UNIVERSITY OF CALIFORNIA,
IRVINE

Improving Mining Performance for Internet Code Search Engines

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Thomas Minsoo Kwak

Thesis Committee:
Professor André van der Hoek, Chair
Associate Professor James A. Jones
Associate Professor Sam Malek

2017

DEDICATION

To the people who helped shape me into the person I am today. My life would not be the same if we never met.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This thesis would not have been possible without the support from many people.

I thank André van der Hoek, my advisor, for guiding me throughout my graduate career with your wisdom and support. Thank you for giving me the opportunity to work in your lab and to be surrounded by incredible researchers. I have learned so much from you, and I still feel that I have so much left to learn.

I am very grateful for my committee members. Thank you, Jim Jones, for showing me the world of software engineering research and software testing through your courses. Thank you, Sam Malek, for giving me the opportunity to teach alongside you during the Internet Applications Engineering course. Teaching students and learning from them was one of the most fulfilling experiences I have ever had.

I am extremely grateful to Lee Martie, who mentored me throughout my graduate career. Thank you for letting me work alongside you and helping me grow both as a programmer and as a researcher. Thank you for closely helping me throughout the thesis process, from bouncing new ideas to revising my writing. Through all the highs and lows of our projects together, working with you is definitely the highlight of my graduate career.

I thank the members of my research group. Thank you, Fernando Spanghero, Consuelo López, Edgar Weidema, and Christian Adriano, for giving me support when I first joined the lab and during the early stages of my thesis. Thank you, Ayushi Rastogi Tariq Ibrahim, and Shinobu Saito, for giving me support each day and engaging in interesting research conversations.

To Panayiotes Kakleas, I thank you for introducing me to the world of programming and sticking by my side since middle school. Without you, I would have never tried out programming in the first place. I am extremely grateful for your constant support and for helping me grow as a person.

To Sonoko Kawakatsu, Katty Chou, Jennifer Ngo, Annie Lau, Danni Jiang, Vincent Tran, and Michelle Don, I thank you all for staying by my side and supporting my endeavors since college. I am grateful for all of the encouragement you have all given me over the years, and I hope that I can do the same someday.

To Neeraj Kumar, Jason Desrosiers, and Kishore Narendran, I thank you for being the best colleagues to work with during my time at UCI. All of you have helped me to become a better programmer and to finish this thesis.

To my siblings, Mike and Tiffany, and to my parents, Timothy and Stella, thank you for being supportive of my decision of going to graduate school. Thank you for motivating me to find my own path of what I want to do for the rest of my life.

ABSTRACT OF THE THESIS

Improving Mining Performance for Internet Code Search Engines

By

Thomas Minsoo Kwak

Master of Science in Software Engineering

University of California, Irvine, 2017

Professor André van der Hoek, Chair

To aid programmers in searching for code on the Internet, researchers and developers have created code search engines. These search engines use ranking algorithms to sort their results based on properties of the code. However, obtaining the properties to use a ranking algorithm requires a resource-intensive process, including downloading, parsing, analyzing, and indexing large amounts of code. Additionally, few guidelines exist for reducing the resources needed for this process without affecting the performance of the ranking algorithm. We explore two techniques for improving the process of mining code from the Internet for code search engines. First, we introduce What-If Ranking Analysis, a novel technique that attempts to find cheaper versions of a ranking algorithm by reducing the number of properties required when ranking. Second, we modify the original Beowulf cluster to optimize on network throughput instead of CPU performance to examine whether the modified cluster can outperform a single high-performance server in mining and uploading code for use in a code search engine. Our findings show that more efficient ranking algorithms exist that perform as well as the original ranking algorithm while reducing the time spent mining by 44% and reducing the disk space used for storage by 41%. Further, we find that the modified Beowulf cluster mines and processes code three times faster than a high-performance server at approximately the same cost.

Chapter 1

Introduction

When developing software, programmers search for code on the Internet regularly. They do this for a variety of reasons. One reason is that they may want to learn a design pattern [26], an example of which might be to learn how to represent the state of a board game. Another reason is that they might need to remember how to write a certain piece of code, either looking for previous code that they have written themselves or looking for a piece of code that they have seen before that does what they need [30, 33]. Finally, they may even search for code that they will want to use as-is so that they can copy the code verbatim into their ongoing project [7].

To search for code on the Internet, programmers frequently use general-purpose search engines (e.g., Google [10], Yahoo [36], Bing [3]), particularly because they are familiar with these search engines and know how to use them. These search engines index a vast amount of web pages, source code, tutorials, and other kinds of information that can be searched by the programmer. Programmers issue a query, typically by entering some keywords. The search engine then matches the query to documents it has indexed and returns matching documents as results to the user, usually as links to the original documents (e.g., Google

returns links to web pages). Based upon the results, the programmer might be content with what they find, might go back and submit an entirely new query, or might adjust some keywords to tailor the previous query. Overall, they follow a process like they do with any normal search.

General purpose search engines are not the only search engines available. Recently, a number of specialized code search engines have emerged (e.g., Krugle [14], Searchcode [27], GitHub Search [8]). Unlike general-purpose search engines, these specialized code search engines only index code and also attempt to present results in a manner more convenient to someone who is looking for code. For instance, Figure 1.1 shows Searchcode's interface after querying for quick sort [27]. While it still works like a general-purpose search engine in that the programmer types in a query as keywords, it differs in returning documents that are shown as code snippets rather than links.

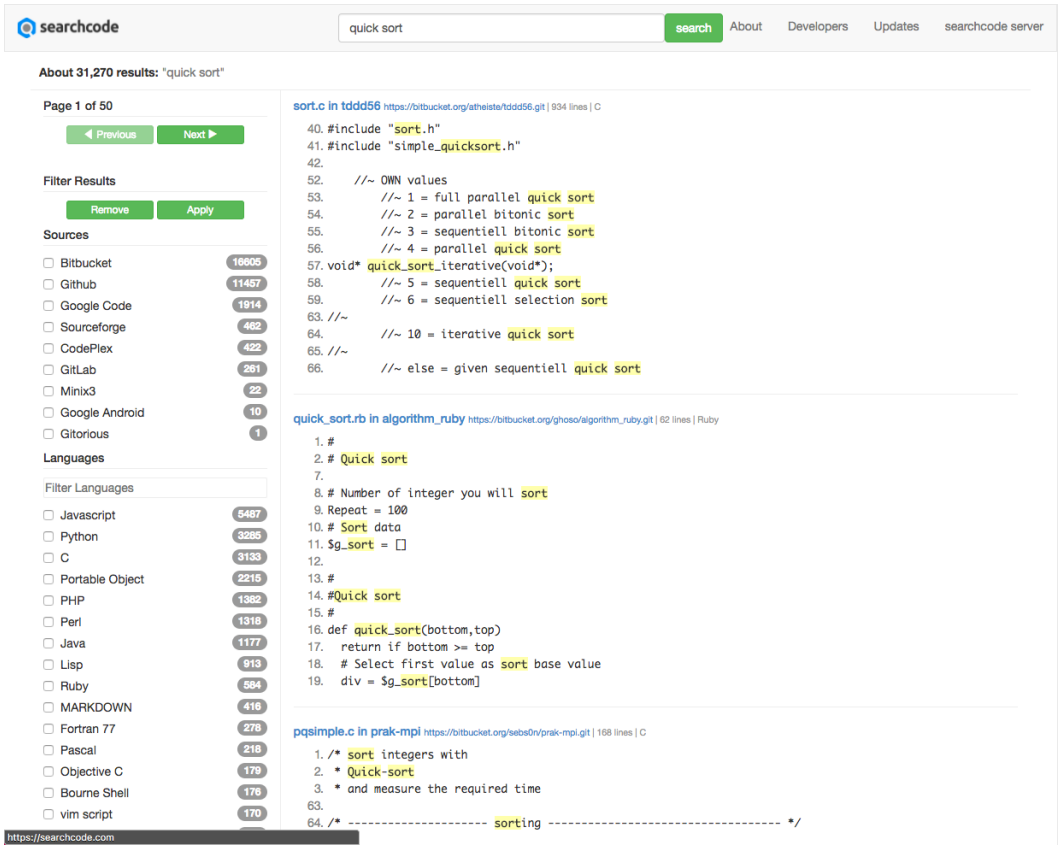

Figure 1.1: Searchcode search engine.

Not only have these specialized code search engines emerged in industry, researchers have also paid much attention to how to better serve the needs of programmers looking for certain types of code. This has taken a variety of forms. Examples of these forms include query expansion, such as through the use of synonyms when searching [17], and query reformulation, as seen in Sisman and Kak's automatic query reformulation [28] and CodeExchange's manual query reformulation support [18].

Regardless of the type of search engine, underneath it is ultimately a ranking algorithm. This ranking algorithm is responsible for providing an ordered list of results that best match the query that has been issued. It is no surprise that these ranking algorithms have received considerable attention. For instance, in Sourcerer, the ranking algorithm more heavily weighs matches against fully qualified names in code [16]. Another example is Example Overflow, which has a ranking algorithm that weighs matches based on terms occurring in the description of the code [37].

This thesis is about ranking algorithms. Specifically, this thesis focuses on two questions. The first question is the following: given a ranking algorithm that operates using multiple properties, does there exist a cheaper version of that ranking algorithm that uses only a subset of those properties? Particularly, in the ranking algorithm to diversity code results, called More With Less-ST Hybrid (MWL-ST Hybrid), it takes into consideration 17 properties of code [19]. For the ranking algorithm to work, these properties need to be downloaded, parsed, and analyzed. Is it possible to have the ranking algorithm working properly using fewer properties?

The second question concerns the fact that, even if we make the mining process cheaper by focusing on fewer properties, the mining process has many bottlenecks when using a single computer for parsing the properties from the source code. Examples of these bottlenecks include bandwidth throttling when downloading code from the Internet, or, if the index is actually stored elsewhere, uploading data fast enough so that the server that handles

indexing is rarely idle. Can we develop a method to reduce the amount of time the mining process takes while minimizing the cost to mine the code?

To address these two questions, we performed two experiments. The first experiment consisted of testing different permutations of properties for the MWL-ST Hybrid ranking algorithm and comparing its accuracy to that of the original ranking algorithm. For the second experiment, we explored a parallel architecture on commodity-based Raspberry Pis to understand whether this architecture could be more efficient in carrying out the mining process.

The remainder of the thesis is organized as follows. Chapter 2 presents background information on ranking algorithms, What-If Analysis, and Beowulf clusters. Chapter 3 presents our What-If Ranking Analysis. Chapter 4 presents our Beowulf cluster for code mining. Chapter 5 describes the experimental designs for two experiments, one for What-If Ranking Analysis and one for the Beowulf cluster. Chapter 6 presents results from both experiments. Chapter 7 presents the discussion. Chapter 8 discusses the threats to validity. Chapter 9 concludes by reviewing the contributions of this work as well as future directions for this work.

Chapter 2

Background

2.1 Ranking Algorithms

When a programmer issues a query to a search engine, the search engine matches the query to the documents stored in its index. To help select the best matching results, ranking algorithms are used to quantify each document by how well it matches the query. In code search engines, this matching occurs by weighing properties of each document and giving each document a score. The documents with the highest scores are then returned to the user.

Many different ranking algorithms are used in code search engines. Examples of different ranking algorithms include, but are not limited to, the following:

- CodeExchange's ranking algorithm uses Apache Lucene's version of term frequency-inverse document frequency [18];
- CodeGenie and Sourcerer both use the Specificity ranking algorithm, which uses fields such as the fully qualified names of code entities [2, 15];

- CodeLikeThis has a ranking algorithm that uses 17 properties, which includes the author's name and the method declaration names [19];
- Example Overflow's ranking algorithm uses the code's title, tag, answer, question, code, and social metadata to rank its results [37];
- Exemplar uses word occurrences, relevant API calls, and the structure of the API call [20];
- Kim et al. use a ranking algorithm that relies on similarity of the code to a cluster, conciseness of the code, and correctness of the code [13];
- ParseWeb ranks using the frequency of a method invocation and the length of the code [32]; and
- Sniff ranks results based on the number of occurrences of the code snippet across client source files [5].

