RICE UNIVERSITY

Code Similarity Search in a Latent Space

by

Letao Qi

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Dr. Christopher M. Jermaine, Chair
Professor of Computer Science

Dr. Swarat Chaudhuri
Associate Professor of Computer Science

Dr. Luay K. Nakhleh
Professor and Chair of Computer Science

HOUSTON, TEXAS
APRIL 2017
ABSTRACT

Code Similarity Search in a Latent Space

by

Letao Qi

A huge database of program source codes that supports fast search via code similarity would be useful for several applications, including automated program synthesis and debugging, and user-facing code search in an integrated development environment. Here, “similar” is defined with respect to a set of application-defined similarity functions. The key difficulty in realizing this goal is that standard database indexing techniques cannot be applied to the problem of querying based on arbitrary similarity functions. To address this difficulty, I propose a dictionary-based approach where I represent each piece of code by a vector of similarities to a set of example database codes. Cosine similarity between the vector representing a query code and the vector representing a database code can be used to measure closeness.

However, the dictionary may need to be very high dimensional if the goal is to accurately index a wide variety of database codes. Hence, I explore the idea of using projection matrix to the reduce dimensionality of the problem. One approach is to use random projection. The other approach that I explore is learning the projection matrix by developing a machine learning algorithm that is supervised using the text/code pairs provided by StackOverflow, a question-answering website for programmers.
Contents

Abstract ii
List of Illustrations iv

1 Introduction 1
1.1 Motivation 2
1.2 A Brief Introduction to the Dictionary-Based Approach 4
1.3 Curse of Dimensionality and Dimensionality Reduction 6
1.4 Contributions 8
1.5 Thesis Organization 9

2 Background 10
2.1 TF-IDF 10
2.2 Cosine Similarity 11
2.3 Jaccard Index 11
2.4 Locality-Sensitive Hashing (LSH) 12
2.5 SimHash: a Variant of LSH for Cosine Similarity 13
2.6 LSH for Nearest Neighbor Search (NNS) 14

3 Proposed Method 17
3.1 Dictionary-Based Approach 17
3.2 Dimensionality Reduction 21
   3.2.1 Random Projection 22
   3.2.2 Learning a Supervised Model for Dimensionality Reduction 23
Illustrations

2.1 The Jaccard index between the sets of API calls inside two code snippets. .................................................. 12
2.2 Locality-Sensitive Hashing ............................................................ 13
2.3 SimHash .......................................................... 14
2.4 The $L$ hash tables used for the locality-sensitive hashing. $K$ hash functions in the original hash family are concatenated to produce a new hash function. .................................................. 16

3.1 The dictionary-based representation of the code. ......................... 18
3.2 A 2-D space partitioned by the k-d tree algorithm. The black points are data. The blue and red lines are splitting hyperplanes. .......... 20
3.3 The flow graph for the dictionary-based code representation and the model proposed for dimensionality reduction. .................. 24
3.4 An example of the latent space and data in the space. .................. 27

4.1 The area under curve (AUC) of the receiver operating characteristics (ROC) for the best training run. ROC = 0.8994.................. 32
4.2 The data used in experiment one. The left is a SOF post with four code snippets. They are assumed to be projected in close locations in the latent space on the right. The red ball is the query code snippet. The three dark blue balls are candidates. The other light blue balls are random code candidates. .................. 33
4.3 The combined result of all the 10 runs in experiment one. . . . . . . . 34
4.4 The results of the ten individual runs in experiment one. . . . . . . . 35
4.5 The result of experiment two. . . . . . . . . . . . . . . . . . . . . . . 36
4.6 Two code snippets in the experiment three dataset. 4 delete is a DB access method. 16 PersonUI is a GUI method. . . . . . . . . . . . . . . 37
4.7 The hierarchical clustering of the original data matrices with no projection. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.8 The hierarchical clustering of the randomly projected data. . . . . . . 38
4.9 The hierarchical clustering of the data projected with the learned matrices. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
Chapter 1

Introduction

In recent years, many large online code repository websites are found, for example, GitHub and SourceForge. Billions of lines of source codes reside in these repositories and contain enormous valuable information. The availability of large code repositories offers great resources for many software engineering applications such as code completion, code repairing, plagiarism detection, etc. [1, 2, 3]

Code completion helps users finish incomplete code fragments with similar codes in codebases. Code repairing debugs and repairs original codes with the help of similar codes. Plagiarism detection looks for exact or approximate clones of codes of interest to detect academic misconducts. The core technique used in those applications is code similarity search.

Code similarity search is a widely used technique. It is the procedure of finding code pieces similar to the query code in a candidate pool according to some similarity functions. The input of code similarity search is a piece of query code and a collection of candidate codes. Based on different similarity functions, different features will be extracted from the codes. Similarity functions use associated features to compute similarity scores between the query code and the set of candidate codes. A set of candidate code pieces that have the highest similarity scores with the query code are
Code features can be categorized into three classes, *lexical features*, *syntactic features* and *semantic features*. Lexical features reflect textual properties of the code, eg. *tf-idf vector of code*. Syntactic features reflect syntactical structures of the code, eg. *abstract syntax tree (AST)*. Semantic features reflect internal semantic meanings inside the code, ex. *program dependency graph (PDG)*. Similarity functions compute similarity scores by comparing code features, such as, *cosine similarities* between tf-idf vectors of codes.

There are many existing works on manufacturing of code features and similarity functions [4, 5, 6, 7], a lot of which show encouraging performance. There is a great interest to take advantage of a combination of existing high-performance code features and similarity functions to improve code similarity search. The goal of this thesis is to propose a general framework for efficient and scalable code similarity search based on a set of arbitrary similarity functions and their associated code features.

### 1.1 Motivation

In this thesis, I assume all the code pieces are stored in databases due to the reason that database systems are renowned for their abilities to conduct fast search with integrated indexing techniques. Indexing techniques construct proper data structures to store relevant information of the data, which make quick search possible without accessing each data point. Conventional database systems normally perform exact
search based on the idea of exact keys. In general, database systems employ various 
indexing techniques to locate tuples that have the same key as the one in the query. 
However, the data we use is code. It is rarely the case that they have keys or we 
compare them by keys. We often compare them by applying similarity functions to 
code features. If no indexing is used, a linear search of the entire code base becomes 
a necessity, which exhibits a great challenge for code similarity search.

