Prediction of Performance Regression

Team 8

Chun Yu Lee
cl5901@rit.edu

Yash Mahesh Bangera
yb7574@rit.edu

Austin Simmons
as7012@rit.edu

I Introduction

The performance of a software is critical. If you have a poor performance it can make the users have a bad experience, or even worse, your application may load or operate for a longer time to perform what you want it to achieve. It can also lead to reputational repercussions. Performance regression is always a big challenge for all developers. For example, after a commit the performance of the same function in the system may be worse than before. To prevent this from happening, a developer can run an overall testing suite before the commit. However, software testing is costly and time-consuming.

Determining the proper avenue of software testing is always a challenge with room for improvement. For instance, a team tries to maintain their performance by doing weekly testing. Within a week there are around twenty or more commits. When it comes to the result, it shows that there is a decline in the levels of performance because of some of these commits. Now they must find the root cause of this performance regression from these twenty or more commits at the same time. It is difficult to determine which commit is causing this issue and which commit is not. On the other hand, a team may try to run a test every commit so that they know the exact commits that are causing the regression. This method is extremely costly and proposes a significant waste in resources. Balancing between these two components is what will lead to an effective solution.

I.I Related Works

Several solutions to this problem have been implemented with varying degrees of success. Solutions range from testing on a static time schedule to using algorithms to predict if a commit needs to be fully tested or not. One of the keys to success for this project is finding an answer that has an optimal balance between accuracy and cost.

Most software is developed with the help of maintaining a test suite and a benchmark suite for exposing performance bugs. The benchmark suite needs to run every time a commit is made to the software repository. However, there is a problem with this method of development. The benchmark suite cannot be run after multiple commits because that can cancel out the effects of one commit with some other commit and not give the root cause of the problem. The solution here is to run the benchmark suite after every commit. This proves to be costly. The paper "Perphecy: Performance Regression Test Selection Made Simple but Effective" focuses on implementing a "light-weight, general and effective approach" [7] where the entire benchmark suite does not to be run every time a performance change is expected to happen. To do this, the paper makes use of Perphecy - a tool that reduces the use of benchmark suites by 83%. This tool allows the developers to detect performance changes earlier and more frequently.

The paper "PerfJIT: Test-level Just-in-time Prediction for Performance Regression Introducing Commits"[6] describes prior research studies on performance regression at the code commit level. These performance regressions are detected by running all of the readily available tests at that

time. This helps the developer to find the root-cause of the performance regression and address it immediately. However, running these tests every time is a time-consuming task. The paper also describes Perphecy in detail, explaining how the tool trains on previous thresholds and how this method, when used on the test level, leads to many false-positive results. Therefore, the paper proposes a method that should be invoked just-in-time which is a prediction model that can predict the tests that can lead to performance regressions with a given code commit. These tests are called "performance-regression-prone" tests. The model first identifies all the tests that affect performance regression in terms of resource utilization, I/O read and writes and the response time. The model then makes use of five classifiers, one for each of the aforementioned metrics to determine the performance regression. Now, developers know that they need to run only these performance-regression-prone tests to determine the performance regression. The developers can then carry out their investigation in these specific tests. These newly executed performance regression tests are then included in the prediction model's training data for future use. In this manner, the performance regressions can be determined timely and proactively by these developers. However, the performance of this prediction model is constrained when it comes down to dynamic problem detection like deadlocks.

The solution here is to build up a model to let the developers know where the performance regression could happen. This will help them narrow down the problems so they don't need to run all the tests. Thus, they can put more resources on the areas with higher potential of performance regression. The challenge is then finding the best way to run these checks. One potential solution is using Test Case Prioritization(TCP) [9], which is a method to select the most appropriate tests to execute and enable faster feedback. Through building up a model to predict the appropriate tests at the commit level, we can drastically reduce the time and resources spent.

II Study Design

The team's original plan was to work on an approach from scratch that includes feeding the original dataset as described in section VI.I into a bag of Machine Learning classifiers. This follows into the validation/testing phase with predictions as to whether a commit is causing performance regression or not. The future work includes working on Test Case Prioritization and Performance Regression Testing for optimization of the approach. This can be seen in Figure 1.

However, due to the poor performance results as shown in Figure 2, a curated dataset was made which is shown as DataSet_v2 in Figure 3. This made the team change their approach into the one as shown in Figure 3. The approach includes more work in the feature selection process because of the multitude of features. The reduced features were then passed into the KNN classifier because of its ability to outperform other classifiers as will be discussed under section III's Research Question 2. The dataset is first put into an oversampler which is taken directly from the imbalancedlearn library and is discussed in the section III.II . The section also talks about the sampling strategy that is being used for oversampling. The K Nearest Neighbors classification works with an unknown test instance placed in a set of multiple train instances as shown in Figure 3. The value of K is manually selected by the user which is an integer value. The team selected the value to be 10, so the random test instance is compared with the 10 nearest train data instances. The distance is calculated from this test instance to the K nearest neighbors, which can be done using one of the

following metrics:

- Euclidean distance:

$$d(x, y) = \sqrt{\sum_i (x_i - y_i)^2} \tag{1}$$

- Cosine Similarity:

$$similarity = cos(\theta) = \frac{A \cdot B}{||A||||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} {A_i}^2}\sqrt{\sum_{i=1}^{n} {B_i}^2}} \tag{2}$$