What is important, however, is that each of these ranking algorithms represents a mathematical formula that uses different properties of each result. Because each ranking algorithm leverages different properties of the results, the results returned by each ranking algorithm tend to be different.

2.2 What-If Analysis

When trying to find the set of decisions that will produce the most desirable outcome, people must weigh each decision and understand its implications. To show the outcome of each set of decisions, What-If Analysis focuses on hypothetical situations involving these decisions to approximate the outcomes. An example use case of What-If Analysis is traveling from one place to another using either a car or a plane while minimizing the amount of money spent.

It is possible that the price for gas increases to the point that using a plane may be cheaper. It is also possible that airline companies could all increase their ticket prices, thus making the car a much cheaper option.

What-If Analysis is used across many different fields, most often in risk evaluation settings [1]. Other examples of usages of What-If Analysis are as follows:

- In software development, What-If Analysis can be used to optimize when the best time is to merge branches into the main product [4] or optimize individual MapReduce jobs to improve performance [12];
- In software release management, it can help in selecting the best release date for a product to maximize revenue [34];
- In biology, it is often used to manipulate and analyze molecular interactions in drug designs [35]; and
- In sustainability science, What-If Analysis is used to predict trends due to interactions between human and natural systems [31].

We will apply the idea of What-If Analysis to ranking algorithms. We would like to understand whether applying this technique to ranking algorithms can provide alternative ranking algorithms that use only a subset of properties while still maintaining the same accuracy as the original ranking algorithm.

2.3 Beowulf Clusters

In computing, there are large problems that require many calculations to complete. Parallel computing was developed to solve these larger problems in less time by dividing the large

problems into smaller problems, which can then be solved simultaneously. The problem with these systems is that they can become very expensive.

One solution to achieve high performance while keeping costs low is the Beowulf cluster. The Beowulf cluster, built at NASA in 1995 [29], is a cluster of commodity-grade computers connected together with a local area network. A problem can be split among these connected computers, which then can each solve a sub-problem. Originally used for earth and space sciences, the Beowulf cluster idea has been applied in multiple different scientific areas, such as mathematical modeling [9] and genome identification [11]. An example of what a Beowulf cluster looks like is shown in Figure 2.1 [21].

Figure 2.1: Beowulf Cluster at Michigan Technological University.

Using a Beowulf cluster of Raspberry Pis, we compare the speed of the mining process against a high-performance standalone computer. Because of the network bottlenecks that can appear in a high-performance standalone computer, we want to understand if having multiple network connections in our Beowulf cluster help increase the speed of the mining process.

Chapter 3

What-If Ranking Analysis

Applying the same techniques as What-If Analysis to the context of ranking algorithms, we introduce What-If Ranking Analysis. Analogous to What-If Analysis, where different decisions are chosen to change the outcome, What-If Ranking Analysis explores how different subsets of properties in the ranking algorithm change the results returned by a code search engine. For instance, if a ranking algorithm considers properties A, B, and C, then all possible subsets of properties for that ranking algorithm are A, B, C, $A+B$, $A+C$, $B+C$, and $A+B+C$.

By understanding how different subsets of properties change the results, it is possible to identify ranking algorithms that both perform as well as, or nearly as well as, the original ranking algorithm, yet use fewer properties. If such a ranking algorithm is found, then, because fewer properties are used, fewer computations would be required for ranking the results for each query and fewer properties of the code would need to be mined from the Internet. With fewer properties of the code mined, less time would be spent mining the code from the Internet, and less disk space would be needed to store the properties.

The general algorithm for What-If Ranking Analysis is as follows:

1. Select a ranking algorithm R, a sample set of queries Q, and a set of code snippets C to mine from the Internet.

2. While mining each property P_i in R from C, measure the time to mine each property as T^{P_i} and index the source code on its respective server as S^{P_i}.

3. For each sample query in Q, issue it to the search engine configured with R and record the top results as I^R.

4. Create all possible subsets of properties from R as $\{M_1, ..., M_n\}$.

   (a) This is accomplished by creating every unique permutation of properties found in R. Each permutation does not have to have every property in it. For example, if $R = a + b + c$, then there would be 6 possible permutations: a, b, c, $a + b$, $a + c$, $b + c$, and $a + b + c$.

5. For each possible subset $M_i \in M_1, ..., M_n$ and each query in Q, configure the search engine using M_i and record the top results as I^{M_i}. Measure the accuracy of M_i by measuring the average number of results that intersect between I^{M_i} and I^R for each query in Q, divide by the number of results returned in each query, and record as A^{M_i}. Calculate the time to mine the properties for a subset as $T^{M_i} = \sum_{j=1}^{k} T^{P_j}$ and the disk space necessary to store the properties as $S^{M_i} = \sum_{j=1}^{k} S^{P_j}$, where j is the set of properties in M_i and k is the number of properties in M_i.

6. For each interval of accuracy $[100, 100]$, $(100, 90]$, $(90, 80]$, $(80, 70]$, $(70, 60]$, and $(60, 50]$, return the ranking algorithm M_x that uses the least time to mine as T^{M_x}, and return the ranking algorithm M_y that uses the smallest amount of space S^{M_y} to support. For comparison, return the original ranking algorithm R with its time to mine T^R and space taken for its index S^R.

In this thesis, we apply What-If Ranking Analysis on the MWL-ST Hybrid ranking algorithm. We use this ranking algorithm because it uses a large number of properties of the code, which can be computationally expensive to examine every property. We use this ranking algorithm because it uses a large number of properties of the code, which can be computationally expensive to examine every property. The MWL-ST Hybrid ranking algorithm is shown in Figure 3.1.

$$
\begin{gathered}
sim_2^{ST}\left(D_i, D_j\right) = \sum \\
\left|authorName\left(D_i\right) \widehat{\cap} authorName\left(D_j\right)\right|, \\
\left|className\left(D_i\right) \widehat{\cap} className\left(D_j\right)\right|, \\
\left(\frac{\widehat{1}}{max\left(abs\left(complexity\left(D_i\right) - complexity\left(D_j\right)\right)\right)}\right), \\
\left(\frac{\widehat{1}}{max\left(abs\left(\left|fields\left(D_i\right)\right| - \left|fields\left(D_j\right)\right|\right)\right)}\right), \\
\left|hasWildCard\left(D_i\right) \widehat{\cap} hasWildCard\left(D_j\right)\right|, \\
\left|isAbstract\left(D_i\right) \widehat{\cap} isAbstract\left(D_j\right)\right|, \\
\left|isGeneric\left(D_i\right) \widehat{\cap} isGeneric\left(D_j\right)\right|, \\
\left|imports\left(D_i\right) \widehat{\cap} imports\left(D_j\right)\right|, \\
\left(\frac{\widehat{1}}{max\left(abs\left(\left|imports\left(D_i\right)\right| - \left|imports\left(D_j\right)\right|\right)\right)}\right), \\
\left|methodCallNames\left(D_i\right) \widehat{\cap} methodCallNames\left(D_j\right)\right|, \\
\left|methodDecNames\left(D_i\right) \widehat{\cap} methodDecNames\left(D_j\right)\right|, \\
\left|ownerName\left(D_i\right) \widehat{\cap} ownerName\left(D_j\right)\right|, \\
\left|package\left(D_i\right) \widehat{\cap} package\left(D_j\right)\right|, \\
\left|parentClass\left(D_i\right) \widehat{\cap} parentClass\left(D_j\right)\right|, \\
\left|projectName\left(D_i\right) \widehat{\cap} projectName\left(D_j\right)\right|, \\
\left(\frac{\widehat{1}}{max\left(abs\left(size\left(D_i\right) - size\left(D_j\right)\right)\right)}\right), \\
\left|variableWords\left(D_i\right) \widehat{\cap} variableWords\left(D_j\right)\right|
\end{gathered}
$$

Figure 3.1: More With Less-Social Technical Hybrid algorithm.

The MWL-ST Hybrid algorithm uses 17 properties of the code that can be extracted either from the code's abstract syntax tree or the code's GitHub repository. The algorithm returns the similarity between two documents by comparing the author name (the list of emails of

the people who contributed to the code from the Git commits), the name of the Java class, the complexity density (the cyclomatic complexity of the code divided by the number of lines of code), the Java fields inside of a class, whether the class uses wildcards, whether the class is abstract, whether the class is generic, the names of the import statements, the number of import statements, the method declaration names, the method invocation names, the person who owns the repository in which the code resides, the name of the Java package, the super class (if any), the name of the project (found from the GitHub repository), the size of the code (in number of characters), and the names of the variables used throughout the class.

Each property in the MWL-ST Hybrid algorithm is normalized to a number between 0 and 1 to prevent any property from affecting the similarity score more than another property. 0.5 is used with the complexity, number of fields, number of imports, and size to prevent division by 0 and, when normalized, returns to 1.

By testing different subsets of properties from the ranking algorithm, we explore whether there exists a cheaper ranking algorithm that uses fewer properties while maintain the same accuracy as the original. If a cheaper ranking algorithm existed while maintaining the same accuracy as the original, then each search that uses the ranking algorithm would be more efficient due to the fewer computations needed because of the fewer properties.

Additionally, we would like to see if subsets that produce higher accuracies have certain properties in common and how much of an effect higher accuracy has on time and space requirements when mining code from the Internet. In subsets that are as accurate as the original ranking algorithm, it is possible that some properties are left out. Therefore, those properties may not contribute to accuracy as much as other properties with respect to the original ranking algorithm. Also, some properties may take more time to mine from the Internet, while others may take more disk space to store. Finding the advantages and disadvantages of each property would help in selecting a subset of properties that takes the least amount of time to mine and the least amount of disk space to store.

Chapter 4

Beowulf for Code Mining

Mining code from the Internet can be a resource-intensive process. Bottlenecks can form at any step of the mining process when mining code from the Internet. For instance, downloading code from GitHub repositories is limited to 5,000 authenticated requests or 60 unauthenticated requests per hour [8]. Another example of a bottleneck can happen when parsing properties from the source code, which is limited by the time and space complexities of the program that parses the properties. As a final example, uploading the parsed properties to storage on a separate server is limited by a number of factors, such as the throughput and latency of the network connection between the server and the computers running the script and the write speed of the server's storage disk.

In this thesis, we aim to address the bottleneck of moving data from the Internet to a single local computer. When using a single computer, downloading repositories from GitHub is a slow process. This occurs because a single computer can only download one file at a time at the fastest network speed. Additionally, the network throughput is limited by the connection the computer has to the Internet (e.g., Ethernet, wireless).

A solution to this problem is to download the repositories in parallel. It is possible to clone

multiple repositories at once on a single computer, but the rate at which each repository is downloaded will be slowed because each repository will compete for bandwidth. Therefore, a parallel system consisting of multiple computers and independent Internet connections is necessary for parallel downloads. The independent Internet connections in the parallel system allow for data to travel across paths without slowing down network throughput, leading to a larger volume of data to travel from the Internet at any given point in time. By using multiple computers, the number of repositories downloaded at one time can be increased, which then increases the amount of data that can be sent to the server for indexing.

To implement this, we use a modified Beowulf cluster that optimizes on network throughput to increase the rate at which code is downloaded, parsed, and uploaded for storage. Like the Beowulf architecture, our Beowulf cluster uses a parallel computing system consisting of commodity-grade computers as seen in Figure 4.1. We are able to use commodity-grade computers for mining code because both parsing properties from the repositories and downloading GitHub repositories from the Internet are not CPU-intensive processes.

