Data Dependence and Indecisiveness for Locality-Sensitive Hashing

Master Thesis

Supervisor: Dr. Michael Cochez
First Examiner: Prof. Dr. Stefan Decker
Second Examiner: Dr. rer. nat. Walter Unger

Iraklis Dimitriadis
Computer Science 5 - Information Systems
RWTH Aachen University
Aachen, June 14, 2019

Abstract

This thesis proposes three different extensions to Random Hyperplane Hashing, a common hashing family for LSH Forest. Hashing similar data points by randomly generated hyperplanes might cause them to be separated and lower the quality of nearest neighbor queries. To tackle this issue, the proposed extensions introduce a fuzziness area for hyperplanes that defines points within it as indecisive and hash them into both subspaces generated by the hyperplane. As a result, data points on different sides of the hyperplane are still hashed to the same corresponding node of the LSH tree and improve the quality of the nearest neighbor queries. While for the first extension the fuzziness area is fixed, the second extension allows the area to be variable, so that a certain fraction of points hashed by a node is covered. The last approach is a mix both approaches, so that, if a certain fraction of points is within that area, all points being hashed by this node will be hashed as indecisive. For comparison, an experimental analysis is performed for three different datasets and provide an empirical performance overview of each extension.

Contents

1 Introduction

During the last couple of decades, the internet developed rapidly and has taken a central role in information exchange. It connects billions of users, companies and data centers, that are generating and exchanging a huge amount of data. This development has opened new opportunities in information retrieval and exchange, as many platforms can provide much more information to their customers and companies have more data to process. Despite all the opportunities that came with the development of Big Data, new kinds of problems arose. Huge amounts of data also include less important information, thus processing the data in order to gain only the important kind of information has become a major task in data processing. Furthermore, Big Data often includes data records with similar information. Albeit they do not provide any new information, they increase the computational effort to process the dataset. Thus, data deduplication is an efficient way to decrease the computational complexity. However, due to the sheer amount of data, processing them has become a challenging task and demands new concepts and algorithms to process them fast and efficiently, without losing any kind of necessary information.

Querying similar or close records in huge datasets has become of major importance in several domains like data compression [**ricci1999data**], data mining [**han2011data**], computer vision [**zhang2006svm**], and other data related areas [**Shakhnarovic2006ANNTheory**]. The (k)-Nearest Neighbor Search problem (NNS or k-NNS) addresses the issue of finding the most similar pairs in a dataset. In NNS, every data record is considered to be a point in a predefined metric space. The idea is to provide a data structure with a set of n given points P, so that for any query point q the set of (k) *closest* points to the query are returned quickly [**Arya94anoptimal**]. There are many known algorithms to solve the NNS efficiently for low dimensional data [**wu2008top**]. However, when dealing with high dimensional data, all of them are either suffer from slow query time or exponential space complexity [**andoni2015optimal**]. Imagine a simple Audio-On-Demand Service, that contains billions of songs and adds new songs to its library continuously. Due to the huge amount of accessible songs, it is helpful for customers to have a list of similar songs recommended, so that one can easily browse through the whole available library. In order to do so, a nearest neighbor search can be used to provide a set of songs, by comparing the current song to each existing song in the library and return the most similar ones. While this naive approach might work for small music libraries, it can not be done efficiently when the amount

of available songs is too big. Even with optimized and task specific k-NN algorithms it is still necessary to compare a huge amount of songs and thus, not possible to be done in a reasonable time. The problem with dealing with high dimensional data in NNS is often referred to as the "curse of dimensionality" [**andoni2015practical**].

Despite the effort in the last years, current solutions for NNS are still unable to overcome this "curse". However, many applications nowadays prioritize time efficiency higher than obtaining the optimal set of nearest neighbors. One can think back to the example of the Audio-On-Demand Service: Customers do not have an advantage of a precise set of similar songs, but are irritated from slow recommendations in their user experience. In fact, many customers do not care about the exactness of recommended songs, but are satisfied if this set of songs matches the genre, style of music or the artist. Therefore, an approximation of the "most similar songs" is sufficient. By allowing an inaccurate (but still good enough) set of near neighbors, the already known Nearest Neighbor Search problem can be reduced to an approximation of it, also known as the approximate (k)-Near Neighbor Search problem (ANN or k-ANN) [**andoni2014beyond**]. Instead of restricting the data structure to return the closest points (neighbors), it is allowed to return any set of data points within a radius. To be more precise, the data points of the result must be within radius cr, where r represents the max distant of the near neighbors to the query point and $c > 1$ the approximation factor of it. This opens new opportunities to create faster and efficient algorithms for Near Neighbor Search.

Locality Sensitive Hashing (*LSH*) has been developed to address the ANN problem and has proven to be an efficient and reliable way in many application domains [**har2012approximate**]. In fact, it provides the best data structure to query points in sub-quadratic space with a constant approximation factor, which has been a key factor for ANN in practice [**andoni2015optimal**]. The core idea of LSH is to hash points such that close points (distant at most r) are more likely to collide with those far away (distant at least cr). As a result, points with the same hash value are considered to be similar. Each used hash function has specific requirements and is therefore grouped in so called LSH-Families, as we will see later. Generally speaking, every point is distributed to a bucket, where its bucket is determined by the outcome of the hashing process. Given such hash functions as mentioned above, close points are likely to have the same computed hash values and will therefore share the same bucket, where far away points are expected to be hashed in other buckets and thus be stored separately. Thus, each bucket contains a set of close points and is represented by the outcome of the hash functions. In the query process, the query point is hashed with same hash functions and distributed to one of the buckets. Now, rather than computing the similarity with each point within the dataset, the query point is only compared to points that share the same bucket. As a result, the amount of comparisons is reduced to the amount of points within the bucket. It is also possible to replace the bucket structure by a tree structure like Bawa et al. introduced

in [**bawa2005lsh**]. This variant of LSH is called LSH Forest (LSHF) and will be excessively studied within this thesis. Instead of distributing points to buckets, a forest consisting of several trees is used as a data structure, where the data points are stored in leafs of the trees. The near neighbor set is then constructed by points that share the longest prefix path as the query point. This structure achieves better results in many practical applications and is therefore considered to be superior to the LSH bucket structure [**bawa2005lsh**].

Hashing data points can be done in many different ways and depend on the given distance metric. Considering the angular distance as metric, the vector space is divided by randomly generated hyperplanes. These planes cut the space into subspaces, where each subspace represents a bucket. Now, any point within that subspace is distributed to the corresponding bucket (or leaf). This hashing technique is also known as *Random Hyperplane Hashing (RHH)*.

Compared to other hashing techniques, RHH gives the opportunity to improve the hashing procedure of points and thus also improves the overall result of the near neighbor search. However, this also introduces a new range of problems, that can have a negative effect to the performance of LSH. Firstly, if points are too close to a hyperplane, it might be better to not make a decision, as this can distribute points to the wrong subspace. Secondly, the hyperplane generation does not take any information about the dataset into account. Especially when the dataset contains dense clusters of points, it can happen that a generated hyperplane separates those points into different subspaces and does not consider them as near neighbor to each other. Hence, the information of such dense points should also be taken into account when generating hyperplanes.

In contradiction to locality-sensitive hashing, data-dependent hashing offers ways to generate hyperplanes dependent of the dataset. While this will improve the quality of the near neighbor search, it will worsen the runtime of the LSH algorithm, as gaining additional information of the dataset will necessarily need more computational effort. As an alternative solution, the locality information can be used to simulate some data-dependency without slowing down the hashing procedure. By taking and processing the information that are already computed by RHH itself, some improvement can be done in order to deal with the aforementioned problems as this thesis will show.

Course of this thesis

This thesis will propose three extensions to Random Hyperplane Hashing in order to improve the correctness of the near neighbor search in LSH. These extensions will be based on the idea of Cochez et al. [**cochez2017large**], which introduces the ability to hash points into multiple subspaces. By defining a small area around the hyperplane, points that are within this area will be considered as indecisive and assigned to both subspaces generated by the hyperplane. Thus, points can be hashed into multiple leafs of a LSH Forest tree and lower the

probability of separating similar points. By doing so, one can directly address the problems that arise with points that are too close to a hyperplane. As a result, the overall correctness of the near neighbor search can be improved.

In the first variant, the indecisive area will be given by a fixed radius around the hyperplane. This algorithm is already known as *Fuzzy Random Hyperplane Hashing* [**cochez2017large**]. The second one will be referred to as *percentage-based Random Hyperplane Hashing* and will hash the $n\%$ closest points to the hyperplane as indecisive. The last variation is *indecisive-based Random Hyperplane Hashing* and represents a mix of the other two RHH variants. Instead of hashing points as indecisive, the hashing outcome of the hyperplane will be ignored for all points, if $n\%$ of the points are within a fixed angle around this hyperplane. While the Fuzzy Random Hyperplane Hashing will remain as presented in [**cochez2017large**], the last two variants are completely new approaches.

All hyperplane variants will only use the information that is obtained by normal Random Hyperplane Hashing and thus can be done without additional computational effort. However, hashing points as indecisive will increase the number of nodes in the data structure and worsen the space and time complexity of LSH Forest. Therefore, it remains to proof if the benefits outweigh the shortcomings of this trade. In order to do so, this thesis will analyze the performance of each extension by testing them for different scenarios. The results of these experiments will be discussed and show if any of those variants can compete with normal RHH in some use cases.

2 Use Case

Locality-sensitive hashing presents an efficient way to solve the nearest neighbor problem for large scale applications. Its diversity and simplicity allows the algorithm to be modified and optimized for any related use case scenario. This section will provide an overview of common areas where LSH is used in its plain or modified version, where general and more concrete examples will be discussed.

2.1 Matching News Article

In [**leskovec2014mining**] the problem of detecting similar news articles was discussed and is very related to detecting similar documents. When organizing repositories of online news article as it is done by many news feeds, it is common to group them together by news article from the same source in order to avoid repeating the same news multiple times. Bigger news companies produce news article and distribute them to different online news platform. Each of those platforms putting the news article online, but adding additional information, such as the platform names, link and advertising. Furthermore, the news article can be published as a modified version, since adding sentences to the article or leaving out smaller parts of the origin text is common. As a result many web sites can offer the same news derived from the same source in a different appearance. In order to avoid publishing the same news multiple times, it is necessary to detect news with similar information or news derived from the similar news source and only distribute them once. This task is closely related to the problem of finding similar documents as both are based on the idea that similar elements will have repetitive text parts in their text body, and therefore LSH is suitable as a tool to do so. However, finding similar news articles differs from documents, such that ignoring specific parts of the news article, like links, headline or advertising improves the overall detection. In addition, texts, as they appear in books or other documents, differ from news articles in a sense that they have a higher frequency of stop words. Therefore, the text has to be preprocessed by detecting and leaving out unnecessary parts and the set of stopwords have to be chosen more carefully in order to still be able to detect. Afterwards, the news article can be tokenized as described in Section 4.3.1 and compared based on their Jaccard Similarity.

2.2 Recommendation Systems

The task of recommendation systems is to recommend a set of items for any given query item. Reducing this task to its core functionality, it becomes essentially the nearest neighbor search problem. As most recommendation system are working with huge datasets in order to recommend the best possible set of similar items, the task of finding the nearest neighbor to a query item becomes unfeasible. For this reason, approximating the set of nearest neighbors by using Locality-sensitive hashing is a common approach to overcome this issue. In fact, many companies, whose business model are build on recommendation systems, are actively researching in the domain of LSH. Alphabet (formerly Google) uses LSH along with their PageRank to improve their image search technology VisualRank [**35244**, **Jing:2008:VAP:1444381.1444396**]. The basic idea is to generate feature vectors of items, where each entry represents a specific feature of the set. In order to to measure the similarity between a query and all items, the angular distance is computed and used as a score to rank the near neighbor candidates. Although most recommendation engines involve other processes to improve these results further, LSH is an essential part to enable a fast and reliable selection of near neighbor candidates.

2.3 Fraud Detection in Uber Systems

Uber is, with five million daily trips world wide, one of the largest transportation network companies [**uber**, **uber2**]. In order to detect fraudulent drivers, Uber is using LSH to detect similar trips based on their spatial properties. A trip is identified by GPS signals represented as a list of latitude, longitude and time tuples. Since these signals are tracked by mobile devices of customers and drivers, the quality of the GPS information can vary a lot. This makes the detection of similar trips harder, as smaller noises can distort the quality of the signal, and therefore same trips can be represented by different signal sequences. Furthermore, similar trips can be completed in different times, due to road traffic or other reasons. As a result, same trips that are completed in a different time window will differ in their frequency of signals. In order to deal with those issues, each signal is converted into a S2 Cell. These Cells are representing a hierarchical decomposition of the earth sphere, so that each Cell represents an area of the earth by an unique index. By doing so, a sequence of latitude, longitude and time tuples is represent by a set of S2 Cell indexes and enables to compare similar trips based on the Jaccard Similarity of those sets. However, since trips involving also the direction, such that two opposite trips are not considered as the same trip, the indexes are shingled so that two consecutive S2 Cells represent an element in the set representing the trip. Therefore, two opposite trips will be represented by different pairs of indexes and their Jaccard Similarity will be low.

Detecting similar trips by computing the Jaccard Similarity of all pairs is unfeasible, since the set of existing trips is very large. To improve the runtime, Uber uses LSH to divide the set of trips in smaller groups of similar trips. This reduces the amount of trips that need to be compared into a small subset and enables to detect similar trips much faster.

3 Background and Related Works

The recent effort to find a solution for the near neighbor search in high dimensional datasets showed the importance of this topic. Despite the existence of efficient solutions for the low dimensional case [**clarkson1988randomized**, **bentley1975multidimensional**, **meiser1993point**], the high dimensional case is believed to have no solution that guarantee the correctness of the near neighbor set, while returning it in a reasonable time. This is due to the "curse of dimensionality", a phenomenon that describes the problem of exact near neighbor algorithms becoming inefficient if the dimensionality of the dataset is high. To overcome this issue, different approximation algorithms were developed in order to solve the near neighbor problem in a reasonable time. Some existing methods use projections in order to project the original data into lower dimensions representing value of the dimension by either one or zero using hashing functions. The data complexity is then reduced significantly so that returning a set of near neighbors can be done very fast. To do so, these algorithm uses hash functions that can be mainly categorized in locality-sensitive and data-dependent hashing functions.