Fortunately, there exists a type of database systems, called metric database, that 
targets at indexing metrics or distance functions [8]. If we query a metric database 
of codes for the set of code pieces with the shortest distances to the query code, we 
equivalently perform code similarity search. Some adjusting of similarity functions 
to distance functions is required here. A metric database uses a metric space model 
where the metric or distance function is defined for all pairs of data points in the 
database. Not all the distance functions are metrics. In order for a distance function 
to become a metric, the function has to meet the properties of positiveness, symmetry, 
strict positiveness and triangle inequality [8]. The formulae for the properties are as 
follows:

\[ d : X \times X \rightarrow [0, \infty), \]

1. \( d(x, y) \geq 0, \) positiveness

2. \( d(x, y) = d(y, x), \) symmetry

3. \( d(x, y) = 0 \Leftrightarrow x = y, \) strict positiveness

4. \( d(x, y) \leq d(x, y) + d(y, z), \) triangle inequality
The triangle inequality is the key to enabling a performance better than brute-force or a linear search. We can observe a linear combination of a set of metrics with non-negative weights is still a metric.

In our code similarity search problem, our goal is searching based on a set of arbitrary similarity functions. After some adjusting, a plateau of functions are well-defined for positiveness, symmetry and strict positiveness. However, making a random function to meet the triangle inequality is hard. Given that we have a set of arbitrary functions, the scenario is even worse. In other words, we have to adhere to the set of metrics, which is limited and finite for code search. This clearly prohibits us from setting a general framework to incorporate arbitrary similarity functions and code features. In order to solve the problem, we propose a \textit{dictionary-based approach} for code representation.

\subsection{A Brief Introduction to the Dictionary-Based Approach}

In this thesis, we propose a dictionary-based approach to represent code. We build a comprehensive set of prototype codes to approximate all the codes in a certain code base. We call the comprehensive set of \textit{prototypes} as \textit{dictionary}. Given the dictionary, a vector of similarity scores between the target code and each prototype code in the dictionary gives a new representation to the target code. The idea is, if we can have a large set of objects to represent all angles of an application field, then a new object’s resemblance to the large set of objects can maintain its identity in the application field.
For example, in the application field of Java IO, the dictionary contains one example code snippet for each standard Java IO API class. Given a new code snippet calling `BufferedReader` inside, we compute the vector of similarities between the new code and each example code. The dimension corresponding to the `BufferedReader` class in the vector will have a high value. The similarity vector succeeds in maintaining the new code’s identity. A graphical explanation of the approach is in Figure 3.3.

The principal advantage of the proposed dictionary-based approach is two-fold. First, the approach enables the incorporation of arbitrary similarity functions and code features into the code search framework. If the size of selected similarity functions is more than one, the vector of similarity scores becomes a matrix, with the number of columns equivalent to the size of functions. The \((i, j)\)th entry in the matrix is simply the similarity score between the target code and the \(i\)th prototype code under the \(j\)th similarity function. Any similarity function can be applied. Second, the approach transforms code or code features into a multi-array of numerals. There exist many works on indexing of multi-dimensional numerical vectors we can take advantage of, such as, metric database, \(k\)-d tree, \(R\) tree and locality-sensitive hashing (LSH). Unlike typical hashing methods, LSH hashes similar data points into the same bucket with high probability. See Equation 1.2.

\[
Pr_h[h(x) = h(y)] = f(sim(x, y))
\] (1.2)

A query code is hashed into the bucket where most candidate codes are similar to it. Then, only those code pieces are further compared. Due to the fact that hashing
is used, the code similarity search will achieve near-constant time complexity for querying. One drawback of LSH is approximate search rather than exact search. This can be relieved by concatenating multiple hash functions together to make new hash functions and maintaining multiple hash tables. Furthermore, the majority of code similarity search applications does not necessarily look for the most similar code among candidates but the ones that are similar enough to perform the task. With the great speedup LSH offers, we consider LSH as our choice of indexing multi-dimensional numerical vectors.

In the proposed dictionary-based approach, we use cosine similarity to compare numerical vectors. The variant of LSH for cosine similarity is SimHash.

### 1.3 Curse of Dimensionality and Dimensionality Reduction

In our dictionary-based approach, the dictionary is a comprehensive set of example codes in some application field. In order for the dictionary to have a broad coverage in the field, the size of the dictionary is quite large. The scenario is even worse when a few similarity functions are applied. The majority of indexing techniques for numerical vectors, k-d tree, LSH, etc., will fail due to the reason that it is very difficult to organize and apply heuristics to high-dimensional numerical vectors. There is also a concern for the space complexity. With hundreds of thousands of similarity scores to represent a piece of code, the disk space is drained easily. There is also a problem for the memory. In LSH, we do not have to store real data in the memory but pointers
to the blocks where data reside. In the case that the dimension of numerical vectors is very high, LSH starts to underperform. In high-dimensional space, we need to concatenate more hash functions to put similar data points into same buckets and more hash tables to ensure most similar data points are found for further evaluation. The size of hash tables needed is sometimes several hundred for a less-than one thousand dimensional space [9]. The pointers to the entire dataset copy as many times as the size of hash tables. The simultaneous increase in hash tables and hash functions put great burden on both the memory and the CPU. The problem is called \textit{curse of dimensionality}. Hence, \textit{dimensionality reduction} is required.

One intuitive solution is \textit{random projection}. In random projection, the original $d$-dimensional data is projected into the $k$-dimensional subspace or latent space ($k \ll d$), with a random $k \times d$ projection matrix. There are various methods for populating the matrix, ex. Gaussian distribution. Experimental studies show random projection succeeds in preserving cosine similarity [10].

The other approach that we explore is learning the projection matrices by developing a machine learning algorithm that is supervised using the code/text pairs provided by \textbf{Stack Overflow (SOF)}, a question-answering website for programmers. On SOF, programmers post programming questions as well as answering questions by writing code snippets. There are rich sources of code snippets and their descriptions on SOF. If we project both the code vector and the text bag-of-words vector into the latent space vectors, with each dimension representing a topic, both vectors should
have matching dimensions with a large value. Based on the conception, we are trying to learn both projection matrices. If the size of similarity functions is more than one, the code is embodied as a two-dimensional similarity matrix. We observe that different entries in the matrix play different roles. We are trying to learn a weight matrix for the code before projecting it into the latent space. See Figure 3.3 for details. We perform experimental studies to evaluate our approach.

We still use cosine similarity to compare similarities of vectors for code in the reduced-dimensional space.

1.4 Contributions

The principal contribution of this thesis is proposing a general framework (the dictionary-based method) that can incorporate arbitrary similarity functions and arbitrary features for code similarity search. The importance of the framework is reflected in the ability of reusing any existing code similarity search methods and the possibility of producing a better method by combining existing methods.

