# COSC 5P07 – Final Project

NAZANIN MEHREGAN: 7549975 and CLEMENT FRIMPONG OSEI: 7274715, Brock University, Canada

## 1 INTRODUCTION

Performance issues emerge from a lot of defects present in applications and these issues, if known before hand, in the development phases of applications, can help stabilize the applications. Observing the performance data of existing applications after benchmarking them, is important to give developers in-depth knowledge of building efficient applications. In their work, Cito et. al. (2018) [4] presented an approach that creates operational awareness of issues with performance in developers' code by creating a plugin that integrates performance traces in runtime into developer workflows. Analysing the changepoint during the execution of an application helps to determine its steady state and whether that state is the peak of its performance or not [2]. In this study, we focus on determining the patterns that arise in the benchmark data generated in the research work [6] by Traini et. al (2022). Our goal is to extract meaningful features from the data, examine the pattern amongst the features and identify which of them effectively impact the state of a benchmark. We propose and answer some research questions that projects us to achieving the goal of this study. We employed the Python programming language and libraries like numpy, pandas and seaborn to finding answers to our research questions.

## 2 RESEARCH QUESTIONS AND CONTRIBUTIONS

Within the scope of this study, we will first propose three research questions:

### 2.1 RQ1: How can we extract meaningful features from the data that was generated by the JVM benchmark?

We have access to the findings of the steady-state study, which are included within a number of JSON files and are partitioned into 3 separate sections. 1) Classification 2) Changepoints, and 3) the time-series.

The purpose of Java microbenchmarking is to evaluate the steady state performance of Java applications by measuring their execution time. The data in the time-series JSON files represent time-series that are the execution durations of each benchmark iteration. We have a JSON file dedicated to each and every benchmarking that was performed on the applications.

Each JSON file within the classification folder reports whether or not a specific benchmark has reached a steady state of performance, as well as when this occurred. Specifically, the file reports the classification of each benchmark (steady state, no steady state, or inconsistent) and fork (steady-state or no steady state), as well as the JMH iterations in which the steady state is reached (-1 indicates no steady state).

In addition to this, we get access to the list of changepoints discovered by the PELT method in each and every fork of each benchmark. In order to determine where and when these changes occurred in the execution time, a changepoint algorithm was applied. They

Authors' address: Nazanin Mehregan: 7549975; Clement Frimpong Osei: 7274715, Brock University, 1812 Sir Isaac Brock Way, St. Catharines, ON, , Canada.

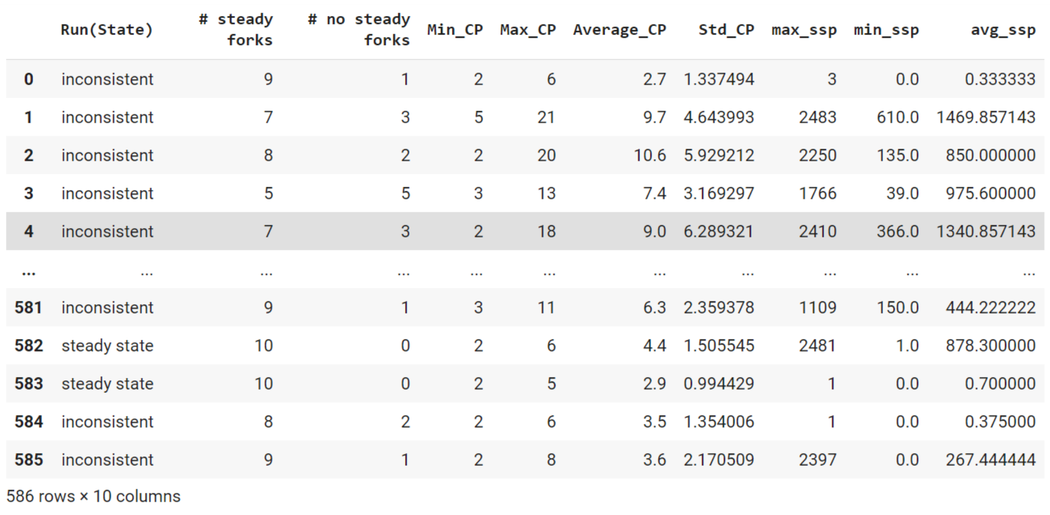
Update date: 7 January 2023

break up the entire time-series into smaller pieces called segments within which the behaviour of the time series is considered unchanged. These will then be used to determine if and when a benchmark execution has reached a steady state.

