Analysis of Code Embeddings as a Code Similarity Metric

Samuel Coburn, Jessica Kim
sc4917, sk4711

1. Synopsis

For this project, we extend the experiments performed in “A comparison of code similarity analysers” to include code embeddings as a code similarity tool [1]. From “A comparison of code similarity analysers”, we recreate three experiments that measure the performance of the code similarity tools in the presence of pervasive modifications. Using the same datasets from the “A comparison of code similarity analysers” paper, we generate code embedding representations of each of the Java files in the datasets. We then compute the cosine similarity between the embeddings of our datasets. Comparing the result of cosine similarity calculation for a pair of embeddings to a threshold value, we are able to get counts of true positives, true negatives, false positives, and false negatives for the threshold value. We then use those values to compute the F1 score. We vary the threshold value from 1 to 100 in order to maximize the F1 score. Overall, code embeddings made from Java files from three different datasets had pairwise comparisons performed on them in the experiments. The three datasets are: 1) the original Java source code with pervasive modifications, 2) normalized Java source code using the decompiler Krakatau, and 3) normalized Java source code using the decompiler Procyon.

The implementation of our experiments is organized by the pre-trained model used to generate the code embeddings for the three datasets [2]. We utilized two different pre-trained code2vec models, called Java14 and Java-Large, in order to create the code embeddings. By repeating the experiments for the code embeddings generated by the two pre-trained models, we can determine whether or not the performance of code embeddings as a similarity metric is affected by the training of the pre-trained models.

The novelty of this project lies within its measurement of code embeddings as a similarity metric. Additionally, it provides value to software developers who specialize in code similarity by comparing the performance of code embeddings to the code similarity tools and techniques observed in the original “A comparison of code similarity analyzers” paper. The code utilized in this project to measure the similarity between generated code embeddings is publicly available on GitHub [3].

2. Research Questions

For this project, we had three research questions that we wanted to answer when recreating the original experiments from “A comparison of code similarity analyzers”:

- RQ1: How accurate are code embeddings at determining the similarity between code source files with pervasive modifications?
- RQ2: How does the performance of code embeddings as a similarity metric compare to the code similarity analyzers observed in the original experiment?

- RQ3: How does the training of the code embedding model impact its performance as a code similarity metric?

2.1 Research Question #1

To test the accuracy of code embeddings as a similarity metric, we followed the experimental method laid out as Scenario 1 in the “A comparison of code similarity analyzers” paper. The experiment begins with ten source files implemented using the programming language Java. The source code files perform introductory computer science tasks, like BubbleSort or implementing a linked list. For each of the ten source files, nine combinations of pervasive modifications were applied systematically to the file, yielding ten files accomplishing the same task; after this step, we have 10 files in our dataset that implement BubbleSort, but have different code, for example. We utilize the 100 resulting data files from this step to form our dataset (in actuality, the datasets utilized for this paper were sourced directly from the GitHub repository of the “A comparison of code similarity analyzers” paper; we are currently illustrating how exactly the data files were formed to better explain our experiment).

In this stage of the experimental process, the dataset is composed of Java files. In order to assess the accuracy of code embeddings as a similarity metric using this dataset, we need to encode the Java files of the dataset as code embeddings. In order to do this, we utilized pre-trained code2vec models. The code2vec script openly available on GitHub allows the user to change the code of an input file and receive the code embeddings via output to the terminal [4]. For the entire dataset, each individual Java file was provided as the input file, and the code embedding was provided as output to the terminal. The code embeddings were then placed in a .csv file and organized by their functionality, i.e. BubbleSort, Linked List, etc. The code embeddings for the dataset were then placed into a .txt file in order to be easily read by our code embeddings analysis Python script. We perform the process of creating code embeddings from our Java file datasets twice using two different pre-trained code2vec models, resulting in two sets of code embeddings for the same dataset. The two pre-trained models used to generate our code embeddings will be referred to as the Java14 model and the Java-Large model.

Following the experimental methodology from the “A comparison of code similarity analyzers paper”, we determine the accuracy of code embeddings as a similarity metric by performing pairwise comparisons for every code embedding in our dataset. Simply put, we compare every file in our dataset to every file in the dataset, including itself. To calculate the similarity of two code embeddings, we perform the cosine similarity of the two code embeddings. Considering that code embeddings are represented as vectors, taking the cosine similarity of the two embeddings tells us how close the two embeddings are in the embedding space using a percentage; however, in isolation, the similarity calculation does not mean anything without a point of reference. As such, we need a threshold value to know whether a similarity score between two code embeddings determines that the embeddings are actually similar. Using the threshold value, we can then sort the similarity scores of the embeddings in the dataset into four categories: true positives, true negatives, false positives, and false negatives. True positives are defined as code embeddings with the same functionality (BubbleSort, Linked Lists, etc) and a

similarity score greater than or equal to the threshold value. True negatives are defined as code embeddings with different functionality and a similarity score less than the threshold value. False positives are defined as code embeddings with different functionality but a similarity score greater than or equal to the threshold value. False negatives are defined as code embeddings with the same functionality but a similarity score less than the threshold value. Performing the pairwise comparisons for all embeddings within the dataset, we are able to get a final count of true positives, true negatives, false positives, and false negatives with respect to a particular threshold value. Using these counts, we can calculate the precision and recall at the current threshold value, which can then be used to calculate the F1 score at the current threshold value. In an effort to find the threshold value which maximizes the F1 score for the dataset, the threshold value is varied from 1 to 100.

Tool	Settings	T	TP	TN	FP	FN	Acc	Prec	Rec	AUC	F1
Java14											
Original Dataset	N/A	66	452	8780	220	548	0.4103	0.6726	0.452	0.8125	0.5407
Java-Large											
Original Dataset	N/A	87	344	8988	12	656	0.4148	0.9663	0.344	0.8301	0.5074

Table 1: Experiment 1 Results

Looking at the results in **Table 1**, code embeddings, specifically code2vec embeddings, do not perform very well as a code similarity metric in determining the similarity between code files with pervasive modifications applied on them. The code embeddings generated by the Java14 model had an F1 score of 0.5407 while the code embeddings generated by the Java-Large model had an F1 score of 0.5074. Framing this task as classifying code embeddings as either similar or dissimilar, we could get a similar F1 score by randomly choosing the result for each pairwise comparison. For each set of code embeddings, less than half of the existing true positives in the dataset (of which there are 1000) were correctly identified. As such, we make the conclusion that code embeddings are not capable of detecting similarity between source code files with pervasive modifications. The graphs depicting the change in F1 score for various thresholds for both pre-trained models' code embeddings are available in the **Appendix**. Additionally, the graphs depicting the ROC curves (used to derive the AUC metric shown in the table) for both pre-trained models' code embeddings can also be found in the **Appendix**.

The second experiment repeats the same methodology of the first experiment for two different datasets. The datasets contain Java source code from the dataset used in experiment 1 that was decompiled using a decompiler. The first dataset used the Krakatau decompiler while the second dataset used the Procyon decompiler. In decompiling the machine code of the compiled Java files, the decompilers effectively normalize the source code files, as they are written using the same technology from machine code with similar functionality. In the original paper, the authors observed that normalizing the source code files using a decompiler increased the accuracy of the code similarity tools. As we did with the first experiment, we generated code embeddings from the two aforementioned datasets twice, using both the Java14 model and the Java-Large model. We used the same pairwise comparison method to determine the performance of the code embeddings in recognizing the similarity between the normalized source files.

Tool	Settings	T	TP	TN	FP	FN	Acc	Prec	Rec	AUC	F1
Java14											
Krakatau	N/A	86	658	9000	0	342	0.4292	1	0.658	0.8891	0.7937
Procyon	N/A	76	734	8972	28	266	0.4314	0.9633	0.734	0.921	0.8331
Java-Large											
Krakatau	N/A	91	628	9000	0	372	0.4279	1	0.628	0.8738	0.7715
Procyon	N/A	88	788	8976	24	212	0.4339	0.9704	0.788	0.9208	0.8698

Table 2: Experiment 2 Results

Looking at **Table 2**, we see a similar result to that of the original paper. The code embeddings generated by both pre-trained models see a vast increase in their performance when comparing normalized source code. A notable metric within **Table 2** is that the code embeddings generated by models had a precision score of 1 on the Krakatau dataset, meaning that neither comparison generated any false positives. Additionally, the F1 scores for the code embeddings from both pre-trained models were greater on the Procyon dataset. From this second experiment, we can conclude that code embeddings are a useful similarity metric in comparing normalized source code. This supports the finding from the original paper, “A comparison of code similarity analysers.” The graphs depicting the change in F1 score for various thresholds for both pre-trained models’ code embeddings are available in the **Appendix**.