The second contribution of this thesis is recognizing some dimensionality reduction is required by the proposed dictionary-based method and proposing a supervised model to learn the dimensionality reduction procedure. We also conduct experiments to show the feasibility of our model against the random projection and the no projection case.
1.5 Thesis Organization

In the Chapter 2, we give more background knowledge on code similarity search including some features, similarity functions and the locality-sensitive hashing (LSH) in detail. In Chapter 3, we elaborate on the dictionary-based approach and the dimensionality reduction methods. In Chapter 4, we introduce the training procedure of the supervised model and evaluate the model through three experimental studies. Chapter 5 will exhibit some related work. In chapter 6, we conclude the thesis.
Chapter 2

Background

In this chapter, we introduce the background knowledge for several techniques related to the code similarity search in this thesis.

2.1 TF-IDF

Tf-idf is short for the term frequency-inverse document frequency. It is an important technique to lexically represent a document in a corpus. The term frequency $tf(t, d)$ is the number of times a word appears in a document. The term frequency is an analogy to the famous bag-of-words representation (without normalization). The inverse document frequency $idf(t, D)$ is a measure of how often a term appears across the documents in a corpus.

$$idf(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$  \hspace{1cm} (2.1)

$$idf(t, D) = \log \frac{N}{|\{d \in D : t \in d\}| + 1}$$  \hspace{1cm} (2.2)

$$tf-idf(t, d) = tf(t, d) \times idf(t, D)$$  \hspace{1cm} (2.3)

Idf is the logarithm of the inverse fraction of how many documents contain the term in the corpus. Frequent terms, such as, get, have, set, etc. have low idf scores. Equation 2.2 is the original form of idf. Equation 2.2 is a variant of idf with smoothing. The
tf-idf of a term in a document is the multiplication of the appearance frequency of the term in the document and the term’s inverse document frequency. Tf-idf is the weighted version of bag-of-words. Tf-idf is superior for decreasing the value for frequent terms that are less indicative of distinguishing a document. In practice, a dictionary of terms are selected beforehand. Tf-idf of a document $d$ is a tf-idf vector $\vec{v}$ with $v_i = tf-idf(t_i, d)$.

### 2.2 Cosine Similarity

The cosine similarity is a widely used technique, which measures the cosine of the angle between two term vectors, eg. tf-idf vectors.

$$cos(A, B) = \frac{A \cdot B}{||A|| \cdot ||B||}$$

(2.4)

It is computed by taking the inner product of two term vectors divided by the product of both vector lengths. $||\cdot||$ is the Euclidean norm of a term vector. Only the directions of the term vectors in the vector space matter for the computation of cosine similarity. Hence, cosine similarity avoids biasing toward larger documents.

### 2.3 Jaccard Index

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{A \cap B}{|A| + |B| - |A \cap B|}$$

(2.5)

Jaccard index is also called Jaccard similarity coefficient, which is used to measure similarity of two sets. It is computed as the intersection of two sets divided by the
Figure 2.1: The Jaccard index between the sets of API calls inside two code snippets.

union of two sets. The range of Jaccard index of two sets is $0 \leq Jaccard(A, B) \leq 1$. In the thesis, the Jaccard index is used to compare the similarity of two sets of API-related elements. For example, in Figure 2.1, there are two code snippets converting a string array into a list and then sorting the list. They only differ in the list implementation, `ArrayList` v.s. `Vector`. Intuitively, they should have a large Jaccard index. The Jaccard index between the code snippets is $\frac{2}{3}$.

### 2.4 Locality-Sensitive Hashing (LSH)

Hashing is the function of mapping a given data vector $x \in \mathbb{R}^D$ to an integer key $\mathbb{R}^D \rightarrow \{0, 1, ..., N-1, N\}$. Each key corresponds to a bucket in the hash table. Conventional hashing algorithms put similar data vectors into different buckets to avoid conflicts.

Unlike conventional algorithms, the property of the locality-sensitive requires the hash functions: $Pr_h[h(x) = h(y)] = f(sim(x, y))$, $h$ is random in an LSH family $\mathcal{F}$. 

```java
String init[] = {"b", "d", "a", "c"};
List list = new ArrayList(Arrays.asList(init));
Collections.sort(list);

String init[] = {"b", "d", "a", "c"};
List list = new Vector(Arrays.asList(init));
Collections.sort(list);
```

$Jaccard\_Index = \frac{2}{3}$
This means the probability of two data vectors being hashed into the same bucket is proportional to their similarity under some similarity function. An example of LSH is in Figure 2.2. As you can see, the two close data points in the coordinates are hashed into the same bucket. The one that is further is hashed into another bucket.

In this thesis, we use LSH to hash the numerical vector of the query code into the bucket where most of the similar code pieces reside. Then we only further compare the code pieces in the bucket. This allows a near-constant time complexity.

2.5 SimHash: a Variant of LSH for Cosine Similarity

We use LSH for indexing numerical vectors and cosine similarity to compare numerical vectors. Hence, we need a variant of LSH that has a family of hash functions, which are able to put two numerical vectors into the same bucket with the probability proportional to their cosine similarity. This variant is called SimHash. SimHash uses a random hyperplane and hash two data points into the same bucket, if they are
on the same side of the hyperplane and vice versa. In Figure 2.3, we can observe intuitively the chance of two data points being hashed into the same bucket keeps increasing as their angle decreases, because cosine similarity is inversely proportional to the angle between the two data points in the interval $[0,\pi)$. Given a random hyperplane, the probability of two data points being hashed into the same bucket is proportional to their cosine similarity. Formally,

$$Pr_h(h_r(x) = h_r(y)) = 1 - \frac{\theta}{\pi}, \quad \theta = \cos^{-1}s$$

(2.6)

### 2.6 LSH for Nearest Neighbor Search (NNS)

One of the important applications of LSH is the nearest neighbor search (NNS). In order to use LSH for NNS, there are two primary parameters, the width parameter $K$ and number of hash tables $L$. This is shown in Figure 2.4. As aforementioned in 2.5, in
SimHash, two numerical vectors with an angle of 179° still have a chance to be hashed into the same bucket. With a single hash function, there exist chances for dissimilar data points to be hashed into the same bucket. When given a query, this will decrease the performance since dissimilar data points are further compared for NNS. To put truly similar data points into the same bucket, we amplify the hash function using AND-construction. We construct a new family of hash functions $\mathcal{G}$ by randomly concatenating $K$ hash functions in the original hash family, i.e. $g(\cdot) = [h_1(\cdot), ..., h_K(\cdot)]$. As a result, this decreases the probability of dissimilar data points being put into the same bucket by $K$ folds. The parameter $K$ and the AND-construction ensure that very few dissimilar data points are further compared. On the other hand, this also leads to some similar data points being put into different buckets as one of the $K$ hash functions decides differently. Therefore, $L$ hash tables are generated simultaneously. Given a query data point, the query is hashed into a bucket in each of those $L$ tables. All the candidate data points in the $L$ hash tables are further compared. This increases the probability of the most similar data points being further compared by $L$ times and also guarantees that very few dissimilar data points are processed.
Figure 2.4: The $L$ hash tables used for the locality-sensitive hashing. $K$ hash functions in the original hash family are concatenated to produce a new hash function.
Chapter 3

