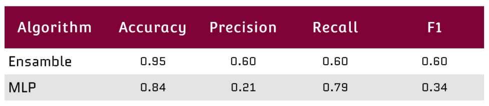


```
Accuracy: 0.963
Precision: 0.627
Recall: 0.712
F1: 0.667
```

Figure 13: Validation Random Forest on test set

3.4. Advanced Methods

The performance achieved with the random forest can be considered very good as we were able to increase the recall by 70%. However, we checked whether better performance could be achieved by using the ensamble between SVM,KNN,RF or with Multilayer perceptron. But in both cases, the performance is inferior.



Algorithm	Accuracy	Precision	Recall	F1
Ensamble	0.95	0.60	0.60	0.60
MLP	0.84	0.21	0.79	0.34

Figure 14: Averaged results

It is interesting to note that with the use of MLP, the starting situation is reversed, we are able to classify flaky tests better than non-flaky tests. A possible explanation for this lies in SMOTE, however we do not consider it appropriate to go into this in more detail as the results obtained with random forest turn out to be to our liking.

3.5. Model testing

Model validation can be seen as a kind of white-box testing, however, during the course we talked about how it is also possible to perform a kind of black-box testing using metamorphic tests. We also tried to introduce such tests into our pipeline. Not with the aim of testing the predictions of the model but to test how the fit of the model responded to different mutations on the training data. The tests performed do not exhaustively check all possible mutations that can be performed on the dataset. However, we still manage to get a general idea about the goodness of our pipeline.

- feature permutation
  - Input: Dataset
  - Mutation: column permutation
  - Metamorphic Relation: The same performans as the model trained on the unmodified dataset
- redundant row addition
  - Input: Dataset
  - Mutation: Adding redundant rows
  - Metamorphic Relation: The same performans as the model trained on
- random label editing
  - Input: Dataset
  - Mutation: Random label editing
  - Metamorphic Relation: Lower performance than the model trained on the original dataset

- random row deletion
  - Input: Dataset
  - Mutation: Random row delation
  - Metamorphic Relation: LPerformance less than or equal to the model trained on the original dataset

3.6. Model explainability

The **explainability** step was realized by employing a *global surrogate*. The global surrogate is an interpretable model, which is trained to approximate our model's performance.

The global surrogate - in our case, a single decision tree - was trained on the same data flowing out the data engineering pipeline, after that we've measured how well the surrogate replicated the model's predicitons. The measure typically used is the R^2, i.e., percentage of variance that is captured by the surrogate. For our global surrogate, R^2 stands between 0.5 and 0.6.

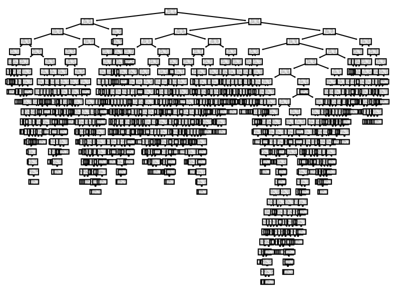
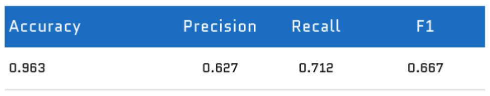


Figure 15: Surrogate

4. Results and implications

This section shows the results obtained by testing the classifier and summarizes the steps performed in the feature engineering pipeline.

After the experiments conducted, our final model shows the following performances:



Accuracy	Precision	Recall	F1
0.963	0.627	0.712	0.667

Figure 16: validated model's performance

Our model's performance are very similar to Alsham-mari's model et al. While our model has higher accuracy (63% vs 60%) their model can boast of higher recall (71% vs 74%).

Starting from the same classification algorithm, thanks to the experiments shown in the previous sections, we showed that by applying appropriate feature engineering operations it is possible to improve performance in the process of test-flaky identification. The described model is trained on the data flowing out the feature engineering pipeline consisting of the following steps:

step	technique/algorithm
Feature Scaling	min-Max normalization
Feature Selection	gini index ordering
Data Balancing	SMOTE

Table 2: algorithms selected for feature engineering steps.

5. Conclusion

Flaky tests make testing unreliable because it is hard to say which test failures represent true regressions. On the basis of FlakeFlagger we experimented with feature engineering techniques in order to achieve improvement on general performance and establish an optimal data engineering pipeline for the problem at hand. Gli sviluppatori sono incoraggiati a sperimentare all'intero di modelli per il riconoscimento di test flaky, le varie tecniche mostrate. The idea, proposed by the authors of the original paper, of expanding the dataset in order to be able to further improve the performance of flaky test classifiers remains relevant.

Data availability

All the code made is available at the following: github repository.