- Minkowski distance:

$$d(x, y) = (\sum_i (|x_i - y_i|)^p)^{\frac{1}{p}} \tag{3}$$

Once the train data instances are fitted, the test data instance is taken one instance at a time and the above distances are calculated with keeping in mind the K-nearest neighbors as explained before. Now, the class label for the majority times appearing neighbor is taken. For instance, in Figure 4, the circle in red is the unknown record and the plus symbols in red are the train data instances. If the k value is taken as 3, the 3 nearest neighbors represent the plus symbol having class. This is taken as the majority and a prediction is given out as either 0 or 1. However, in situations where there is a conflict, i.e., two class labels have equal weight in the number of nearest neighbors as shown in Figure 2. This concern can be mitigated using the majority vote rule where the weight factor is calculated as follows:

$$w = \frac{1}{d^2} \tag{4}$$

In this scenario, the highest weight-having class is taken as the majority and given out as either 0 or 1, which means whether the majority class is a hit or a dismiss.

Figure 1: Approach Overview Diagram

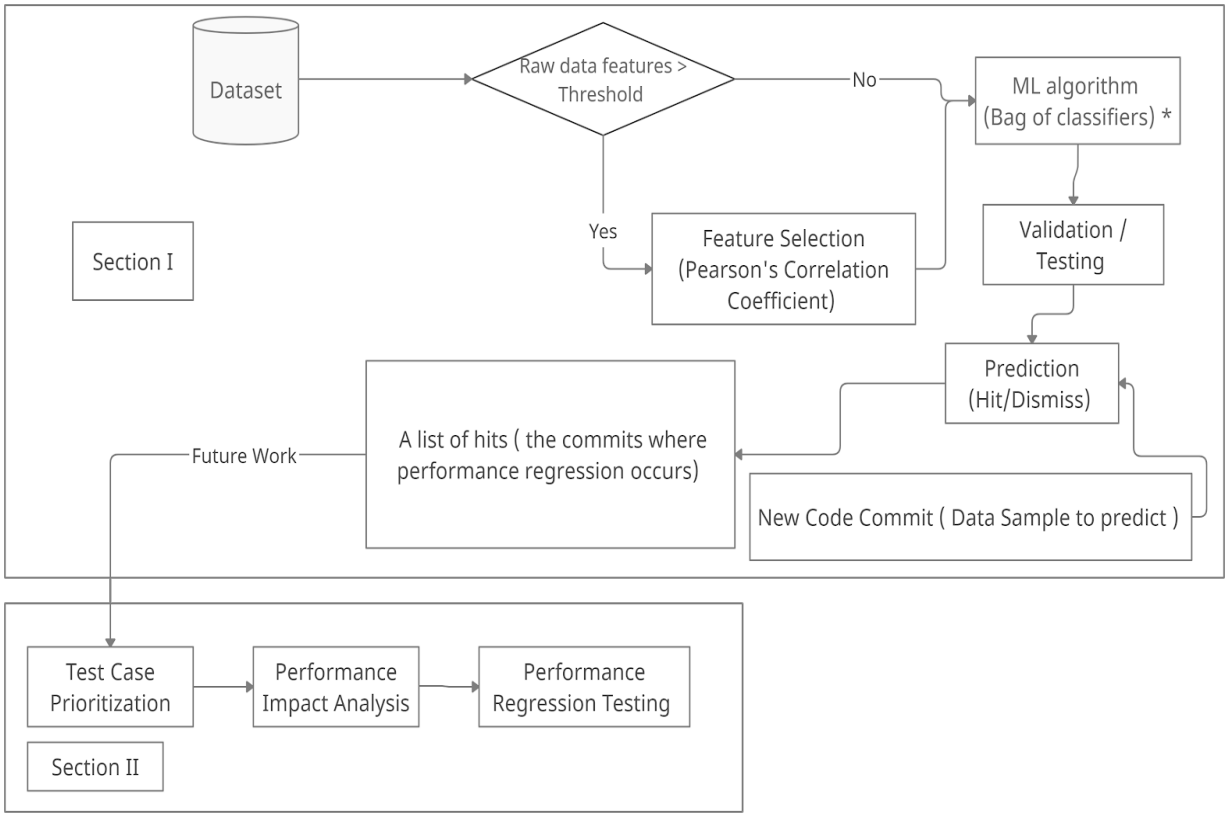


Figure 2: KNN Conflict Case When k = 4 [2]

