Predictive Test Selection - Red Hat Inc.

Alon Mannor, Rubi Arviv

Submitted as final project report for Projects with the industry workshop, IDC, 2021

1 Introduction

With ever increasing code repositories and tests needed to cover said code, the need for test prioritization schemes (i.e. producing a prioritized list of test to run, given a new code change) is becoming ever more essential. Various ways have been proposed to deterministically select which tests to run given a code change (for further details see the introduction section of the FB article, see below) .

A non-deterministic and promising approach is to use an ML-based model that will be trained on past code deployment runs, that contain code changes and the tests that ran as part of their deployment cycle. Given the historical data, such a model will be given a code change and a list of potential tests. It'll produce a prioritized list of the tests - from most likely to fail - to least likely.

We have partnered with Red Hat to implement the approach provided in the FB article. The project chosen is Red Hat OpenShift.

1.1 Related Works

This project is based on a Facebook article by Machalica et al (hereinafter "the FB article"). [1]

2 Solution

2.1 General approach

We plan to go over the historical output of the CI and the OpenShift's GitHub repository. From these 2 sources, we'll scrape the historical raw data (containing things like all the metadata of CI deployments, test results, historical changes made to the project's files and more). After having enough raw data, we'll run it through one or more stages of pre-processing and eventually use it to train an xgboost-based model. From there, we'll run both a classifier (outputting a binary fail/success prediction for a specific test) and a regressor (outputting a probability that a test will fail). In the latter approach, we'll use a cut-off value

to be the threshold from which we recommend to run the test for a given code change (e.g. all the tests that have at least 80% chance of failing).

There are many alternative approaches we tried, the main ones are:

In data scraping: The project's CI output identifies a test by it's "locator" (a string value which can be thought of as the macro command use to run the test in the CI system). The task of mapping between the locator and the file/s that hold the test's code was no small feat. This required many cycles of trial-and-error. Eventually we decided to implement a straight-forward approach of iterating through the output's xml nodes, finding the one that holds the locator and looking at the node's text to look for any URLs starting the project's GitHub repo address.

In data analysis: An important question we faced was whether to use a binary classifier or a regressor. The FB article used a regressor and 2 hyperparameters to calculate adhoc metrics (to asses how well the model performed). We implemented both a classifier and a regressor. We used off-the-shelf metrics (e.g. precision, recall, F1 score etc.). The classifier implementation produced results which weren't so good - so we implemented the regressor (with a final cutoff value of 0.7) - which improved the results.

In addition, there's a theoretical alternative we initially considered on how to get enough data. The idea was to deploy OpenShift on our own built environments and to introduce small changes (that'll break the code) ourselves in order to produce known results (i.e. that a lot of tests would fail). This was rejected for a host of reasons, e.g. that deploying such a system can be painstakingly long and tedious, the code changes we'll introduce will be very simplistic and unnatural (since we don't have knowledge in the domain of the code, we would have done things like breaking compilation in a inelegant ways). Furthermore, we saw that the real-world data is good enough, in both quantity and quality.

2.2 Data set

We scraped data from 2 sources:

1) The OpenShift GitHub repo, located at:

```
https://github.com/openshift/origin
```

2) The OpenShift CI dashboard, located at:

```
https://prow.ci.openshift.org/
```

We performed a pre-preprocessing step on these 2 (raw) data sources and produced the following 3 datasets:

i) Mapping changesets to tests: Each test is identified by it's locator and also has the result of the specific run (i.e. "success" or "fail" - other state like "skipped" are ignored).

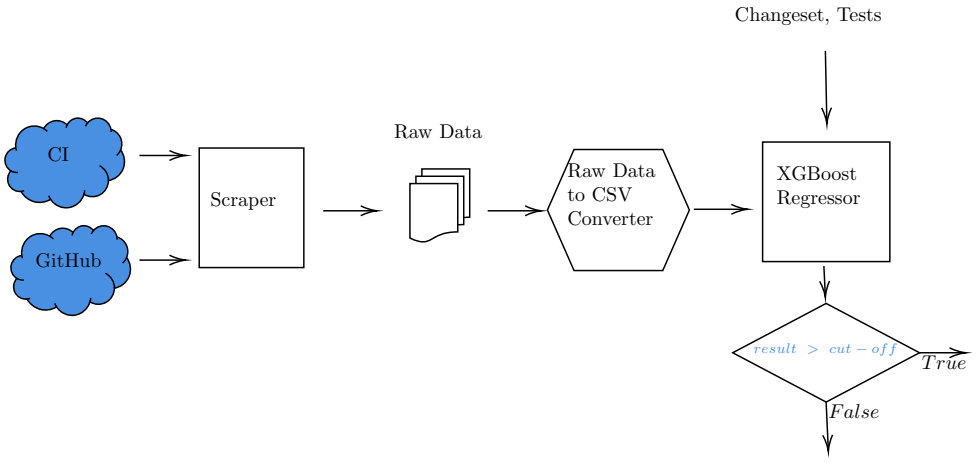

Figure 1: Predictive test selection pipeline design.

ii) Mapping test locators to path/s: As aforementioned, we used heuristics to deduce this mapping. We weren't successful in all of the times and some mapping are empty.

iii) Files changes history: We used GitHub's API to scrape the project's repo and created a list of all the project's files and their changes (i.e. commits) history.