We had to begin by obtaining the JSON data and constructing a DataFrame from them before we could proceed with the analysis of the data and the extraction of patterns among the consistent and inconsistent benchmarks. These patterns can thus be useful in determining if the benchmarking being used is useful or not. For instance, let's say that there were never more than two changepoints among the benchmarks that have attained steady states. If we were monitoring a benchmark and noticed that it had beyond two changing points, we could come to the conclusion that this benchmark is probably not relevant and issue a warning that the benchmark needs to be revised. In addition, if we receive five changing points during this benchmark, we know for certain that there is a problem, and that it will prevent us from reaching the steady state.

For merging the JSON data prepared for us, we used Pandas and Numpy libraries in python, created a data frame and named the column of it. In this data frame, each row represents a benchmark of an application.

We set out to collect the data of 586 different benchmarks across 28 different applications. Given that we had the data for each fork of every benchmark and that each benchmark was run for 10 different forks, this part presented a challenge for us because we needed to find a way to effectively summarize the data from 10 different forks of benchmarking into one row that provides us with an overview of the data for that particular benchmark. The features that we obtained from this data frame include Run (the benchmark state), the number of steady and no steady forks, as well as the minimum, maximum, and average number of the changing points. These features are illustrated in figure 1. In addition to this, we have the starting steady point minimum, maximum, average, and standard deviation for each and every fork that actually reaches the steady state. This data proves to be useful when we move on to the next steps and attempt to identify patterns and correlations between the various features of a benchmark and whether or not it has reached a steady state.



| | Run(State) | # steady forks | # no steady forks | Min_CP | Max_CP | Average_CP | Std_CP | max_ssp | min_ssp | avg_ssp |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | inconsistent | 9 | 1 | 2 | 6 | 2.7 | 1.337494 | 3 | 0.0 | 0.333333 |
| **1** | inconsistent | 7 | 3 | 5 | 21 | 9.7 | 4.643993 | 2483 | 610.0 | 1469.857143 |
| **2** | inconsistent | 8 | 2 | 2 | 20 | 10.6 | 5.929212 | 2250 | 135.0 | 850.000000 |
| **3** | inconsistent | 5 | 5 | 3 | 13 | 7.4 | 3.169297 | 1766 | 39.0 | 975.600000 |
| **4** | inconsistent | 7 | 3 | 2 | 18 | 9.0 | 6.289321 | 2410 | 366.0 | 1340.857143 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **581** | inconsistent | 9 | 1 | 3 | 11 | 6.3 | 2.359378 | 1109 | 150.0 | 444.222222 |
| **582** | steady state | 10 | 0 | 2 | 6 | 4.4 | 1.505545 | 2481 | 1.0 | 878.300000 |
| **583** | steady state | 10 | 0 | 2 | 5 | 2.9 | 0.994429 | 1 | 0.0 | 0.700000 |
| **584** | inconsistent | 8 | 2 | 2 | 6 | 3.5 | 1.354006 | 1 | 0.0 | 0.375000 |
| **585** | inconsistent | 9 | 1 | 2 | 8 | 3.6 | 2.170509 | 2397 | 0.0 | 267.444444 |

586 rows × 10 columns

Fig. 1. The extracted data frame

### 2.2 RQ2: What are the patterns among different features of benchmarks and how can we classify them?

We made use of 3 different methods to answer this question. After removing the outliers in the data, 22 rows were dropped and we ended up with 564 benchmarks to apply our algorithms on.

*2.2.1 K Nearest Neighbor Classification.* The first method is called the k nearest neighbor classification. The KNN algorithm is a method for classifying data that estimates the likelihood that a data point will become a member of one group or another based on what group the data points that are closest to it belong to. If the value of K is equal to one, then the only neighbour that will be considered for deciding the category of a data point is the one that is geographically closest to it. If the value of K is equal to 10, then we will use the ten neighbours who are geographically closest to the point of interest, and so on [1]. We specified the sizes for the train set and test set as 80 and 20 percent respectively and we got the precision of approximately 72 percent. Figure 2 illustrates the output confusion matrix. It is worth mentioning that whether a benchmark is steady or not is encoded as 1 or 0 in this data set.