The third and final experiment recreated for this project calculates performance metrics other than F1 scores to limit the dependency of the chosen threshold value on the performance of the tool. The three metrics calculated for this experiment are precision@n, average-r-precision (ARP), and mean-average-precision (MAP). Precision@n measures the performance of the comparison tool by calculating the precision of the top n results with the highest similarity scores from the pairwise comparisons. Because there are 1000 true positives possible when performing the pairwise comparisons, the precision@n value shown in **Table 3** is the precision@n when n is equal to 1000 for the dataset from experiment 1. The next two metrics consider similarity scores for a single source file rather than pairwise comparisons for the entire dataset. To calculate these metrics, we compute the similarity score for a single source file from the dataset to every other file in the dataset. We then rank the results of the comparisons and consider the top 10 scores (the top 10 are considered because, for each file in the dataset, there are 10 files with the same functionality). From these top 10 results, the ARP is calculated by counting the number of true positives divided by 10. The MAP is calculated by taking the precision@n for the 10 results, varying n from 1 to 10. The scores shown in **Table 3** are the average ARP and MAP for all source files in the dataset used for experiment 1.

Tool	Settings	Precision@N	ARP	MAP
Java14				
Original	N/A	0.506	0.556	0.7691
Java-Large				
Original	N/A	0.48	0.556	0.7616

Table 3: Experiment 3 Results

Table 3 shows the result for the third experiment of this project. For the precision@n, the value for both models hovers around 0.5. This means that, for the 1000 comparisons with the highest similarity scores, only half (or less than half in the case of Java-Large's embeddings) were able to correctly determine when the code embeddings were similar (ie, came from the same functionality grouping). Similarly for ARP, the embeddings from both models score 0.556. For all the comparisons on the dataset for a single file, the top 10 highest similarity scores contained only 5 true positive comparisons on average. The most promising metric from this experiment was MAP, with both models scoring around 0.76. In comparison to the ARP, the high MAP scores imply that the true positives front-loaded the top 10 highest similarity scores for a given file in the dataset. From these alternative performance metrics, we can again conclude that code embeddings, specifically code2vec embeddings, are not capable of accurately determining the similarity between code files with pervasive modifications. The graphs showing the precision@n curves for both models can be found in the **Appendix**.

2.2 Research Question #2

To compare the performance of code embeddings as a similarity metric to other analyzers observed in the original experiment, we compared metrics that we have obtained by following the experimental methods defined in Scenarios 1, 3 and 4 against those derived by the original paper.

Looking at **Table 1** for Scenario 1, the F-score values we obtained for the Java14 and Java-Large models are 0.5407 and 0.5074, respectively. Table 7 in the paper ranks the performance of 30 other similarity analyzers in terms of F-scores, highest to lowest. The tool at the top is ccfx, a token-based clone detector, with an F-score of 0.9760. The tool at the bottom is bsdiff, a binary file comparison tool, with an F-score of 0.5216. In comparison to these other tools, the Java14 and Java-Large models would come in third-to-last and last, respectively. We can conclude that the two models we used in our experiment have relatively low F-scores compared to other similarity tools analyzed in the paper.

Looking at **Table 2** for Scenario 3, the F-score values we obtained for the Java14 model using the decompilers Krakatau and Procyon are 0.7937 and 0.8331, respectively. The F-score values for Java-large using Krakatau and Procyon are 0.7715 and 0.8698, respectively. Figure 11 in the paper depicts the F-scores of the same 30 similarity analyzers with and without using the decompilers mentioned above. The chart shows that both Krakatau and Procyon boost the F-scores significantly, with the minimum F-score of all tools as around 0.8 and most ranging from 0.9 to 1.0. The F-scores obtained for the Java14 and Java-Large models with the decompilers are comparable to these values but seem to be on the low to medium end of all scores.

Looking at **Table 3** for Scenario 4, the precision@n values for the Java14 and Java-large models are 0.506 and 0.48, respectively. The ARP values for both these models are 0.556, and the MAP values for Java14 and Java-Large are 0.7691 and 0.7616, respectively. Table 14 in the paper shows the top 10 rankings using these top metrics. The greatest precision@n value is

0.976 while the least (tenth place) is 0.820. The greatest ARP value is 1.000 while the least is 0.895. The greatest MAP value is 1.000 while the least is 0.929. Neither Java14 nor Java-Large models are in the top 10 for any of these metrics. It is unclear where these two models are placed among all 30 similarity analyzers due to the lack of data in the paper.

Overall, the performance of the Java14 and Java-large models seems to be comparable to other tools analyzed in the paper, but hovers near the bottom (i.e. worst performing) tools, in terms of F-scores (with and without decompilation), precision@n, ARP, and MAP.

2.3 Research Question #3

In order to analyze how the performance as a code similarity metric is affected by the training of the code embedding model, we can compare the performance between Java14 and Java-Large models using the metrics derived and analyzed in **Sections 2.1** and **2.2**.

In terms of F-scores without decompilation, Java14 (0.5407) has a slightly higher score than Java-Large (0.5047). For F-scores using Krakatau, Java14 (0.7937) has a slightly higher score than Java-Large (0.7715). However, for F-scores using Procyon, Java-Large (0.8698) has a slightly higher score than Java14 (0.8331).

In terms of other metrics, with the exception of ARP where both models have the same value (0.556), Java14 has slightly higher values than Java-Large for both precision@n (0.506 vs. 0.48) and MAP (0.7691 vs. 0.7616).

Overall, it seems that Java14 performs slightly better than Java-Large, though both produce similar results. A possible explanation could be the types of Java source files used to train the models. The Java files we used to create code embeddings are all class-based. Since the Java-Large model was trained on a much larger dataset, it is possible that the examples the model was trained on could have been too different, resulting in a slightly less accurate similarity analysis.

3. Deliverables

The link to our GitHub repository is below:

https://github.com/Sam-Coburn/COMS6156_FinalProject

For more information about the content of our GitHub repository, please consult the readme file in the repository.

4. Self Evaluations

The final project for this course was worked on by both Samuel Coburn and Jessica Kim. Their takeaways from executing this project can be read below.

4.1 Samuel Coburn

I had the idea for this project after completing my midterm literature review paper. I wrote about code completion technologies, and I read about how many code completion technologies provide code suggestions to their users. One paper I read mentioned clustering methods based on their similarity, and it brought the subject to my attention more prominently (I imagine many computer science students are aware of code similarity detection in the form of plagiarism detection for programming assignments, but they probably aren't aware of it more explicitly in terms of a software development topic). I also read a paper about code embeddings, and it reminded me of a lesson I learned in my NLP class that embeddings are useful because you can observe the similarities between embeddings. I also stumbled across the code similarity analyzers paper whose experiments we repeat, so it all came together from reading research papers for class.

For this project, I implemented the code that determines the F1 scores of the similarity analysis across various threshold values. I created all of the embeddings for the project due to a hardware restriction on Jessica's end. I also took the measurements for the first two experiments that we recreated.

From implementing this project, I learned a lot about the research process. I gained a lot of exposure to research by reading the papers discussed during lecture, but actually having to implement a research project gave me a better understanding of the process. The biggest challenge that I had during the implementation of this project was actually being able to bring all the pieces we had available together in order to get the result that we were trying to achieve. For instance, I initially thought that it would be much easier to get the embeddings directly by creating a script to load the pre-trained models, but code2vec basically requires you to utilize their code in order to retrieve the embeddings. Because of that, I basically had to get all of the embeddings one-by-one by hand. Overall, the project was a good experience that I was very passionate about doing to the best of our abilities.

4.2 Jessica Kim

I also wrote my midterm paper based on code completion technology, but did not have the chance to perform any experiments previously. I was intrigued to extend my understanding in this area by learning how the similarity between code snippets can be derived based on their embeddings.

For this project, I derived the Python functions for ARP (average r-precision) and MAP (mean average precision) based on the formulas provided by the research paper. I also created the program for the class demo, which demonstrates all the analytical functions we developed on a few sample code embeddings.