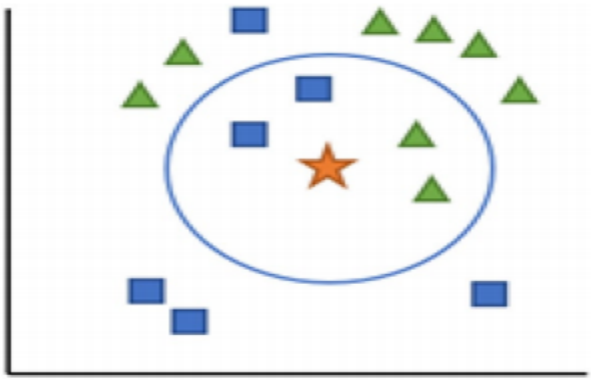


Figure 3: Whitebox KNN Solution

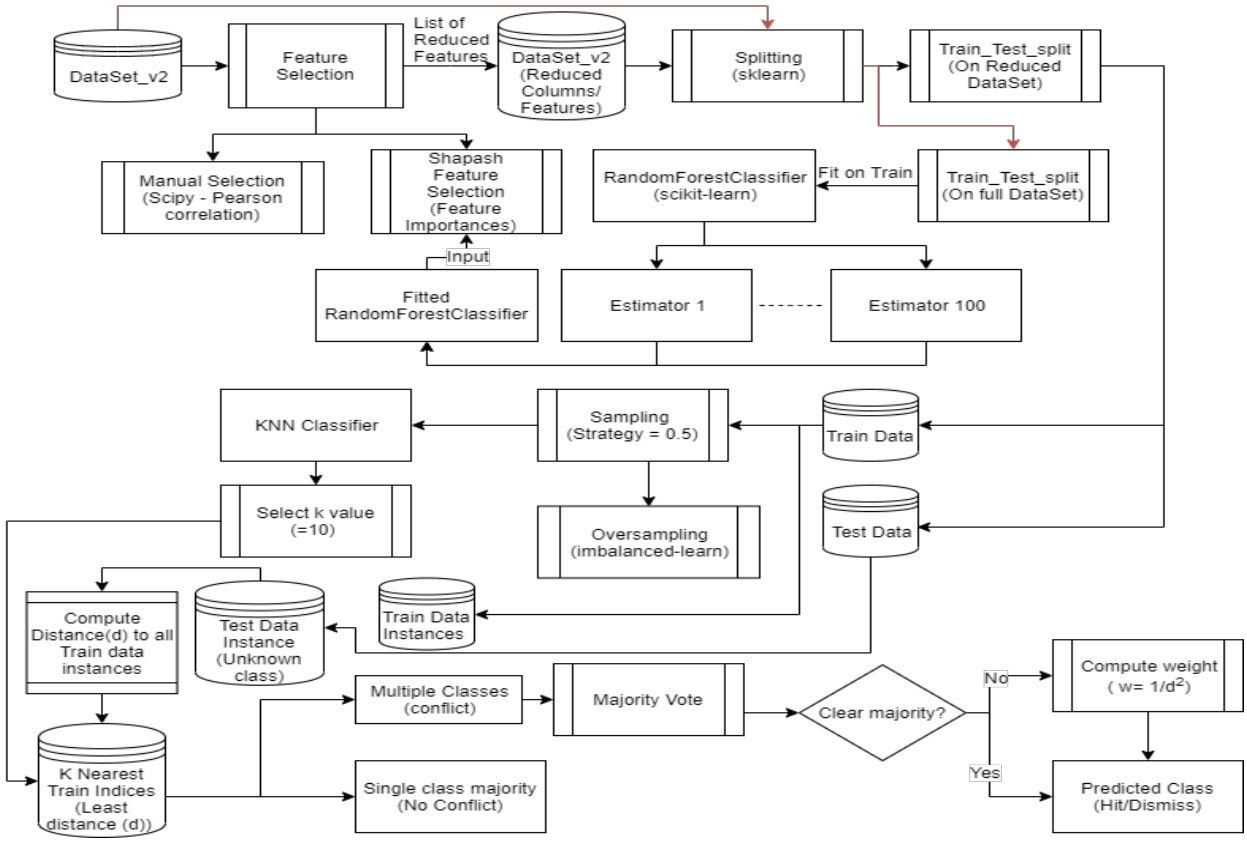
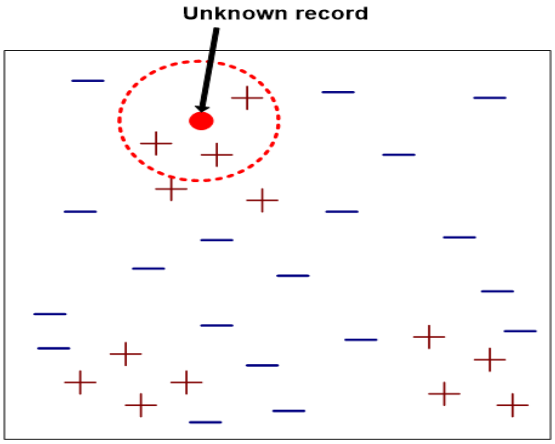


Figure 4: KNN Clear Majority When k = 3 [11]