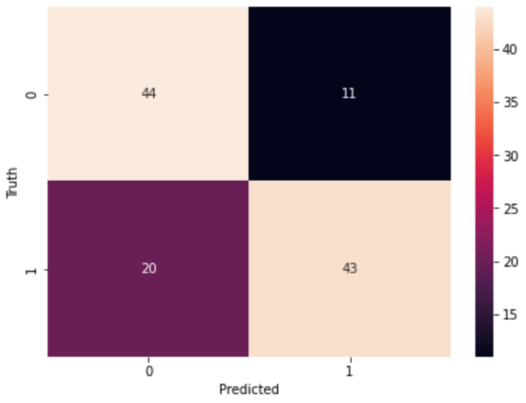


Fig. 2. Output KNN Confusion Matrix

*2.2.2 SVM Classification.* We made use of the SVM method to classify different benchmarks based on the extracted features. SVMs are a group of associated supervised learning techniques used for regression and classification and are a member of the generalized linear classification family. The Structural Risk Minimization is the foundation of SVM (SRM). Input vectors from SVM are mapped to a higher-dimensional space, where a maximum separation hyperplane is built. On each side of the hyperplane that divides the data, two parallel hyperplanes are built. The hyperplane that maximizes the separation between the two parallel hyperplanes is known as the separating hyperplane. It is presumpted that the classifier's generalization error will be lower the wider the margin or separation between these parallel hyperplanes [3]. As shown in figure , the output of this algorithm on our benchmarking data resulted in a 99 percent accuracy in identifying which benchmarks reach steady states or not. However, it is possible that this algorithm has faced an overfitting problem which could be further explored in the future work.

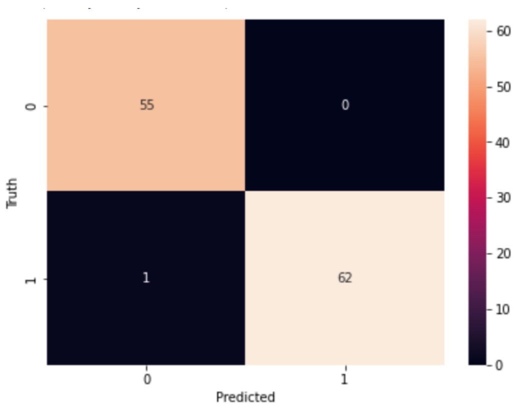


Fig. 3. Confusion Matrix– Output of SVM Classification

*2.2.3 K-Means Clustering.* A cluster is represented by its centroid in the k-means method, which is the mean of the points inside the cluster. This is more practical for numerical qualities solely and may be adversely impacted by a single outlier. The term "centroid" derives from the fact that each of the k clusters C is represented by the mean c of its points. The objective function is the total of the differences between a point and its centroid represented as an acceptable distance. The cluster with the nearest centroid is given to each point [5]. K refers to the total number of clusters, which in this case is equal to 2. First, we went through our data set and assigned a value of 0 for inconsistent and 1 for steady to each of the benchmarks. After that, we took our data set, excluded the benchmark state feature from it, and then applied the k-means clustering method to it. This resulted in the data being split into two sections without making use of the consistent and inconsistent features. Figure 3 illustrates the clustering output. After that, we checked the accuracy of the k-means algorithm's clustering of the data by comparing its output to our own labelled data. This allowed us to see how well the algorithm did its job. The comparison reveals that there is roughly a 74 percent similarity between the two data sets. This indicates that 74 percent of our data was successfully clustered in the appropriate group that corresponds to the state feature.

## 2.3 RQ3: Which feature has the most effect on the state of a benchmark?

In this part, we examined the clustered output from the previous section in order to determine which features have the most impact on whether or not a clustering is achieved correctly. We separated the 74 percent part, which consisted of around 416 rows that were successfully clustered, and then we computed the mean of all of the attributes of those rows to combine them into a single row. In a similar manner, we determined the mean for the 26 percent of the data that was improperly grouped. After that, we compared the two sets of compressed data and searched for the feature(s) that had the most significant disparity between the two sets of data. As shown in figure 4, we can conclude that maximum, minimum, and average of the starting steady points in benchmarks have the biggest differences among the groups of data that can suggest these features have the most effect on clustering the data sets.