Proposed Method

3.1 Dictionary-Based Approach

As aforementioned in the introduction, conventional key-based and metric-based database indexing techniques do not apply to the code similarity search based on arbitrary similarity functions and code features. Therefore, we propose a dictionary-based approach by representing the target code with a vector of similarities to a comprehensive set of codes in some application domain. If the size of similarity functions is more than one, the vector of similarity scores is a similarity matrix. In this thesis, the number of prototype codes is $n$. The number of similarity functions is $d$. Then, the similarity matrix for each code piece is $S : n \times d$, where $S_{i,j}$ is the similarity score between the target code and the $ith$ prototype code under the $jth$ similarity function. Figure 3.1 shows a transposed similarity matrix due to the space constraint.

In order for the dictionary-based approach to work, the first thing is to select a set of prototypes, i.e. the dictionary, that captures all the features of code in some application domain. Different features spread over different prototype codes. Then, the resemblance of the target code to each prototype can reflect the features of the target code. It is not always possible for the dictionary to cover all features of the
code in an application domain. Our goal is to select a large set of prototypes that contains all the major features. Even if the set is not exhaustive, the number of prototypes is still quite large.

Despite the fact that the dictionary-based approach is able to incorporate arbitrary similarity functions and code features in the code’s representation, the other principal merit of the approach is transforming the code into a similarity matrix. If we flatten the similarity matrix, the matrix can be treated as a multi-dimensional vector. There exist lots of works on indexing multi-dimensional vectors we can take advantage of [11, 12, 13]. Some widely used techniques include metric-based methods, methods using space-partitioning data structures and hashing methods.

Metric databases cannot index the original form of the code. After the code is
transformed into a similarity matrix or multi-dimensional vector, metric databases can index the codebase based on the *angular distance* between pairs of codes. Angular distance between the two vectors is the angle between them, which is a valid metric. Angular distance is monotonic and inversely proportional to cosine similarity. When we compare vectors by cosine similarity, metric databases with angular distance is an alternative.

Another group of indexing methods for multi-dimensional vectors are based on constructing space-partitioning data structures. Some well-known approaches include *k-d tree, R tree*, etc. K-d tree partitions the high-dimensional space with hyperplanes and cycles through all the axes for next splitting planes. When a new partition is produced, k-d tree picks a data point in the partition and splits the partition with a hyperplane through the point and perpendicular to the next axis in the cycle. The procedure is repeated iteratively until all the points are used up. According to the affiliation of partitions in different levels, there is a tree structure with splitting data points as tree nodes. When querying, the query data point is pushed down to the bottom of the tree and moves up to update its closest neighbor in terms of the Euclidean distance. As the query goes up, it can eliminate all the data points in the other side of the current hyperplane if the perpendicular distance to the hyperplane is larger than the distance to the current closest neighbor. Since all the splitting planes are either perpendicular or parallel to each axis, the perpendicular distance to a splitting plane is always shorter than the distances to the data points in the other
Figure 3.2: A 2-D space partitioned by the k-d tree algorithm. The black points are data. The blue and red lines are splitting hyperplanes.

K-d tree is efficient since many data points are eliminated during querying. An example of k-d tree partitioning space is shown in Figure 3.2. The R tree explores a similar idea as the k-d tree. It tries to group nearby data points in a minimal Rectangle box and uses spatial relationship for efficient querying. Indexing methods based on space partitioning is appropriate for our dictionary-based method when the Euclidean distance is used for measuring closeness.

In our work, we choose a hashing method, locality-sensitive hashing (LSH), as the preferred indexing method for our dictionary-based code representation. LSH hashes similar data points into the same bucket with high probability. This achieves a near-constant time complexity by hashing the query code into a bucket and further comparing codes in that bucket only. The time efficiency is the primary reason why
we choose LSH. We are using cosine similarity to compare code vectors. The variant of LSH for cosine similarity is SimHash. To perform NNS, \( K \) hash functions are concatenated to exclude dissimilar codes in the query’s bucket. \( L \) hash tables are generated respectively to include all the similar codes for further comparison. LSH-related topics are elaborated in Chapter 2.

### 3.2 Dimensionality Reduction

In the dictionary-based approach, the dictionary has a large number of example codes to realize a broad coverage in some application field. As a result, the similarity matrix is of a large dimension. When the dimension increases, the indexing techniques introduced in Section 3.1 start to fail for various reasons. The phenomenon is called the **curse of dimensionality.** For metric-related methods and methods using partitioning data structures, in high-dimensional space, there are not enough data points for each dimension. It is hard to apply heuristics to avoid a linear search of the database. For locality-sensitive hashing, large \( K \) and \( L \) are needed to ensure the accuracy of code similarity search. Experimental study shows the size of hash tables is sometimes several hundreds for a less-than one thousand dimensional space [9]. Given that the dimension of the space we are dealing with is easily more than ten thousand, we have to probe \( O(KL) \) buckets for further comparison, which are sometimes infeasible. Also, a large amount of memory is needed to store hundreds or thousands of hash tables, with each table having an entire copy of pointers to
each data. Storing a large amount of data as similarity matrices on disk is also space-consuming. We want to perform dimensionality reduction to solve the curse of dimensionality problem.

### 3.2.1 Random Projection

Random projection is a widely used technique for dimensionality reduction. It is called “random” since the projection matrix used is populated with a statistical distribution. The idea of random projection is based on the Johnson-Lindenstrauss lemma [14], which indicates that a random subspace of relatively high dimension will approximately preserve the distances in the original space. Formally, in random projection, the original $d$-dimensional vector is projected into a $k$-dimensional subspace ($k \ll d$), following Equation 3.1.

$$D_{k \times N}^{RP} = R_{k \times d} \times D_{d \times N}$$

(3.1)

$R_{k \times d}$ is the random projection matrix. $D_{d \times N}$ is the data matrix with $N$ $d$-dimensional data points. $D_{k \times N}^{RP}$ is the approximate data matrix in the $k$-dimensional subspace after random projection. In this thesis, we use a Gaussian distribution to populate $R_{k \times d}$, i.e. $R_{i,j} \sim \mathcal{N}(0,1)$. Achlioptas [15] has shown the Gaussian distribution can be replaced by a much simpler distribution:

$$R_{i,j} = \sqrt{3} \cdot \begin{cases} +1 \text{ with probability } \frac{1}{6} \\ 0 \text{ with probability } \frac{2}{3} \\ -1 \text{ with probability } \frac{1}{6} \end{cases}$$

(3.2)
In this thesis, we flatten a similarity matrix into a high-dimensional vector first and project the vector with a Gaussian random projection matrix.