Locality-sensitive hashing functions represent a class of hash functions, where each hash function is given by a randomly generated projection function and hashes the points by their locality in a given metric space [**andoni2006near**]. Due to their random generation, these functions are independent of the dataset and allow a fast generation. As described in the introduction, close points are likely to have the same hash value computed and grouped into buckets of similar points. This property enhances the query to perform the near neighbor search only on points that have the same hash value computed. Locality-sensitive hashing represents the most popular algorithm that hashes points based on their locality and is an approximation algorithm that solves the nearest neighbor search problem time efficient, while the guarantee for correctness is very high. First mentioned in [**andoni2006near**] this algorithm was steadily advanced and due to its flexibility, many different extensions have been developed to be suitable for specific metrics and use cases [**gionis1999similarity**, **andoni2015practical**, **buhler2001efficient**, **bhushan2015big**]. The most remarkably change to this algorithm was presented by Bawa et al. in [**bawa2005lsh**] and replace the bucket structure of the original LSH with a tree structure, that showed to allows faster queries in many practical applications and proven by [**andoni2015practical**]. Being key components of this thesis, LSH and LSH Forest will be excessively discussed in Chapter 4 and provide a general introduction into how this algorithms work.

Unlike locality-sensitive, data-dependent hashing functions are explicitly generated uniquely for the given dataset. In particular, these hash functions are learned from the training data by usually involving some feature of the dataset. As a result, each hash function is tailored and designed to optimize the outcome of the near neighbor search for a given dataset. Some representative data-dependent hashing functions are given by Spectral Hashing [**abdullah2014spectral**, **weiss2009spectral**, **joly2011random**] that is based on graph partitioning theory, K-mean based Hashing [**jegou2011product**], Random Maximum Margin Hashing [**joly2011random**] and Isotropic hashing [**kong2012isotropic**]. All named hashing functions differ in their hashing generation, since data-dependent hashing involves more information about the data and thus allows different ways for hashing. Most recent research showed that the improvements of data-dependent hashing outperforms the quality of the near neighbor in locality-sensitive hashing, as they providing better selectivity of the data [**weiss2009spectral**]. However, this improvements occurred only in specific cases and have the general drawback that the hashing procedure becomes slower as for locality-sensitive hashing functions, due to data dependency.

In order to improve the quality of the near neighbor search in Locality-sensitive hashing Cochez et al. introduced Fuzzy Random Hyperplane Hashing (f-RHH) [**cochez2017large**]. Unlike traditional Random Hyperplane, f-RHH allows data points to have two hash values for the same dimension computed. The dimension is defined by the amount of hyperplanes, so that each hyperplane represents one dimension. By applying a small angle around the generated hyperplane, data points that are within this area are considered as indecisive and will have multiple hash values computed. Experiments in this work showed that f-RHH increases the quality of the near neighbor set, but worsen the space complexity of the LSH algorithm. Fuzzy Random Hyperplane Hashing will be excessively discussed in Chapter 5, as this idea will be extended and used for alternative Random Hyperplane Hashing approaches proposed by this thesis.

4 Locality-Sensitive Hashing

The task to find nearest neighbors in datasets is very common and of major importance in many areas. It is used to find duplicates, similar objects or even for clustering. Although many techniques exist to solve this problem efficiently, they are not performing good enough, if the dimension of the dataset is high. Consider the following example:

Imagine a dataset D which consists of elements R^d where d is the dimension of properties describing each element. Furthermore, we define two elements to be nearest neighbors to each other, if they share the exact same properties. Now assume, two elements $a, b \in R^d$ are given. We can check if those two elements are nearest neighbors to each other by comparing each property of a to each property of b. Hence, $\mathcal{O}(d^2)$ operations are needed to check if two elements are near neighbors.

We can find the set of near neighbors for an element by simply repeating this step on multiple other elements and select those, that are the closest. Although this brute force approach will return the exact set of nearest neighbors, it depends on the size of the dataset if this can be done in a reasonable time. As the dimension of the properties growths, this algorithm will become slower. For example, due to the "curse of dimensionality" the kd-tree needs to visit more paths, which let the algorithm perform only slightly better than linear search over all points [**andoni2006near**].

Approximating algorithms have proven to be a solid way to overcome this issue. Although these algorithms can not guarantee to deliver the exact set of near neighbors, they will do it most of the time. In particular, they may not provide the exact set of near neighbors, but will return a good approximation of it, which is good enough for most domains, where the near neighbor problem is relevant.

Locality-Sensitive hashing (LSH) is one of those approximation algorithms and has been proven to be the most solid one in terms of run time and space complexity [**andoni2015optimal**]. This section will provide a general introduction to the LSH algorithm. First, we will describe the main algorithm and see how the hashing of items is done. This also covers the explanation of metric spaces, distance metrics, and Locality-sensitive hashing families with some concrete examples. In the end, we will discuss the most popular extension of LSH, that improves the LSH algorithm by introducing a new data structure.

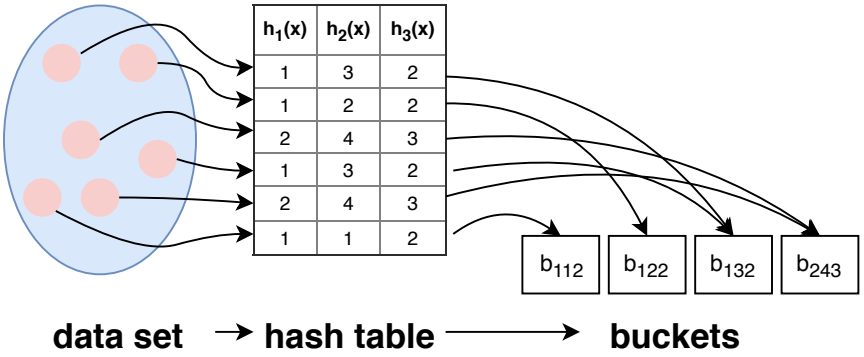


Figure 1: Schematic drawing of buckets in LSH

4.1 The Algorithm

Locality-sensitive hashing is an algorithm to identify approximately the set of nearest neighbors. Its idea is to hash points using several hash functions such that close points are more likely to "collide" than points far apart. More formally, two points are close, if the distance between them is at most r, and far apart if its cr, where c is the approximation factor for the radius r [**andoni2015practical**]. Each point is then mapped to a bucket, such that the bucket corresponds to the outcome of the hashing procedure. Hence, points that are very likely to collide will be mapped to the same bucket with a high probability. Therefore, each bucket represents a set of points that are very likely to be close to each other. Given such a data structure, one can retrieve the set of nearest neighbor by hashing the query point into a bucket and return all points inside it.

Figure 1 illustrates a single mapping procedure in LSH. Three hash functions $h_1(x), h_2(x), h_3(x) \in \mathcal{F}$ are given to hash a point. So in this case, each point will have exactly three hash values, one for each hash function. These hash values taken together are also known as the hash vector or fingerprint of a data point. Now, each point is mapped to a bucket, such that the label of the bucket is corresponding to the hash vector. Since the hash functions will likely generate the same hash value for data points that are close, each bucket represents a set of similar items.

To increase the accuracy of the near neighbor set, one can add another layer of buckets by repeating the mapping procedure with another set of hash functions. The query point is then hashed to multiple buckets, hence multiple sets of near neighbors are available. If now a point shares the same bucket (in any of the bucket layers), it will be considered as a near neighbor candidate. To retrieve the set of near neighbors, one can either return all candidates or filter them by doing an exact near neighbor search on the candidate set and return only the closest ones.

In order to measure the closeness of two points, a distance between them has to be de-

fined. This is done by projecting the points into an metric space, where the locality of each point is defined uniquely and thus their closeness can then be computed by a distance function. The next subsection will give a short overview and show how such metric spaces are defined.

4.1.1 Metric Spaces

A metric space is a space in which every distance between two elements is defined. In particular, each pair of elements is mapped to a real number.

To measure the distance between two points, a metric function (also called distance function) $d(x,y)$ is used. A metric is defined by the following properties:

Definition 1 (metric)**.** *Given a set S a metric is defined as a function $d(x,y)$ so that each $x,y \in S$ is mapped to $\mathbb{R}$ and satisfies the following conditions [**leskovec2014mining**]*

1. $\forall x,y \in S, d(x,y) \geq 0$ *(Distance is positive)*
2. $d(x,y) = 0 \iff x = y$ *(Only identical points have a distance of 0)*
3. $d(x,y) = d(y,x)$ *(Symmetry)*
4. $d(x,y) \leq d(x,z) + d(z,y)$ *(Triangle inequality)*

There are many different distance functions, but not all of them are commonly used with LSH. We will introduce some of them and focus on these, that are used in this thesis.

Euclidean Distance

The *Euclidean Distance* is the most popular distance metric. It is defined on an euclidean space, where points are represented by n-dimensional vectors of real numbers. Given such space, the euclidean distance $d_e(\vec{x},\vec{y})$ can be computed based on different norms. For example, the L_2-norm is defined as straight distance between two given points and can be computed as follows [**leskovec2014mining**]:

$$d_e(\vec{x},\vec{y}) = d_e([x_1,x_2,..,x_n],[y_1,y_2,..,y_n]) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Jaccard Distance

The *Jaccard Distance* $d_j(A,B)$ is a distance metric defined on sets. It computes the similarity between two sets, by using the *Jaccard Similarity* of those. In particular, the similarity between two sets is defined by the ratio between their intersection and their union. Subtracting the similarity from 1 defines the Jaccard Distance as a real number between 0 and 1. This

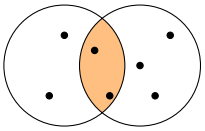


Figure 2: The highlighted area shows the intersection of A and B, including two points

distance metric is mostly used with document and text based elements, as we will see later. The Jaccard Similarity can be defined as follows:

Definition 2 (Jaccard Similarity)**.** *Given two sets A and B, the Jaccard Similarity is defined as*

$$SIM_j(A,B) = \frac{A \bigcap B}{A \bigcup B}$$

Given this, the Jaccard Distance is defined as follows:

Definition 3 (Jaccard Distance)**.** *Given two sets A and B, the Jaccard Distance is defined as*

$$d_j(A,B) = 1 - SIM_j(A,B) \tag{4.1}$$

where d_S is the Jaccard Similarity between them.

Figure 2 shows two sets A and B, where A has 4 elements and B has 5. The highlighted area shows the intersection area between them, which in this case is 2. With these two together, we can compute the Jaccard Distance for this example as $d_{jd}(A,B) = 1 - \frac{2}{7} = \frac{5}{7}$.

Cosine Distance

The *Cosine Distance* (also called *angular distance*) is one of the most popular distances for LSH and is used as main metric in this thesis. It is defined for spaces with a dimension, such that points can be considered as vectors as shown in Figure 3. Rather than to distinguish vectors by their length, the Cosine Distance is defined as an angle between two vectors. On any dimension, the angle between two vectors always ranges from $0°$ to $180°$, which is then used to define the distance.

The cosine distance is 1 minus the cosine similarity of two vectors:

Definition 4 (Cosine Similarity)**.** *Given a n dimensional space $\mathbb{S}$, the Cosine Similarity of two*

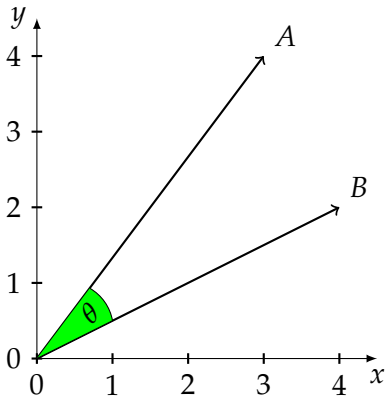


Figure 3: Represents the angle θ between A and B

vectors $a, b \in \mathbb{S}$ is defined as

$$SIM_{cos}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \cdot \left\|\vec{b}\right\|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \sqrt{\sum_{i=1}^{n} b_i^2}}$$

Hence, the definition for the Cosine Distance is as follows:

Definition 5 (Cosine Distance)**.** *Given an n dimensional space $\mathbb{S}$, the Cosine Distance of two vectors $a, b \in \mathbb{S}$ is defined as*

$$d_c(\vec{a}, \vec{b}) = 1 - SIM_{cos}(\vec{a}, \vec{b})$$

where $SIM_{cos}(\vec{a}, \vec{b})$ is the Cosine Similarity.

If two vectors have the same orientation the Cosine Similarity between them is 1, and therefore their distance is 0. If two vectors are orthogonal to each other, their similarity will be 0 and the distance 1.

4.2 LSH-Family

A *LSH-Family* defines a set of Locality-sensitive hash functions. Unlike traditional hashing, these hash functions ignore smaller "distortions" of close points. To do so, the points are hashed respective to their locality in a metric space, while all points within a small locality area hashed to the same hash value. As a result close points are likely to have same hashing outcome, which has the same effect as ignoring smaller differences in their locality. How small the difference can be is defined by the LSH-Family of the hash function [**cochez2016taming**]:

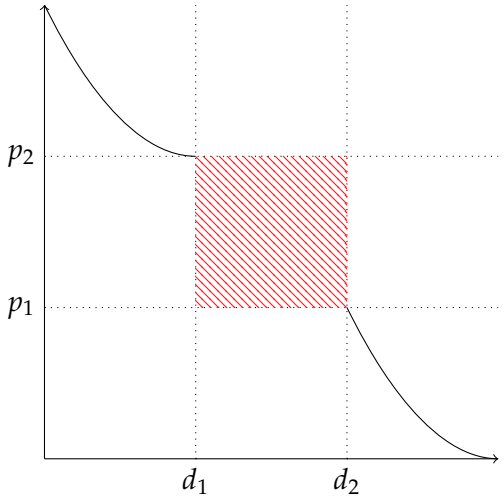


Figure 4: Illustration of LSH-Family boundaries (inspired by [**leskovec2014mining**])

Definition 6 (LSH-Family)**.** *Given a distance function d and two distances d_1 and d_2, a family of functions $\mathcal{F}$ is called (d_1, d_2, p_1, p_2)-sensitive if for every $f \in \mathcal{F}$ the following conditions are true:*

1. *If $d(x, y) < d_1$, then the probability of $h(x) = h(y)$ is at least p_1,*
2. *If $d(x, y) \geq d_2$, then the probability of $h(x) = h(y)$ is at most p_2,*

where $p_1 < p_2$ and $d_1 < d_2$.