From implementing this project, I learned a lot about following and extending experiments performed by researchers. I also learned what a code embedding may actually look like and how to implement functions that can be used to analyze the similarity between different embeddings. The biggest challenge was understanding these different analytical metrics that

were derived in the original paper, and creating our own Python programs to replicate these functions.

References

[1] Ragkhitwetsagul, C., Krinke, J. & Clark, D. A comparison of code similarity analysers. *Empir Software Eng* 23, 2464–2519 (2018). https://doi.org/10.1007/s10664-017-9564-7

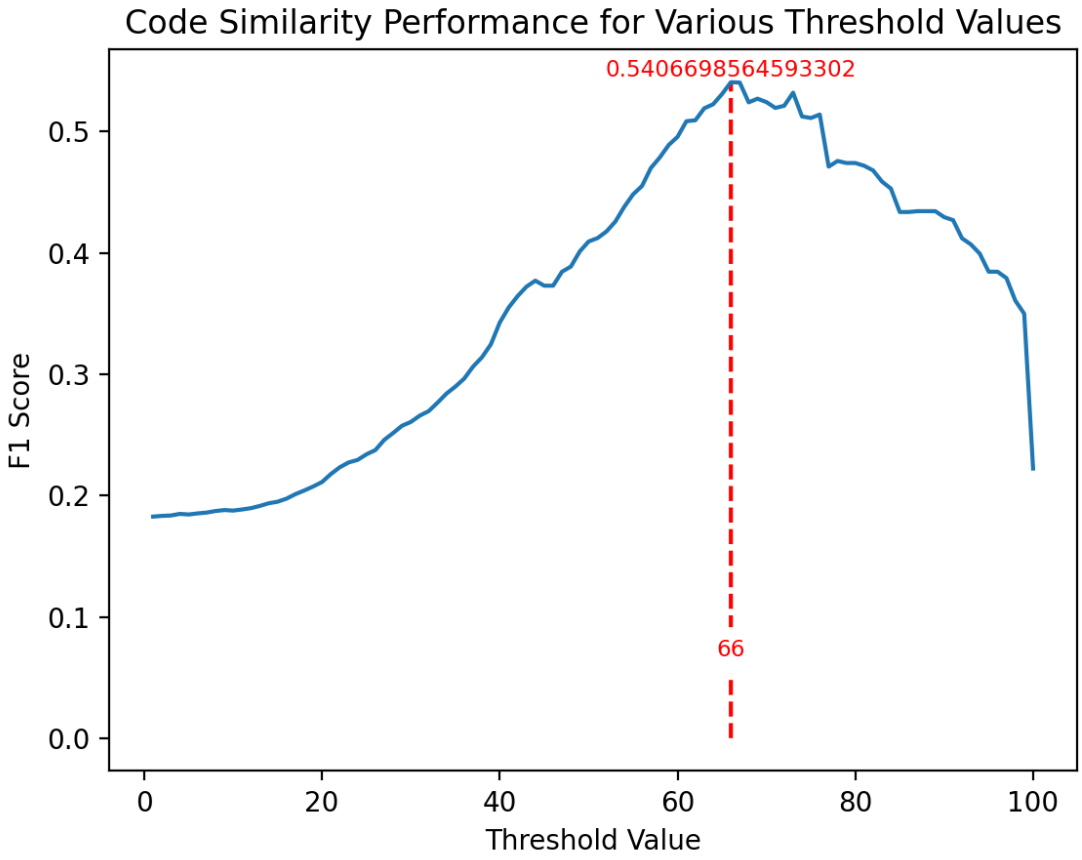
[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: learning distributed representations of code. Proc. ACM Program. Lang. 3, POPL, Article 40 (January 2019), 29 pages. https://doi.org/10.1145/3290353

[3] Sam-Coburn. "Sam-Coburn/coms6156_finalproject." *GitHub*, https://github.com/Sam-Coburn/COMS6156_FinalProject

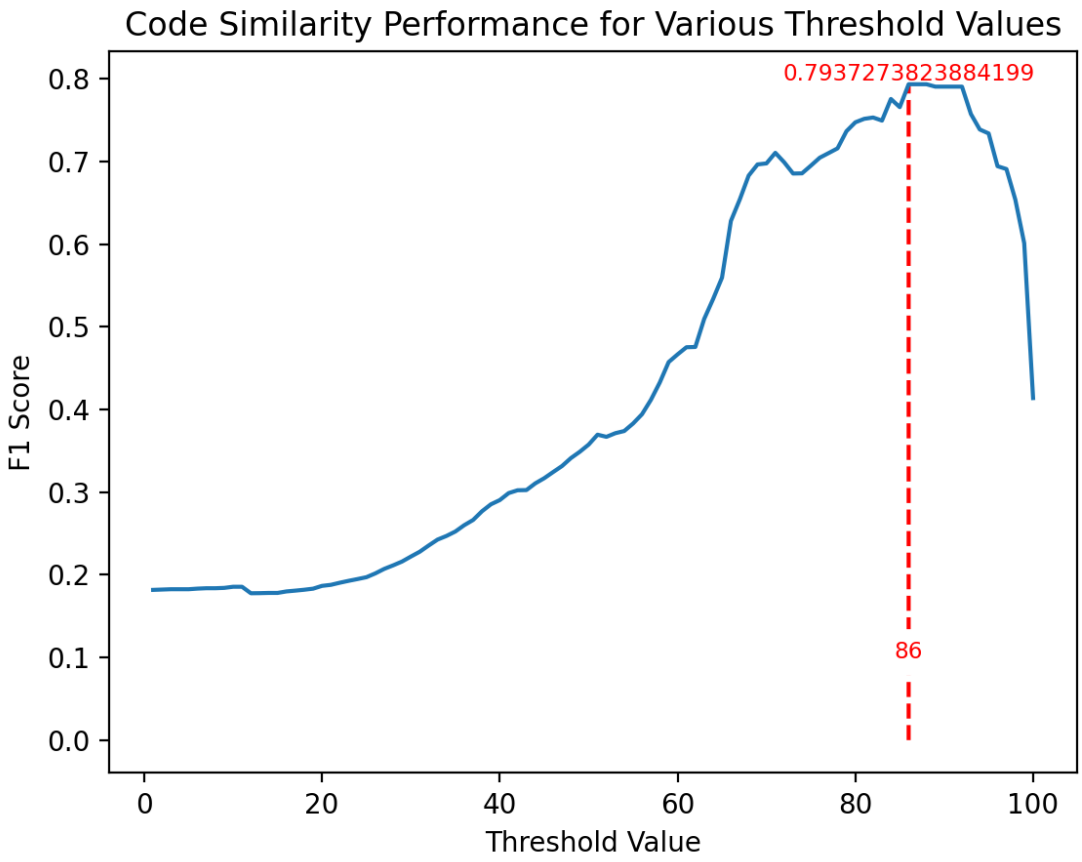
[4] Tech-Srl. "Tech-SRL/code2vec: Tensorflow Code for the Neural Network Presented in the Paper: 'code2vec: Learning Distributed Representations of Code.'" *GitHub*, https://github.com/tech-srl/code2vec