3.2.2 Learning a Supervised Model for Dimensionality Reduction

Besides the random projection, we are trying to learn the dimensionality reduction to outperform random projection in preserving similarities and solve some intrinsic weakness of random projection in our code similarity search problem. Our idea is to project the similarity matrix into a latent subspace where each dimension represents some latent topic. If we know the latent topic vector in the subspace, the projection matrix is going to be learned. The problem ends up being finding supervision for the subspace location of the code. In other words, for each piece of code, it requests a piece of data talking about the same set of latent topics. The most natural choice for the data is the textual description of the code. The text can be represented as a tf-idf vector. The expectation is, given some projection matrix, the text is projected to a location close to the code in the latent topic subspace. Since the two latent space vectors are close, they share large values in same dimensions. This indicates a large inner product value. If we perform a machine learning task to maximize the inner product, we are able to find out the projection matrices for both the code and the textual description. Since the subspace is of a much lower dimension, dimensionality reduction is conducted. Due to the fact that the code is projected into a latent subspace with semantical meaning, the expectation is this gives some advantages
Figure 3.3: The flow graph for the dictionary-based code representation and the model proposed for dimensionality reduction.

In our experiments, we use the code/text pairs provided by Stack Overflow (SOF), a question-answering website for programmers. SOF is a live forum for programmers. A large amount of posts containing code snippets and users’ discussions on the codes are posted everyday. The code and the discussion text in a post is a code/text pair. SOF data is publicly available. We use the data to learn the dimensionality reduction procedure.

**Model.** We formalize the learning process discussed in 3.2.2 with the equation of
a supervised model (a graphical flow graph of the model is in Figure 3.3):

\[ f(x, y) = \left( N \left( \sum_{j=1}^{d} \Delta_j(x, X) \circ W_{s,j} \right) \right)^T M y \tag{3.3} \]

- \(n\): number of prototypes
- \(d\): number of similarity functions
- \(k\): dimension of latent space, \(k \ll n\)
- \(m\): size of dictionary for text
- \(x\): code; \(X\): \(n \times 1\), prototype codes
- \(y\): \(m \times 1\), textual description
- \(\Delta_j(x, X)\): \(n \times 1\), similarity vector between \(x\) and each in \(X\) under the \(j\)th similarity function; \(\Delta(x, X)\) or \(S\): \(n \times d\), similarity matrix
- \(W\): \(n \times d\), weight matrix; \(W_{s,j}\): \(n \times 1\), the \(j\) column in \(W\)
- \(\circ\): elementwise multiplication
- \(N\): \(k \times n\), projection matrix of code
- \(M\): \(k \times m\), projection matrix of text
- \(f(x, y)\): literal, similarity score, inner product of projected latent space vectors

In our model, the similarities \(S\) between the target code and each prototype under all similarity functions are computed. Then, the similarity matrix \(S\) and a weight vector \(W\) is multiplied element-wise. The result is summed across columns into \(\overrightarrow{v}^t\): \(n \times 1\). \(N\) projects \(\overrightarrow{v}\) into \(\overrightarrow{left}\): \(k \times 1\). \(M\) projects \(y\) into \(\overrightarrow{right}\): \(k \times 1\). \(f(x, y)\) is the inner product of \(\overrightarrow{left}\) and \(\overrightarrow{right}\).

In the model, there exists a weight matrix \(W\) to adjust importance of entries in \(S\).
The key idea is that different similarity functions have different roles in determining the overall similarity to some prototype code. For example, *the edit distance between function names is more indicative in determining two code snippets’ similarity than the common key words they share*. Also, same similarity function has different importance in various code prototypes. For example, *the Jaccard index between sets of API calls is more important for a GUI-related code than a sorting algorithm code implemented from scratch*. The weight matrix $W$ is learned along with two projection matrices $N$ and $M$ simultaneously.

In $N$ and $M$, each row is a weight vector for $\vec{v}$ and $y$ respectively. $\vec{v}$ is a vector of weighted overall similarities to prototypes. $y$ is a tf-idf vector that stands for the textual description. The weighted sum for each row is the extent of the corresponding latent topic being discussed. $N_{i,j}$ represents how much the $j$th prototype code is talking about the $i$th latent topic. $M_{i,j}$ represents how much the $j$th word in the text dictionary is talking about the $i$th latent topic.

If the learning of $W$, $N$ and $M$ succeeds, two code snippets and their descriptions will be projected into the marked locations in the latent space in Figure 3.4. The vertical dimension represents the latent topic of internet worm. The horizontal dimension represents the latent topic of binary search. The top code snippet is an *addOne* function and its description contains words like *add* and *one*. Both the code and the text are related to internet worm because worms can copy themselves but not really related to binary search. So both the code and the text are placed in the
coordinates with a high value for the internet worm latent dimension and a low value for the binary search latent dimension. As a result, they are close to each other in the latent space. Also, they have a large inner product value since they share a latent dimension with big numbers.

**Learning and the Objective Function.** In this work, the positive data instances are the code/text pairs from the same post in SOF. We generate negative data instances by picking up a piece of code from one post and a piece of text from another post. Our assumption is that the latent vectors of code and text from the same post are close in the latent space. They share large values in similar dimensions. They have a larger inner product. The latent vectors of code and text from different posts are separate in the latent space. Their dimensions of large values don’t match,
which results in a smaller inner product. Then our model can push inner products of positive instances to a larger value and negative instances to a smaller value in order to learn the desired parameter matrices $W$, $N$ and $M$. We can learn the model with logistic regression and maximum likelihood estimation (MLE).

Logistic regression models the likelihood of a positive instance with a logistic function and the likelihood of a negative instance with one minus the likelihood.

Logistic function: $\sigma(f) = \frac{1}{1 + e^{-f}}, \sigma \in (0, 1), f = f(x, y)$  \hspace{1cm} (3.4)

Likelihood of positive instance: $\frac{1}{1 + e^{-f}}$  \hspace{1cm} (3.5)

Likelihood of negative instance: $1 - \frac{1}{1 + e^{-f}}$  \hspace{1cm} (3.6)

Then we can use MLE to estimate $W$, $N$ and $M$ through $f(x, y)$, where $f(x, y) = f(x, y|W, N, M)$.