The parameter d_1 describes the point when the distance between points is considered to be close, which in return means that points are very likely to collide. In this case, the probability that h will compute the same hash value for x and y is defined by p_1. Condition 2 is similar to 1 with the difference, that it defines the distance and probability for points far apart. In addition, p_1 has to be smaller than p_2 and d_1 smaller than d_2. Those parameters together define the (d_1, d_2, p_1, p_2)-sensitive LSH family. By defining such a family, any hash function that is member of it will has the same probability to hash close and points far apart to the same hash value.

Figure 4 illustrates what the bounds for a LSH-Family can look like. As we can see, there is still a space between p_1 and p_2 that is undefined. This space represents the approximation area of candidates for the near neighbor set. In other words, if an element is within this area, it can still be considered as an element in the near neighbor set. We can decrease the size of this area by repeating the hashing of two points with another hash function. This step is called *amplification*.

Amplification

As mentioned before, each (d_1, d_2, p_1, p_2)-sensitive family $\mathcal{F}$ has an undefined area between its two distances. This area can be considered as the approximation area in a sense that points that are within this area can still be considered as close points, although they are not. Selecting d_1 and d_2 closer to each other reduces the approximation area, but typically means that p_1 and p_2 will be closer to each other.

In order to keep p_1 and p_2 far apart, while moving d_1 and d_2 closer to each other, one can use the effect of *amplification*. We define a new Family $\mathcal{F}'$ where each member $h' \in \mathcal{F}'$ is constructed by a set of functions of $\mathcal{F}$. Given such a family, we will distinguish between two different types of amplifications [**leskovec2014mining**]:
´

Definition 7 (And-construction)**.** *Given a (d_1, d_2, p_1, p_2)-sensitive family $\mathcal{F}$ and a fix $r \in \mathbb{N}$, a family $\mathcal{F}'$ is $(d_1, d_2, (p_1)^r, (p_2)^r)$-sensitive if:*

1. *if $h \in \mathcal{F}'$, then h is constructed from a set $\{h_1, h_2, ..., h_r\} \in \mathcal{F}$.*
2. *if $h(x) = h(y)$ is true, then for all $i = 1, 2, ..., r$ each function $h_i(x) = h_i(y)$ is also true.*

Definition 8 (Or-construction)**.** *Given a (d_1, d_2, p_1, p_2)-sensitive family $\mathcal{F}$ and a fix $r \in \mathbb{N}$, a family $\mathcal{F}'$ is $(d_1, d_2, 1-(1-p_1)^l, 1-(1-p_2)^l)$-sensitive if:*

1. *if $h \in \mathcal{F}'$, then h is constructed from a set $\{h_1, h_2, ..., h_r\} \in \mathcal{F}$.*
2. *if $h(x) = h(y)$ is true, then there is at least one $i = 1, 2, ..., r$, where $h_i(x) = h_i(y)$ is also true.*

The effect of an And-construction can be best explained with the bucket structure. Consider $H = \{h_1, h_2, ...h_r\}$ to be the set of functions of which $h \in \mathcal{F}'$ is constructed of. Each function of H has its own layer of buckets and thus an individual mapping. Now, if a data point a and the query point q share the same bucket on each layer, they are considered to be similar. This is equal to $\prod_{i=0}^{r}(h_i(a) = h_i(q)) = 1$. Knowing this, the probability of $\mathcal{F}'$ is straight forward: Since each function of H has the probability of p, the probability of h is given by

$$p_h = \prod_{i=0}^{r} p_i = (p)^r.$$

The Or-construction can be explained quite similar. Again, we consider H as the set of functions of $h \in \mathcal{F}'$, where each function has its own layer of buckets. Now, unlike before, the Or-construct considers a data point a to be similar to a query point q if they share the same bucket on at least one layer. This in return is similar to $\sum_{i=0}^{r}(h_i(a) = h_i(q)) \geq 1$. $1-p$ is the probability that a function of H will define a and q as not similar, therefore the probability that that all hash function will define them as not similar is given by $(1-p)^r$. Knowing this, we can compute the probability of a and b being at least for one of the functions similar by

$$p_h = 1 - \prod_{i=0}^{r}(1 - p_i) = 1 - (1 - p)^r.$$

The benefit of using such constructions is, that depending on how the functions in $\mathcal{F}'$ are constructed, the probabilities can be moved either closer or far away from each other. By using an And-construction we can keep the probability p_1 close to 0, while having p_2 far away from 0. The Or-construction on the other hand can keep p_1 close to 1 and p_2 far away from it. If we recall Figure 4, we can now change the undefined space between p_1 and p_2, without changing d_1 and d_2, since amplification only affects the probabilities.

4.3 Concrete LSH-Families

This section will introduce two LSH-Families and discuss how they are constructed. The first one is called *Minhashing*, which refers to an algorithm that maps a data point to its minimum hash. We will use a concrete example to explain Minhashing and see how it is usually applied when working with documents. However, documents can not be used directly, therefore we will introduce two alternative representations of them [**broder1997resemblance**, **leskovec2014mining**].

The second one is *Random Hyperplane Hashing*, which uses a certain amount of random hyperplanes to divide the given space and distribute points to subspaces. This algorithm measures the distance based on the cosine distance from Chapter 4.1.1 and therefore needs the data points as vectors.

After explaining both algorithms, we will show that they are indeed a (p_1, p_2, d_1, d_2)-Family by proving so.

4.3.1 Minhashing

Minhashing was invented by Andrei Broder [**broder1997resemblance**] and refers to an algorithm that maps a data point to the minimum hash of it. Each data point is represented by a set, where each element contains a different information of the data point. Afterwards, the elements are hashed by a function h and the lowest hash value of all elements is taken to represent this set. By repeating this process multiple times, with an fixed order of hash functions, the data point can be compressed to a sequence of hash values. This sequence represents the fingerprint of the data point, as it was described in the introduction to this chapter. Minhashing a data point as described above has two advantages: Firstly, the size of a data point becomes smaller. And secondly, we can reduce the amount of comparisons drastically.

Imagine we have a set of documents and want to apply the Minhashing algorithm. Representing a document by a set is usually done by dividing the text of the document in smaller strings. One way to do so are k-Shingles, which are defined as follows:

Definition 9 (k-Shingles [**broder1997resemblance**])**.** *Given a continuous sequence of characters D, $S(D,k)$ is defined as the set of all k consecutive character sequences in D.*

Especially for documents, reducing them to a set of smaller strings has proven to be an effective way in order to improve further text analysis [**leskovec2014mining**]. Based on the use case, these strings can be a sequence of k consecutive characters, words, or whole sentences. The advantages of representing documents in such sets is that lexical similarity between them can be computed by calculating the intersection of those sets. Another advantage is that neither the order nor the frequency of a string is important, which are unnecessary factors in lexical similarity.

Shingling Documents is one of the simplest, but most common approach to generate such sets. Given a document D, one can define its *k-Shingle* as $S(D,w)$ being every sub sequence of the length k within document D.

Assume a shingle is defined by a sequence of consecutive words. For a given document

$$D = \texttt{"processing large data sets"}$$

$S(D,2)$ is defined as the set of 2 consecutive words:

$$S(D,2) = \{\texttt{"processing large"}, \texttt{"large data"}, \texttt{"data sets"}\}$$

Now that the documents can be represented as sets, the next step is to generate the fingerprint of it. This is done by taking advantage of Minhashing, which is defined as follows:
Definition 10 (Minhash)**.** *Minhashing defines a family of functions $h_\pi(K) = min\{\pi(k)|k \in K\}$, where π is a random permutation of the given universe $\mathcal{U}$.*

We will explain Minhashing with an example. Assume the set of two documents D_1 and D_2 are given as follows:

- $S(D_1,2) = \{$processing large, large data, data sets $\}$
- $S(D_2,2) = \{$working with, with large, large data, data sets $\}$

Additionally, we assume that each element of the universe $U = \bigcup_{i=1}^{n} S(D_i,k)$ is mapped to a unique number. Moreover, we can represent them in a matrix (inspired by [**leskovec2014mining**]), where the rows represent an element of $\mathcal{U}$ and each column represents the set of a document. Notice that the order of elements is defined by Minhashing the universe $\pi(\mathcal{U})$. The entry (x,y) of this matrix indicates if the shingle x is present in document y. In case x is present in y, the entry will be set to 1, otherwise it will be set to 0.

Figure 5 shows two tables, where the left table is the matrix before replacing the elements

	$S(D_1,2)$	$S(D_2,2)$
processing large	1	0
large data	1	1
data sets	1	1
working with	0	1
with large	0	1

$=>$

	$S(D_1,2)$	$S(D_2,2)$
0	1	0
1	1	1
2	1	1
3	0	1
4	0	1

Figure 5: matrix representation of sets over the universe $\mathcal{U}$ [**leskovec2014mining**]

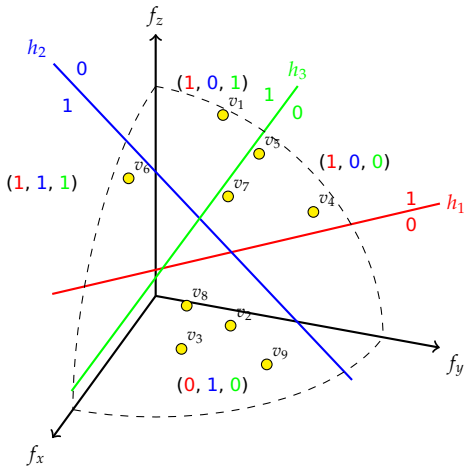
of universe $\mathcal{U}$ with numbers. What the Minhash algorithm does is it permutes the universe $\mathcal{U}$ and selects the first occurring element of a set for the given permutation of $\mathcal{U}$. As it was already mentioned, the elements are mapped to numbers, so in case of D_1 the Minhash algorithm will return a 0 which respectively stands for "processing large". By doing this for multiple permutations, we can generate a fingerprint for each document. However, generating a permutation over the whole universe is a costly operation. For this reason, Minhash is defined by a set of hash functions, where each function simulates a permutation. In particular, each hash function is applied on every element of a given set. Afterwards, the lowest hash value is stored as the first fingerprint value for the respective document.

We can prove that Minhash is a $(d_1, d_2, 1-d_1, 1-d_2)$-sensitive family as follows:

*Proof [***leskovec2014mining***]:*

- Assume d_j to be the jaccard distance. We can proof that Minhash is a $(d_1, d_2, 1-d_1, 1-d_2)$-sensitive family:
  - Let $d_1 < d_2$
  - If $d_j(x,y) = d_1$, the similarity between x and y is given by $1-d_1$. This is equal to the statement that $1-d_1$ items in x and y are similar. Therefore d_1 items are left to consider their hash value as the minimum for the hashing outcome. This is equal to the statement of $(1-d_1)$ being the probability that x and y will have the computed hash value.
  - If $d_j(x,y) = d_2$, the similarity between x and y is given by $1-d_2$. This is equal to the statement that $1-d_2$ items in x and y are similar. Therefore, d_2 items are left to consider their hash value as the minimum for the hashing outcome. This is equal to the statement of $(1-d_2)$ being the probability that x and y will have the computed hash value.

In other words, the probability that Minhash will permute the sets such that both will have the first row set to 1 is exactly the difference between both sets. For the example given by Figure 5, the probability that both documents D_1 and D_2 will have the first entry set to 1 is $\frac{2}{5}$.



(a)

bucket	vectors
...	...
(0, 1, 0)	v_2, v_3, v_8, v_9
...	...
(1, 0, 0)	v_4, v_5, v_7
(1, 0, 1)	v_1
...	...
(1, 1, 1)	v_6

(b)

Figure 6: (a) Illustrates subspace generation of RHH with 3 hyperplanes and (b) the bucket distribution of vectors

4.3.1.1 Random Hyperplane Hashing

Random Hyperplane Hashing (RHH) is commonly used with LSH and refers to an algorithm that hashes points with a randomly generated set of hyperplanes. The hyperplanes are generated so that each one is going through the origin of the vector space and divides it in two equal halves. The points, given as projected vector in that space, are then hashed to either 0 or 1, depending on which side of the hyperplane they are.

By repeating this step for k hyperplanes, the vector space is divided in 2^k subspaces [**rajaram2008client**], where points that are within the same subspace will be hashed to exactly the same hash value. Therefore, given a set $\mathcal{H}$ of k hyperplanes, the fingerprint of a vector v is defined by $H(v) = [h_1(p), ..., h_k(p)]$.

The hash value of a point can be computed from an explicit definition of the hyperplane. However, computing hash values with the hyperplane itself is a costly operation and therefore not reasonable. A more efficient implementation is to hash points based on their angular distance to the normal vector of a hyperplane. The angular distance between any vector v and the normal of a hyperplane n is defined as follows:

$$\arccos\left(\frac{v \cdot n}{\|v\| \, \|n\|}\right)$$

Since the angle of the hyperplane and its normal is given by $\frac{\pi}{2}$, we can compute the angle between the vector v and the hyperplane as follows:

$$\frac{\pi}{2} - \arccos\left(\frac{v \cdot n}{\|v\| \, \|n\|}\right)$$

If the outcome is positive, the vector will be hashed to 0 and if negative to 1. However, since for the hashing procedure only the side of the hyperplane is necessary, we can reduce this whole computation to $v \cdot n$ and compute the hashing outcome by:

$$h(p_i) = \begin{cases} 0, & p \cdot n < 0 \\ 1, & p \cdot n \geq 1 \end{cases}$$

Figure 6(a) shows how 3 hyperplanes h_1, h_2, h_3 divide the space in 8 subspaces. As one can see, the subspaces are not equally large, because the hyperplanes are randomly chosen. The table in (b) shows how the vectors are distributed to the different subspaces. Notice, that the left column represents the bucket in which the vectors is distributed. However, the distribution of vectors to subspaces is equal to the distribution of vectors to buckets and therefore dividing the space into subspaces serves the same purpose as hashing vectors to different buckets.