2.3 Design

The design is as follows, currently there is a scraping process to collect raw data from CI (to collect code change to test name, and test locator data) and GitHub (to collect changes history). This raw data is then aggregated into a flattened json file. Since xgboost can't handle raw text, in this stage we also encoded these fields. The flattened json is then translated to a CSV file using a customable configuration file. The CSV file is the input for the XGBoost regressor. The prediction's input will be the changeset and the set of tests (each with either a 1 or 0, representing if the test has failed or succeeded, respectively). We will also include a cut-off parameter (a number between 0 and 1) that will decide whether the result is to to take the test or not with the current change that have been made. As been mentioned previously, the model that was selected is the xgboost regressor because the purpose is to decide whether a test should be selected for the changeset that have been made with a level of confidence (at the end we can't run all the tests, just the ones with a higher probability to fail). That is why the model returns a value between 0 and 1. A cut off parameter will decide whether a test has a high probability to be related to the change and we are not just interested just in returning *True* or *False*. This is why logistic regression was not used here. See Figure 1 for the predictive test selection pipeline design.

3 Experimental results

For the purpose of collecting results during the early data preparation, the data was split into 2 subsets: data before the last week and data in the last week. In that manner we were able to imitate the way it will be used in production. We were able to accomplish this using a parameter that is provided in the json configuration file. The results were collected for binary xgboost (for sanity) and for the regressor model. Since we ran our model on code that is very close to being deployed in production, almost all the tests succeeded. This lead to our data being highly imbalanced. The parameter we used to overcome the imbalanced data is a parameter in xgboost called *scale_pos_weight*. We set it to 9 (we reached it according to the recommendations in xgboost documentations). Also the *cut-off* parameter was set to the value 0.7. The way that the table is arranged is by collecting all the performance evaluation metrics to understand whether we are in a good direction. We collected:

- *precision* - how correct are our model results
- *recall* - the percentage of positive results the model caught
- *f1-score* - percentage of positive results are correct
- *support* - number of actual occurrences in the data-set.

3.1 Binary Model Results.

As we can see the result of the binary model 3.1 are not good enough, because of 2 reasons:

- The imbalance in the data between *positive* (1) and *negative* (0) results.
- The binary cut-off which is not flexible.

	precision	**recall**	**f1-score**	**support**
0	0.95	0.98	0.96	41896
1	0.32	0.16	0.22	2642
accuracy			0.93	44538
macro avg	0.63	0.57	0.59	44538
weighted avg	0.91	0.93	0.92	44538

3.2 Regressor Model Results.

In the xgboost regressor, we got more accurate results (since we used the cut-off parameter to decide whether the result is *positive* (1) or *negative* (0)). The accuracy has increased to 77% as opposed to 32% in the binary model which is a huge improvement. Again the results are not perfect but show that choosing

the regressor over the classifier leads us in a good direction. Furthermore, it's worth reminding that our focus was on setting a pipeline to gather data and process it until it will reach the model learner.

	precision	recall	f1-score	support
0	0.96	0.99	0.98	41896
1	0.77	0.33	0.47	2642
accuracy			0.95	44538
macro avg	0.86	0.66	0.72	44538
weighted avg	0.95	0.95	0.95	44538

4 Discussion + What we have learned

This project has been an enjoyable and learning experience. Some of the major takeaways for us (and hopefully for the readers as well :)) are:

- Old straightforward deterministic ways to preform test selection in CI are not good enough. A better approach is needed. Our implementation of the xgboost (as suggested by Facebook) is one such way.
- It's no trivial task to implement this approach on any given project. It's evident that Facebook utilized specific characteristics of their repo when implementing their solution (e.g. an existing or easily-calculated mapping between a test and the file it resides in).
- One straightforward insight gained from the experiments is that the regressor approach (along with a respective cut-off) is preferable to the binary classifier one.
- Most of out effort (in both writing the code, running it etc.) was invested in fetching the data and organizing it correctly. This fact should be known and taken into consideration in similar future projects.

5 Potential future work

We see this project as a preliminary step in a direction that should be further explored. We believe there are important further steps to be done in this project (and in similar projects in the future). These include:

- Running a model that implements a wider part of the approach described in the FB article; Due to a (very) limited time-frame, we weren't able to recreate our version of the 2 functions of ScoreCutoff & CountCutoff (figures 6 & 7 in the FB article, respectively).

- For future extensions of this work, it's a good idea to include a more generic (i.e. implementation-independent) version of the features *Minimal distance* and *Common tokens*. There are software that can check code coverage (e.g. when running a test on a given code change). These are more generic and should be used instead or in addition to aforementioned non-generic features.

- Go neural! We followed in the footsteps of Facebook and ran xgboost but there's a very good reason to believe that neural network based approach will provide better results.

- Imbalanced data & business POV: The data we gathered is of code that's very mature (the tests ran just before the code is being dropped to production). Naturally, this entails very low failures (which matches what we saw in our data).

  This has 2 issues:

  - From a data scientist POV (that data is highly imbalanced and thus isn't ideal for training, to say the least).

  - From the business (stakeholders) POV: This means there's a good chance we could have used this approach earlier in the CI process - thus reducing the time it takes for bugs to surface. As any person in R&D knows, the earlier you catch a bug - the better.

  The way to deal with these issues is simple - take CI data from an earlier stage (let's say the first or second "round" of tests a developer runs on their code). This will provide a more balanced data and will cause potential bugs to surface earlier, thus reducing R&D time.

6 Code

All of resources (e.g. code, the data we used, the project's presentation, this report etc.) can be found in our GitHub repository at:

`https://github.com/Amannor/redhat_final_proj`

Be sure to check our README file for more details!

References

[1] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive test selection. In Helen Sharp and Mike Whalen, editors, *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 91–100. IEEE / ACM, 2019.