III Experiment

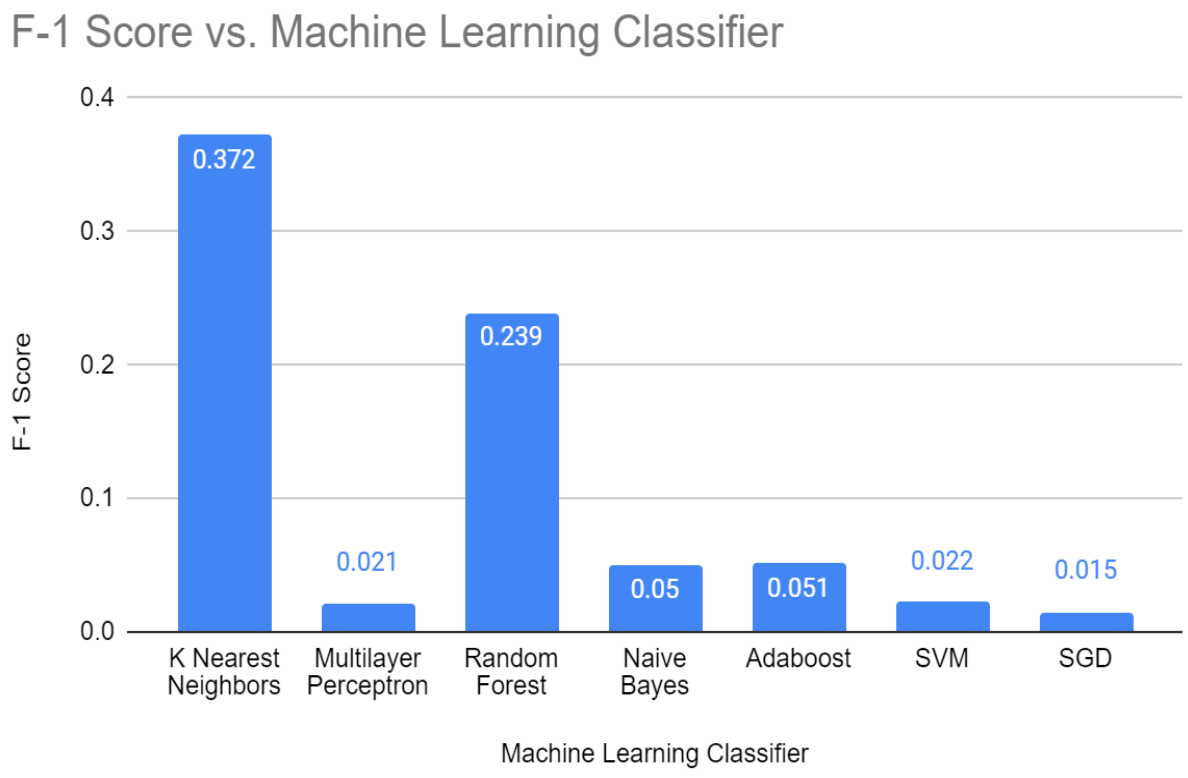
Choosing the appropriate ML algorithm is vital to the performance of the proposed final solution. The approach consists of testing various algorithms and comparing them to see which gives the best results. These result in the following research questions:

RQ1: Which algorithm to use for the classification of hit/dismiss for our given dataset?
Answer: The team carried out a comparative analysis between multiple Machine Learning classification algorithms, namely, the K-Nearest Neighbors classifier , Neural Network , Random Forest Classifier and Naive Bayes Classifier.

RQ2: How can we improve the accuracy of our model?
Answer: Future work will require trials of hyperparameter tuning. For better optimization of the model's performance, the team applied two methods of feature selection. This will be explained in section III.I.

For the results of our model testing, we look at an average F-1 score over 10 iterations.

Figure 5: Model Testing Comparison



There are advantages and disadvantages associated with using every algorithm. There is an imbalance data issue in the dataset, so it is important to choose a classifier that can effectively handle this problem. The advantage of using the MLP classifier is that it trains on the previous iteration of results which has a possibility of outperforming the other classifiers when trained for longer periods of time. However, the MLP and Adaboost algorithms performed poorly compared to the other algorithms with our dataset. The team also attempted to use Naive Bayes, SGD, and Random Forest classifiers. While the Random Forest classifier performed well, a recent study[3] showed that K nearest neighbor was able to outperform other classifiers. The results from these experiments showed that PRICE [3] was correct in that K Nearest Neighbors has the highest overall performance compared to other classification algorithms.

III.I Sampling

As mentioned before, the team faced an imbalanced data issue. The detailed information can be seen in table 2 in appendix VII.I. Only 6.31% of the data was a "hit" for regression. Oversampling and undersampling techniques were used in order to solve this problem. Undersampling is the process of decreasing the number of dismisses in order to give the data more balance. The results with this technique were poor. This is a common occurrence with undersampling since a lot of information is dropped. Important data may be dropped that can heavily affect the results of the model. With these poor results from undersampling, it was necessary to test oversampling. Oversampling duplicates data in the minority class, allowing for the data to become more balanced by having more samples. The oversampling technique was implemented with a sampling strategy of 0.5 in the training data. This means that the majority data consists of 500 examples and the minority data consists of 100 examples. After implementing the oversampling the minority data will have 250 examples. The predictions are then run by these features. This technique proved to be the most effective for the imbalanced data issue in this dataset, giving the best results on the testing models as can be seen below.

	precision	recall	f1-score	support
0	0.97	0.99	0.98	1193
1	0.82	0.56	0.67	84
accuracy			0.96	1277
macro avg	0.90	0.78	0.82	1277
weighted avg	0.96	0.96	0.96	1277

Figure 6: All features with oversampling

	precision	recall	f1-score	support
0	0.94	1.00	0.97	1202
1	1.00	0.03	0.05	75
accuracy			0.94	1277
macro avg	0.97	0.51	0.51	1277
weighted avg	0.95	0.94	0.92	1277

Figure 7: All features without oversampling

III.II Feature Selection

Feature Selection is an important process when the input dataset has a plethora of features. The dataset presented for this project started with a total of 7 features. After running several models with poor results, as can be seen in figure 5, it was determined that more features would be needed in order to accurately train a model. The team was then given a dataset with 46 features,

a substantial increase from the previous 7. Running a model on this many features can cause inaccuracy due to the redundancy of correlating features. Selecting the proper features to use became a top priority in improving results.

The first approach to feature selection was using Pearson's Correlation Coefficient. This method is known for giving a quantifiable relationship between two entities. Understanding which features highly correlate with each other allows for certain features to be removed. This is due to highly correlating features causing redundancy as mentioned before. Figure 8 shows a Pearson Correlation plot for the 46 features of the datset.