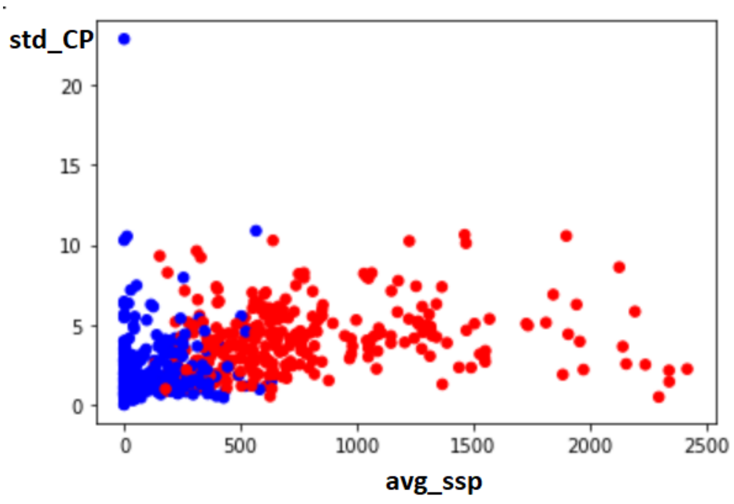
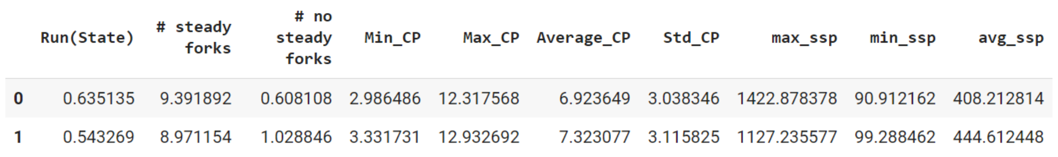


Fig. 4. Output of K-Means Clustering Method



| | Run(State) | # steady forks | # no steady forks | Min_CP | Max_CP | Average_CP | Std_CP | max_ssp | min_ssp | avg_ssp |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.635135 | 9.391892 | 0.608108 | 2.986486 | 12.317568 | 6.923649 | 3.038346 | 1422.878378 | 90.912162 | 408.212814 |
| **1** | 0.543269 | 8.971154 | 1.028846 | 3.331731 | 12.932692 | 7.323077 | 3.115825 | 1127.235577 | 99.288462 | 444.612448 |

Fig. 5. Output K-Means Clustering Method

## 3 CONCLUSIONS

We initially presented three research topics in this literature study, which were concerning feature extraction, pattern mining, and result analysis. We retrieved the data from 586 JVM benchmarking applications and defined meaningful features for the data set. The data was then mined using several machine learning methods such as KNN and SVM classification to uncover relevant connections between the state of each benchmark. They were both quite accurate, with accuracy rates of 72 and 99 percent, respectively. Furthermore, K-Means clustering was performed on the data, and the results were compared with the original labelled (steady or inconsistent) data, yielding a similarity of 74 percent. We discovered that the minimum, maximum, and average of the initial steady points in benchmarks had the most influence on whether or not they attain a steady state. The link to our github repository where the source code can be found is: https://github.com/nazaninmehregan/5P07-Final-Project.

## REFERENCES

[1] S Archana and K Elangovan. 2014. Survey of classification techniques in data mining. *International Journal of Computer Science and Mobile Applications* 2, 2 (2014), 65–71.

[2] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (oct 2017), 27 pages. DOI: http://dx.doi.org/10.1145/3133876

[3] Himani Bhavsar and Mahesh H Panchal. 2012. A review on support vector machine for data classification. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)* 1, 10 (2012), 185–189.

[4] Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, and Harald C. Gall. 2018. PerformanceHat: Augmenting Source Code with Runtime Performance Traces in the IDE. (2018), 41–44. DOI:http://dx.doi.org/10.1145/3183440.3183481

[5] Pradeep Rai and Shubha Singh. 2010. A survey of clustering techniques. *International Journal of Computer Applications* 7, 12 (2010), 1–5.

[6] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. 2022. Towards effective assessment of steady state performance in Java software: are we there yet? *Empirical Software Engineering* 28 (2022). https://doi.org/10.1007/s10664-022-10247-x