Appendices

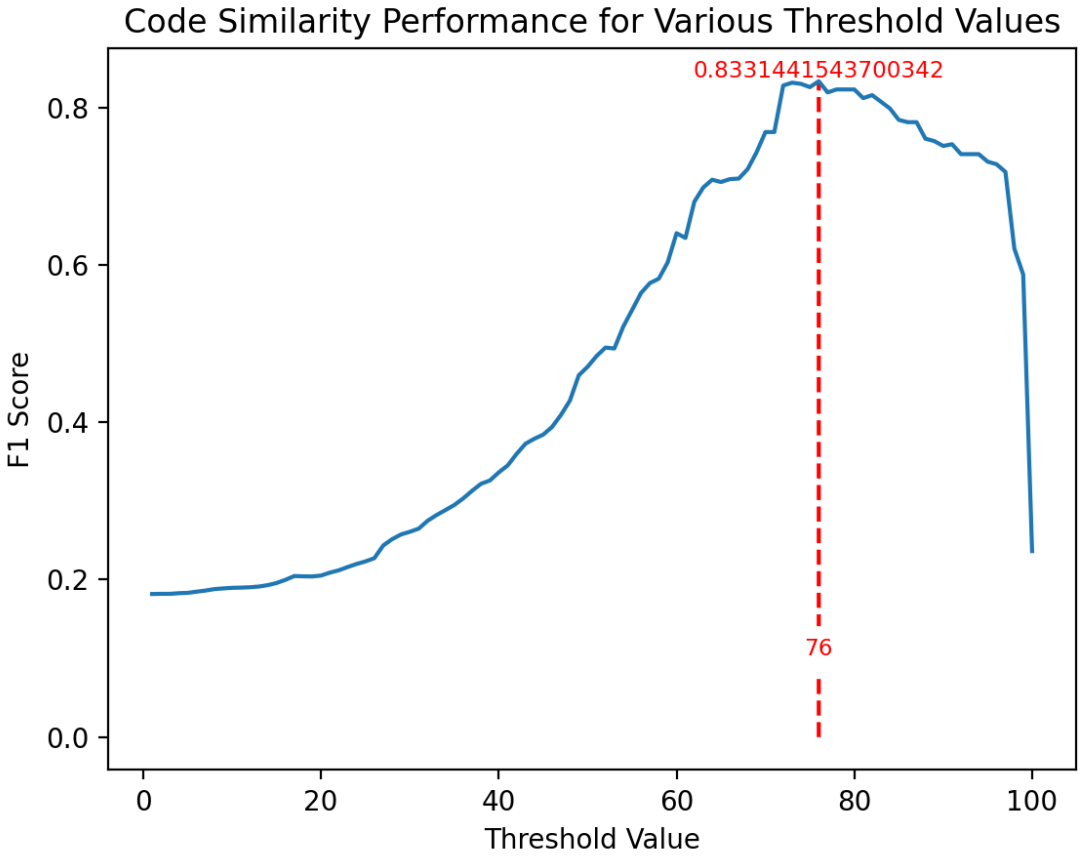
Java14 F1 Curves



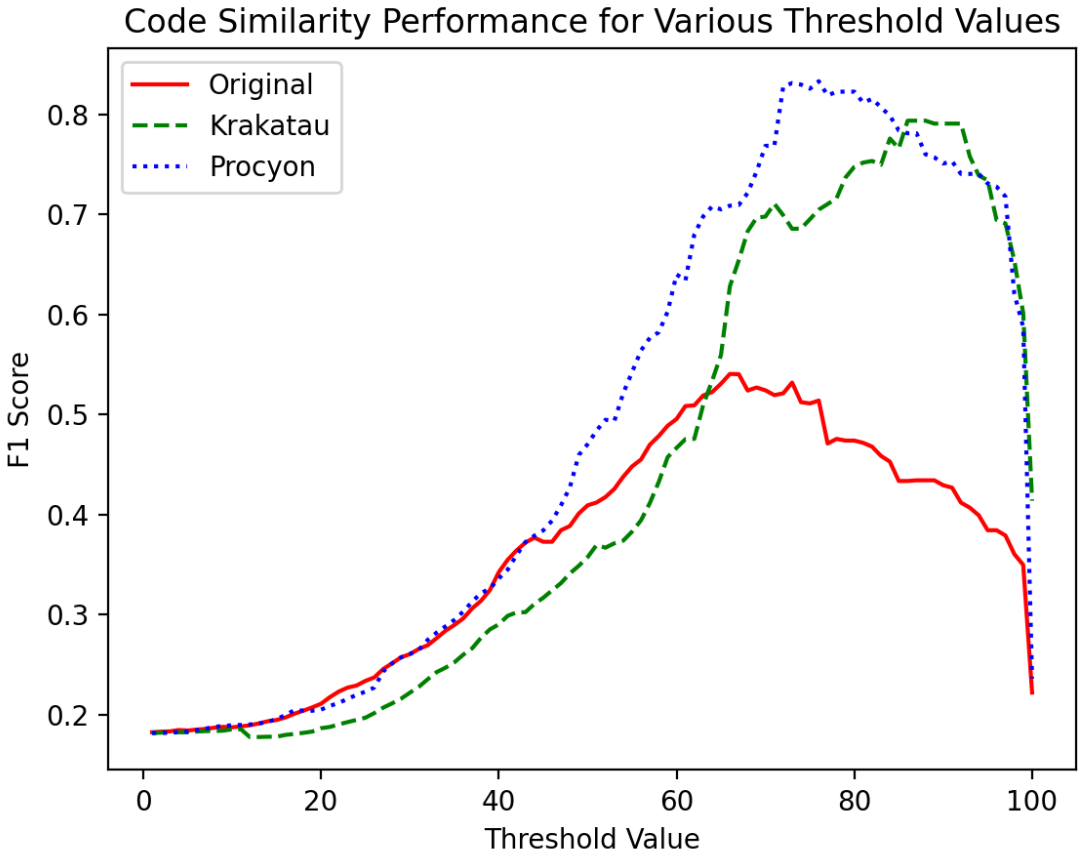
Graph 1: F1 Curve for Dataset Used in Experiment 1 (Original)



Graph 2: F1 Curve for Dataset Used in Experiment 2 (Krakatau)

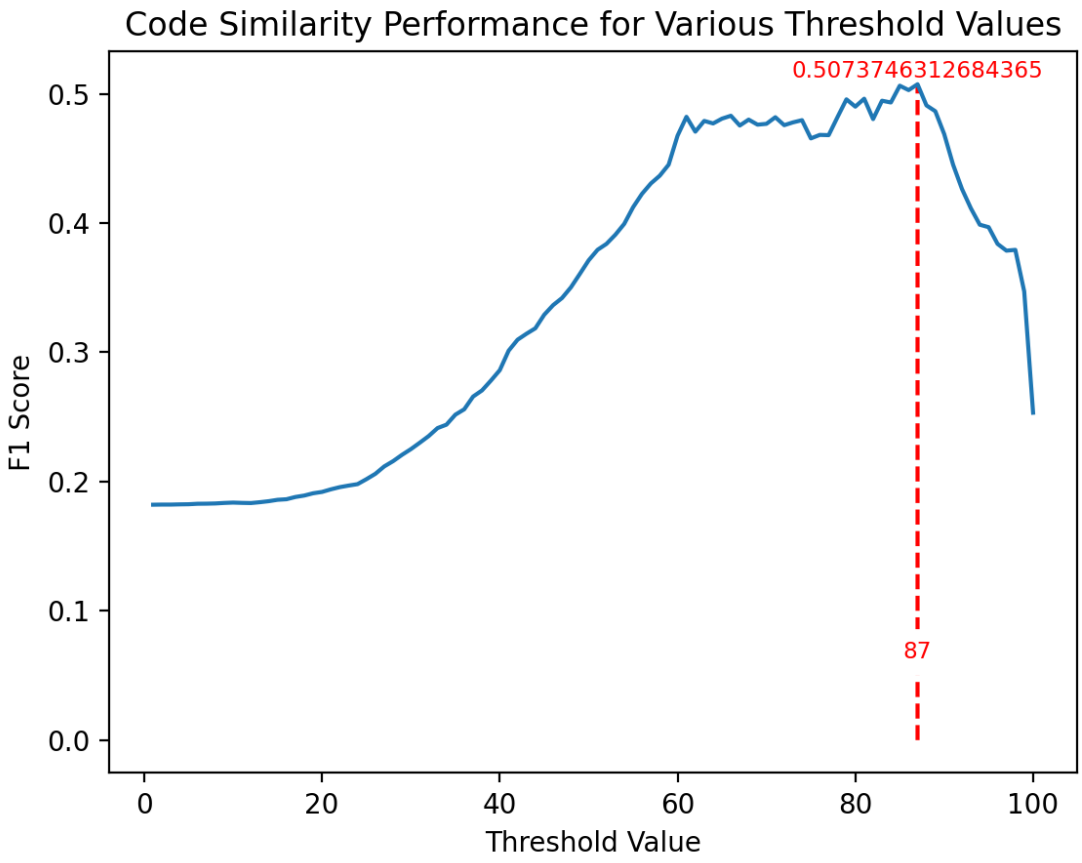


Graph 3: F1 Curve for Dataset Used in Experiment 3 (Procyon)