Each hyperplane is considered to be a hash function and together represent the LSH-Family for the cosine distance [**rajaram2008client**]. To show, that this family of functions is indeed a LSH-Family, we need to prove that the properties of a LSH-Family are true. The following proof will show, that RHH is a $(d_1, d_2, \frac{1-d_1}{180}\frac{1-d_2}{180})$-sensitive family for the cosine distance:

*Proof [***leskovec2014mining***]:*

- Given d_c as the cosine distance and $h(x) = h(y)$ if two vectors are on the same side of a hyperplane $h \in \mathcal{H}$. Furthermore, let d_1, d_2 being the distances and p_1, p_2 the probabilities for close and far apart vectors, where $d_1 < d_2$ and $p_1 < p_2$. We can prove that RHH is a $(d_1, d_2, \frac{1-d_1}{180}\frac{1-d_2}{180})$-sensitive family:
  - The probability a randomly chosen hyperplane h will be between two vectors x and y is equal to $\frac{d_c(x,y)}{180}$.
  - If d_1 is defined as the distance between two close vectors, we can prove that $p_1 = 1 - \frac{d_1}{180}$:
    - Since $d_c(x, y) \Leftrightarrow d_1$, the probability for close vectors is $1 - \frac{d_1}{180}$.
    - If $\frac{d_1}{180}$ is the probability that two vectors are separated by h, then $1 - \frac{d_1}{180}$ is the probability that they are on the same side, which the same as $h(x) = h(y)$.
    - If d_1 is the distance for close vectors, the probability of $h(x) = h(y)$ is $1 - \frac{d_1}{180}$.
    - Therefore $p_1 = 1 - \frac{d_1}{180}$
  - If d_2 is defined as the distance between two vectors far apart, we can prove that $p_2 = 1 - \frac{d_2}{180}$:
    - Since $d_c(x, y) \Leftrightarrow d_2$, the probability for vectors far apart is $1 - \frac{d_2}{180}$.

* Thus $p_2 = 1 - \frac{d_2}{180}$
- This holds for any random hyperplane $h \in \mathcal{H}$
- Therefore any set of random hyperplanes $\mathcal{H}$ is a $(d_1, d_2, \frac{1-d_1}{180}\frac{1-d_3}{180})$-sensitive family for the cosine distance. □

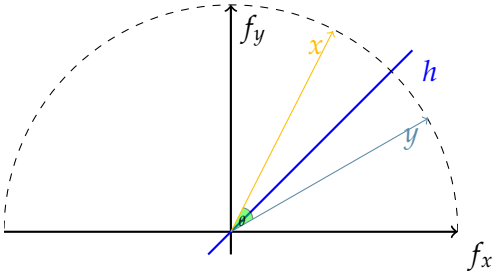


Figure 7: Illustrates a hyperplane that divides the two given vectors in an 2-Dimensional space.

Figure 7 provides a better understanding of what exactly happens. In this example, two vectors x and y are divided by a hyperplane h. The angle between those two vectors is given by θ which is equal to the cosine distance $d_c(x, y)$. Note, that θ is always the smallest angle between two vectors, and is therefore never above $180°$. Therefore, the probability that a hyperplane will be within that angle of θ is equal to $\frac{\theta}{180°}$. Given this, $1 - \frac{\theta}{180°}$ is the probability that the hyperplane h will not separate those vectors, which in turn is equal to $h(x) = h(y)$.

4.4 Runtime Complexity

The main application of LSH is to solve the nearest neighbor search problem fast and efficient. Due to the variety of existing improvements and extensions, it is impossible to argue about the runtime complexity in general. However, restricting it to be for a specific version of LSH, we can provide a formula to compute it. In order to fit into the scope of this thesis, we will explain the runtime complexity of the query when using Random Hyperplane Hashing. Furthermore, the algorithm is defined by two parameters, being k the amount of functions to compute the hash string of a point and L the amount of bucket layers.

The runtime for the LSH algorithm is given by the following definition:

Theorem 1. *Given N d-dimensional points with k hyperplanes dividing the vector space. The runtime complexity costs of LSH with L iterations is*

$$\mathcal{O}\left(Ldk + Ld\left(\frac{N}{2^k}\right)\right)$$

The left side of the equation computes the time that is needed to distribute the query point to L buckets. In order to be distributed to a single bucket, the dot product of a point and

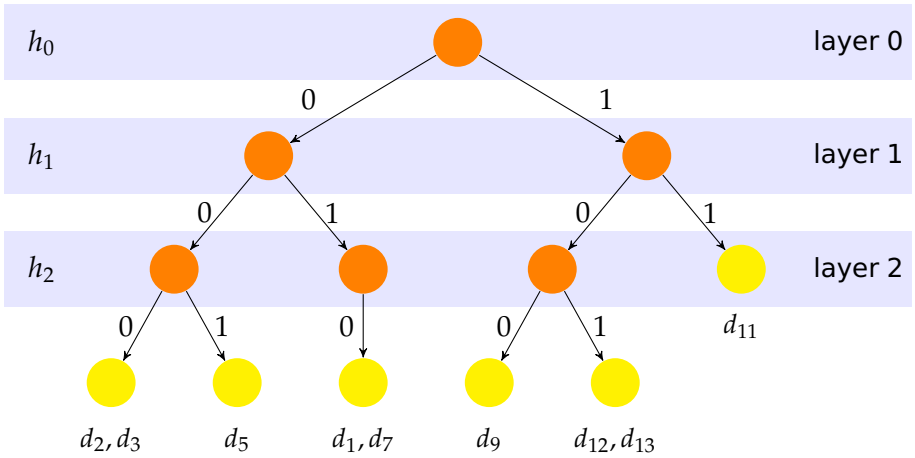


Figure 8: A single tree in LSH Forest. The orange nodes illustrate the inner nodes and yellow nodes visualize the leafs of the tree. The blue highlighted area shows which hash function is used for the respective tree layer. The hashed points are displayed below the leafs.

each hyperplane has to be computed for all k hyperplanes. The second part of the equation computes the time that is needed to find the nearest neighbor in each of the L buckets. The amount of points in a single bucket is represented by $\frac{N}{2^k}$. Note that for this proof, we assume that the points are equally distributed over all buckets. The average of points in a bucket is N divided by all possible regions generated by k hyperplanes. Since each hyperplane divides the space in two subspaces, the amount of overall buckets is given by 2^k. This has to be done for d dimensions and L bucket layers. Summing this up, the runtime of finding the approximated nearest neighbor of a point is $\mathcal{O}\left(Ldk + Ld\left(\frac{N}{2^k}\right)\right)$.

4.5 Locality-Sensitive Hashing Forest

Although LSH has a good runtime while the error rate remains low, it still lacks simplicity. In addition to assigning a point to a bucket, it demands the fingerprint to have a length equal to the bucket label, since each bucket is labeled by the hashing outcome of k consecutive hash functions. However, in some cases this seems to be an unnecessary computational effort. Imagine we have a set of 10 hash functions $\{h_1, h_2 ... h_{10}\} = \mathcal{H}$ and a dataset D consists of 5 points $\{d_1, d_2, d_3, d_4, d_5\}$. Now, in order to distribute the points to buckets, each point is hashed with each of the 10 hash functions (as explained in Section 4.3.1). Assume that for the first hash function the first 4 data points have the hash value of 1, while for the last data point it is 0 (more precise $h_1(d_1) = h_1(d_2) = h_1(d_3) = h_1(d_4) = 1$ and $h_1(d_5) = 0$). If we now query a point q, where $h_1(q) = 0$ it would be favorable to not compute the hash values of the other hash functions, since d_5 would be the only possible element to be considered as a candidate for the near neighbor set. In particular, it is preferable to save the computational effort of hashing the query and the data point multiple times (In this case 10 times). So

in this case, a variable fingerprint length to determine the bucket for a data point could improve the overall run-time of the hashing procedure. However, since in the original LSH algorithm the fingerprint corresponds to the bucket id, it has to be of a fixed length. Bawa et al. [**bawa2005lsh**] introduced a new data structure for LSH that allows fingerprints of variable length, called *Locality-sensitive Hashing Forest (LSH Forest/LSHF)*. Instead of storing data points in buckets, a set of k trees is grouped to a forest, where each data point is hashed into one leaf of every tree. In the query process, the query point is also hashed in the same way and the near neighbor set is constructed by selecting the data points of every tree, that share the longest prefix path.

One can imagine a single tree as a set of paths, where the maximum length of a path is d. Each path is constructed by a concatenation of nodes, beginning from the root node. The last node in a path is called *leaf* and stores a set of data points, while every node in between is called *inner node*. Taken all those paths together, a tree with at most d layers can be constructed, since d bounds the maximum length of a single path. Moreover, each layer has a unique hash function distributed, which in turn is used by all nodes of that layer to compute a hash value of a data point. Each edge between a node and its successor represents one hash value of a data point. The fingerprint is represented by the edges that are visited, while walking from the root node to the leaf it is stored to.

The illustration in Figure 8 represents a tree with a maximum depth of 3, where inner nodes are represented as orange and leafs as yellow. Note that all hash function in this tree hash points to either 0 or 1 and therefore, every node can have at most 2 direct successor nodes. Those trees are also known as *Binary Trees* [**skiena1998algorithm**]. The blue area highlights the hash functions and represents the mapping of one hash function to a layer. Every node within that area shares the same hash function, which in turn is used to compute the hash value of a data point (corresponding to that particular layer). As one can see, data point d_2 is hashed into the left most leaf, which means it was hashed by h_0,h_1, and h_2. If we recall the definition of a fingerprint, we know that the fingerprint for a data point d is represented by a hashing vector $(h_1(d),...,h_n(d))$. So in this case, the fingerprint of d_2 is represented by the hashing vector $(h_0(d_2),h_1(d_2),h_2(d_2))$. If we follow the path from the root node to that leaf we can get the fingerprint of d_2 by taking the label of each visited edge. As a result, we get the hash vector $(0,0,0)$, which is equal to $(h_0(d_2),h_1(d_2),h_2(d_2))$.

The illustration also shows, that not every node is on the bottom most layer. The node of point d_{11} for example is on layer 2. This has the reason, that no other element was hashed down the prefixed path of $(1,1)$ and therefore the computation of the hash value for layer 2 is not necessary to distinguish d_{11} from other points. This in turn means, that the fingerprint of $(1,1)$ is sufficient to identify d_{11}. For a better understanding, we first need to explain how data points are inserted into a tree: beginning from the root node, the insertion of a data point d is done by walking from node to node. If the current node is an inner node, $h_i(d)$ is

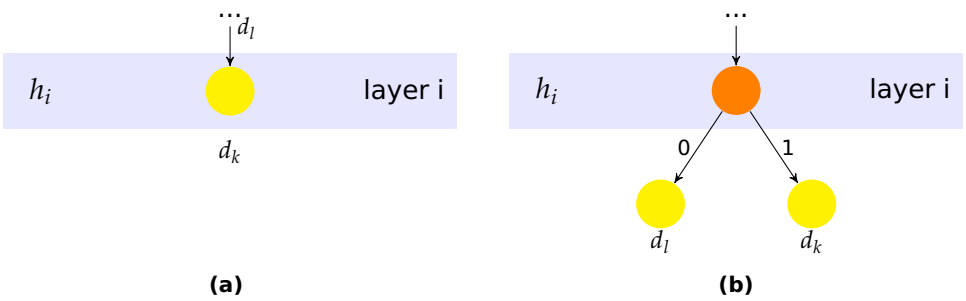


Figure 9: Illustrating the split and branching operation on a leaf. (a) Shows a leaf with a single point d_1 right before the insertion of d_2 and (b) shows the outcome of the branching operation, after d_2 is inserted.

computed and the point will continue to the next node by walking the edge labeled $h_i(d)$. In case that no such edge exists, we can assume that no other data point shares the same path with d. Consequently, a new leaf will be generated and connected to the node with an edge labeled $h_i(d)$.

If the current node is a leaf, we first need to validate if it is on the bottom most layer. Given such a leaf n and the data points stored in n as $n_{datapoints}$, we simply add d to $n_{datapoints}$. Note that in order to uniquely identify a data point by its fingerprint, it has to be the only data point in $n_{datapoints}$. However, if the bottom most layer is reached, we are unable to distinguish q from the data points in $n_{datapoints}$. Therefore, they are stored in the same leaf and share the same fingerprint. In case that the leaf is not on the bottom most layer, the leaf is transformed into an inner node, and all data points $d \cup n_{datapoints}$ are hashed in the same way as described above. More precisely, the insertion process for each point in the set $d \cup n_{datapoints}$ is repeated, beginning from the node that was transformed to an inner node. We will refer to this operation as *Branching*.

Figure 9 shows how the branching process is done in general. The left figure shows a leaf that stores data point d_k, before d_l is also hashed to the same leaf. As soon as d_l visits the same leaf, the leaf is transformed to an inner node and both items are distributed to a node one layer deeper. As one can see $h_i(d_l) = 0$ and $h_i(d_k) = 1$, which means, that no more split is necessary to distinguish those items.

By inserting data points into a tree and branching when necessary, the tree grows. It can happen, that some points are so similar that the depth of the tree would be very large. To avoid this behavior, each tree has a maximum length of d that defines the maximum depth of a node, where a split can be performed. For the case, that elements are still hashed into the same leaf and the maximum depth of the tree is reached, no more splitting will be performed and the elements will remain in the same leaf.

The query process itself differs from the bucket structure, since the tree structure allows fingerprints of any length. Consider a LSH Forest consisting of l trees and $\mathcal{T}$ is the set of

Input: query q, layer level x_i, current node s
if s *is leaf* **then**
  Return (x_i, s)
else
  h = Hash(q, x_i)
  t = NodeFromBranch(h)
  (p, z) = TopDown($q, x_i + 1, t$)
  Return (p, z)

Algorithm 1: TopDown(q, x_i, s)

Input: $\mathcal{M}$, consists of $(layer, node)$ tuples (x, s)
$x = max^l(x_l)$ over all $(x_l, s_l) \in \mathcal{M}$.
$\mathcal{S} = \emptyset$
while $(x > 0)$ *and* $(|S| < l)$ **do**
  foreach $(x_i, s_i) \in \mathcal{M}$ **do**
    if $x_i == x$ **then**
      if s_i *is leaf* **then**
        dp = getDataPointsOf(s_i)
        $S = S \cup dp$
      else
        dp = getDescendantDpOf(s_i)
        $S = S \cup dp$
  x = x - 1
Return $\mathcal{S}$

Algorithm 2: BottomUp($\mathcal{M}$)

Figure 10: Shows the two phases of the query process in LSH Forest: (a)The top-down phase, where all nodes from each tree with the longest prefix path to q are collected and (b) the bottom-up phase, where the closest data point to q of all previously collected nodes are returned. The algorithms are inspired by [**bawa2005lsh**]

tree that are used. A query for k-nearest neighbors of point q is performed by hashing q into each tree. Ideally, q will be hashed to a leaf and all data points that share the same leaf will be considered as near neighbors to q. However, it may happen that the hashing procedure would hash q to a leaf that does not exist and no near neighbor set can be retrieved. To overcome this issue, the query process is divided in two phases: The first *top-down* phase, where potential nodes with near neighbor candidates are collected and the second *bottom-up* phase, where only the k closest candidates out of those nodes are selected.