Figure 4.1: Beowulf cluster for mining code from the Internet.

However, unlike a traditional Beowulf cluster, each computer in our cluster is given its own Internet connection for downloading code repositories and uploading parsed data to storage. This allows n simultaneous repositories to be downloaded from GitHub, where n is the number of computers in our cluster. Because these repositories are independent from one another, our Beowulf cluster uses loosely coupled computers to independently parse the properties from the code.

In this thesis, we use our Beowulf cluster to maximize the network throughput so that more repositories are downloaded in less time. By doing so, the amount of code which is mined, parsed, and stored in an index will be greater than if we used a high-performance standalone computer. In addition, we compare the performance of a Beowulf cluster consisting of 43 Raspberry Pi 1 Model B+ against the performance of a high-performance standalone computer, both costing approximately the same amount of money, when mining code from the Internet.

Chapter 5

Experimental Design

This chapter presents the experimental designs for both the What-If Ranking Analysis experiment and the Beowulf code mining experiment. The first section presents the experimental design for the What-If Ranking Analysis experiment, covering how to find the accuracies of each subset of properties, selecting the sample of Java classes to mine, and measuring the time and space requirements for each property. The second section presents the experimental design for the Beowulf code mining experiment, showing the parameters for mining between the Beowulf cluster and the high-performance standalone computer.

5.1 What-If Ranking Analysis

The ranking algorithm used to test What-If Ranking Analysis is the MWL-ST Hybrid algorithm. This section explains the experimental design for What-If Ranking Analysis, the methods to measure the accuracies for each subset, and the methods to gather the time and space requirements for each subset.

To run these experiments, we used a Beowulf cluster of 43 Raspberry Pi 1 Model B+ com-

puters [25]. In addition, we used 2 high-performance standalone computers. All programs for parsing and uploading data to the server were written in Java.

Additionally, we previously mined and indexed 10M Java classes from GitHub repositories. For each Java class, the index contains its source code, a unique identifier, and the data for all 17 properties found in the MWL-ST Hybrid algorithm.

5.1.1 Accuracies of Each Subset of Properties

Table 5.1: Keyword queries.

database connection manager
ftp client
quick sort
depth first search
tic tac toe
api amazon
mail sender
array multiplication
algorithm for parsing string integer
binary search tree
file writer
regular expressions
concatenating strings
awt events
date arithmetic
JSpinner
prime factors
fibonacci
combinations n per k
input stream to byte array
spring rest template

To set up this experiment, we configured a code search engine with the MWL-ST Hybrid algorithm and the index of 10M Java classes. For querying the code search engine, we used the 21 keyword queries in Table 5.1, which were used as representative queries from multiple code search engines in a previous study [19].

For each keyword query, we needed to gather the ground truth, or the top 10 code results for a given keyword query when using the original ranking algorithm. The ground truth is used to compare whether another ranking algorithm returns the same results as the original ranking algorithm. To obtain the ground truth, we queried the code search engine using the original ranking algorithm and recorded the unique identifiers for each of the top 10 code results returned.

We created every possible subset of properties from the original ranking algorithm. We represented a subset as a 17-digit bit vector, where each bit represented one property. Table 5.2 shows the property represented at each position in the bit vector. A bit with value 1 would use the property in the ranking algorithm, and a bit with value 0 would not use the property in the ranking algorithm. For example, the bit vector 10000000000000001 is a subset that uses the Import Names property and the Project Owner Name property. As another example, the bit vector 11111111111111111 is a subset that uses all properties, which is equivalent to the original ranking algorithm.

Table 5.2: Property represented at each position in bit vector.

Code Property	**Position in Bit Vector**
Import Names	1
Variable Names	2
Class Names	3
Author Name	4
Project Name	5
Method Invocation Names	6
Method Declaration Names	7
Size	8
Number of Imports	9
Cyclomatic Complexity	10
Parent Class Name	11
Package Name	12
Field Names	13
Is Generic	14
Is Abstract	15
Has Wild Cards	16
Project Owner Name	17

In total, there were 131,072 unique subsets, which is derived from 2^{17} total permutations by removing any duplicates. To use a subset in the code search engine, we configured the code search engine to accept the 17-digit bit vector as a query parameter in addition to the keyword query. The ranking algorithm will then search by weighing only those properties noted in the bit vector.

After querying the code search engine with both the keyword query and the 17-digit bit vector, the number of matching IDs between the top 10 returned results and the ground truth were recorded in a file. Figure 5.1 shows a snippet from one file, where the bit vector is shown to the left of the underscore, and the number of matching IDs is shown to the right.

After all of the subsets were queried with all queries outlined in Table A, we calculated the accuracy of each subset with respect to the original ranking algorithm. The number of matching IDs were summed across all queries and divided by the total number of possible

```
00000000000010110_5
00000000000010111_5
00000000000011000_5
00000000000011001_5
00000000000011010_5
00000000000011011_5
00000000000011100_5
00000000000011101_5
00000000000011110_5
00000000000011111_5
00000000000100000_4
00000000000100001_3
00000000000100010_4
00000000000100011_4
00000000000100100_4
00000000000100101_4
00000000000100110_4
```

Figure 5.1: Bit vector (left) to number of matching IDs to the ground truth (right).

matching IDs. Since we had 21 queries with each query returning 10 results, the total number of possible matching IDs is 210.

5.1.2 Selecting the Sample Java Classes for Mining

To estimate the amount of time each property takes to mine and the amount of disk space each property consumes relative to one another, we selected a sample from the 10M Java classes in the index of the code search engine used in 5.1.1, as performing the experiment with all 10M classes is too prohibitive. Rather, we must repeat each analysis with some subset of classes.

To select the sample size, we picked a number between an upper bound of the number of projects on GitHub and a lower bound created using the pwr package in R [23]. We calculated a statistically significant lower bound of 4,607 Java classes. We selected 146,200 because this number seemed achievable during a week of mining Java classes and it was an easy number to divide amongst each computer in the Beowulf. We then selected 146,200 URLs from the

10M Java classes previously mentioned.

5.1.3 Mining the Properties from the Sample Java Classes

Prior to mining each property, an Apache Solr instance was created for each property for storing the data for each individual property. Using the Beowulf cluster, the 146,200 repository URLs were evenly distributed to each node. The format for each URL is seen in Figure 5.2.

```
https://raw.github.com/[author]/[project]/[commit hash]/[path]/[file]?start=[number]&end=[number]
```

Figure 5.2: URL Format.

To mine the URL, the repository was cloned to a local directory. The property data was extracted for each property using two methods. For technical properties, an abstract syntax tree was created from the file using the Eclipse Java Development Tools API [6]. For social properties, we queried our 10M Java class server. After gathering either the technical data or social data, the data was uploaded to its respective Apache Solr instance.

During the mining process, we also timed how long each individual action took. The start and end times of the following actions during the mining process were recorded in a text file as Unix timestamps:

- Starting the program to parse the data from the repository;
- Cloning the repository;
- Creating variables for storing the parsed data;
- Parsing the source code and creating the abstract syntax tree;
- Extracting social properties;

- Extracting technical properties; and
- Uploading the parsed data to the server for indexing.

An example of the text file is shown in Figure 5.3. Each process was delimited using different punctuation marks to indicate different pieces of information about the process. For example, the double colon when used after an action (i.e., ::) is followed by the Unix timestamp. The text between the two dollar symbols (i.e., $$) and the double colon is the subset of properties currently being mined.

```
URL::https://raw.github.com/uw-it-aca/spacescout-android/
bc29ac0fa16933623434c618635c67615ac93dc1/SpaceScout/src/main/java/edu/uw/
spacescout_android/CustomDialogFragment.java?start=4871&end=5010
Started process::1485414357664
Started cloning::1485414357672
Finished cloning::1485414361669
Started setup after cloning::1485414361671
Finished setup after cloning::1485414361796
Started creating empty solr doc$$00000000000000010::1485414361800
Finished creating empty solr doc$$00000000000000010::1485414362049
Started source code$$00000000000000010::1485414362050
Finished source code$$00000000000000010::1485414362185
Started technical$$00000000000000010::1485414362186
Finished technical$$00000000000000010::1485414371030
Started social$$00000000000000010::1485414371041
Finished social$$00000000000000010::1485414371042
Started uploading$$00000000000000010::1485414371446
Finished uploading$$00000000000000010::1485414373683
Started creating empty solr doc$$00010000000000000::1485414373684
Finished creating empty solr doc$$00010000000000000::1485414373688
Started source code$$00010000000000000::1485414373689
Finished source code$$00010000000000000::1485414373698
Started technical$$00010000000000000::1485414373698
Finished technical$$00010000000000000::1485414373699
Started social$$00010000000000000::1485414373700
Finished social$$00010000000000000::1485414373736
Started uploading$$00010000000000000::1485414373738
Finished uploading$$00010000000000000::1485414374831
Started creating empty solr doc$$00001000000000000::1485414374833
Finished creating empty solr doc$$00001000000000000::1485414374834
```

Figure 5.3: Output file from mining repositories with timestamps for each process.

5.1.4 Measuring Time and Space Usage for Each Property

The files generated in 5.1.3 contained the timestamps to parse and upload a single property. The difference in time between uploading and parsing were calculated using the timestamps, giving the total time to parse each property.

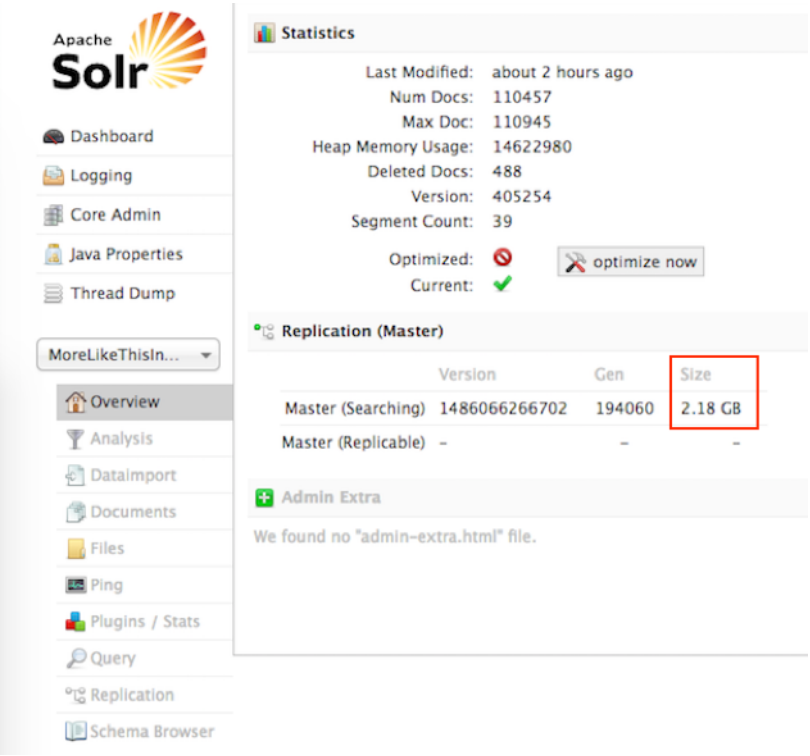

Figure 5.4: Size of Apache Solr index.

The disk space usage of each property was estimated as the size of the Apache Solr index to which each property was uploaded. In Figure 5.4, the size of the index is outlined with a red square.

5.2 Comparison of Performances

To compare the performance between the Beowulf cluster and the high-performance standalone computer for downloading, parsing, and uploading property data, we compared the number of classes that could be mined by either system within a 5-day period. The classes to be mined were the same 146,200 GitHub URLs used in Section 5.1.2. Both systems had the tasks of cloning the repository, traversing to the exact class file, parsing the property data, and uploading the parsed data to another server on the local network for indexing.

The programs used to carry out these tasks were identical for both systems.