Figure 8: Pearson Correlation Plot

The plot shows all scores that are above 0.7. This allows for an easier time viewing which features have high correlation. Each feature is then carefully reviewed and filtered until there are only 9 features remaining. This method of feature selection resulted in the following 9 columns remaining: ['Del Func >= X', 'New Func >= X', 'Reached Del Func >= X', 'Top Chg by Call >= X%', 'Top > X% by Call Chg by >= 10%', 'Top Chg Len >= X%', 'Top Reached Chg Len >= X%', 'MaxCyclomatic', 'AvgLine']. A Model is then ran based on the feature selection results, as can be seen in figure 9.

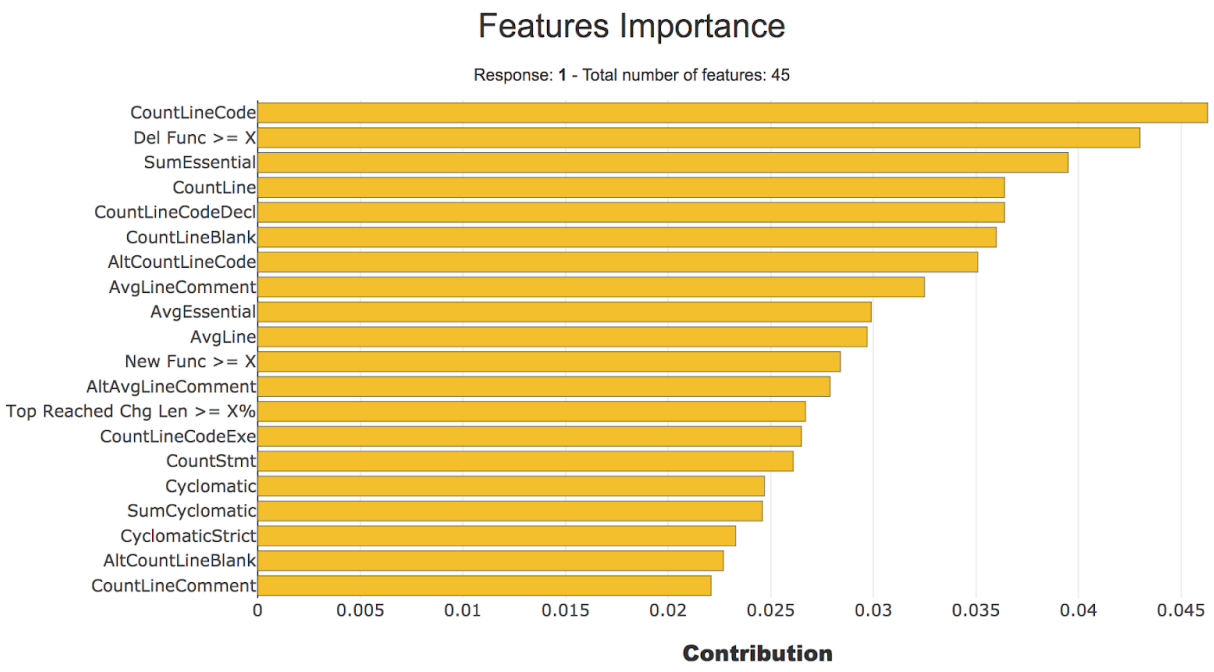
Figure 9: KNN Model Results Using Pearson Feature Selection

```
print(classification_report(y_test,predictions))
```

	precision	recall	f1-score	support
0	0.96	0.96	0.96	1198
1	0.41	0.41	0.41	79
accuracy			0.93	1277
macro avg	0.68	0.68	0.68	1277
weighted avg	0.93	0.93	0.93	1277

However, the pearson correlation does not perform well with the numerical input and categorical output. It was decided to run a different feature selection technique to see whether a better result could be obtained. The Shapash library is a package specializing in feature selection techniques that allows for easy visualization of the most important features. It requires an already trained model as input to calculate the most important features. For this scenario we used the Random Forest Classifier with all 46 features as the input.

Figure 10: Feature Importance Chart



In this feature selection, contributions of feature importance that are over 0.025 were picked to be the features, resulting in these 13 features being used: ["SumEssential", "CountLine", "Count-

LineCodeDecl", "CountLineBlank", "AltCountLineCode", "AvgLineComment", 'AvgEssential', 'AvgLine', "New Func >= X", "AltAvgLineComment", "Top Reached Chg Len >= X%", "CountLineCodeExe", 'CountStmt']. The results from the KNN and Random Forest classifiers can be seen in figures 11 and 12 below. The KNN model is shown to outperform the Random Forest model. Several iterations were ran to verify the consistency of the results.

	precision	recall	f1-score	support
0	0.98	0.99	0.99	1193
1	0.90	0.64	0.75	84
accuracy			0.97	1277
macro avg	0.94	0.82	0.87	1277
weighted avg	0.97	0.97	0.97	1277

Figure 11: KNN Results with Shapash Feature Selection

	precision	recall	f1-score	support
0	0.97	0.99	0.98	1193
1	0.86	0.60	0.70	84
accuracy			0.97	1277
macro avg	0.92	0.79	0.84	1277
weighted avg	0.96	0.97	0.96	1277