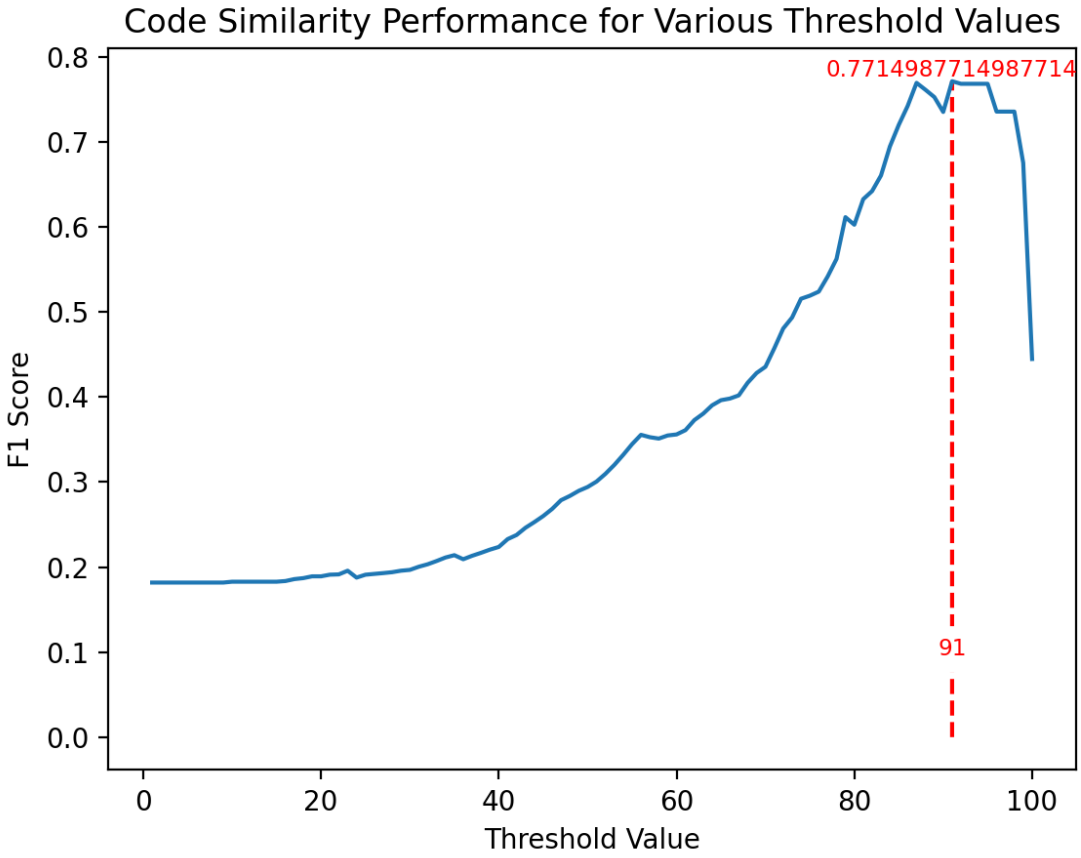


Graph 4: F1 Curves of All Three Datasets

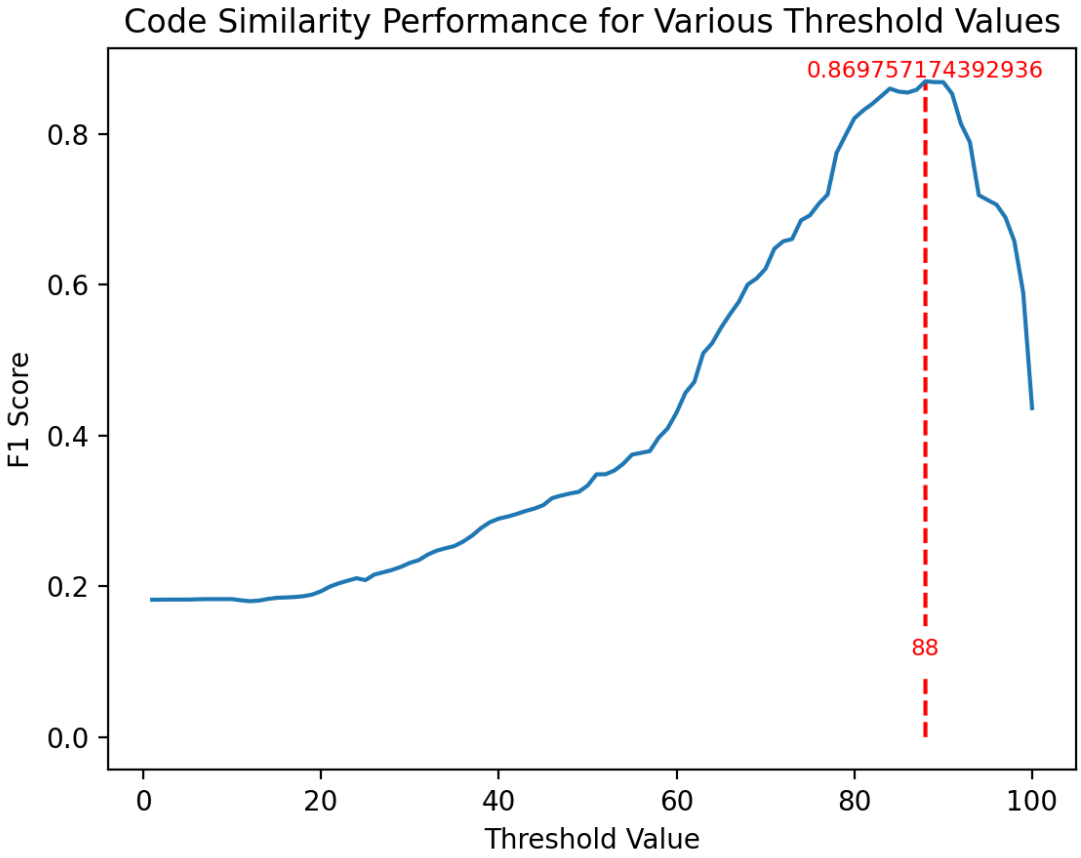
Java-Large F1 Curves



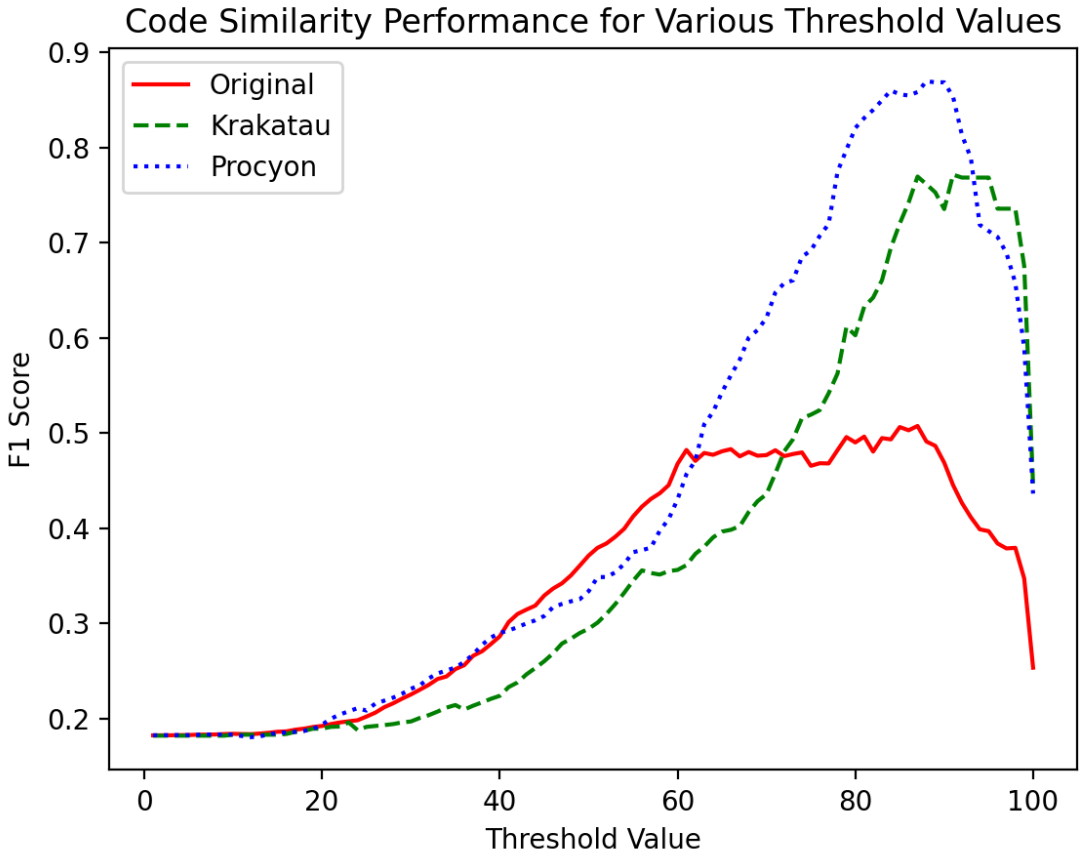
Graph 5: F1 Curve for Dataset Used in Experiment 1 (Original)