We used three computing systems for this experiment: our Beowulf cluster and two different high-performance standalone computers (HPC1 and HPC2). The Beowulf cluster uploaded the parsed data to HPC2, and the high-performance standalone computer HPC2 uploaded to HPC1[1]. Each test was run independently from one another.

To show the differences in CPU and disk performances between the three systems used in this experiment, we use multiple tests from the benchmarking suite, Phoronix Testing Suite [24]. We use two I/O intensive tests (Flexible-IO and IOzone) to show the read and write speeds of the disks of each computer in Figures 5.5, 5.6, and 5.7, and we use two CPU intensive tests (FFTW and N-Queens) to show the CPU performance of each computer in Figures 5.8 and 5.9.

HPC2 had the highest disk read and write speeds out of all computers at 8163.13 MB/s and 116.27 MB/s, respectively. HPC1 had the second highest disk read and write speeds at 5167.34 MB/s and 93.25 MB/s, respectively. The Raspberry Pi Model 1 B+ had the lowest disk read and write speeds at 18.09 MB/s and 12.31 MB/s, respectively.

For CPU performance, HPC2 also had the best performance with 5628.68 MFLOPs. HPC1 had the second performance at 3331.68 MFLOPs. The Raspberry Pi Model 1 B+ had the lowest performance at 71.37 MFLOPs. Given an N-Queens problem, HPC2 solved the problem in 40.26 seconds, HPC1 solved the problem in 42.93 seconds, and the Raspberry Pi Model 1 B+ solved the problem in 1557.92 seconds.

[1]The difference in setups was due to experimental necessity, as the computers were in use for other experiments, too. This causes an imbalance in one-to-one comparisons between our Beowulf cluster and HPC1. However, HPC2 is faster in CPU performance, disk read speeds, and disk write speeds than HPC1, so this should not affect the results.

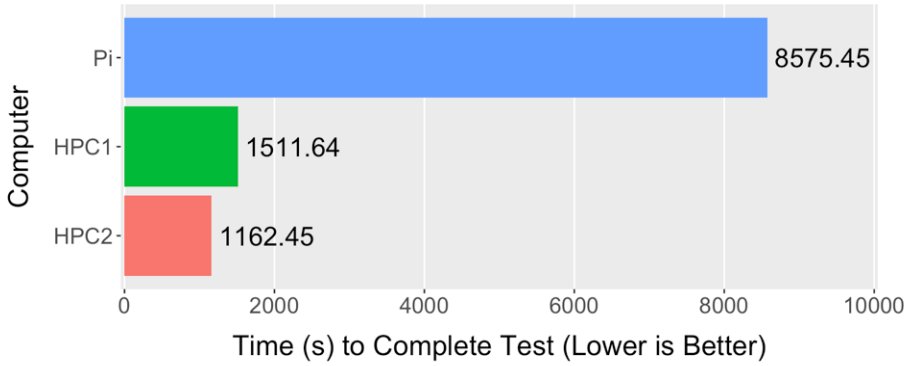

Figure 5.5: Flexible I/O disk benchmark results.

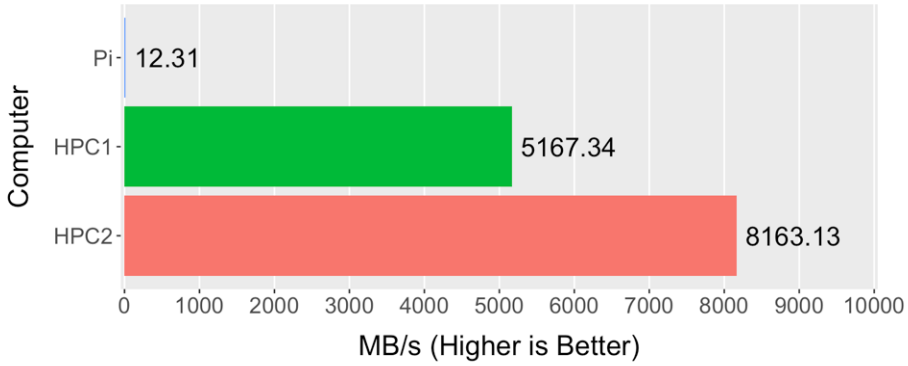

Figure 5.6: IO-Zone disk read speed benchmark results.

Figure 5.7: IO-Zone disk write speed benchmark results.

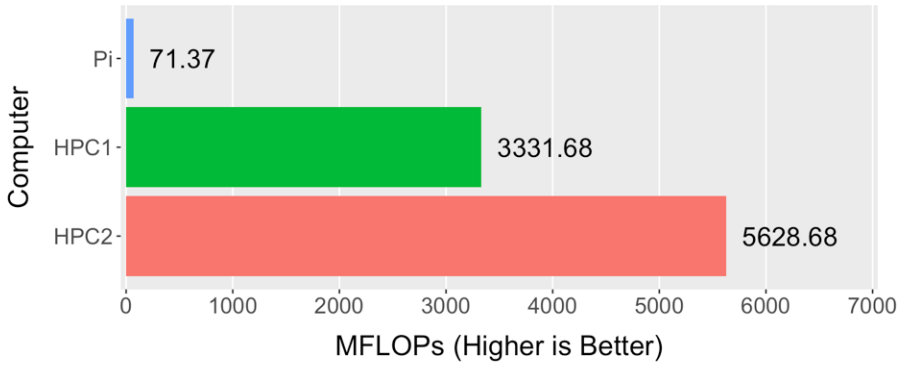

Figure 5.8: FFTW benchmark results.

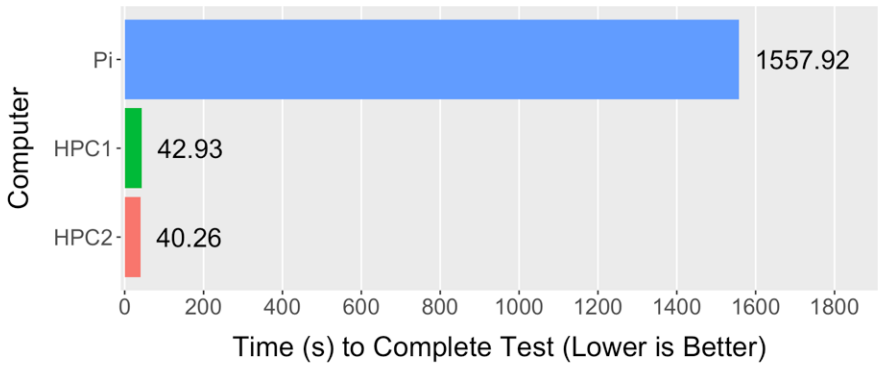

Figure 5.9: N-Queens benchmark results.

Chapter 6

Results

This chapter presents the results from both the What-If Ranking Analysis experiment and the Beowulf code mining experiment. The first section presents the results from the What-If Ranking Analysis: time to mine each property individually, index size of each property, and the subsets of properties with the lowest times to mine and smallest index sizes. The second section presents the results from the Beowulf code mining experiment, showing the difference in mining speeds between the Beowulf cluster and the high-performance standalone computer.

6.1 What-If Ranking Analysis Results

In total, we mined 110,437 Java classes out of the 146,200 possible Java classes. This was caused by some of the computers in the Beowulf cluster shutting down due to problems, such as out-of-memory errors or hard disks failures.

6.1.1 Time to Mine Individual Properties

Figure 6.1 shows the time to mine each property individually from the sample of 110K Java classes. Figure 6.2 is a zoomed in graph of Figure 6.1, showing the difference in time to mine for Properties 1-3 and Properties 6-15. Table 6.1 shows each property's time to mine rounded to the nearest thousandth of a day.

Table 6.1: Time to mine each code property individually.

Code Property	**Time to Mine (days)**
Import Names	1.62
Variable Names	1.58
Class Names	1.57
Author Name	0.66
Project Name	0.28
Method Invocation Names	1.59
Method Declaration Names	1.65
Size	1.57
Number of Imports	1.69
Cyclomatic Complexity	1.56
Parent Class Name	1.63
Package Name	1.60
Field Names	1.59
Is Generic	1.56
Is Abstract	1.56
Has Wild Cards	2.49
Project Owner Name	0.23

The property that took the most amount of time to mine was Property 16, Has Wild Cards, taking 2.49 days. The property that took the least amount of time to mine was Property 17, Project Owner Name, taking 0.23 days. The mean time to mine one property was 1.44 days, and the median time to mine one property was 1.58 days.

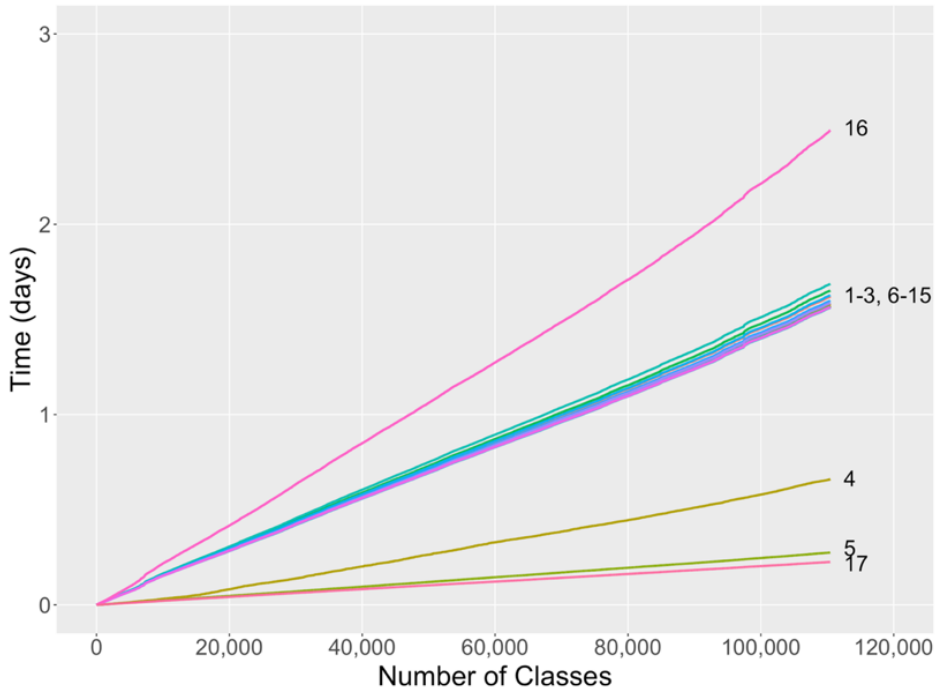

Figure 6.1: Time to mine each property across 110K Java classes.

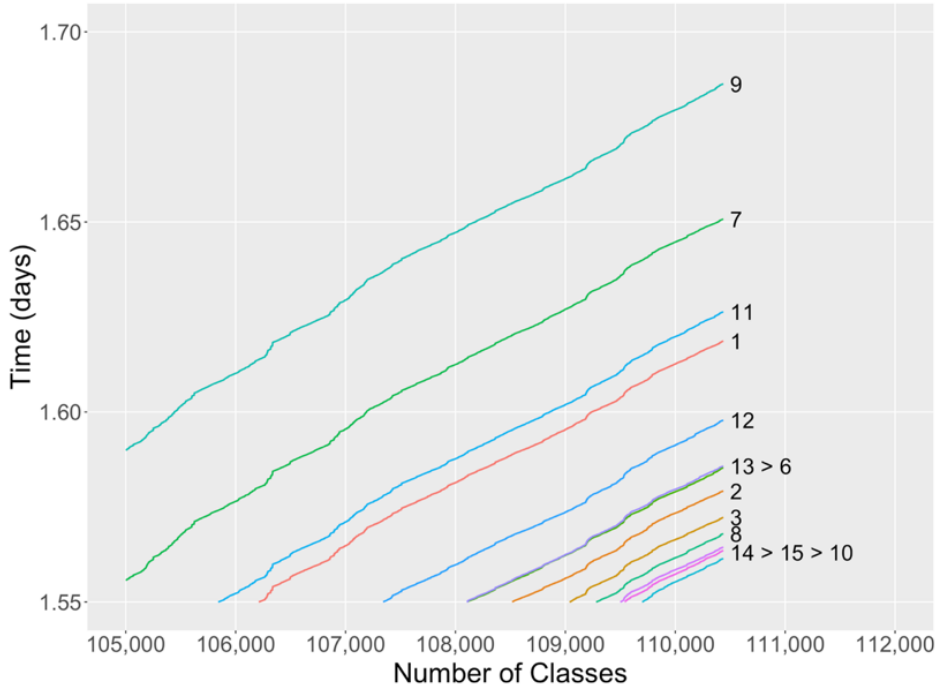

Figure 6.2: Time to mine Properties 1-3 and 6-15 across 110K Java classes.