Figure 12: Random Forest Results with Shapash Feature Selection

III.III Cross-validation & Hyper-parameter Tuning

To enhance the model further, the team implemented cross-validation and hyper-parameter tuning. Cross validation is a commonly used method of re-sampling that helps to evaluate models with different samples of testing and validation data. This study used a manual implementation of k-fold cross-validation that splits the samples into a number of groups, determined by the single parameter "k" (5 in this scenario). The process for this can be seen in figure 13, along with the results in figure 14.

```
[ ] dictionary = {}
    for i in range(5):
      if (i!=4):
        dictionary[i] = {}
        m=2*639
        dictionary[i]['test'] = X[i*m:(i+1)*m]
        index_values = [j for j in range(i*m,(i+1)*m,1)]
        dictionary[i]['train'] = X.drop(index_values,axis=0)
      else:
        dictionary[i] = {}
        m=2*639
        dictionary[i]['test'] = X[i*m:6383]
        index_values = [j for j in range(i*m,6383,1)]
        dictionary[i]['train'] = X.drop(index_values,axis=0)
```

Figure 13: Kfold Cross-validation code

	precision	recall	f1-score	support
0	0.93	0.81	0.87	1192
1	0.06	0.17	0.09	86
accuracy			0.77	1278
macro avg	0.50	0.49	0.48	1278
weighted avg	0.87	0.77	0.81	1278

Figure 14: KNN Model w/ cross-validation

As can be seen, the results from cross-validation were very poor. The future work will include a stratified k-fold technique. This technique will be applied to the oversampled dataset, which will ensure that the ratio of the target classes remain the same across each fold.

Hyper-parameter tuning consists of finding the optimal set of hyper-parameters for the model. The method used here was a grid search taken from the scikit-learn library. Pre-determined values of hyper-parameters are selected, and then ran on the KNN model. The results of this process can be seen in figure 15. These results are almost identical to the results we previously had without performing the hyper-parameter tuning.

Figure 15: KNN Model Hyper-parameter Tuning

```
[ ] gridSearchCV.fit(X_train_over,y_train_over)

    GridSearchCV(cv=None, error_score=nan,
                 estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
                                                metric='minkowski',
                                                metric_params=None, n_jobs=None,
                                                n_neighbors=10, p=2,
                                                weights='uniform'),
                 iid='deprecated', n_jobs=None,
                 param_grid={'algorithm': ('auto', 'ball_tree', 'kd_tree', 'brute'),
                             'leaf_size': (1, 5, 10, 15, 30),
                             'n_neighbors': (5, 10, 15, 30), 'p': [1, 2],
                             'weights': ('uniform', 'distance')},
                 pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
                 scoring='f1', verbose=0)

[ ] gridSearchCV.best_estimator_.get_params()

    {'algorithm': 'auto',
     'leaf_size': 30,
     'metric': 'minkowski',
     'metric_params': None,
     'n_jobs': None,
     'n_neighbors': 30,
     'p': 1,
     'weights': 'distance'}

[ ] gridSearchCV.best_score_

    0.725876608148037
```

III.IV Results

To understand the results of this experiment, one must understand recall and precision. Recall is defined as true positive over (false negative + true positive). In the case for this study, recall means how often the system correctly detects the performance regression. When the recall is low, it shows that the system labels the performance regression commit as non-performance regression commits. However, these are our main concerns. Precision is defined as true positive over (true positive + false positive). It determines how accurate the model is in the predictions. When the precision is low, the system is tagging the non-performance regression commits as performance regressions. It wastes resources, but the correctly detected regressions are still being fixed. That's why the team is more concerned about the recall and uses recall as the major metric for evaluating performance. Upon analysis, an instance of a false negative can be seen in figure 16. There are two reasons behind the recall. First, there is only one record of this commit. Secondly, the relevant commits following the false negative commit (yellow mark) have more changes, but they are non-performance regression. So it is challenging for the model to distinguish this commit as performance regression.

Figure 16: False Negative Example

Commit A	Commit B	SumEssential	CountLine	CountLineCodeDecl	CountLineBlank	AltCountLineCode	AvgLineComment	AvgEssential	AvgLine	New Func >=X	AltAvgLineComment	Top Reached Chg Len >=X%	CountLineCodeExe	CountStmt	hit/dismiss
52debb68	5f30bb4a	-3	-22	-3	-4	-15	0	0	0.001876	1	0	0	-9	-12	0
52debb68	5f30bb4a	-3	-22	-3	-4	-15	0	0	0.001876	1	0	0	-9	-12	0
52debb68	5f30bb4a	-3	-22	-3	-4	-15	0	0	0.001876	1	0	0	-9	-12	0
52debb68	5f30bb4a	-3	-22	-3	-4	-15	0	0	0.001876	1	0	0	-9	-12	0
52debb68	5f30bb4a	-3	-22	-3	-4	-15	0	0	0.001876	1	0	0	-9	-12	0
52debb68	5f30bb4a	-3	-22	-3	-4	-15	0	0	0.001876	1	0	0	-9	-12	0
52debb68	5f30bb4a	-3	-22	-3	-4	-15	0	0	0.001876	1	0	0	-9	-12	0
52debb68	5f30bb4a	-3	-22	-3	-4	-15	0	0	0.001876	1	0	0	-9	-12	0
52debb68	5f30bb4a	-3	-22	-3	-4	-15	0	0	0.001876	1	0	0	-9	-12	0
5220b758	76180a2b	0	0	0	0	0	0	0	0	4	0	0	0	0	1
5c9159de	b1ec08fd	-37	-846	-299	-96	-688	0.003752	0.007505	0.030019	112	0.001876	0	-366	-411	1
b1ec08fd	52fcec75	163	2870	720	387	2183	-0.008318	-0.0253	-0.19902	207	-0.006567	0.06383	1333	1544	1
52fcec75	a5229cc9	924	13342	2435	1739	9231	-0.024106	-0.048592	-0.009716	457	-0.026627	1	5844	6364	0
52fcec75	a5229cc9	924	13342	2435	1739	9231	-0.024106	-0.048592	-0.009716	457	-0.026627	1	5844	6364	0
52fcec75	a5229cc9	924	13342	2435	1739	9231	-0.024106	-0.048592	-0.009716	457	-0.026627	0.0625	5844	6364	0
52fcec75	a5229cc9	924	13342	2435	1739	9231	-0.024106	-0.048592	-0.009716	457	-0.026627	1	5844	6364	0
52fcec75	a5229cc9	924	13342	2435	1739	9231	-0.024106	-0.048592	-0.009716	457	-0.026627	1	5844	6364	0
52fcec75	a5229cc9	924	13342	2435	1739	9231	-0.024106	-0.048592	-0.009716	457	-0.026627	1	5844	6364	0
52fcec75	a5229cc9	924	13342	2435	1739	9231	-0.024106	-0.048592	-0.009716	457	-0.026627	1	5844	6364	0
52fcec75	a5229cc9	924	13342	2435	1739	9231	-0.024106	-0.048592	-0.009716	457	-0.026627	1	5844	6364	0