In the top-down phase as depicted in Figure 10, the query point q is inserted into each tree by descending them, while searching for the longest matching path to q. Expressed more simply, at query time t, q is hashed to a node of layer t of each tree $T_i \in \mathcal{T}$. If the current node in T_i is a leaf, the layer and the leaf will be returned. Afterwards, the tree will be disabled for all upcoming iterations of the first phase. If the current node is an inner node, the hash value of the current layer will be computed and compared to all outgoing edges of that node. If an edge is labeled with the hash value, the query point q will continue down that path and wait for the next iteration to repeat the process on the next layer. If no edge exists, the current node and layer will be returned and the tree disabled so that it wont be considered in the next iteration. This process will be repeated until all trees are processed. As a result, a set $\mathcal{M}_{candidates}$ is generated, consisting of tuples (x, s) where x is the layer and s the node that was returned. Note that $|\mathcal{M}_{candidates}| = |\mathcal{T}|$, since every tree has exactly one prefix path that matches to the query point. All nodes in $\mathcal{M}_{candidates}$ are considered to have potentially near

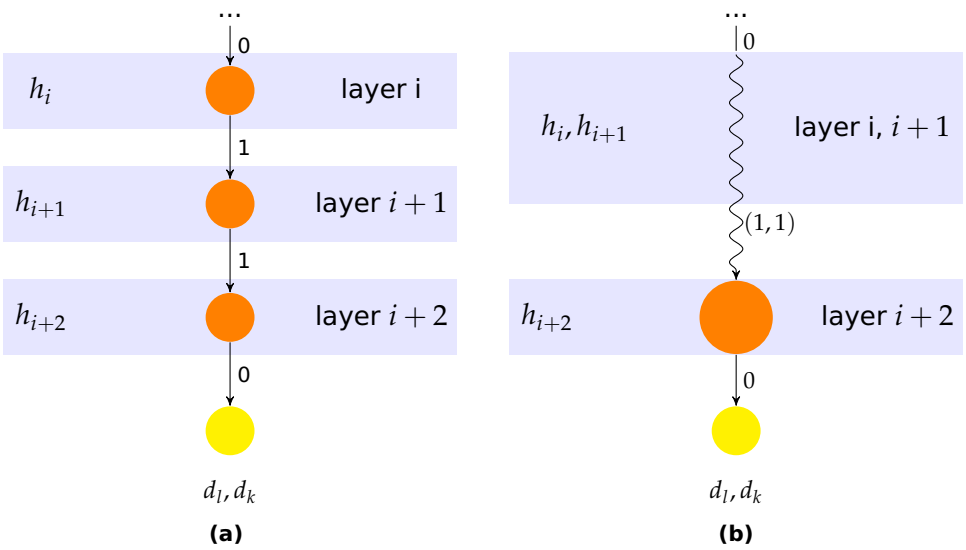


Figure 11: Illustration compressed nodes. (a) Shows a chained sequence of nodes, that have only one outgoing edge, while (b) represents a compressed version of this sequence as a single node

neighbors and consequently form the set of nodes with near neighbor candidates. However, not all of them have the same probability to be similar to q. In particular, the deeper the layer of a corresponding node is, the more likely it contains a neighbor close to q. In addition, the number of candidates can exceed the amount of near neighbors we are looking for and therefore only the closest candidates should be selected.

In the second *bottom-up* phase, as depicted on the right side of Figure 10, we solve this problem by collecting k points from the deepest nodes of $\mathcal{M}_{candidates}$. Those points will be collected to $\mathcal{M}_{nn}$. The bottom-up phase starts from layer x, where $x = max^l(x_l)$ over all $(x_l, s_l) \in \mathcal{M}_{candidates}$. While moving up from layer x to the root of each tree, we merge $\mathcal{M}_{nn}$ with all points from nodes that are in the set $\mathcal{M}_{candidates}$. If the node in $\mathcal{M}_{candidates}$ was not a leaf but an inner node, we will collect all data points of every leaf, that is descendant to this node. After each iteration, we check if $M >= k$ and if so, the process stops and M is returned as near neighbor set. Otherwise, the algorithm moves up to the next layer and is repeated. Note, that although a data point can be considered as a near neighbor candidate in more than one tree, it will not be collected multiple times in $\mathcal{M}_{nn}$, since $\mathcal{M}_{nn}$ is a set. After both phases are finished, the points in $\mathcal{M}_{nn}$ will be sorted by their similarity to q and the k most similar points will be returned.

If comparing LSH Forest to the best known LSH algorithm, no quantitative improvement will be noticed. However, being based on practical heuristics, it is provably better than most of them [**andoni2017lsh**]. The tree structure as described above is not efficient in its space allocation, since the amount of nodes to hash data points could become very large. Although each tree is restricted by its maximum depth, two very similar points are very likely to be

hashed to the bottom most layer of the tree. Assume we hash two very close points and want to hash them into a tree with the maximum depth of k. As both items are close, it is very likely that hash functions compute the same hash value for both of them. If we assume that this is indeed the case, both items will be branched until the bottom most layer is reached and both items are stored in the same leaf. Consequently, $k+1$ nodes are needed to store those points. However, each node in the path will have exactly one outgoing edge, which seems quite redundant to store just 2 points. In order to save the space of such unnecessary node chaining, sequences of nodes with only 1 outgoing edge can be compressed into a single node without affecting either the insertion nor the query process of the LSH trees. An example of compressed nodes is given in Figure 11. The left illustration shows a chained sequence of nodes, where each node has only 1 outgoing edge. The right illustration shows a compressed version of this sequence. In particular, the first two nodes are merged into the third node, while their edges are merged into a single edge. The new edge is then labeled with both labels of the merged edges. While in the query process, it is still necessary to compute the whole path for a query point, the amount of nodes to store the data points will be reduced. In fact, the amount of internal nodes is one less than the overall number of leafs and ensures, that the storage complexity is linear to the number of overall data points [**bawa2005lsh**]. Trees that support such kind of compression are also known as *PATRICIA* trees [**morrison1968patricia**].

A forest as described above represents a more simple data structure compared to the original LSH. Especially when combining LSH Forest with Random Hyperplane Hashing, the overall quality of the near neighbor set is good. However, one shortcoming of both LSH variants is that none of them take any data dependent information into consideration. For example, if we have a dataset with dense clusters it would be favorable to keep data points of one cluster close to each other and avoid "splitting" them down into different pieces. The following section will highlight some of the shortcomings of LSH Forest combined with Random Hyperplane hashing and present solutions to overcome them.

5 Challenges and Improvements for LSH

When using LSH one profits from its data structure that enables a time efficient query for the near neighbor search, even for a large amount of data. Combined with Random Hyperplane Hashing, LSH can achieve a good run-time. By dividing the data space in sub-spaces, the query process is only performed in smaller portions of it. A special property of the algorithm is that it allows to control how close the approximation is, by trading time and space efficiency for a better approximation. Nevertheless, this property is also one of its most challenging parts, since it depends on many factors like the LSH parameter settings or the dataset itself.
.

First of all we will talk about how the approximation can be improved. If we recall the structure of LSH Forest as it was presented in Chapter 4.5 we can notice that the correctness of the near neighbor search correlates to (a) the amounts of trees in a forest and (b) the depth of each tree. There is also a noticeable similarity to the bucket structure: Increasing the amount of hash functions also increases the probability that two points in the same bucket are similar to each other. This statement is equal to (b), since the depth of the tree represents the (maximum) amount of hash functions that hash points to leafs. Literally speaking, the deeper a leaf is in the tree, the more likely it is that its stored points are similar to each other. This has the reason that deeper trees offer a higher granulation of data, since each layer represents a possibility to split data points in different branches. Despite the fact that two very close points have a high probability to be hashed into the same leaf, it may happen that they are split into different branches very early. This has an impact to the approximation. In fact, it is enough if one hash function on a layer close to the root node, computes different hash values for both points.

To overcome this issue, we can increase the amount of trees, since each tree works with a separate and independent set of hash functions and therefore the hashing outcome can differ for each of those sets. More precise, all trees are independent from each other and will distribute the points differently. In conclusion, each pair of points has multiple chances to be hashed into the same leaf or at least, be split into different branches quite late. If we recall the bucket structure we will see (a) is very similar to adding different layer of buckets. Actually, both serve the same purpose, which is to lower the impact of unfortunately hashed

points. Since both, LSH and LSH Forest, are very similar in handling this kind of problems, we could also improve the approximation by using either Or- as well as the And-Construction as described in Section 4.2.

As already mentioned, improving the approximation comes also with the downside of worsening either query complexity, space efficiency or the correctness of the near neighbor set. In fact, both also correlate to (a) and (b). Increasing the depth of each single tree also increases the amount of space that is needed to store all nodes. Even with PATRICIA trees, each additional layer is likely to add new nodes. Furthermore, each tree works completely independent from all other trees and thus has to be stored separately. The space complexity is given by the approximate amount of nodes that each tree needs to store the whole dataset in, multiplied by the amount of trees. Considering time complexity, the query time increases with each additional layer in a tree, since on each layer, the query point has to be hashed. Therefore, finding a good balance between correctness and time/space efficiency is a necessary, but also challenging task.

Another impacting factor is the LSH Family, which provides the set of locality-sensitive hash functions. As per definition, these hash functions are generated independent from the data and decide if points are close to each other or not. While these functions are typically generated randomly, they still offer ways to improve the correctness of the approximation. By carefully selecting the LSH Family, we can change the distance between close points and influence the result of the near neighbor search. In contrast to (a) and (b), the amount of trees and their depth can be left untouched, while the improvement is done within the hashing procedure.

One reason that locality-sensitive hashes are preferable when working with LSH is that their generation can be done very easy and independent of information provided by the dataset. Furthermore, they work very efficient, since the items are hashed only based on their distance, which is a fast computational operation. As a drawback, data-dependent information that could improve the overall correctness of the near neighbor search are ignored and not considered during this hashing procedure. Although it is possible to add limited amount of data-dependent information to the locality-sensitive hashing procedure additional computational effort is needed. In fact, the hashing procedure in LSH is a common operation and should remain simple. By adding more factors to the hashing process, hash functions can be overburdened with too much computational effort and lead to a higher time complexity of each single hashing operation. This in fact will worsen the query procedure, since the query point has to be hashed in the same way.

In some cases, the benefits from taking data-dependent information into account will overweight the drawback of worsening the time and space complexity. Imagine a set that contains a dense cluster of points. A dense cluster is an indicator that all points within it have a high probability to be logically related to each other. In this case, it would be preferable that a

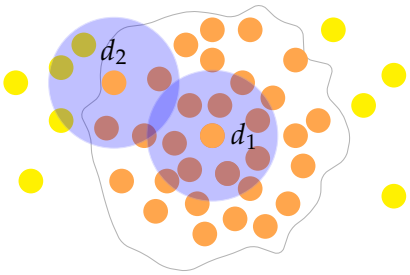


Figure 12: Shows the locality-sensitive area of two points of a cluster. The orange nodes represent points of the cluster, while the yellow nodes represent other points of the dataset. The two blue areas show the locality-sensitive area of datapoint d_1 and d_2.

locality sensitive hash function does not separate them by computing a different hash value, since this would result in the tree distributing them into different leafs. Despite the fact that these points are logically connected to each other, they may not be located close to each other. Especially points on the outer border of the cluster are unlikely to be hashed to the same leaf as some points in the center.

Figure 12 illustrates a cluster that is part of a dataset. The orange nodes represent data points that are part of the cluster, while the yellow are just some other points of the dataset. The two blue areas represent the locality-sensitive area of point d_1 and d_2. On first glance we can see, that none of those areas cover the whole cluster. To fix this problem, one could take the maximum distance between two cluster points as the radius of the locality-sensitive area. While this would help to define all cluster points as potential near neighbors, the hash functions may become too imprecise for the overall performance of LSH. A completely different approach is to generate the hash functions dependent on the dataset. Instead of randomly generating locality-sensitive hash functions, a function will be generated based on the data. In fact these functions are already known as *Data-dependent hashing functions*, which contradict the idea of locality-sensitive hash functions. These hash functions are defined uniquely only for the given dataset and include some feature information of the data. The objective is to distribute the points based on features of the data. There are different approaches that address this issues, like Spectral Hashing [**weiss2009spectral**, **abdullah2014spectral**], K-mean based hashing [**pauleve2010locality**] and Random Maximum Margin Hashing [**joly2011random**]. While data-dependent functions show good results, they are not applicable for every use case. In addition, some of them demand of more computational effort, when comparing them to locality sensitive hashing.

Nevertheless, there are also problems that are hardly solved by either of locality-sensitive or data-dependent hash functions. If we recall the cluster in Figure 12 we can see, that some points are covered by both locality-sensitive areas. For this case, hash functions have to decide which locality-sensitive area a point should be distributed to. In fact, these areas represent a leaf in a tree. As one already knows, points can only be distributed in one leaf.

So in this case, hash functions will always hash the point into a wrong leaf, since either d_2 or d_1 will miss it in its nearest neighbor set. Thus it would be preferable to avoid hashing a data point in one leaf exclusively.

It is shown, that each hash variant offers many possibilities to improve the approximation of LSH. However, both of them have their downsides, since each approach introduces new problems that have impact to the overall performance of the algorithm. Although it seems not to be possible, it is obvious that mixing the benefits of both variants without adapting their worse parts, will be the best solution to the stated problem.