Graph 6: F1 Curve for Dataset Used in Experiment 2 (Krakatau)

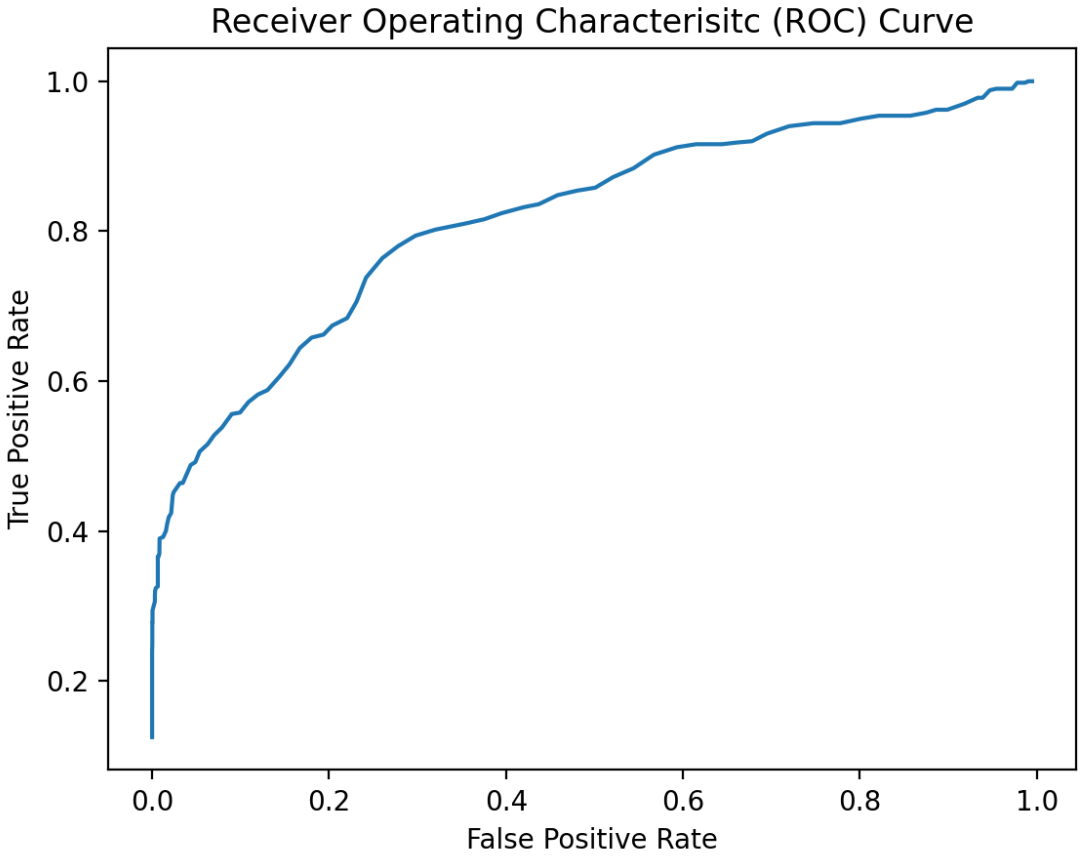


Graph 7: F1 Curve for Dataset Used in Experiment 3 (Procyon)

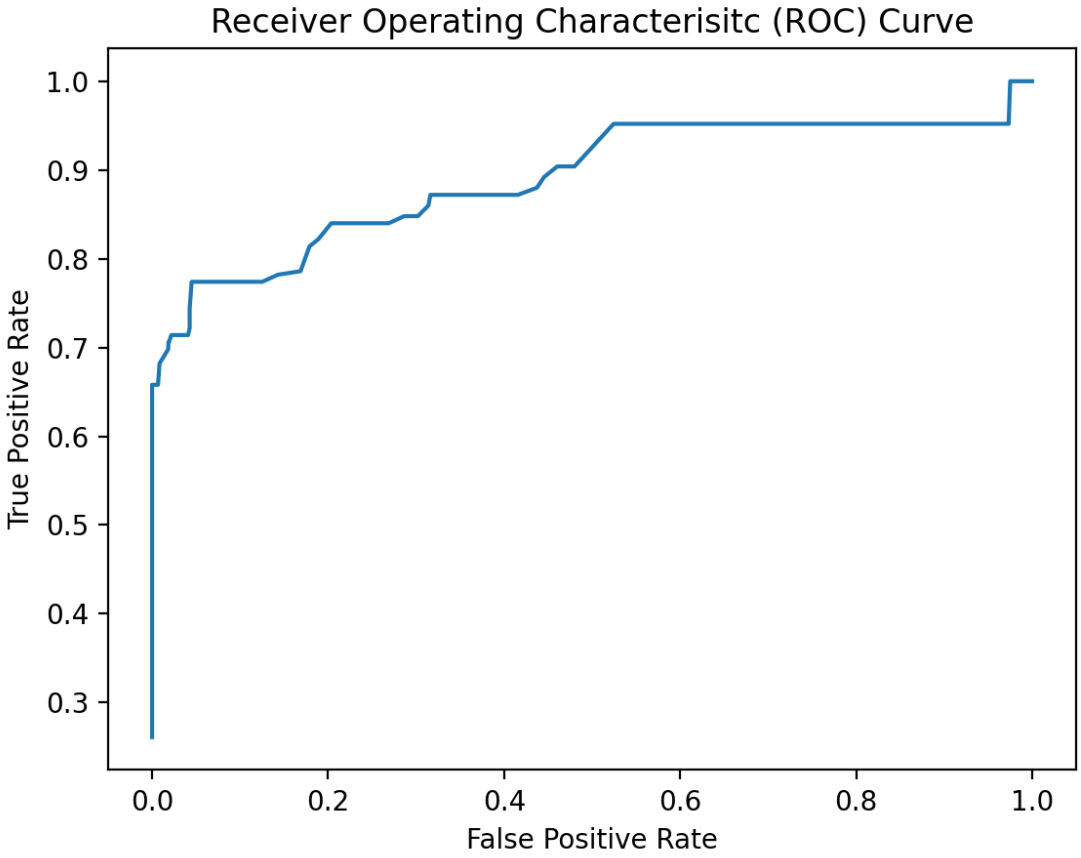


Graph 8: F1 Curves of All Three Datasets

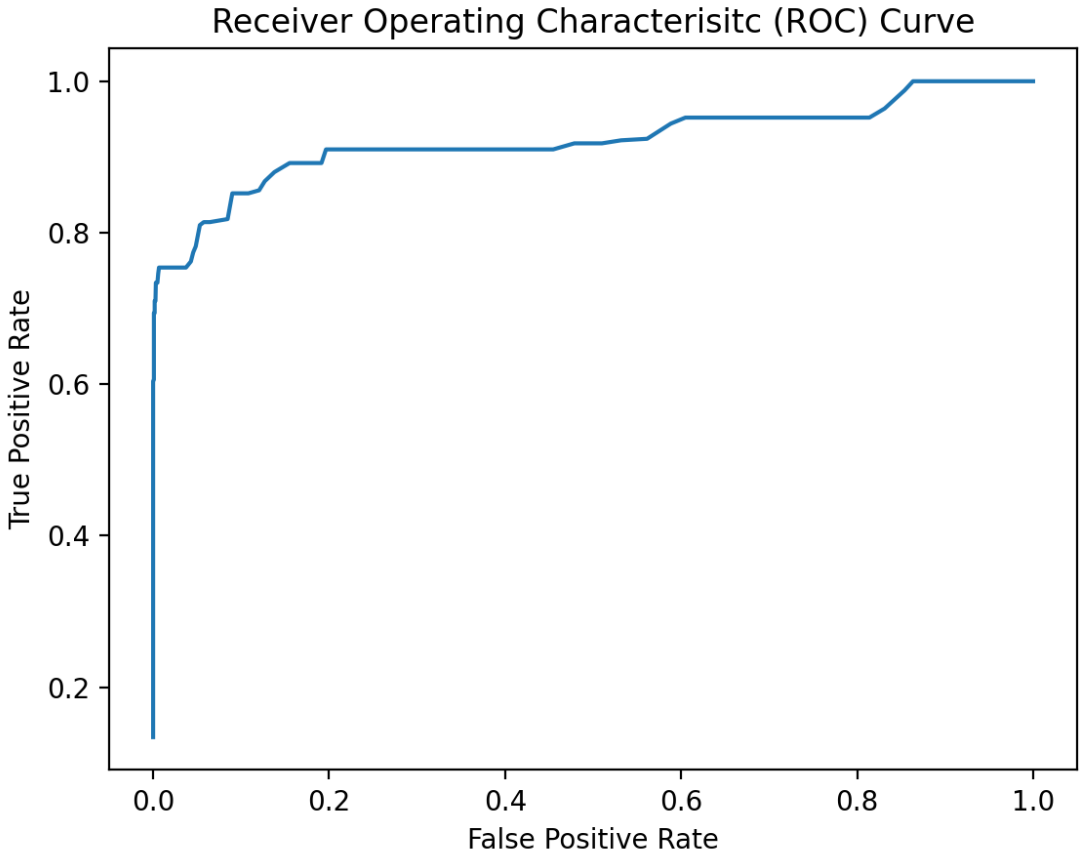
Java14 ROC Curves



Graph 9: ROC Curve for Dataset Used in Experiment 1 (Original)