MLE: $f(x_1, \ldots, x_n|\theta) = f(x_1|\theta) \times \ldots \times f(x_n|\theta), x_i$ is a data point \hspace{1cm} (3.7)

\begin{equation}
\prod_{i=1}^{n} \left( \frac{1}{1 + e^{-f_i}} \right)^{l_i} \left( 1 - \frac{1}{1 + e^{-f_i}} \right)^{1-l_i}, l_i \in \{0, 1\} \hspace{1cm} (3.8)
\end{equation}

\begin{equation}
\sum_{i=1}^{n} l_i \log \left( \frac{1}{1 + e^{-f_i}} \right) + (1 - l_i) \log \left( 1 - \frac{1}{1 + e^{-f_i}} \right) - c_1\|W\|_F - c_2\|N\|_F - c_3\|M\|_F \hspace{1cm} (3.9)
\end{equation}

When we maximize Equation 3.8, the likelihood of both types of instances will increase, which means positive instances have large $f$-value and negative instances have small $f$-value. Parameter matrices are adjusted during MLE to make the desired behaviors happen, which leads to the code and its related text being projected to nearby points in the latent subspace and vice versa. Equation 3.9 computes Equation 3.8 in
log space for the ease of computation and add Frobenius norm to avoid large values in parameter matrices, which could falsely dominate the projection process.

\[
\text{Frobenius Norm: } \| A \|_F = \left( \sum_{i,j=1}^{n} |a_{ij}|^2 \right)^{1/2}
\]

(3.10)
Chapter 4

Experiments

4.1 Training

Data Set. The data set we use for code/text pairs is Stack Overflow (SOF). A post in SOF contains a question and a few answers. Code snippets are embedded in questions and answers. The other parts in a post are discussions related to the code snippets embedded. We first download a May 2015 snapshot of SOF. Then, we extract all the posts tagged Java. We filter out posts with less than 500 characters in length of code. We combine all the textual content in a post as the text and combine all the code snippets in the post as the code. Each post becomes a code/text pair. We shuffle and randomly choose 50,000 pairs for further processing. The code features we choose for the training are CodeTfidf and SetOfAPICalls. The similarity functions associated with the two features are cosine similarity and Jaccard Index. For CodeTfidf, we treat the code as pure text and convert it into a tf-idf vector. We preprocess the code by splitting tokens in camel case (CamelCase) and snake case (snake_case), removing non-word tokens, lowercasing, lemmatizing and stemming. We use the TfidfVectorizer in the Python Natural Language Toolkit to convert codes into tf-idf vectors in a corpus of 50,000 codes. The TfidfVectorizer filters out common English stop words, words
appearing in more than 10 percent of all the code snippets and words appearing less than five times. For SetOfAPICalls, we extracted all the distinct API calls in Java 1.8 Standard Library along with Java 1.8 package names from the 50,000 pieces of code. For the text, we follow a similar way the CodeTfidf feature uses and convert all text into tf-idf vectors. The text dictionary has 9,947 words.

**Experimental Setup and Results.** During training, we produce negative data instances by picking code and text from two different posts with no common tag. We use 5,000 as the size of code prototypes. We use 32,000 data instances for training (+:16,000, -:16,000) and 8,000 (+:4,000, -:4,000) instances for testing. We use batch gradient descent (BGD) to optimize the objective function and update parameter matrices. We evaluate the testing result with ROC, the receiver operating characteristic curve. ROC measures the performance of a binary classifier with different decision thresholds and evaluates the binary classifier with a comprehensive consideration of its performance at different thresholds. A ROC of 1 means the binary classifier splits the positive and negative instances completely. A ROC of 0.5 means the binary classification is random. Our best result achieves a ROC of 0.8994. See Figure 4.1. A ROC of 0.9 means an excellent classifier. The result shows, after training, our model clearly distinguishes positive pairs and negative pairs. The model projects the code and its related textual description into nearby points in the latent space and vice versa. Our best result gives a latent space dimension of 20. We record the learned parameter matrices in our best run to evaluate the dimensionality reduction performance.
Figure 4.1: The area under curve (AUC) of the receiver operating characteristics (ROC) for the best training run. ROC = 0.8994.

4.2 Evaluation

In this section, we conduct three experiments to evaluate the performance of the random projection, the projection learned by our model, i.e. the learned projection, in abilities of preserving similarity matrices during dimensionality reduction. We use similarities between the original data, i.e. no projection, as the gold standard. We denote the three methods, random-projection, learned-projection and no-projection.

Experiment One. Ranking similar code among random code. In this experiment, we look for SOF posts with at least four code snippets in it. See an example of
Figure 4.2: The data used in experiment one. The left is a SOF post with four code snippets. They are assumed to be projected in close locations in the latent space on the right. The red ball is the query code snippet. The three dark blue balls are candidates. The other light blue balls are random code candidates.

experiment one in Figure 4.2. In order for a code snippet to be counted, it has to have at least 200 characters to show significance. We assume the code snippets in the same post are similar. We use four code snippets in each post. We preprocess the code snippets into similarity matrices and text tf-idf vectors as what we did during training. In each post, the first snippet is selected as the query. The other three are put into a pool of 1,000 code snippets as candidates, with the other 997 as random codes. We apply the random projection and the learned projection methods to similarity matrices and use cosine similarity to compare similarities of latent vectors. The results are the ranks of the three similar codes in 1,000 candidates in terms of similarity with
the query code under different projection methods. A rank of 0 means the most similar and a rank of 999 means the most different. In our experiment, each run has 500 posts and 1,500 ranks. The posts used in different runs don’t overlap. We compute the number of ranks for each interval of rank value. We run this 10 times and report the results of the three projection methods in Figure 4.4. We also combine the 10 runs and report the result in Figure 4.3.

![Figure 4.3](image.png)

Figure 4.3: The combined result of all the 10 runs in experiment one.

**Experiment Two.** *Human evaluation of quality of retrieved code.* In this experiment, we pick ten code snippets with commonly used API calls and search for the three most similar code snippets in a candidate pool of 100,000 under the three methods. See Table 4.1 for the names of query codes. The processing of code snippets is same as before. We ask ten experienced programmers to compare the three code
Figure 4.4: The results of the ten individual runs in experiment one.

<table>
<thead>
<tr>
<th>ActionListener</th>
<th>BufferedReader</th>
<th>BufferedWriter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calendar</td>
<td>DescriptionStatistics</td>
<td>JButton</td>
</tr>
<tr>
<td>JScrollPane</td>
<td>SQLConnection</td>
<td>StringTokenizer</td>
</tr>
<tr>
<td></td>
<td>URLConnection</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Names of all ten queries in experiment two.

snippets retrieved under different projection methods based on relevance to the query code for each query. Programmers use 1 for the best method and 3 for the worst. Ties are allowed. During the survey, the order of different methods’ answers is random and is hidden from the volunteers. The result is shown in 4.5.