This thesis will propose three different Random Hyperplane Hashing extensions for LSH Forest to solve the aforementioned issues. All of them are designed so that a small portion of the space complexity and time efficiency is traded for an improved correctness of the near neighbor set. This will be done by introducing a "fuzzy" area for each hyperplane that is not assigned to any subspace exclusively, but represents an overlapping area between both subspaces and define the space for points that are too close to the hyperplane. Therefore every point within this area will be distributed in both subspaces. This has the same effect as ignoring the hashing outcome for a particular hyperplane completely as the point will be distributed in multiple child nodes. We will refer to points that are too close to a hyperplane as *indecisive points*. This comes with some computational effort, but it should only be noticeable in the initialization phase of each tree, since the query procedure remains unchanged. We hope, that this approach can improve the overall correctness of the near neighbor search, while the query time can compete with traditional LSH algorithms. As an unavoidable downside, the space complexity will increase. This demands further analysis and will be covered by Chapter 7.

As for the first solution we will classify points by considering their closeness to a hyperplane as described above. While small modifications will try to improve the aforementioned algorithm, it will mainly remain the same. We will refer to this variant as *Fuzzy Random Hyperplane Hashing (f-RHH)* [**cochez2017large**]. The second solution will also include a fuzzy factor, but instead of branching points if necessary, a fixed percentage of points on each layer will be hashed into multiple nodes on the lower layer. In order to select a specific percent of the points, we will compute the closeness of a point to the hyperplane and take only the demanded percentage of closest points. The goal is to simulate an appropriate behavior when data points are very dense. Despite the fact that this can slightly increase the amount of wrongly hashed points, it is assumed that the overall correctness of the near neighbor set will be increased. This algorithm will be refereed to as *Percentage-based Random Hyperplane Hashing (p-RHH)*. The third and last algorithm is *Indecisive Random Hyperplane Hashing (i-RHH)* and will be a hybrid of both variants. Instead of hashing points as indecisive, the hashing outcome of all points for a particular hyperplane will be ignored, if a certain percentage of points is within the given indecisive area.

While all three extensions define indecisive points different from each other, they all address the problem of a point being hashed into the wrong space and thus being separated from potentially closer points. It remains to prove if the benefit of increasing the correctness of the near neighbor search can outweigh the worsening of the time and space efficiency. The following section will discuss and highlight the key aspects of the different extensions. However, all of them are based on the idea of Fuzzy Random Hyperplane hashing and therefore the discussion of this variant will also give a more excessive explanation of fuzzy and indecisive points.

5.1 Fuzzy Random Hyperplane Hashing

When using RHH combined with LSH, points are projected as vectors in a n-dimensional space. Each side of the hyperplane cuts the space into halves and when comparing a vector to it, the vector can be on either of the sides. As a result, every point that is hashed with a hyperplane will have either 1 or 0 as hashing outcome. However, in some cases it is hard to make any decision about the projected vector, since this vector can be very close to the hyperplane. More precise, hashing a close points exclusively into one half space could cause that near neighbors are split into different branches of the tree. To ease this problem, Cochez et al. introduced *Fuzzy Random Hyperplane Hashing (f-RHH)* [**cochez2017large**]. The idea is, instead of hashing points exclusively to one side, close points can be hashed to both sides. As a consequence, points can have multiple hash values for close hyperplanes, which has the same effect as ignoring the hashing outcome of those planes. In order to extend normal hyperplane hashing to support indecisive hashed points, an area around the hyperplane has to be defined. This area describes the fuzzy or indecisive area for a hyperplane and every node within it will be hashed as such. Intuitively, one could take the angular distance between the hyperplane and the vector. While this is indeed a good measurement for the closeness distance, it comes with additional computational effort, which will slow down the hashing procedure. However, as described in Section 4.1.1 it can be observed that the angle between the hyperplane and the vector is defined as $\frac{\pi}{2} - \alpha$, where $\alpha = \arccos\left(\frac{v \cdot n_h}{\|v\| \cdot \|n_h\|}\right)$ describes the angle between the vector and the normal of the hyperplane. Therefore, computing this distance, one can obtain the distance of a point to the hyperplane. In fact, this computation will provide us the information for both, the hash value as computed in RHH and the indecisiveness of a point. Assume we have a given parameter k describing the angle of the indecisive area, then for any given vector v and normal of a hyperplane n_h, a vector within this angle will be have both hash values computed by the transformation of the following expression [**cochez2017large**]:

$$\frac{\pi}{2} - \alpha = \frac{\pi}{2} - \arccos\left(\frac{v \cdot n_h}{\|v\| \cdot \|n_h\|}\right) < k$$

Which in turn can be transformed to an equivalent expression:

$$\arcsin\left(\frac{v \cdot n_h}{\|v\| \cdot \|n_h\|}\right) < k$$

Since n_h is the same for all vectors v, we can conclude that $\|n_h\|$ is a (positive) constant. This allows us to replace the expression $\|n_h\|$ by a constant value R. Moreover, if we normalize vector v beforehand, the angle will remain identical. Therefore, we can replace v by its normalized vector $\bar{v}$ where $\|v\| = 1$. Taking all this together, the previous transformation can be written as:

$$\arcsin\left(\frac{\bar{v} \cdot n_h}{1 \cdot R}\right) < k$$

Which will finally be transformed to the equation:

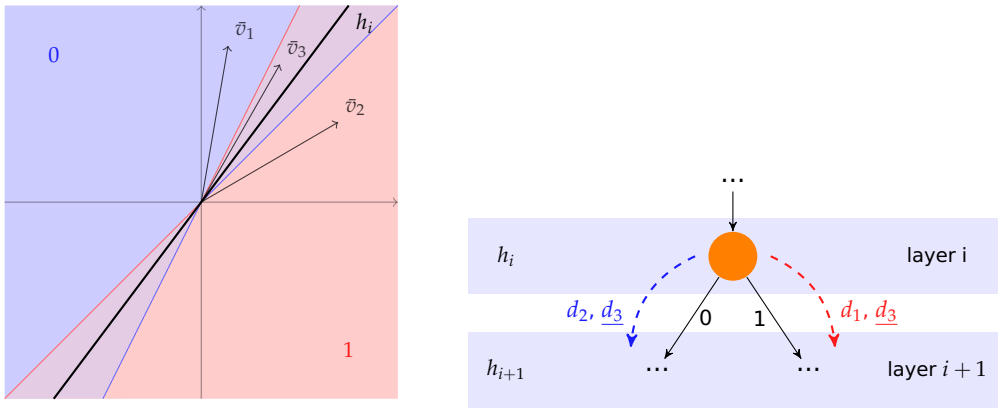
$$|\bar{v} \cdot n_h| < \sin(k) \cdot R = C$$

By replacing the $\sin(k) \cdot R$ with C we can say, that if the angle between the vector v and the hyperplane is smaller than k, it is equivalent to the dot product of the normalized vector v and the hyperplanes normal vector n_h being smaller than the constant parameter C.

If we have a closer look at the expression $|\bar{v} \cdot n_h|$ one may notice, that this is very similar to the way that the hash value for a vector is computed. More precisely, the only difference is that we compute the distance of the normalized vector $\bar{v}$ instead of just v. By providing the vector v already in its normalized form, we can compute the hash value and its indecisiveness in the same time. Thus no additional computational effort is needed in order to determine if a point is too close to a hyperplane and therefore $|\bar{v} \cdot n| < C$ can be implemented very efficiently.

A visualization of the indecisive area of a hyperplane for a 2-dimensional space is given by Figure 13(a). A hyperplane h_i is dividing the space, where the blue area represents the hash value 0 while the red area represents 1. The shaded area around the hyperplane represents the indecisive area defined as the angle to the hyperplane. As one can notice, instead of dividing the space in two halves exclusively, the fuzzy area represents the overlapping part of both subspaces. Furthermore, three points d_1,d_2, and d_3 are given by their normalized projection vector $\bar{v}_1$, $\bar{v}_2$, and $\bar{v}_3$. While $\bar{v}_1$ is clearly on the blue half and $\bar{v}_2$ on the red, $\bar{v}_3$ is within the fuzzy area. Therefore, data point d_3 will have both hash values computed. As a consequence, d_2 will be hashed down the branch 0 and d_1 will follow the path 1, while d_3 will be split into both branches, as depicted in Figure 13(b).

By allowing a point to have multiple hash outcomes for a (RHH) hash function and proceed the insertion as illustrated in Figure 13(b), we essentially ignore the hashing outcome of it.



(a) **(b)**

Figure 13: Illustrating a fuzzy hashing for the data points d_1, d_2, d_3 and their corresponding vector projections v_1, v_2, v_3. (a) Shows a hyperplane that divides the space in two halves and with an indecisive area around it, while v_3 being within it (inspired by [**cochez2017large**]). (b) Shows the outcome of that hyperplane in a tree, so that point d_3 is hashed down into both branches.

Rather than hashing the point into one successor node exclusively, we skipped the hyperplane only for this particular point, while all other points that are not too close will be hashed as in normal RHH. By doing so, even if d_1 and d_2 will be hashed in different branches, d_3 can be hashed in both and thus can be a near neighbor of both points.

The essential idea of f-RHH is to lower the amount of points that are separated from their near neighbors by restricting hash points to either 0 and 1. While this hashing approach remains locality-sensitive and thus, its computation can be relatively fast (compared to data dependent hashing), it still "simulates" a certain behavior of data dependent hashing. To be more clear, by adding a new feature to a data point, which is its closeness to the hyperplane, we basically have more information about the data itself without increasing the computational effort to gain it. In addition, these changes do not have any effect on the query time, since the query process remains exactly the same. By doing so, the correctness of the near neighbor search can be increased. However, what still remains to be answered is how to set the parameter C in order to define the fuzzy area as depicted in 13(a). Generally, this parameter heavily depends on the use case and needs to be defined only for a certain dataset. While this goes in the direction of analyzing the dataset and thus including some data-dependency, the parameter setting can be achieved just by smaller observations of the information gained from locality-sensitive hashing procedure. In order to find a good setting for the indecisive angle, we will run some tests for different datasets. This will be covered in Chapter 7.

5.2 Percentage-based Random Hyperplane Hashing

A general problem in LSH is its inability to react to datasets that are shaped in a certain way. Especially when working with datasets that contain clusters, there is nothing that prevents normal RHH from accidentally dividing it in different subtrees, and therefore worsening the overall correctness of the near neighbor set. This has the reason, that the hyperplanes are generated randomly without taking any information about the actual shape of the data into account.

Using f-RHH instead of RHH can slightly improve the correctness, although f-RHH addresses a completely different problem. This is due to the way it treats points that are close to the hyperplane. If a hyperplane cuts a dense cluster of points, all the points within the indecisive area will be assigned to both subspaces. We can distinguish between two cases: (a) all points of the cluster are within the indecisive area or (b) only a subset of the cluster is within the indecisive area. If (a) is the case, then all points will result in the same subspace and thus also in the same subtree or leaf. if (b) is the case, then only the points within that area will be hashed in the same subspace, while all other will be divided in either of the subspaces.

One way to deal with this issue is to set the angle for the indecisive area to the maximum distance between two points within that cluster. While this setup will solve the problem for a specific cluster, it will not work in general for other clusters within the same dataset, since their shapes (and thus their maximum distance) can differ from each other. Additionally, in order to obtain the information of the maximum distance between two points within a cluster, some additional computational effort is needed and will worsen the overall runtime of the algorithm. Even if it is possible to get the information without any computational effort, the angle for the indecisive area will be the same for every hyperplane, and can therefore not be defined explicitly for a specific cluster.

Alternatively, the angle for the indecisive area can be kept variably, and therefore allow the angle to be different for every hyperplane. *Percentage-based Hyperplane Hashing (p-RHH)* extends the idea of f-RHH, so that the angle of the indecisive area is not fixed, but adaptive to the points that are hashed with it. Instead of providing the angle as a constant parameter for all hyperplanes, each hyperplane will define the angle to always cover a fixed fraction of the points hashed in the given node. This parameter is represented by a percentage threshold and is defined for all hyperplanes. Although this parameter is also constant, the angle of each hyperplane will be variable, since the angle depends on the points that are hashed with it. In order to select a percentage of points, all points that will be hashed with the hyperplane will be ordered by their distance to it and only the closest points, equal to the amount of the threshold, will be hashed as indecisive. Assume we have a hyperplane h_1 that hashes 200 points in total. If the percentage parameter is set to 0.1, 20 of the 200 points will be hashed as indecisive. This is equal to setting the indecisive angle so that 20 points out of the 200

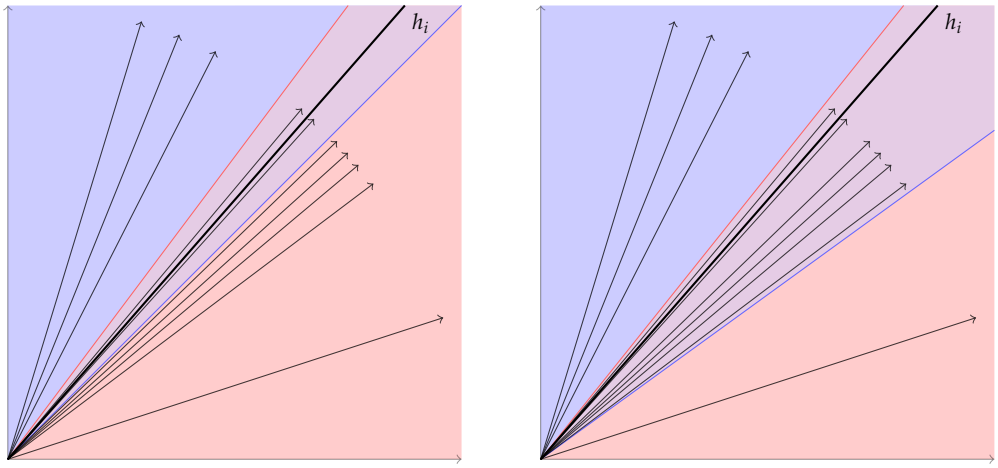


Figure 14: Shows the difference between f-RHH (a) and p-RHH (b) for a dense cluster in a dataset. The indecisive area in p-RHH is equally distributed to both sides, while for p-RHH it is uneven. In addition, p-RHH covers the whole cluster, which contains 5 very close vectors.

points are covered. If, for example, a second hyperplane h_2 hashes 100 points in total, only 10 points will be hashed as indecisive, and therefore the angle for the indecisive area will be different as the angle of h_1.