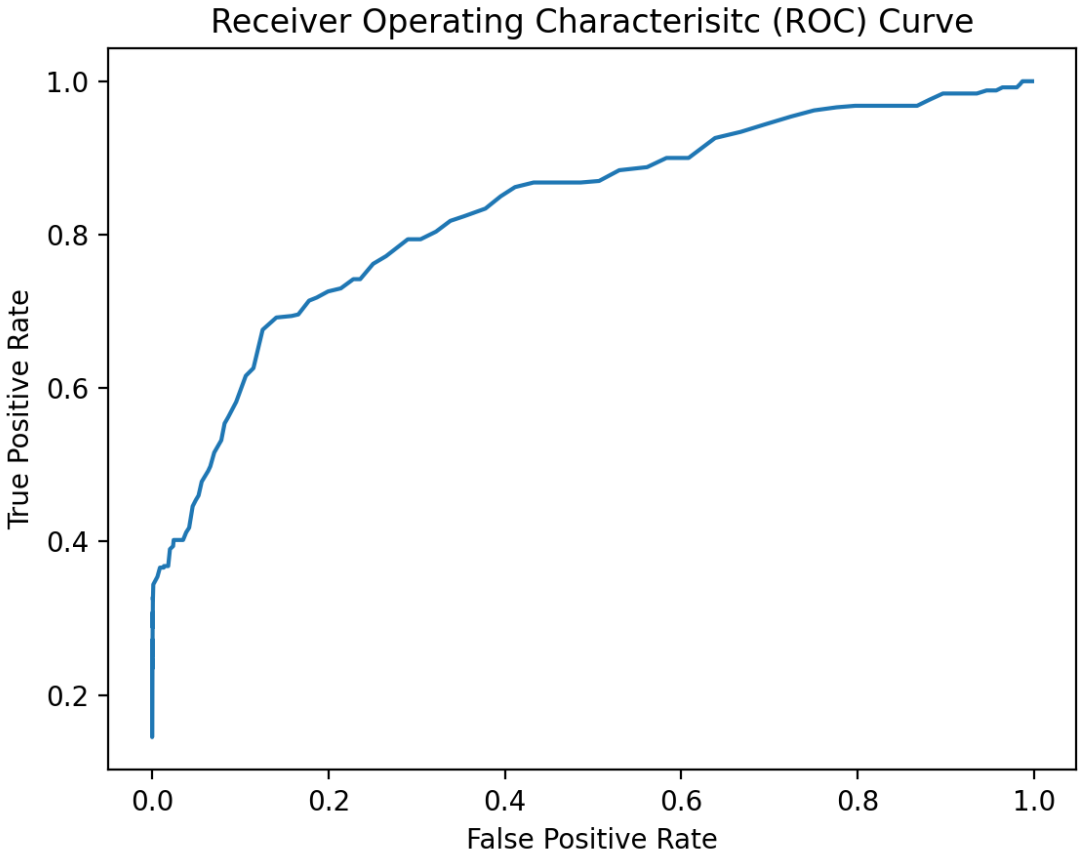


Graph 10: ROC Curve for Dataset Used in Experiment 2 (Krakatau)

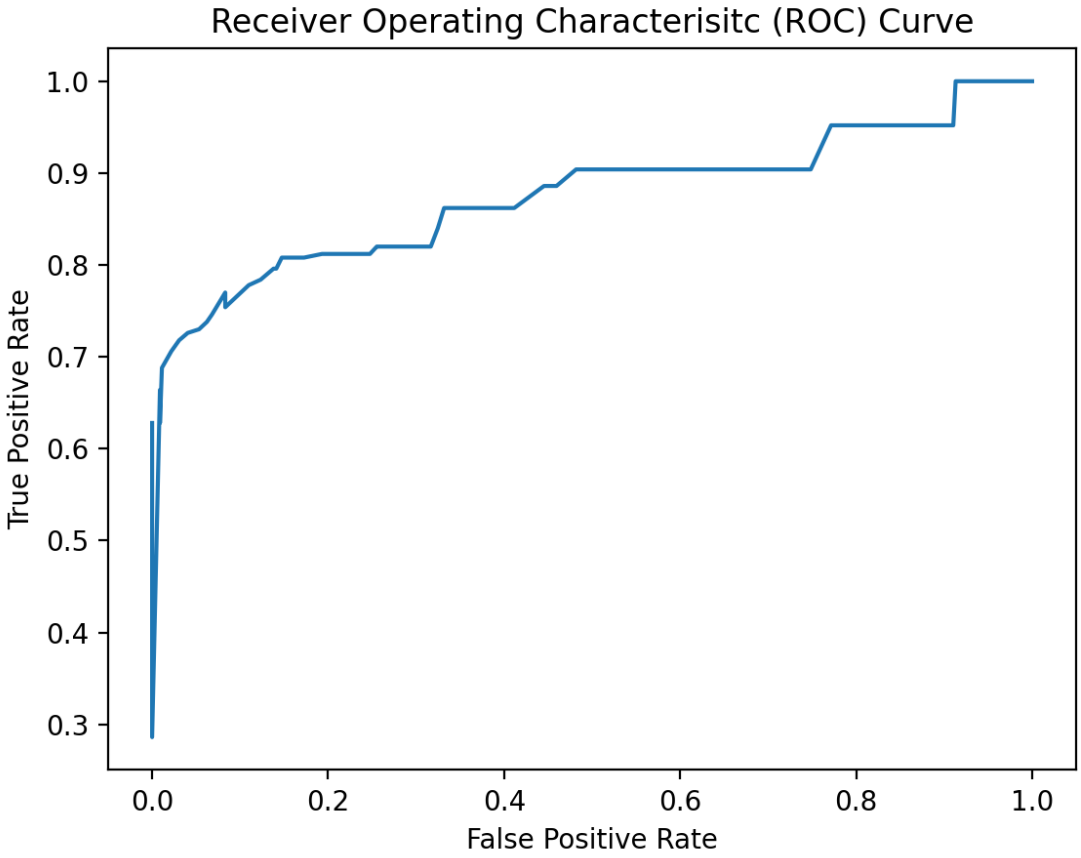


Graph 11: ROC Curve for Dataset Used in Experiment 3 (Procyon)