IV Threats to Validity

Threats to validity are broken down into four main categories: conclusion validity, internal validity, construct validity, and external validity.Understanding the potential problems with the solution is necessary to make improvements. Threats to conclusion validity can be clearly seen in the limited features of the dataset, as well as the imbalance in the hits and dismisses within the data. Data imbalance and poor features makes it difficult to accurately train a model. To mitigate this threat, the team was provided with a new dataset with an additional 46 features to work with. This allowed them to appropriately select the features that worked best on the selected model. As for the data imbalance, it was found that oversampling the data worked best so solve that issue. Internal threats to validity are shown with model and feature selection. The team has tried multiple methods to select features and acquired different results with different feature combinations. Finding the best match of features and models can be an infinite task, but a competitive result is eventually found. Improvements may be seen in later iterations as the team shifts their focus to hyper-parameter tuning. Construct threats to validity can be seen in the evaluation of the models. The f-1 score is often the focus, but as models are fine-tuned it will become important to focus on the precision and recall scores. For this task specifically, the recall score is important as it determines the amount of false negatives that are presented from the model. The data is composed of real world commits. Finally, there are external threats to validity. These can be seen in the dimension of the dataset. Data collected from one project may vary entirely from data collected within another project. This could potentially create an artificial environment where the model is fine-tuned to work on these specific types of commits, but fail on other projects commits. The solution to this problem will likely be a topic in future works.

V Conclusion

It was found that the KNN model gave the best results while selecting a minimal amount of features out of the 50 columns presented. The Random Forest Classifier offers very similar results, but slightly worse. With this iteration results went from an average f-1 score of .13 to .42, and finally to .75. This jump in performance shows the importance of properly understanding the data and the impact feature selection can have on a model. The team looks forward to performing hyper-parameter optimization on both of these models to see how much the performance can be improved upon. Cross-validation will also be examined to ensure the model is efficient on unseen data.

VI Future Work

The future work includes section 2, as shown in figure 1, which includes test case prioritization. This will allow for better resource utilization along with prioritizing which type of test cases/commits are to be analyzed first. Upon further analysis of some data instances, we see that there are some noticeable trends in these instances. First, in table 1, there are three commits which show performance regression. These commits are then followed with a commit which does not show performance regression. The team wondered about why this was the case. Further, the first three instances show that there is a new function added to the software that causes performance regression. Only conclusion as to why the next commit does not exhibit performance regression is that this particular function was deleted from the software code repository. This therefore implies that multiple trends like these could be present within the dataset and could be a part of some time series modelling.

Table 1: Data Trends For Time Series Modelling

Commit A	Commit B	Del Func >= X	New Func >= X	Hit/Dismiss
8f449614	12913a78	569	995	1
bb0ba997	7a3eb9e2	0	1	1
bb0ba997	7a3eb9e2	0	1	1
dcde8b3d	2fc0f184	8	8	0

VII Appendix

VII.I Dataset Overview - old dataset

Table 2: Features

No.	Feature Name	Description	Rationale
1.	Commit A		
2.	Commit B		
3.	Benchmark	Suites measure the performance of hardware, software, or computer.	The suite that needs to be run for performance regression testing.
4.	Del Fun >= X%	Number of deleted functions	The number of deleted functions indicating refactoring which may lead to performance changes.
5.	New Fun >= X%	Number of new functions added	Added functions indicating new functionality which may lead to performance changes.
6.	Reached Del Fun >= X%	Number of deleted functions reached by the benchmark	Deleting a function which was part of the benchmark execution could lead to a performance change.
7.	Top Chg by Call >= X%	The percent overhead of the top most called function that was changed.	Altering a function that takes up a large portion of the processing time of a benchmark has a high risk of causing a performance regression because it is such a large portion of the test.
8.	Top > X% by Call Chg by >= 10%	The percent overhead of the top X% most called function that was changed by more than 10% of its instruction length.	This takes into account that the change affects a reasonable portion of the function in question. Bigger changes may mean higher risk.
9.	Top Chg Len >= X%	The highest percent function length change.	Large changes to functions are more likely to cause regressions than small ones.
10.	Top Reached Chg Len >= X%	The highest percent function length change that is called by the benchmark.	Here we guarantee that the functions are actually called by the benchmark in question.
11.	Hit/Dismiss	Predicted performance Regression	1 - Hit (Performance Change) 0 - Dismiss (No change).