Another difference between f-RHH and p-RHH is the location of the indecisive area. In f-RHH the angle is set so that each side of the hyperplane have the same area covered as indecisive, while in p-RHH the these angles can be unequal. The reason is that we do not necessarily know where exactly the hyperplane cuts the cluster, and therefore it is necessary to "move" the area so that the whole cluster is covered.

Figure 14 shows the difference between both RHH variants, where (a) represents f-RHH with a constant angle and (b) p-RHH with an angle so that 50% of the points are covered. Just by looking at (b) it is noticeable that the indecisive area for the lower red area is larger than the blue one. As a result, the whole cluster of points is within the indecisive area and will be hashed into the same subspaces, while for (a) only a small portion of the cluster will do so. Therefore, we can assume that for a query point that is part of the cluster, the near neighbor set for p-RHH will be presumably better than for f-RHH, since all the points of the cluster will have a higher probability to result in the same subtree.

While for f-RHH the space complexity can not be computed in general, since the amount of points that will be hashed as indecisive is not limited by any parameter, the percentage threshold of p-RHH enables the ability to compute the worst case space complexity of p-RHH, as the amount of indecisive hashed points per layer is limited. Assuming that the points of the dataset are equally distributed into the next layer, the total amount of points given by the initial dataset and the indecisive hashed points is maximized. Given this, we can compute

the maximum depth of a binary tree where each data point is stored into its own leaf as follows:

Theorem 2 (Depth of p-RHH binary tree with maximized data points)**.** *Given a data set $\mathcal{T}$ of n points, an percentage threshold as $0 \leq p < 1$ and l being the current depth initialized with 0, we can compute the depth of a binary tree, while the amount of hashed items by p-RHH is maximized by*

$$k_{prhh}(n, p, l) = \begin{cases} n, & n \leq 1 \\ k_{prhh}\left((\frac{n}{2} + n \cdot p), p, l+1\right) & n > 1 \end{cases}$$

Given this, we can compute the number of nodes in a full binary tree as follows:

Theorem 3 (Maximum number of nodes p-RHH)**.** *Given a data set $\mathcal{T}$ of n points, a percentage threshold as $p < 1$ and the maximum tree depth as m we can compute the maximum number of nodes in p-RHH by*

$$nodes_{prhh} = 2^{k+1} - 1,$$

where k is given by:

$$\min\left(k_{prhh}(n, p, 0), m\right)$$

In definition 11 the maximum layer is computed recursively, where $\frac{n}{2} + n \cdot p$ defines the number of points in each node of the current layer l. Since p is limited to $[0, 1)$, the amount of points per node will decrease with each layer. If the amount of points for a particular node is 1, the current layer will be returned as tree depth, otherwise the recursion will continue. Note that this is not the maximum depth that a binary tree can have while hashing points with p-RHH, but represents the maximum depth of a binary tree, when the amount of hashed points is maximized. In definition 12 this depth is then taken to compute the number of nodes in a full binary tree, that is restricted by the maximum tree depth in an LSH Forest.

Finding a good percentage parameter is an essential part of this extension, as it defines the amount of points that will be hashed as indecisive. In addition, the amount of points that will be hashed with a certain hyperplane has to be known, so that the algorithm can work properly. Therefore, the initialisation of the trees in LSH Forest has to be changed a bit. Instead of inserting the data points sequentially into each node, they will be inserted simultaneously. By doing so, the amount of data points that will be hashed with a certain hyperplane will be known in advance. Moreover, we can firstly compute the distances of all points to a hyperplane and therefore define the angle of the indecisive area. Afterwards, the hashing procedure will be similar to f-RHH.

5.3 Indecisive Random Hyperplane Hashing

As an alternative approach to improve the NNS correctness, we will introduce *Indecisive Random Hyperplane Hashing (i-RHH)*. This RHH variant represents a hybrid of f-RHH and p-RHH. Its essential idea is instead of hashing points as indecisive, it will ignore the hashing outcome of a node if a certain fraction of points is too close to it. Similar to p-RHH, we first compute the distance to the hyperplane of the node for every point that will be hashed with it. As this process needs to pre-compute the distances, the initialization phase will be exactly like it is described for p-RHH. Afterwards, two parameter are necessary to decide how the points will be hashed:

1. A constant parameter C that defines the angle for the indecisive area.
2. A constant parameter p that describes the fraction of points, for which the outcome of a node will be ignored.

Like in f-RHH, the parameter C defines the distance for which a point will be considered as indecisive. The second parameter p describes a percentage threshold and will be used to decide if the hyperplane of a node will be set as indecisive, and therefore ignored. If the fraction of points that are too close to the hyperplane is lower than the given threshold p, the algorithm will behave similar to RHH. If, however, this fraction exceed p, all points that are hashed with this node will be hashed as indecisive. Hashing all points as indecisive will have essentially the same effect as ignoring the outcome of the hyperplane completely. As an effect we avoid to make any decision about the points, if a certain amount of points is to close to it. Different from p-RHH and i-RHH, the points will not be hashed into both trees. In order to avoid hashing points multiple times, the points will be hashed into one leaf on the lower layer, while all outgoing edges of the indecisive node (representing the indecisive hyperplane) will be redirected to the new node.

The query procedure will remain mostly the same. However, computing the hashing outcome for a query of an indecisive hyperplane is an unnecessary computational effort, since all outgoing edges are directed to same node. Therefore, the hash value will not be computed for this particular plane, but the query point will just take any edge to the next node. This way we can ensure, that no additional computational effort has to be made if planes are set as indecisive.

However, ignoring the hashing outcome of a hyperplane means that the amount of hyperplanes to hash points into the forest will be reduced by 1. Although this will increase the fingerprint length, the query time will remain the same as for RHH, since no computational effort is necessary to compute the hash value for an indecisive plane. Therefore, the amount of hashing computation will remain exactly the same, while the fingerprints can be different.

Percentage-based as well as indecisive Random Hyperplane Hashing addressing the same issue, while both behave slightly different. In case of i-RHH, we will try to improve the hashing

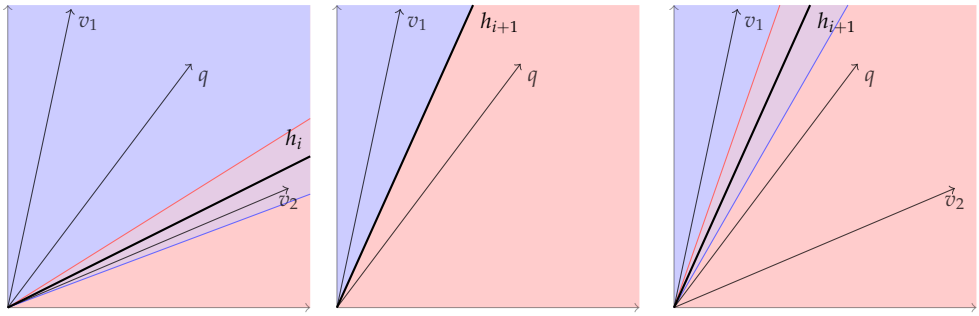
procedure by ignoring "badly" generated hyperplanes. As a consequent, points that can not be assigned clearly to one subspace will still result in the same leaf. Additionally, the percentage threshold for p-RHH have to be selected carefully, since it has to cover all points of the cluster in order to keep them together. If the threshold is chosen so that not all points are within the indecisive area, outer points of the cluster will be separated. The same case is also applicable to i-RHH, with the difference that not only the threshold, but also the angle for the indecisive area has to be defined so that clusters can be detected. A closer comparison of both algorithm will be covered in Section 7.

5.4 Error rates

There are very few cases, were RHH can perform better as the fuzzy variants. While this is expected to happen very rarely, it depends on the query point and its angular distance to its closest points as the following example will show:

Assume that for RHH, the query vector q and two vectors v_1 and v_2 are given, where v_1 is the closest neighbor to q. On layer n, the training vectors are separated, so that v_1 is hashed to 0 and v_2 to 1. On the next layer $n+1$, v_1 will again be hashed to 0 and v_2 to 1, so that for the fingerprints for the vectors are given by $h_{n,m}(v_1) = (0,0)$ and $h_{n,m}(v_2) = (1,1)$. Note that on layer m both points are already separated into different branches. Now assume that for the same layers the fingerprint of q is given by $h_{n,m}(q) = (0,1)$. On layer n, q would be hashed into the same branch as v_1, the hashing procedure would be stopped on layer m and v_1 as a near neighbor candidate returned. For f-RHH, the same scenario is given, but this time n hashed v_2 as indecisive so that the fingerprint of v_2 is given by $h_{nm}(v_1) = [(0,1),(1,1)]$. If the fingerprint of q is again given by $h_{n,m}(v_1) = (0,1)$, instead of stopping at layer m, the hashing procedure would continue since q and v_2 have the same fingerprint. As a result, v_1 would not be in the set of near neighbor candidates. Note that finding v_1 as nearest neighbor was only possible in p-RHH since it was hashed as indecisive and thus resulted in the same subspace as q.

Figure 15 illustrates this scenario for a 2-dimensional space. The left image 15 (a) shows the the random hyperplane hashing on layer n for both variants. Figure 15 (b) shows the hashing procedure on layer m for RHH. Since v_1 is the only vector and the hyperplane would separate q and v_1 in different subspaces, the hashing procedure would stop and return v_1 as a near neighbor candidate. Figure 15 (c) shows the the hashing procedure on the same layer for RHH. Again, the hyperplane separates v_1 from q, but this time the hashing would continue to the layer below, since q is assigned to the same subspace, as vector v_2. Note that v_2 is only considered in this hashing procedure, because it was within the fuzzy area as depicted in (a).



(a) **(b)** **(c)**

Figure 15: Illustrates the case where RHH performs better than f-RHH. (a) Shows the hashing procedure for both variants, while (b) shows the hashing procedure for RHH on layer $i+1$ and (c) for the one for f-RHH. In (b) the hashing procedure for RHH would stop, since h_{i+1} would separate v_1 and q. while in (c) it would continue, since q would result in the same subspace as v_2.

While this case can indeed happen, the probability that f-RHH will separated q from its nearest neighbor is smaller then keeping them together. In fact, we can compute this probability as follows:

Assume that the probability of a query vector q and its closest neighbor v with angle β between them being separated by RHH is given by $Pr[h(q) \neq h(v)]_{rhh}$ and the probability for the same case in f-RHH for an indecisive angle of α is given by $Pr[h(q) \neq h(v)]_{frhh}$. We can show that the probability of $Pr[h(q) \neq h(v)]_{rhh} \geq Pr[h(q) \neq h(v)]_{frhh}$ by contradiction:

- The probability of $Pr[h(q) \neq h(v)]_{rhh}$ is given by $\max\left(\frac{\beta}{\pi}, 0\right)$.
- The probability of $Pr[h(q) \neq h(v)]_{frhh}$ is given by $\max\left(\frac{\beta-\alpha}{\pi}, 0\right)$.
- If we assume that $Pr[h(q) \neq h(v)]_{rhh} < Pr[h(q) \neq h(v)]_{frhh}$, then $\max\left(\frac{\beta}{\pi}, 0\right) < \max\left(\frac{\beta-\alpha}{\pi}, 0\right)$
- This is only true if $\alpha > \beta$, but in this case f-RHH would have hashed v as indecisive and thus, it would result in the same subspace as q ↯.
- Therefore for any given $\alpha > 0$, the probability that RHH separates q from its closest neighbor v is always $Pr[h(q) \neq h(v)]_{rhh} > Pr[h(q) \neq h(v)]_{frhh}$ □.

What this proof essentially shows is that the probability of making a "bad" hashing so that q is separated from v is always smaller than the probability that a fuzzy random hyperplane will hash q and v to the same subspaces. This probability correlates to (a) the distance of q and v and (b) the indecisive angle α. Knowing this, we can compute the probability of q being separated from its closest neighbor as follows:

Theorem 4 (error rate f-RHH)**.** *Given a query vector q, the dataset W, and its closest vector v*

so that $\min\{d_{cos}(q,w_i)|w_i \in W\} = d_{cos}(q,v)$*, the probability that* $\min\{d_{cos}(q,w_i)\} \neq d_{cos}(q,v)$ *for f-RHH with the indecisive angle of* α *can be computed by*

$$Err_{frhh}(q,v) = \max\left(\frac{d_{cos}(q,v) - \alpha}{\pi}, 0\right).$$

Comparing the correctness of p-RHH and RHH, we can make a similar assumption. If we assume that the percentage range is set so that at least 1 point will be hashed as indecisive it will always be the closest point to the query. In fact, p-RHH will always perform better as RHH, if the percentage of points is set so that the amount of fuzzy hashed points for all layers matches to the requested amount of near neighbors to q. This has the reason, that if a hyperplane will be between the query point and its nearest neighbors, the next closest point will be hashed as indecisive. Since there is no other closer point to q as its nearest neighbor, it will be hashed as indecisive and thus always distribute to the same subspace as the query point. If now on each layer, the amount of indecisive hashed points matches the requested amount of k nearest neighbors, the k closest neighbors to q will always be hashed in the same subspace and thus always in the returned nearest neighbor set. However, proving this is not obvious and more challenging as it is for f-RHH, since the angle of the indecisive area is not fixed. The same case is also true for i-RHH: Although the angle in i-RHH is fixed, it needs to be determine how many points are within that area. This in turn depends on the dataset and the distribution of the data points and thus can not be answered in general.

6 Conceptual Approach

In order to proof and evaluate the performance, all RHH extensions presented in Chapter 5 were implemented and tested for different scenarios[1]. The results are compared to a standard implementation of Random Hyperplane Hashing. We will refer to the dataset that is feed to the LSH Forest as training set.