6.1.2 Index Size of Individual Properties

Table 6.2 shows each property's index size rounded to the nearest hundredth, and Figure 6.3 shows the same information as a histogram.

Table 6.2: Index size for each code property across 110K Java classes.

Code Property	**Index Size (GB)**
Import Names	2.19
Variable Names	2.18
Class Names	2.09
Author Name	2.15
Project Name	2.17
Method Invocation Names	2.17
Method Declaration Names	2.18
Size	2.16
Number of Imports	2.18
Cyclomatic Complexity	2.03
Parent Class Name	2.29
Package Name	2.34
Field Names	2.12
Is Generic	2.05
Is Abstract	2.07
Has Wild Cards	2.18
Project Owner Name	2.13

The property that had the largest index size was Property 12, package name, with an index size of 2.34 GB. The property that had the smallest index size was Property 10, cyclomatic complexity, with an index size of 2.03 GB. The mean index size was 2.16 GB, and the median index size was 2.12 GB.

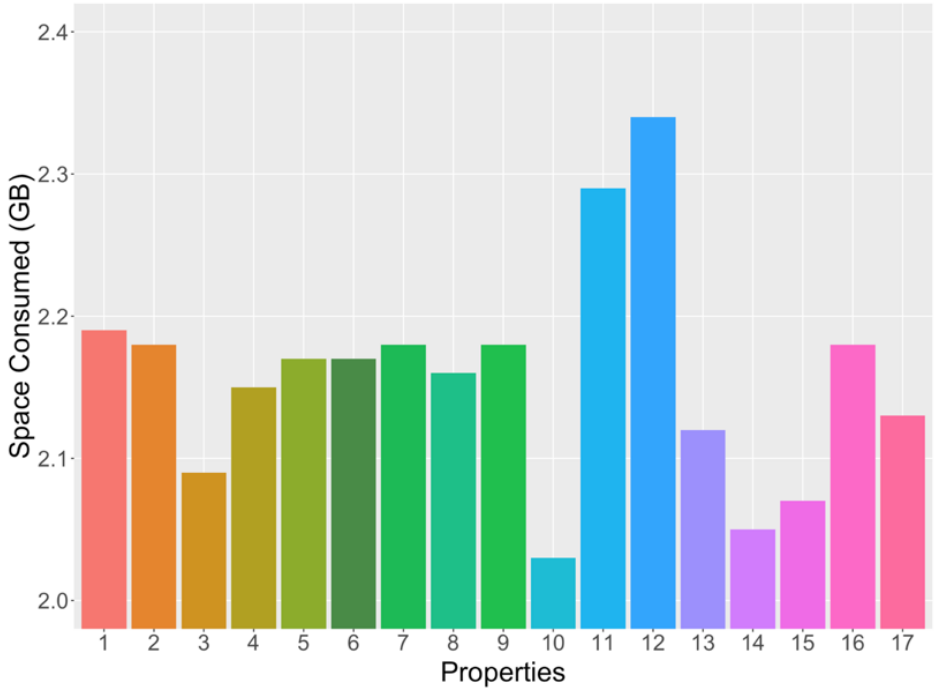

Figure 6.3: Index size for each code property from Table 5.2 across 110K Java classes.

6.1.3 Subset Accuracy

In total, there were 131,072 unique subsets of properties created from the original ranking algorithm. To run a single query, it took approximately 29 hours to test all subsets. Figure 6.4 shows the different accuracies for each subset. We categorized them into bins with ranges of 10% accuracy. The lowest accuracy was 0.44, which was obtained by 4 subsets. The highest accuracy was 1.0, which was obtained by 224 subsets (223 cheaper subsets and the original ranking algorithm).

After finding the accuracies of all subsets, the time to mine each property individually, and the space usage of each property individually, we selected subsets that were the fastest to mine for in each range of 10% accuracy and had the smallest space usage for each range of 10% accuracy. The subset that was the fastest to mine for in each range of 10% accuracy also had the smallest space usage. Table 6.3 shows the original ranking algorithm, denoted as R, as well as the subsets with the fastest time to mine selected from each range of 10%

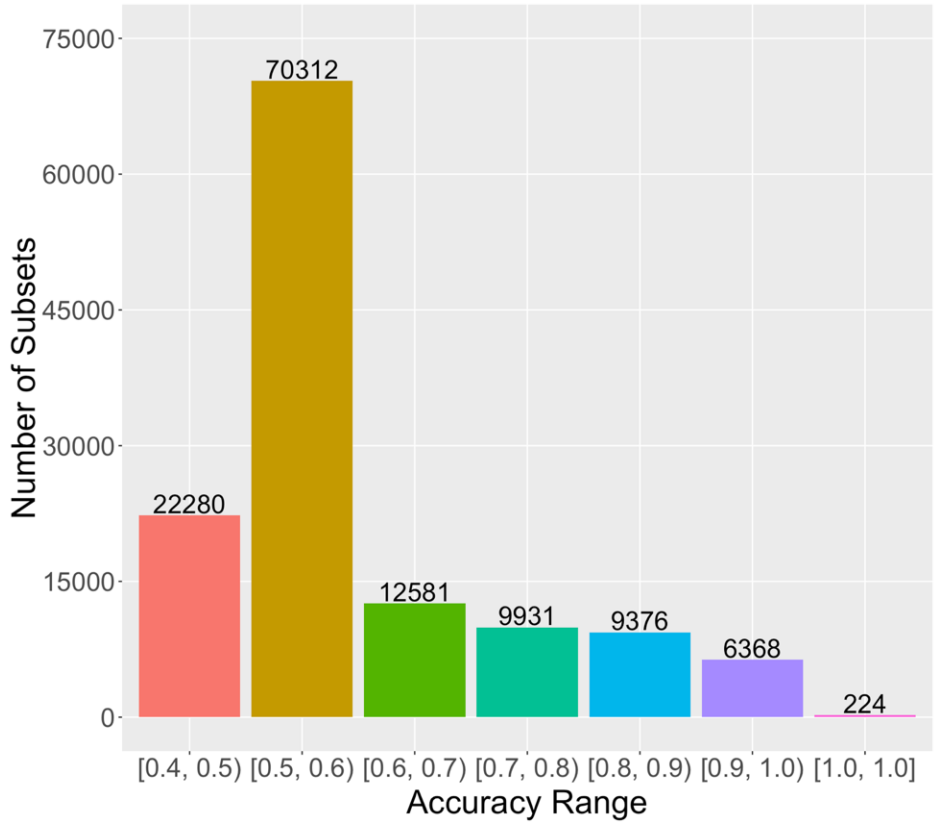

Figure 6.4: All 131,072 subsets of properties binned based on accuracy.

accuracy, denoted as M_1 to M_6. M_1 is the subset that has 100% accuracy to the original ranking algorithm with the fastest time to mine. M_2 is the subset from the (100, 90] accuracy range with the fastest time to mine. M_3 is the subset from the $(90, 80]$ accuracy range with the fastest time to mine. M_4 is the subset from the (80, 70] accuracy range with the fastest time to mine. M_5 is the subset from the (70, 60] accuracy range with the fastest time to mine. M_6 is the subset from the (60, 50] accuracy range with the fastest time to mine.

Table 6.3: Code properties of the original ranking algorithm and each subset.

Code Property	R	M_1	M_2	M_3	M_4	M_5	M_6
Import Names (1)	✓	✓					
Variable Names (2)	✓	✓					
Class Names (3)	✓	✓	✓				
Author Name (4)	✓	✓			✓		
Project Name (5)	✓				✓		
Method Invocation Names (6)	✓	✓		✓			
Method Declaration Names (7)	✓						
Size (8)	✓	✓	✓	✓			
Number of Imports (9)	✓	✓	✓	✓	✓	✓	
Cyclomatic Complexity (10)	✓	✓	✓	✓	✓		
Parent Class Name (11)	✓						
Package Name (12)	✓						
Field Names (13)	✓	✓	✓	✓	✓	✓	✓
Is Generic (14)	✓						
Is Abstract (15)	✓						
Has Wild Cards (16)	✓						
Project Owner Name (17)	✓	✓	✓		✓		

Each of these subsets are outlined in Figure 6.5 and Figure 6.6 with its name and accuracy in parentheses. In Figure 6.5, we show the cumulative time needed to mine each property for a certain ranking algorithm across all 110K Java classes, which is calculated as the sum of the times to mine each property individually from each class. Using the original ranking algorithm R as a baseline, M_1 takes 44.12% less time to mine. M_2 takes 66.42% less time to mine than R, M_3 takes 78.15% less time than R, M_4 takes 80.20% less time than R, M_5 takes 86.60% less time than R, and M_6 takes 93.50% less time than R.

In Figure 6.6, we show the cumulative size of the index of the original ranking algorithm and same subsets across all 110K Java classes, which is the sum of the index sizes all properties present in a ranking algorithm. As said before, these subsets are the subsets that have the smallest cumulative index sizes after mining 110K Java classes for each range of 10% accuracy. Using the original ranking algorithm R as a baseline, M_1 uses 41.66% less disk space than R. M_2 uses 65.35% less disk space than R. M_3 uses 77.04% less disk space than

R. M_4 uses 82.74% less disk space than R. M_5 uses 88.28% less disk space than R. M_6 uses 94.22% less disk space than R.

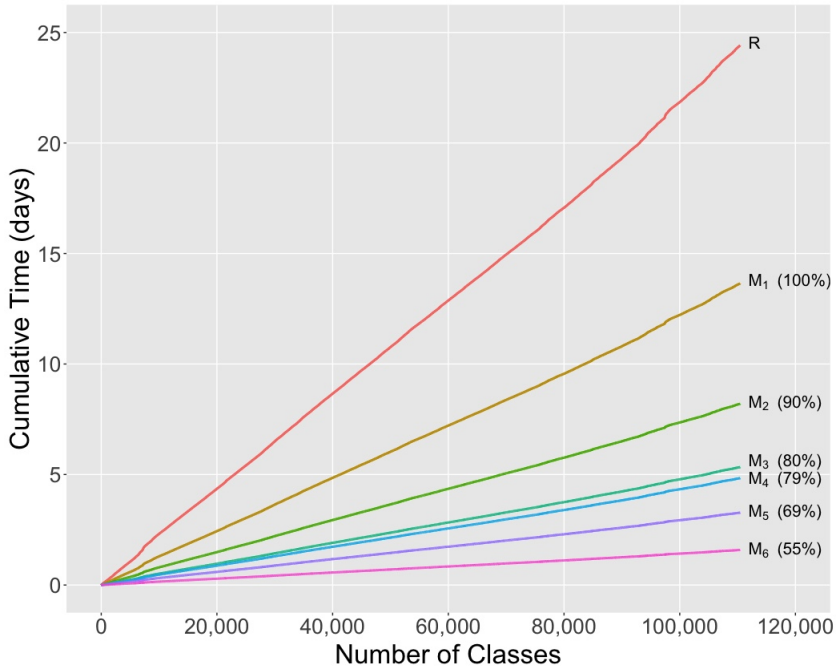

Figure 6.5: Cumulative time to mine properties for each subset of properties from 110K Java classes.