Table 3: Hits/Dismiss

Total number of data points	6353
Number of hits	401
Number of dismiss	5952
Percentage hits	6.31%

Table 4: Benchmark

	The number of each unique type of benchmark	**The percentage of each unique type of benchmark**
p4001-diff-no-index.sh-perf-report	650	10%
p0001-rev-list.sh-perf-report	513	8%
p0000-perf-lib-sanity.sh-perf-report	489	8%
p0002-read-cache.sh-perf-report	489	8%
p4211-line-log.sh-perf-report	477	8%
p4000-diff-algorithms.sh-perf-report	457	7%
p5302-pack-index.sh-perf-report	456	7%
p7810-grep.sh-perf-report	456	7%
p5310-pack-bitmaps.sh-perf-report	453	7%
p7000-filter-branch.sh-perf-report	433	7%
p7300-clean.sh-perf-report	421	7%
p3404-rebase-interactive.sh-perf-report	292	5%
p5303-many-packs.sh-perf-report	151	2%
p3400-rebase.sh-perf-report	136	2%
p0003-delta-base-cache.sh-perf-report	134	2%
p5550-fetch-tags.sh-perf-report	133	2%
p0071-sort.sh-perf-report	123	2%
p0005-status.sh-perf-report	46	1%
p0006-read-tree-checkout.sh-perf-report	44	1%
Total	**6353**	**100%**

Table 5: Benchmark_hit

	Number of hits per benchmark	**Percentage of hits per benchmark**
p4001-diff-no-index.sh-perf-report	199	50%
p0001-rev-list.sh-perf-report	61	15%
p0002-read-cache.sh-perf-report	37	9%
p0000-perf-lib-sanity.sh-perf-report	36	9%
p4211-line-log.sh-perf-report	25	6%
p7000-filter-branch.sh-perf-report	24	6%
p4000-diff-algorithms.sh-perf-report	5	1%
p5302-pack-index.sh-perf-report	4	1%
p7810-grep.sh-perf-report	4	1%
p5303-many-packs.sh-perf-report	3	1%
p5310-pack-bitmaps.sh-perf-report	2	0%
p0071-sort.sh-perf-report	1	0%
Total	**401**	**100%**

Table 6: Benchmark_dismiss

	Number of dismisses per benchmark	**Percentage of dismisses per benchmark**
p0000-perf-lib-sanity.sh-perf-report	453	8%
p7810-grep.sh-perf-report	452	8%
p0001-rev-list.sh-perf-report	452	8%
p0002-read-cache.sh-perf-report	452	8%
p5302-pack-index.sh-perf-report	452	8%
p4000-diff-algorithms.sh-perf-report	452	8%
p4211-line-log.sh-perf-report	452	8%
p5310-pack-bitmaps.sh-perf-report	451	8%
p4001-diff-no-index.sh-perf-report	451	8%
p7300-clean.sh-perf-report	421	7%
p7000-filter-branch.sh-perf-report	409	7%
p3404-rebase-interactive.sh-perf-report	292	5%
p5303-many-packs.sh-perf-report	148	2%
p3400-rebase.sh-perf-report	136	2%
p0003-delta-base-cache.sh-perf-report	134	2%
p5550-fetch-tags.sh-perf-report	133	2%
p0071-sort.sh-perf-report	122	2%
p0005-status.sh-perf-report	46	1%
p0006-read-tree-checkout.sh-perf-report	44	1%
Total	**5952**	**100%**

References

[1] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. Prioritizing versions for performance regression testing: The pharo case. *Science of Computer Programming*, 191:102415, 2020.

[2] Najat Ali, Daniel Neagu, and Paul Trundle. Evaluation of k-nearest neighbour classifier performance for heterogeneous data sets. *SN Applied Sciences*, 1(12):1–15, 2019.

[3] Deema Alshoaibi, Kevin Hannigan, Hiten Gupta, and Mohamed Wiem Mkaouer. Price: Detection of performance regression introducing code changes using static and dynamic metrics. In Shiva Nejati and Gregory Gay, editors, *Search-Based Software Engineering*, pages 75–88, Cham, 2019. Springer International Publishing.

[4] Jinfu Chen. Performance regression detection in devops. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, page 206–209, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Jinfu Chen and Weiyi Shang. An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 341–352, 2017.

[6] Jinfu Chen, Weiyi Shang, and Emad Shihab. Perfjit: Test-level just-in-time prediction for performance regression introducing commits. *IEEE Transactions on Software Engineering*, 2020.

[7] Augusto Born De Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Perphecy: performance regression test selection made simple but effective. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 103–113. IEEE, 2017.

[8] Shadi Ghaith, Miao Wang, Philip Perry, Zhen Ming Jiang, Pat O'Sullivan, and John Murphy. Anomaly detection in performance regression testing by transaction profile estimation. *Software Testing, Verification and Reliability*, 26(1):4–39, 2016.

[9] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. Perfranker: Prioritization of performance regression tests for collection-intensive software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 23–34, 2017.

[10] Thanh HD Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 299–310, 2012.

[11] Zhe Yu. https://docs.google.com/presentation/d/18ko8awpp$_{i}iyodpy3bneugmslgvup2hnc - okbrmvhmy/editslide = id.p.$