**Experiment Three.** *Evaluation of dimensionality reduction with clustering.* In this experiment, we conduct a case study. We perform a hierarchical clustering algorithm based on the transformed vectors for each method. We use the clustering results to evaluate projection methods on their abilities of preserving similarities.
The dataset contains 21 Java methods of DB access and GUI interaction. Examples of methods are in Figure 4.6. The first nine are for DB access (indexed 1-9). The next twelve are for GUI interaction (indexed 10-21). We process Java methods into corresponding vectors as before for each method. We run a hierarchical clustering algorithm with the average linkage and cosine similarity for each method. The results are in Figure 4.7, 4.8, 4.9.

4.2.1 Discussions

In experiment one, in general, the no-projection method puts the most similar code snippets in low ranks. Our learned-projection method is outperforming the random-projection method across different runs. Sometimes, in the low-rank area, the curves for our learned-projection intersect with the no-projection curves, which means learned-
Figure 4.6: Two code snippets in the experiment three dataset. 4 delete is a DB access method. 16 PersonUI is a GUI method.

```java
public void delete() {
    try {
        rowSet.moveToCurrentRow();
        rowSet.deleteRow();
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
}
```

```java
public personUI() {
    setPadding(new Insets(10, 10, 10, 10));
    setTop(msgLabel);
    setCenter(initButtons());
    setBottom(initButtons());
    setFieldData(bean.moveFirst());
}
```

Figure 4.7: The hierarchical clustering of the original data matrices with no projection.

```plaintext
Figure 4.7: The hierarchical clustering of the original data matrices with no projection.
```
Figure 4.8: The hierarchical clustering of the randomly projected data.

Figure 4.9: The hierarchical clustering of the data projected with the learned matrices.
projection has comparable performance with no-projection. Rand-projection is consistently the worst in finding similar snippets across different runs. The reason why no-projection is better is due to the fact that no-projection has no loss of information that is generated during dimensionality reduction. Learned-projection is better than random-projection because our model intentionally picks important information through the use of weights in the three parameter matrices and intentionally allocates the important information across latent dimensions. This is better than a random approach.

In experiment two, the no-projection method is again the best in getting volunteers’ top picks due to a similar reason mentioned above. There is no much difference between learned-projection and random-projection in this experiment. The reason is that there are code snippets very similar to the query code in the candidate pool. No matter what projection methods are used, the query code and the the most similar candidate code will be projected into very close positions in the latent space and have very high cosine similarity.

In experiment three, no-projection and learned-projection are able to cluster the data set into two correct clusters of DB-related methods and GUI-related methods but random-projection doesn’t. To further compare the first two methods, we observe that learned-projection is better than no-projection for the clustering inside the GUI cluster. Learned-projection is able to separate constructors and initializers from getters, setters and checkers. The two clusters represent two separate phases of program exe-
cution. We believe the reason why learned-projection performs the best is the method extracts important information and filters out less relevant information. Since the latent space the method projects into is of a much lower dimension, learned-projection has to be smart in putting what information into these dimensions. The method also gets rid of the disturbing of irrelevant information that exists in the no-projection method. Adding learned-projection is trained with code/text pairs, the method has a sense of code semantics in the latent space. It is not accidental learned-projection outperforms no-projection in some cases. Actually, learned-projection outperforms no-projection in some runs of experiments in experiment one.

Given the scale of training and experiments we perform, the supervised model and associated dimensionality reduction methods are very promising. Even some of the experiments are not extensive but the results in case studies show performance in the direction we expect.
Chapter 5

Related Work

In this thesis, we develop techniques for code similarity search, which is closely related to the topic of code search. In our problem, the input to the search is a piece of code. Code search is a broader area. It does not limit the types of input of search. The input can be a piece of code. In a majority of work [16, 17, 18, 19], the input is a piece of text describing the code. The input can also be a portion of the code that is looked for, such as using a sequence of API calls to find the complete program [20]. The comparison algorithm is also more flexible. Besides similarity search, which is approximate search, code search can also conduct exact search, such as code clone detection [21].

In code search, many methods use textual query to search code. McMillan, et al. [16] develop a source code search engine called Exemplar. The goal of Exemplar is finding a set of software applications given a textual query. The query is first split into a set of key words. Then the key words are used to match words in help documents of API calls. The set of API calls are used to locate applications using them. Lv, et al. [17] explore a similar idea by comprehending users’ query and expanding a user’s query with potential API calls. Then they retrieve related code snippets in the codebase based the API calls. There are also works on code search with textual
query using Stack Overflow Data [18, 19]. Sinai and Yahav propose a system that retrieves code snippets whose descriptions have high text-based similarities with the query code’s text. They use Stack Overflow data as their codebase. The intuition is same as ours, which is the discussion around the code snippet in a post is describing the snippet.

There are also many existing works on code similarity search [22, 23], which use code as input to search code. Yuan and Guo [22] create a system called Boreas. The system compares two code snippets by counting variables in different counting environments such as \textit{if-statement} and \textit{for-loop}. A counting matrix $M$ is used to represent the code, with $M_{i,j}$ as the count of the $i$th variable in the $j$th counting environment. The rows of two counting matrices are matched. The summation of matched rows’ cosine similarities is used for the similarity of two code snippets. Smith and Horwitz [23] propose to use fingerprints to compute codes’ block-level similarities. For each statement, all possible $n$-grams are generated. The $k$ least frequent $n$-grams are used to represent the statement. The fingerprint for an $n$-gram is the indices of tokens. $k$ fingerprints are concatenated as the new fingerprint. The intersection of fingerprints of statements in two blocks measures the block-level similarity. The primary difference between their methods and our method is they spend a lot of time on exploring features. And our method can inherit features and associated similarity functions that have already been proved to work. Our method is also easier to extend to include new features. The extension can lead to a better search method.
When the approximate search becomes the exact search, code similarity search becomes code clone detection [24]. There is a long history of research on code clone detection [25, 26, 27, 28, 29, 30]. Kamiya et al. constructed the well-known system *CCFinder* for code clone [21]. The system first performs lexical analysis on each line of code and then transforms the code with a pre-defined set of rules such as removing template parameters. It compares the query code with candidates using a suffix-tree algorithm. Code clone detection and code similarity search generally have different usage and applications. However, code similarity search is more flexible in querying and is sometimes able to be converted into clone detection by querying code with the highest possible similarity score.