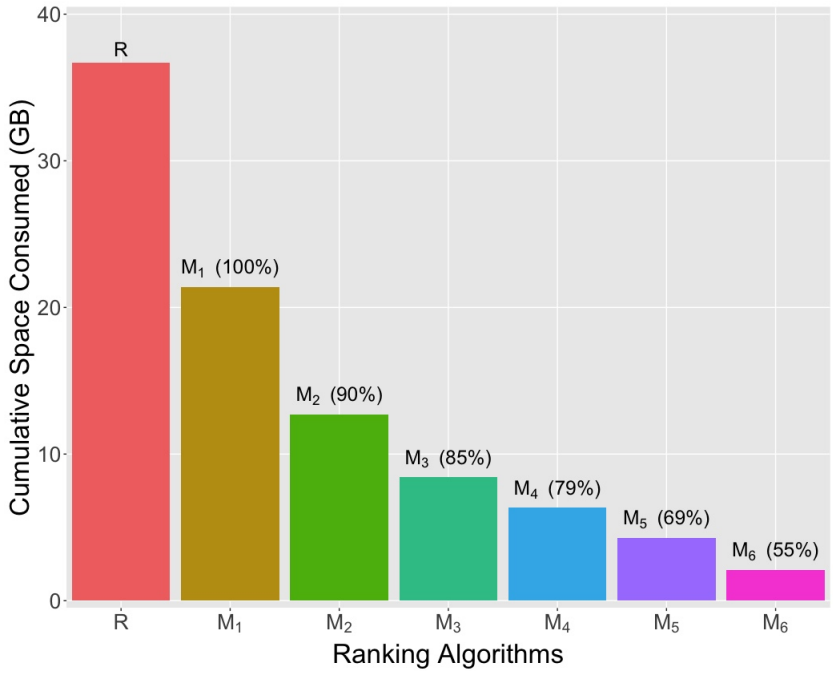

Figure 6.6: Cumulative index size for each subset of properties from 110K Java classes.

6.1.4 Correlations

This section focuses on correlations between 4 factors: number of properties, accuracy, index size, and time to mine. For each figure, we plotted all 131,072 subsets to see if there was a correlation across two properties. We used the Harrell Miscellaneous library in R [22] to get both the Pearson correlation and Spearman correlation.

In Figure 6.7, we plot all subsets based on their number of properties and accuracies. The Pearson correlation between the number of properties and time is 0.40, supporting a weak linear relationship. The Spearman correlation between the number of properties and time is 0.42, supporting a weak monotonic relationship.

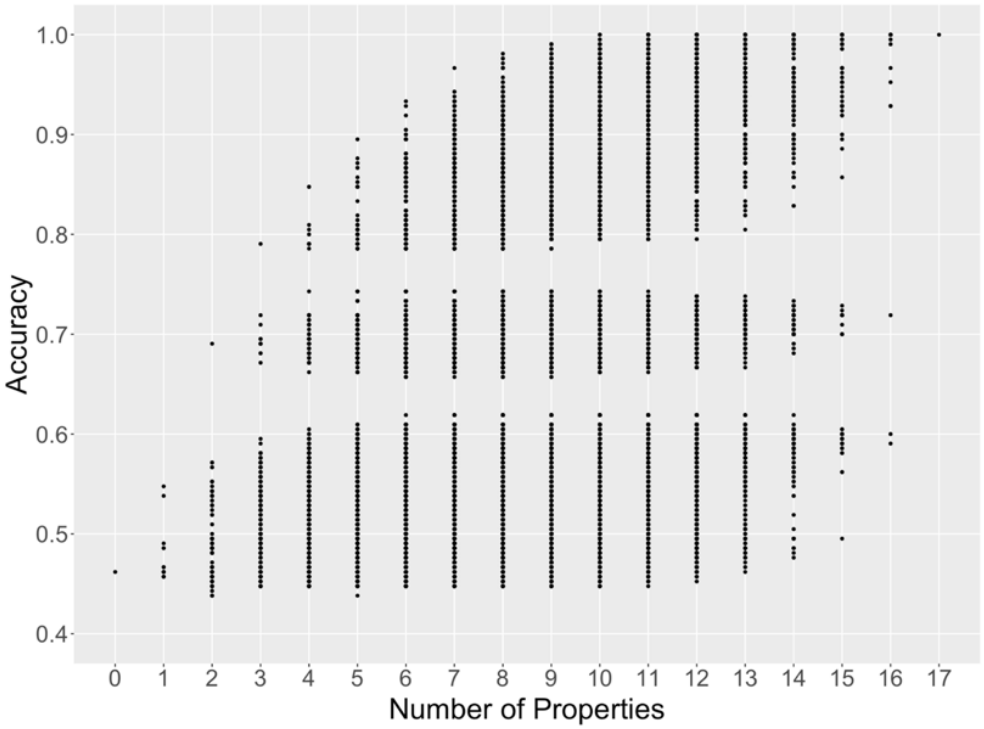

Figure 6.7: Number of properties versus accuracy for all subsets.

In Figure 6.8, we plot all subsets based on their number of properties and index sizes. The Pearson correlation between the number of properties and time is 1.00, supporting a strong linear relationship. The Spearman correlation between the number of properties and time is 0.99, supporting a strong monotonic relationship.

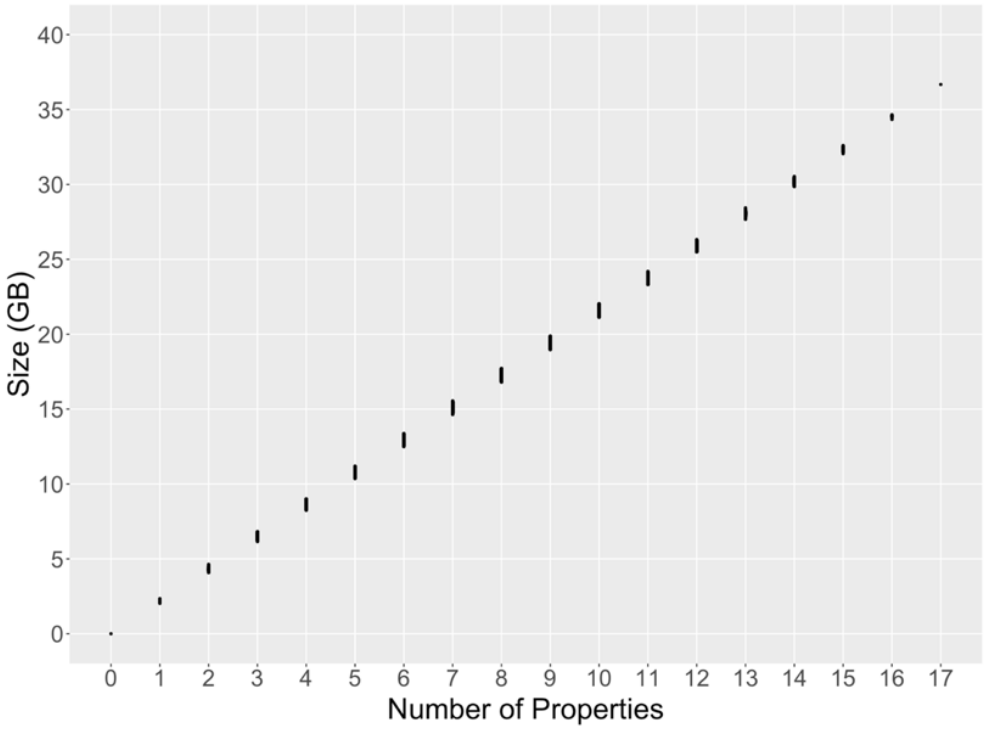

Figure 6.8: Number of properties versus index size for all subsets.

In Figure 6.9, we plot all subsets based on their number of properties and times to mine. The Pearson correlation between the number of properties and time is 0.94, supporting a strong linear relationship. The Spearman correlation between the number of properties and time is 0.93, supporting a strong monotonic relationship.

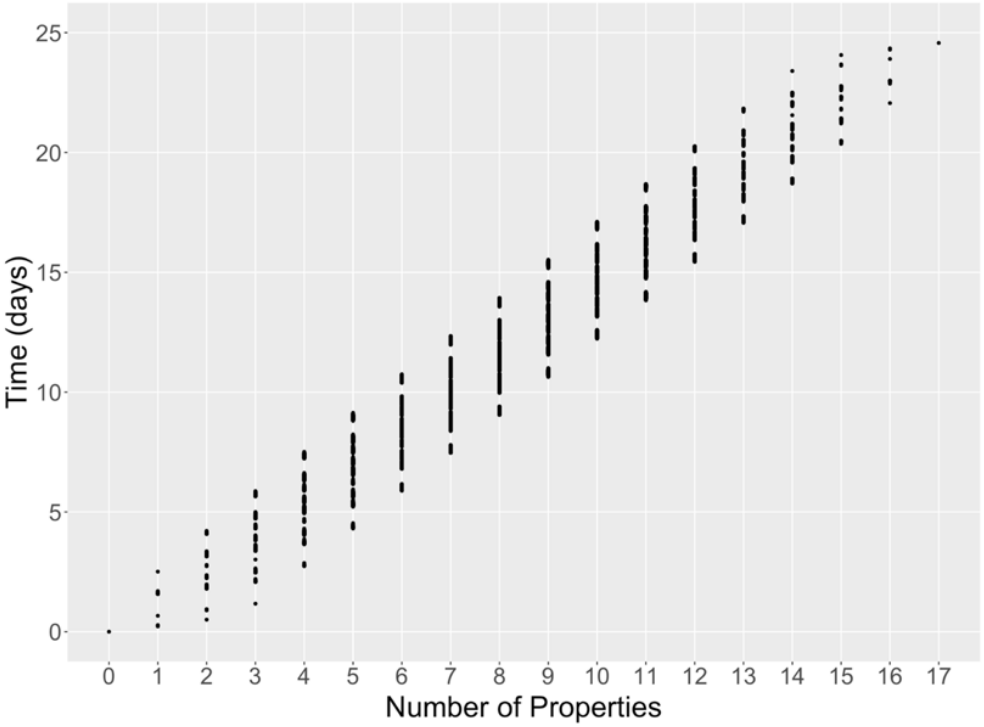

Figure 6.9: Number of properties versus time to mine for all subsets.

In Figure 6.10, we plot all subsets based on their times to mine and accuracies. The Pearson correlation between the number of properties and time is 0.42, supporting a weak linear relationship. The Spearman correlation between the number of properties and time is 0.43, supporting a weak monotonic relationship.