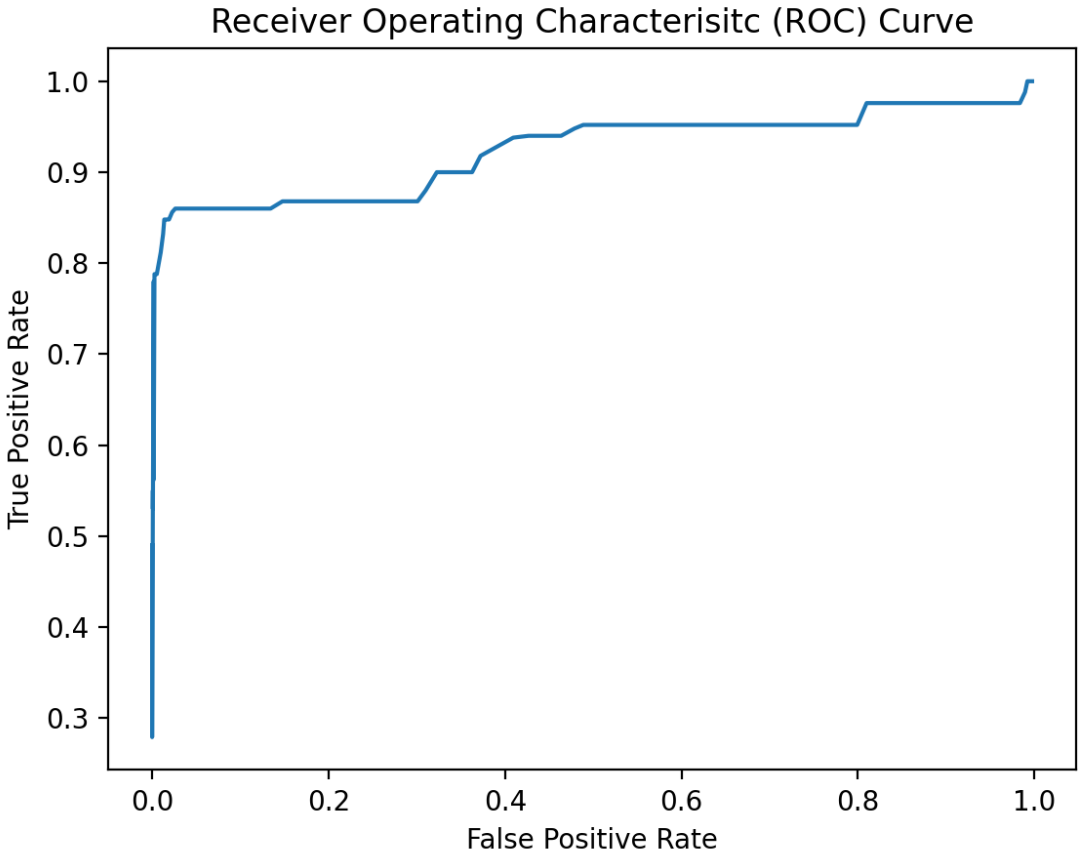
Java-Large ROC Curves



Graph 12: ROC Curve for Dataset Used in Experiment 1 (Original)

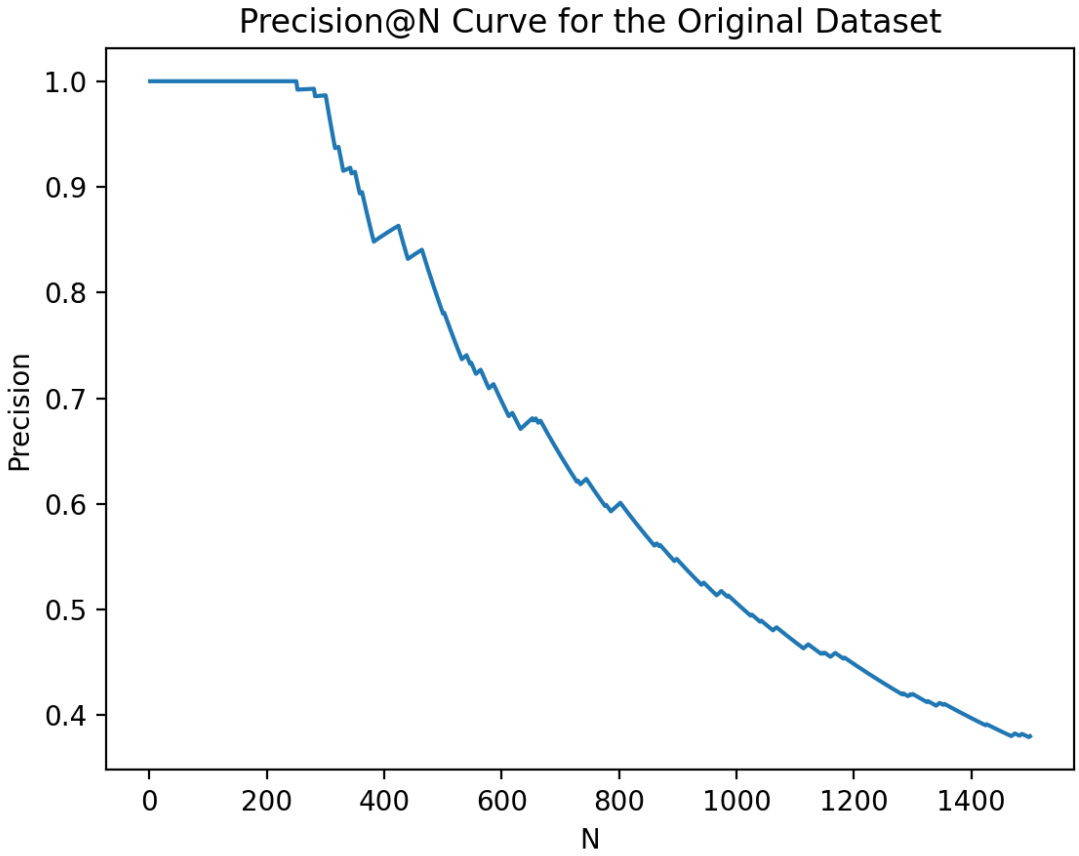


Graph 13: ROC Curve for Dataset Used in Experiment 2 (Krakatau)



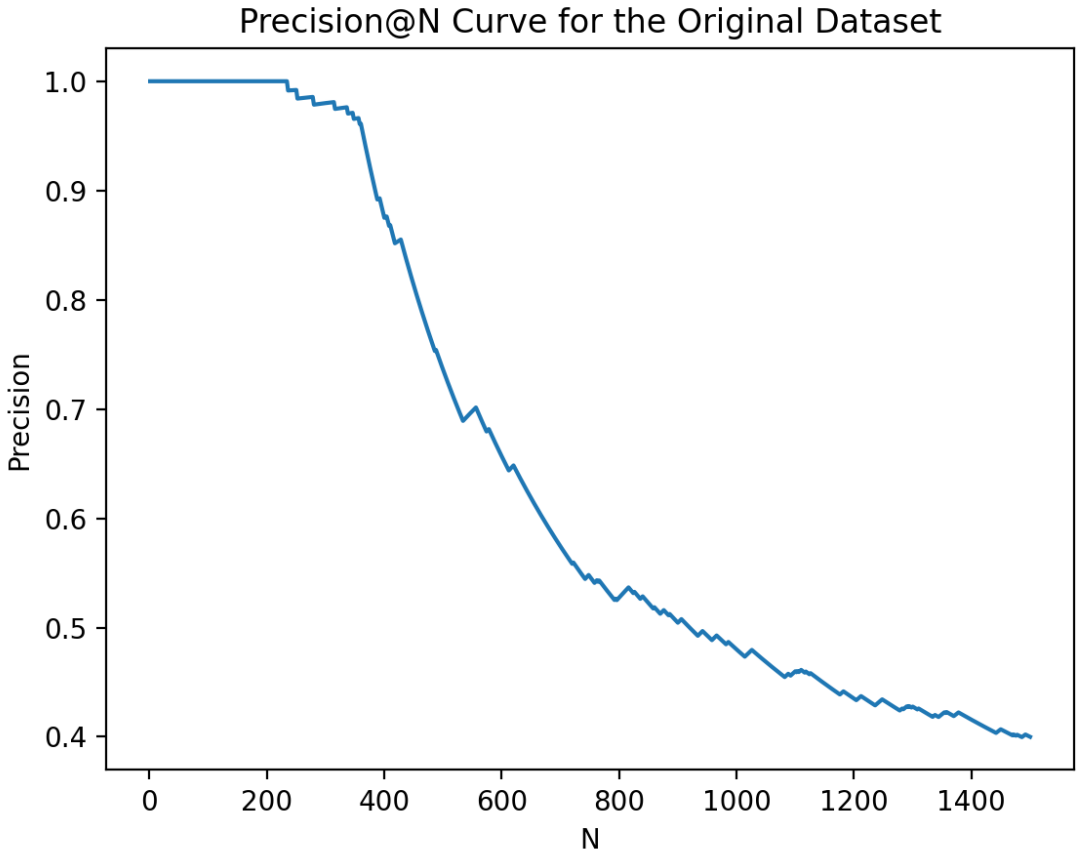
Graph 14: ROC Curve for Dataset Used in Experiment 3 (Procyon)

Java14 Precision@N



Graph 12: Precision@N for Dataset Used in Experiment 1 (Original)

Java-Large Precision@N



Graph 12: Precision@N for Dataset Used in Experiment 1 (Original)