Our work is also closely related to the topic of similarity search. The most renowned algorithm for similarity search is *nearest neighbor search* (NNS) and a generalization of NNS, *k-nearest neighbor search* (KNN) [31]. KNN needs to find the *k* closest data points instead of one. In our dictionary-based approach, we use NNS to query similar code snippets. In the field of code search, Bruch et al. use KNN to search nearest snippets for code completion [1]. Their goal is to automatically suggest API calls programmers are likely to use when they are writing programs in integrated development environments (IDE). Instead of suggesting all the functions relevant to the elements in the current program, they propose to suggest functions inside the nearest neighbors of the current program. They compute the frequency each function in the nearest snippets appear and set a threshold for only suggesting
the significant functions over the threshold. Compared with our method, they do
not use locality-sensitive hashing (LSH) as the implementation for NNS, which might
potentially increase their time complexity. Considering code completion is a real-time
application, LSH is a good fit for their method.

Another approach to similarity search is learning the metric or metric learning
[32]. The learning of a distance metric is generally in this form:

\[ d(x, y) = d_A(x, y) = \|x - y\|_A = \sqrt{(x - y)^T A (x - y)}, \quad x, y \in \mathbb{R}^n, \quad A \in \mathbb{R}^{n \times n} \]

When \( A \) is a symmetric positive semi-definite matrix, then \( A = L^T L, L \in \mathbb{R}^{e \times d}, e \leq \text{rank}(A) \). And, \( d_A(x, y)^2 = \|L(x - y)\|^2_2 = \|Lx - Ly\|^2_2 \). As the formula indicates, metric learning is sometimes like computing distances in a new space after the projection of \( L \). We can perform dimensionality reduction by using metric learning and the \( L \) projection matrix. But I don’t think metric learning will outperform our method. Even metric learning learns \( A \) and \( L \) instead of using random numbers, it still shares the same problem with the random projection, which is not considering the weights of similarity matrix.

There is not much work of metric learning in code search but many in information
retrieval [33, 34, 35, 36]. Lee et al. [34] propose a rank-based method for distance metric learning. The distance between data points does not always mean relevancy. This is especially the case when some point is far away from all the points and some point is close to a lot of points, among which many are irrelevant. Their method learns the matrix \( A \) by making the distance between the point known to be relevant
smaller than the distance between the point known to be irrelevant to the same query in the training data. They apply their method to image retrieval and get promising results.

To step further, our work is related to the implementation technique of similarity search. The two widely used implementation techniques of NNS is locality-sensitive hashing (LSH) and k-d tree. LSH is elaborated in Chapter 2. k-d is introduced in Section 3.1.

There are many applications and adaptations of LSH in similarity search [9, 37, 38, 39]. One example in the code search domain is Balachandran’s work [7]. Like us, he also performs code similarity search. He retrieves the abstract syntax tree (AST) from each code and computes l-level characteristic vectors for the AST tree and its subtrees. Instead of once, he applies LSH twice subsequently in order to hash characteristic vectors into buckets. First, he generates a $f$-bit fingerprint using SimHash. The $f$-bit fingerprint is based on the signs in a $f$-dimensional vector. The characteristic vector increases or decreases the values in the $f$-dimensional vector with its own values based on the results of SimHash of characteristic vectors. Since the characteristic vectors have all non-negative numbers, similar characteristic vectors have a large chance to have the same signs in the $f$-dimensional vector and the same fingerprint. The fingerprint is then appropriate to be further hashed using AND-construction on the LSH family for hamming distance. The benefits of the subsequent use of two LSH methods is the first LSH can retrieve the targeted features from the
original data and also keep the similarity between original data. The second LSH 
hashes the output of the first LSH and performs NNS. Our dictionary-based approach 
can also benefit from a 2-LSH system for dimensionality reduction. The first LSH 
plays a role like a weighting matrix. The second LSH plays a role like the projection 
matrix. We will evaluate the feasibility of a 2-LSH system for our approach in the 
future.

There are also many works on variants of k-d tree for similarity search [40, 41, 
42, 43]. The intrinsic reason why we do not choose k-d tree for our approach is the 
concern of time. k-d tree has an average query time of $O(\log n)$ but LSH has a near 
constant time complexity.

In our work, we transformed the representation of a piece of code into a dictionary-
based similarity matrix. There are many other methods to represent code in code 
search. These include bag-of-AST-nodes [4], bag-of-words/TFIDF [5], word embed-
ding [6], l-level characteristic vector [7], etc. Lee et al. [4] use a vector to represent 
an AST. Each dimension of the vector stands for a unique syntactic element of AST. 
There are 261 syntactical elements in total. They use a variant of $R$ tree, the $R^*$ tree, 
to index the vectors. Nevertheless, 261 is too large for the $R^*$ tree to work. They re-
duce dimensionality by removing a majority of dimensions with high variances across 
the data set. Their work is similar to ours since they vectorize their code and re-
duce dimensionality. Compared with other methods, our dictionary-based approach 
is better in reflecting multiple sides of the code by incorporating multiple features or
code representations.

Our method is also closely related to dimensionality reduction. Two widely used dimensionality reduction techniques are principal component analysis (PCA) [44] and canonical correlation analysis (CCA) [45]. Intuitively, PCA uses a set of orthogonal principal components (directions in the space) to capture the most variance in the data. The first principal component is in the direction of the most variability. The second principal component is in the direction orthogonal to the first principal component and captures the second largest variability in the data, which is uncorrelated to the first one due to the orthogonality. Hence, PCA can use only a small set of principal components to capture most of the variance in the data, which achieves dimensionality reduction. CCA is for two sets of correlated variables, $X = (x_1, ..., x_n)^T, Y = (y_1, ..., y_n)^T$. Canonical variables $a$ and $b$ are two vectors (directions) for $X$ and $Y$ respectively. Canonical variables maximize the correlation $\rho = corr(a^T X, b^T Y)$. The first pair maximizes the correlation. The second pair are maximizing the same correlation but uncorrelated with the first pair of canonical variables. As a result, CCA will find most of the correlation in the data with a small set of canonical variable pairs, which are linear transformations of $X$ and $Y$. CCA decreases the number of dimensions needed to keep the correlation of two sets of correlated variables. CCA is closely connected to our dimensionality reduction model and is one of the fundamental reasons why our model will work.
Chapter 6

Conclusion

In this thesis, we present a novel dictionary-based approach to representing a piece of code as a set of similarities to prototype code pieces. We show our dictionary-based approach is able to incorporate arbitrary features and arbitrary similarity functions for the code similarity search. We recognize some dimensionality reduction is necessary for our dictionary-based representation of the code. We propose and learn a supervised model to project data into a low-dimensional latent space using the Stack Overflow data. Our experiments show our model is promising in reducing dimensionality.
Bibliography


on Object-oriented programming, systems, languages, and applications, pp. 140–141, ACM, 2005.


[38] B. Kulis and K. Grauman, “Kernelized locality-sensitive hashing for scalable