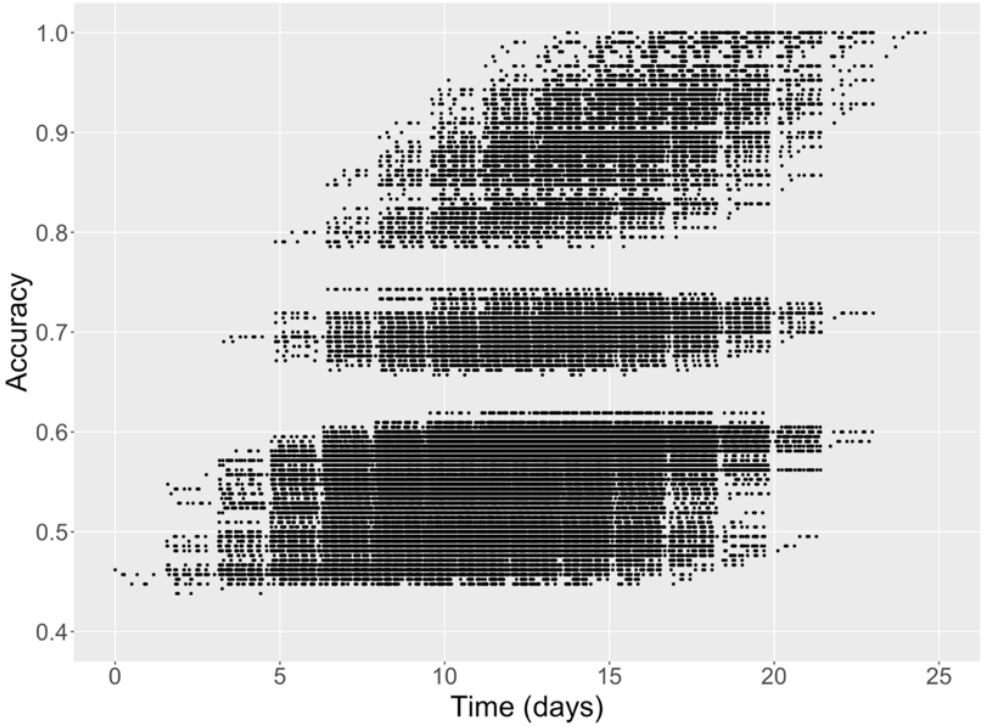

Figure 6.10: Time to mine versus accuracy for all subsets.

In Figure 6.11, we plot all subsets based on their index sizes and times to mine. The Pearson correlation between the number of properties and time is 0.94, supporting a strong linear relationship. The Spearman correlation between the number of properties and time is 0.93, supporting a strong monotonic relationship.

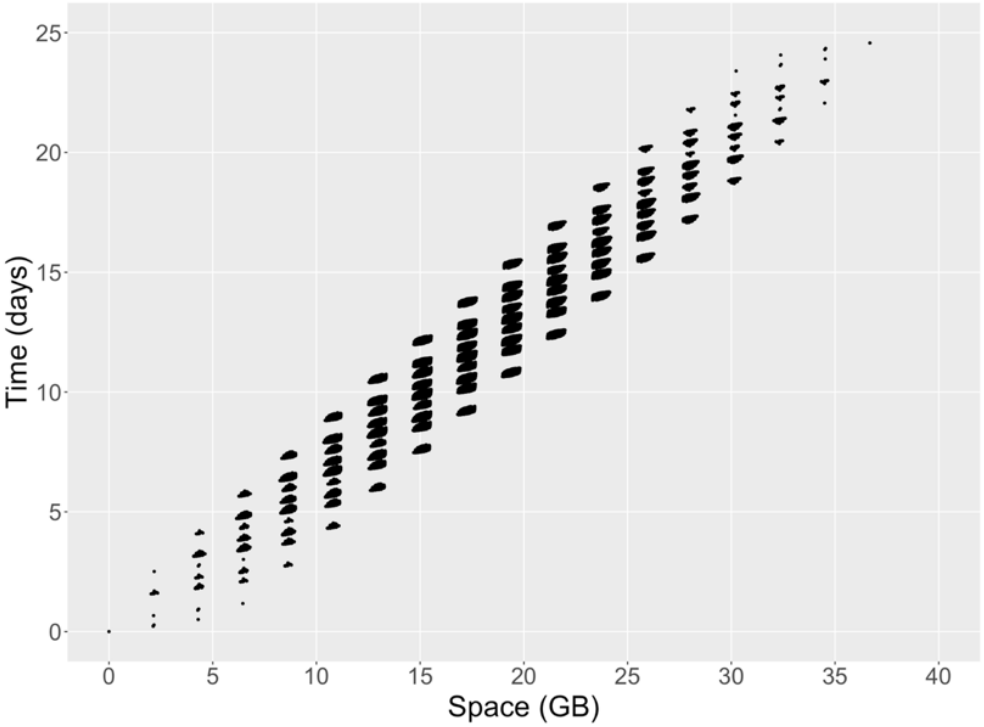

Figure 6.11: Index size versus time to mine for all subsets.

In Figure 6.12, we plot all subsets based on their index sizes and accuracies. The Pearson correlation between the number of properties and time is 0.40, supporting a weak linear relationship. The Spearman correlation between the number of properties and time is 0.39, supporting a weak monotonic relationship.

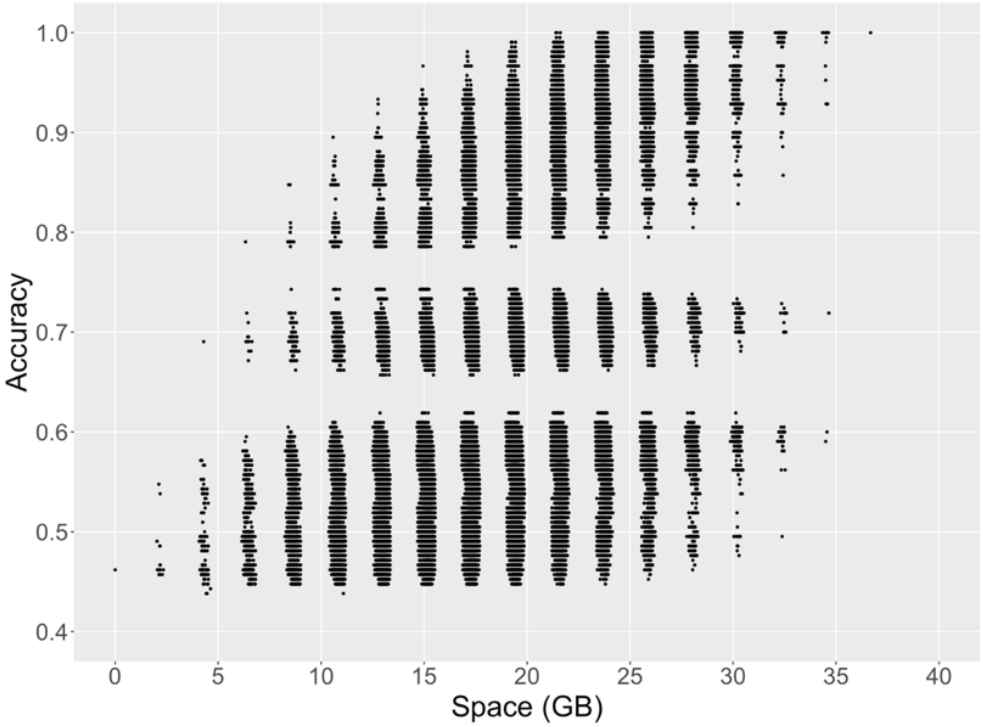

Figure 6.12: Index size versus accuracy for all subsets.

6.2 Beowulf for Code Mining Results

Figure 6.13 shows the cumulative number of classes mined by both the Beowulf cluster and the high-performance standalone computer after each day for 5 days.

Overall, the Beowulf cluster mined 97,016 Java classes in five days, and the high-performance standalone computer mined 32,286 Java classes in five days. The Beowulf cluster mined 20,026 classes the first day, 19,019 classes the second day, 19,632 classes the third day, 19,452

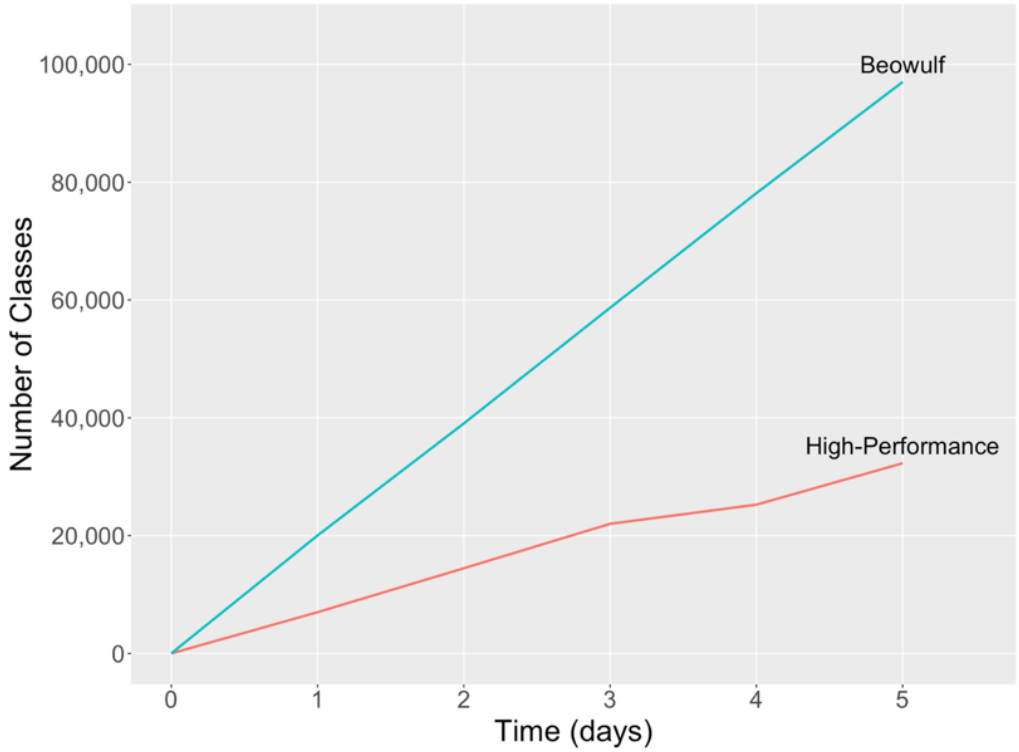

Figure 6.13: Number of Java classes mined over five days by both systems.

classes the fourth day, and 18,887 classes the fifth day. The high-performance standalone computer mined 7,003 classes the first day, 7,460 classes the second day, 7,543 classes the third day, 3,251 classes the fourth day, and 7,029 classes the fifth day.

Chapter 7

Discussion

Our study explores whether What-If Ranking Analysis and a Beowulf cluster for code mining help in improving mining code from the Internet. We first discuss the results from using What-If Ranking Analysis on the MWL-ST Hybrid ranking algorithm. Then we discuss the results from the mining experiment between the Beowulf cluster and the high-performance standalone computer.

We were able to reduce the original ranking algorithm to a cheaper ranking algorithm using What-If Ranking Analysis. We found 223 alternative ranking algorithms to the MWL-ST Hybrid ranking algorithm that performed as well as the original ranking algorithm and used fewer properties. This showed that What-If Ranking Analysis was successful in finding a cheaper ranking algorithm, which answers our first research question.

One possible explanation for cheaper ranking algorithms appearing is that some properties fulfill duplicate roles in the calculation of similarity between two documents. For example, a ranking algorithm $A + B$ and another ranking algorithm $A + C$ may give the same score between two documents if $score(B) = score(C)$. As another example, a ranking algorithm

$A + B$ and a ranking algorithm $A + C + D$ may give the same score between two documents of $score(B) = score(C) + score(D)$. Further analysis would reveal whether there are relationships between certain properties used in the ranking algorithm.

There were certain properties in common with all of the 223 cheaper ranking algorithms that performed as well as the original ranking algorithm. The properties in common among the 223 cheaper ranking algorithms were the following 9 properties: import names, variable names, class name, author name, method invocation names, size, number of imports, cyclomatic complexity, and field names. Because these properties appeared in every cheaper ranking algorithm that was as accurate as the original, we can conclude that these 9 properties are essential for the MWL-ST Hybrid ranking algorithm to work as expected.

However, the ranking algorithm that only contains exactly these 9 properties has an accuracy of 99%. This alternate ranking algorithm supports the idea that, in addition to these 9 properties, additional properties are necessary to achieve the same accuracy as the original ranking algorithm.

The greatest amount of savings when dropping to 90% accuracy with respect to the original gives a 66.42% savings in time to mine and a 65.35% savings in disk space usage compared to the original ranking algorithm. Depending on the developer of the code search engine, achieving 100% accuracy may not be necessary. The purpose of What-If Ranking Analysis is to show the time to mine and disk space usage tradeoffs between different combinations of terms in the original ranking algorithm. The choice of selecting a ranking algorithm that fulfills the time and space requirements is left to the developer.

Compared to the greatest amount of savings at 100% accuracy (44.12% in time to mine and 41.66% in disk space usage), an additional 21.23% savings in time to mine and 23.69% savings in disk space usage when using a ranking algorithm that achieves 90% accuracy can

be very noticeable when the number of classes mined increases. For example, if the developer chooses to mine ten times as many classes as our What-If Ranking Analysis experiment, the developer could potentially save up to approximately 55 days and 81 GB of disk space choosing the 90% accuracy option over the 100% accuracy. However, when a developer chooses to mine one-tenth as many classes as our experiment, the developer only saves 0.55 days and 0.81 of disk space when choosing the 90% accuracy ranking algorithm over the 100% accuracy ranking algorithm. Therefore, if the developer would like to maintain a high accuracy and save on other resources while mining a large number of classes, such as greater than 1,000,000, the developer would benefit more by sacrificing accuracy to lower the time spent mining and disk space usage.

The Beowulf cluster mined three times as many Java classes than that of the standalone high-performance standalone computer in the same amount of time. With this finding, we are able to answer our second research question of creating a method that reduces the time to mine code from the Internet. We learn that using a distributed system of commodity-grade computers is more efficient at mining code from the Internet than a high-performance standalone computer at the same cost. Although the core counts between the two systems is very different (43 in the Beowulf cluster versus 8 in the high-performance standalone computer), Figures 5.8-5.10 shows that the high-performance standalone computer performs significantly better during the benchmark tests. However, in the context of mining code from the Internet, the Beowulf cluster was able to mine more classes because of the increased throughput from the Beowulf cluster to the indexing server. In the case of the high-performance standalone computer, we believe that the downloading of the repositories from GitHub caused the slowdown, which constituted the lower number of repositories mined.

Although the Beowulf cluster mined three times as many classes as the high-performance standalone computer, we note that, based on the number of cores alone, the Beowulf cluster

should mine 5.375 times as many classes as the high-performance standalone computer. This number comes from the number of cores in our Beowulf cluster divided by the number of cores in the high-performance standalone computer (43/8) while ignoring the core's clock rate. Because the Beowulf cluster mined Java classes that the high-performance standalone computer did not, there are many possible explanations as to why the Beowulf cluster did not perform as intended. This may have been caused by larger repositories being mined later, larger property values in those repositories, or the computers in the Beowulf cluster shutting down due to software issues. Although these reasons may all have contributed to the difference, we believe the main contributor to the slowdown is the CPU performance. Because the CPUs found in the computers of the Beowulf cluster are significantly less powerful than that of the high-performance standalone computer, mining the same Java classes between the high-performance standalone computer and a single Raspberry Pi computer would result in the Raspberry Pi taking more time.

Chapter 8

Threats to Validity

In this chapter, we discuss the threats to validity to our work. The first section covers internal threats to validity: threats to the study's design. The second section covers external threats to validity: threats to the generalizability of the results of the study.

8.1 Internal Threats to Validity

When mining the sample using the Beowulf cluster, not all of the URLs in the sample were mined. One reason this happened was that a URL of a repository would no longer be available. Possible causes of this are the repository being deleted by its owner or renaming of the repository. Another reason that not all of the URLs in the sample were mined is that some of the computers in the Beowulf cluster shut down during the mining process. This occurred for a number of reasons, such as out-of-memory errors with the Java parsing program, overheating of the computer, or a failure in the computer's main disk storage. It is possible that our results may have differed when using all 146,200 original URLs instead of the smaller sample size. This difference, however, would have likely worked out in favor

of the Beowulf cluster — more parallel work would have taken place.

8.2 External Threats to Validity

First, only Java source code was tested in both the What-If Ranking Analysis experiment and the Beowulf code mining experiment. This limits our results only to Java source code, since not all of the properties used in the ranking algorithm are available in other languages. Examples of this include the absence of abstract classes in JavaScript or generics in Python.

Second, only one ranking algorithm was tested with What-If Ranking Analysis. The MWL-ST Hybrid ranking algorithm specifically uses the sum of scores of certain properties. This poses two separate threats. One threat is that What-If Ranking Analysis only works on certain properties, which would limit the applications of What-If Ranking Analysis. The other threat is that What-If Ranking Analysis only works on the MWL-ST Hybrid algorithm. Further research needs to be conducted to see if certain ranking algorithms may not find cheaper alternatives through the use of What-If Ranking Analysis.

Third, only 110K classes were used when checking the time and space consumption. This number was selected to be larger than the sample size required for statistical significance. However, these 110K classes are not the same as all classes found on GitHub, which limits the results to only these 110K classes. It is possible that selecting 110K different Java classes for the sample could have provided drastically different results than the results of this study.

Fourth, only 21 queries were tested when using the search engine. These queries are only representative of the context in which they were selected and not of all possible queries. Using different queries than the ones used in the study could have increased or decreased the number of modified ranking algorithms that were as accurate as the original ranking algorithm.

Fifth, the computers used in this study have specifications that are not the same as all other computers. Typically, computers will vary in terms of hardware and software, and the performance of computers depends on these factors in addition to environmental factors (e.g., age of hardware, number of background processes). Repeating this study with different computers will likely produce different results.

Chapter 9

Conclusion

Code search engines are one type of tool to help programmers search for code on the Internet. Although many code search engines exist, there are few guidelines for improving the process of creating code search engines and the performance of code search engines.

In this thesis, we explore two techniques to improve the process of mining source code from the Internet. We show the results of What-If Ranking Analysis to find a cheaper ranking algorithm that performs as well as the original and a modification to the original Beowulf cluster that optimizes on network throughput instead of performance. Using a ranking algorithm called More With Less-Social Technical Hybrid, we use What-If Ranking Analysis to explore whether a cheaper ranking algorithm exists. In addition, we compared the performance of a Beowulf cluster to a high-performance standalone computer by mining as many Java classes from GitHub over 5 days on each system.

We found that multiple cheaper ranking algorithms exist that return the same results as the original ranking algorithm across multiple queries while using both fewer properties and less disk space. We found that the ranking algorithm could be reduced by 44% in time to mine all the source code and 41% in index disk space. Using a cheaper ranking algorithm improves

the speed of the code search engine with fewer computations for each search. Additionally, the cheaper ranking algorithm will have less properties to mine from the Internet, reducing both the time to mine the properties and the size of the index.

We also found that, using the Beowulf cluster, we could mine three times as fast compared to using a high-performance standalone computer at approximately the same monetary cost. This shows that not only is mining using a distributed system faster at the same cost, but also a large amount of CPU power is not necessary when mining.

There are many aspects in which this work can be extended. One area of future works is testing different ranking algorithms to see if What-If Ranking Analysis is more broadly. We could also try our study on mining different programming languages, seeing which properties are the most important for each ranking algorithm. Another area of improvement could be comparing the performance of cloud solutions to that of local solutions. We could see whether it is cheaper to use a cloud solution of similarly performing instances than to build a Beowulf setup.

Bibliography

[1] A. Ahmed, B. Kayis, and S. Amornsawadwatana. A review of techniques for risk management in projects. *Benchmarking: An International Journal*, 14(1):22–36, Mar. 2007.

[2] S. K. Bajracharya. *Facilitating Internet-Scale Code Retrieval*. Dissertation, University of California, Irvine, 2010.

[3] Bing. `http://www.bing.com`.

[4] C. Bird and T. Zimmermann. Assessing the Value of Branches with What-if Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 45:1–45:11, New York, NY, USA, 2012. ACM.

[5] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Fundamental Approaches to Software Engineering*, pages 385–400. Springer, Berlin, Heidelberg, Mar. 2009. DOI: 10.1007/978-3-642-00593-0_26.

[6] Eclipse Java Development Tools (JDT). `http://www.eclipse.org/jdt/`.

[7] R. E. Gallardo-Valencia and S. E. Sim. Internet-Scale Code Search. In *Tools and Evaluation 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure*, pages 49–52, May 2009.

[8] Github. `https://github.com`.

[9] M. K. Gobbert. Configuration and performance of a Beowulf cluster for large-scale scientific simulations. *Computing in Science Engineering*, 7(2):14–26, Mar. 2005.

[10] Google. `http://www.google.com`.

[11] J. D. Grant, R. L. Dunbrack, F. J. Manion, and M. F. Ochs. BeoBLAST: distributed BLAST and PSI-BLAST on a Beowulf cluster. *Bioinformatics*, 18(5):765–766, May 2002.

[12] H. Herodotou and S. Babu. *Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs*.

[13] J. Kim, S. Lee, S.-w. Hwang, and S. Kim. Towards an Intelligent Code Search Engine. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pages 1358–1363, Atlanta, Georgia, 2010. AAAI Press.

[14] Krugle. http://www.krugle.com.

[15] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. CodeGenie: Using Test-cases to Search and Reuse Source Code. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 525–526, New York, NY, USA, 2007. ACM.

[16] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, Apr. 2009.

[17] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan. Query expansion via WordNet for effective code search. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 545–549, Mar. 2015.

[18] L. Martie, T. D. LaToza, and A. v. d. Hoek. CodeExchange: Supporting Reformulation of Internet-Scale Code Queries in Context. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 24–35, Nov. 2015.

[19] L. Martie and A. van der Hoek. Sameness: An Experiment in Code Search. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 76–87, Piscataway, NJ, USA, 2015. IEEE Press.

[20] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, Sept. 2012.

[21] Michigan Technological University Beowulf Cluster. http://www.cs.mtu.edu/beowulf/misc/cluster.jpg.

[22] Package 'hmisc'. https://cran.r-project.org/web/packages/Hmisc/index.html.

[23] Package 'pwr'. https://cran.r-project.org/web/packages/pwr/pwr.pdf.

[24] Phoronix Test Suite - Linux Testing & Benchmarking Platform, Automated Testing, Open-Source Benchmarking.

[25] Raspberry Pi 1 Model B+. https://www.raspberrypi.org/products/model-b-plus/.

[26] C. Sadowski, K. T. Stolee, and S. Elbaum. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 191–201, New York, NY, USA, 2015. ACM.

[27] Searchcode. https://www.searchcode.com.

[28] B. Sisman and A. C. Kak. Assisting code search with automatic Query Reformulation for bug localization. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 309–318, May 2013.

[29] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A Parallel Workstation For Scientific Computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[30] K. T. Stolee, S. Elbaum, and D. Dobos. Solving the Search for Source Code. *ACM Trans. Softw. Eng. Methodol.*, 23(3):26:1–26:45, June 2014.

[31] R. J. Swart, P. Raskin, and J. Robinson. The problem of the future: sustainability science and scenario analysis. *Global Environmental Change*, 14(2):137–146, July 2004.

[32] S. Thummalapenta and T. Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[33] M. Umarji, S. E. Sim, and C. Lopes. Archetypal Internet-Scale Source Code Searching. In *Open Source Development, Communities and Quality*, pages 257–263. Springer, Boston, MA, Sept. 2008. DOI: 10.1007/978-0-387-09684-1_21.

[34] M. van den Akker, S. Brinkkemper, G. Diepen, and J. Versendaal. Software product release planning through optimization and what-if analysis. *Information and Software Technology*, 50(12):101–111, Jan. 2008.

[35] G. Vriend. WHAT IF: A molecular modeling and drug design program. *Journal of Molecular Graphics*, 8(1):52–56, Mar. 1990.

[36] Yahoo. `http://www.yahoo.com`.

[37] A. Zagalsky, O. Barzilay, and A. Yehudai. Example Overflow: Using Social Media for Code Recommendation. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, RSSE '12, pages 38–42, Piscataway, NJ, USA, 2012. IEEE Press.