All extensions can be used with the normal LSH Forest, but in this case some smaller modifications to the algorithm need to be done. For this reason a modified version of the LSH Forest was implemented and used among all extensions to ensure the comparability of the results. The forest consists of multiple binary trees. Moreover, each layer of each tree has a unique hash functions assigned, which is used by its layer nodes to compute the hash value. The initialization phase for RHH and f-RHH is done by inserting each point of the training set sequentially into the forest, while for p-RHH and i-RHH the points were inserted at the same time, so that both algorithm can work correctly. However, since the experiments will focus on comparing the different RHH extensions, the training set will be determined once and does not support any insertion of training points afterwards. Hence, there will be no performance difference if the training points are inserted at once or consecutively. The query procedure is exactly the same for RHH and all 3 extensions. Therefore, for any query point q the query procedure for the k nearest neighbors is as follows:

1. Point q is inserted into each tree (top-down phase):
    a) For each level of each tree, q is hashed with either of the RHH extensions
    b) If a leaf or dead end is encountered, the found near neighbor candidates are marked and the tree is deactivated
2. After all trees are processed, the near neighbor candidates are synchronously collected from all trees, until the near neighbor limit k is exceeded (bottom-up phase)
3. The near neighbor candidates are filtered by their similarity to q and only the k closest are returned.

As Random Hyperplane Hashing demands the points to be given in their vector representation, the near neighbor candidates will be filtered by their angular distance to the query point.

[1] The implementation is available under https://git.rwth-aachen.de/iraklis.dimitriadis/lsh

When using LSH Forest with Random Hyperplane Hashing, many opportunities to improve the performance of the algorithm are available. However, the performance of the algorithm can not be simply compared by a single factor, but depends on the goal that is defined. More precisely, the performance of LSH Forest can be measured by different characteristics and depends on the use case, which makes some of the characteristics more important than others. This makes the comparison of different RHH extensions more complex, since each one focuses on improving only some of those characteristics. Therefore, the experiments in this thesis will focus on mainly three characteristics: (a) query time (b) space allocation and (c) correctness of the near neighbor set. In order to understand the computational results in Chapter 7, we will shortly highlight the key factors for each of them.

(a) Query Time: The query time is recorded from the moment the query point is inserted into the forest until the near neighbor set is computed and returned. Since the query procedure is exactly the same for all four RHH algorithms, the query time will depend on their hashing efficiency and the amount of times a query point needs to be hashed. How often a query point is hashed depend on the average depth of each tree, which can be different for any of the implemented RHH algorithms.

Another factor that has an effect on the query time is the amount of near neighbor candidates. Since the similarity of each candidate has to be compared to the query point, some computational effort is necessary to compute the distance between them. Therefore, the amount of near neighbor candidates has to remain low, while the correctness of the near neighbor set is high. The number of candidates strongly depend on the hashing procedure of each extension, and therefore it is important that a single point is only hashed into multiple subtrees when necessary.

(b) Space Allocation: The space allocation is represented by the amount of nodes that are necessary to structure the trees of the forest. This amount depends on the depth d of each tree, the forest size t and how deep the data points of the training set are hashed into each tree. The RHH extensions differ in the amount of generated nodes, as each one follows an alternative approach to increase the correctness of the near neighbor search. As a result, the training points are distributed differently, where the amount of necessary nodes can vary. As one of the main reasons, hashing points as indecisive necessarily increases the amount of nodes in a tree. The reason for that is that hashing points as indecisive increases the size of the dataset in each iteration and can expand the tree significantly, which makes it important to keep the amount of those points low.

(c) Near Neighbor correctness: The performance of the near neighbor correctness is essentially given by the quality of the returned near neighbor set. The quality itself depends on the distances between the query point and all points of the approximated near neighbor

search relatively to the ones of the optimal set. The quality of a near neighbor search is then computed as follows:

Definition 11. *The quality Q of an approximated k near neighbor set $P = \{p_0, ..., p_k\}$ for its given optimal set $\bar{P} = \{\bar{p_0}, ..., \bar{p_k}\}$ is computed by*

$$Q = \frac{\sum_{i=0}^{k} d(\bar{p_i}, q)}{\sum_{i=0}^{k} d(p_i, q)},$$

where d is a distance function and q the query point.

The value of Q is between 0 and 1, where 1 represents the best possible case as this is only the case if the approximated near neighbor set is the same as the optimal one. Therefore, the closer the quality Q is to 1, the better is the approximated near neighbor set. In order to measure this quality, it is necessary to compute the exact distance between the query point and each of the k near neighbor points. Since the data will be processed and represented as vectors, the distance of two data points is represented by the angular distance between their projected vectors.

The quality of the near neighbor set strongly depends on the selected candidates, which in return is related to the hashing procedure of the RHH algorithms. RHH distribute the data points so that the query point is more likely to be hashed in the same leaf as it nearest neighbors. While this is true for all implemented RHH algorithms, the RHH extensions will increase this probability by hashing some points multiple times. However, this will have a negative effect to (a) and (b) and it remains to find a good balance between the quality of the near neighbor set and the query time as well as the space allocation.

The performance of each algorithm is tested by a series of experiments. In order to have a good and representative evaluation, the same experiments were repeated for each RHH algorithm and the results compared among each other. As an additional task, some test scenarios will validate the classification efficiency of the RHH algorithms and see if any of the extension can outperform RHH. The scenarios will consist of three different datasets, where two of them will be text and one sound based.

6.1 Datasets

The datasets to test the RHH algorithms were selected so that each one challenges the LSH Forest algorithm on another domain. We will discuss each one in detail and explain the vector generation process for each dataset. This will be followed by a more detail explanation about

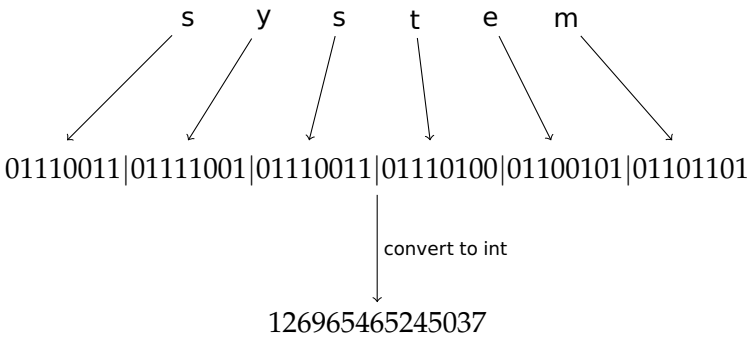


Figure 16: Shows the process of transforming the word "system" into a number.

the hyperplane generation, since the generated vectors will be of infinite dimension.

6.1.1 ACM Papers (ACM)

This dataset is a collection of different ACM papers [**rahm:2017**]. While it contains bibliographic data, with 4 attributes title, authors, venue, year, only the information of the title will be used to generate the vector. The short title of every data record does not contain much information and will challenge the LSH algorithm such that it will be more difficult to keep similar titles close to each other. In addition, this dataset contains data records with the same title and will help to investigate the behavior of the different RHH variants for datasets with duplicate records, as LSH Forest will hash same data records with the same information in leafs on the bottom most available layer.

In order to to transform the data to its vector representation, each data record will be tokenized into single words and stopwords removed. The numerical representation of each word is then used as the index of the vector, while the frequency of it represented its value. In order to represent a word as a number, every character was converted into an 8-bit vector. This bit vectors are then concatenated and the sequence of bit translated to its integer value. Figure 16 shows how the word "system" is translated into its decimal representation. The binary string of this word consists of 5×8 bit blocks, where each block is representing a character. For example the bit representation of character "s" is 01110011, and therefore the bit string begins with exactly that prefix. This bit block is translated into a number by transforming it as follows:

Definition 12. *Given a bit string $D = d_0, d_1, ...d_{n-1}, d_n$ its decimal number is equal to:*

$$decimal(D) = d_0 \cdot 2^0 + d_1 \cdot 2^1 + ... + d_{n-1} \cdot 2^{n-1} + d_n \cdot 2^n$$

By using the decimal number of a word as index and its frequency as the value to it, we obtain a vector that represent the data record by its words and frequency. As these numbers will be very large while the amount of words in the titles will be usually less than 20, the

generated vector will become very sparse. Furthermore, the amount of words is basically unlimited, since each concatenation of random characters can be interpreted as a word. Therefore, the dimension of the vector will be $+\infty$. However, since each text document, or in this case the title of the data record, is limited by the amount of words, this vector will have a limited number of non-zero entries. This fact will enable us to compute the distance of the vector and the hyperplane, as we will see in Section 6.2.

6.1.2 Bag of Words (BOW)

This dataset consists of five text collections in the form of bag-of-words [**Dua:2017**, **newman:2008**]. This means that the documents are not given in text form, but are represented by a list of unique numerical word id's and their frequency. This list does not contain any stopwords and is filtered so that it includes only words with a frequency of 10 or higher. The dataset is ideal for clustering and classification topic modeling experiments and fits very good into the scope of this thesis. At the time this thesis was written, one of the collection was not available, and therefore not included in the tests. The remaining four text collections consists of the following sources [Name (documents, unique words, total words)]: Enron Emails (39861, 28102, 6400000), NIPS full papers (1500, 12419, 1900000), KOS blog entries (3430, 6906, 467714), NYTimes news articles (300000, 102660, 100000000).

The vector generation of this dataset will be similar to the ACM paper. However, removing stopwords and transforming the words of the text document into its decimal number was not necessary, since each document was already represented by a unique word id and stopwords removed. Therefore each word id represents the index to an vector, while their frequency was taken as its value. In addition, each data record will be labeled with the name of its class collection which will then be used to validate the results of the classification tests.

6.1.3 Urbansound8k (U8K)

This dataset contains 8732 labeled sound excerpts (<=4s) of urban sounds from 10 classes: air conditioner, car horn, children playing, dog bark, drilling, engine idling, gun shot, jackhammer, siren, and street music [**urbansounds:2014**]. It was designed for audio recognition and sound classification and fits perfectly as dataset for the test scenarios of this thesis. The work of Salamon et. al [**Salamon:UrbanSound:ACMMM:14**] gives a close look into the composition of this dataset.

The vector generation of the sound excerpts is more complex, as the data model to generate them is not as simple as for text based elements. For example, a text can be transformed into a vector based on its words and their frequency. Thus, the cosine similarity is high, if two documents have many words in common. For sound files this is more difficult, such that we first need to find a property for which we can compare two sound

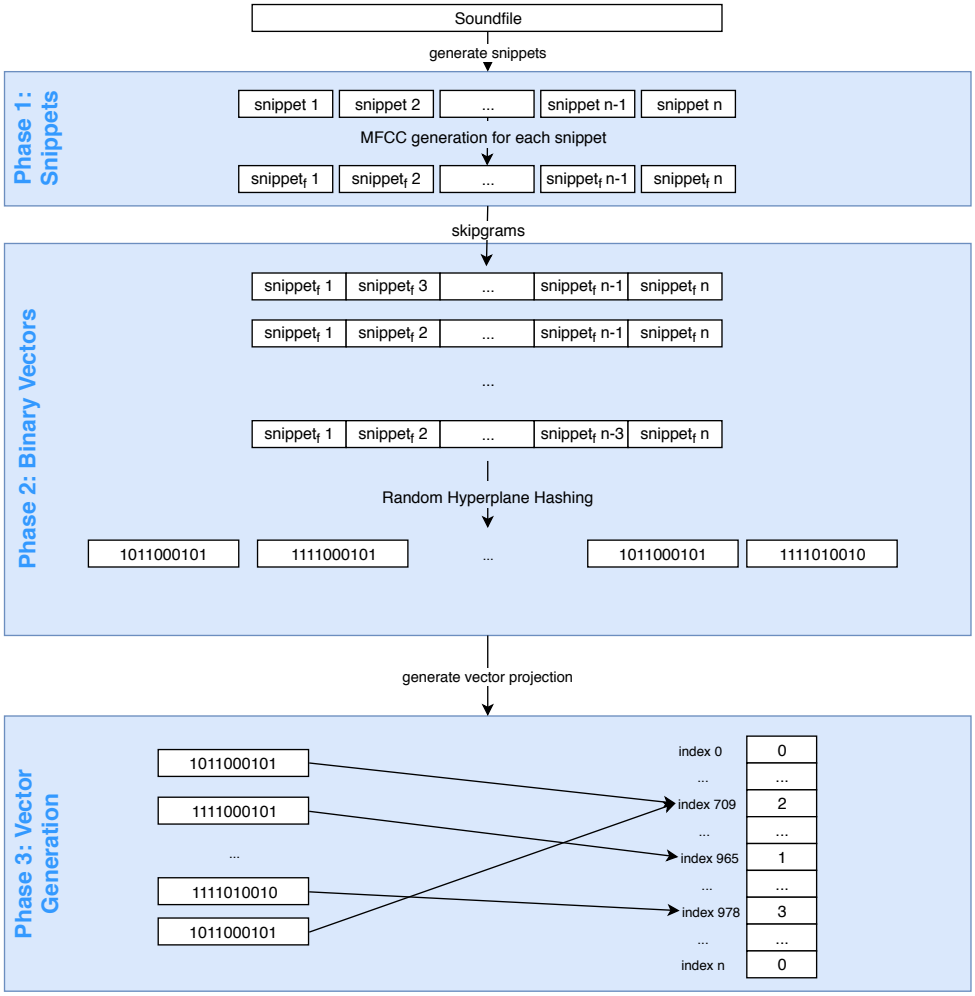


Figure 17: Shows the processing of a single sound excerpt in 3 phases [**cochezCom**]

files. The *Mel Frequency Cepstral Coefficients (MFCC)* has its origin in speech recognition [**rabiner1993fundamentals**]. Human generated vocals are shaped by the tract of the tongue, the movement of the mouth, teeth etc. Therefore, its sound is determined by this shape. The MFCC offers a way to represent this shape by a short time power spectrum of the phoneme and has proven itself as property to compare human generated sounds. The work of Logan et al. [**logan2000mel**] showed that this approach of speech recognition is also applicable for music modeling.

This thesis will use MFCC in order to transform a the sound excerpts into a vector, so that two sound files can be compared based on their MFCC properties. However, comparing just the MFCC vectors of the sound excerpts is not enough, since some of them contain noises that distort the MFCC generation and make them useless as such. For this reason, a data model was developed which will reduce the distortion of noises by leaving out some smaller parts of the sound excerpt. In order to do so, the vector of a sound excerpt will be obtained

by processing it three phases:

1. snippet generation
2. binary vector generation
3. final vector generation that represents the sound excerpt

The illustration in 17 shows the whole process of the vector generation. The first phase divides the sound excerpt in to smaller ordered snippets. Afterwards, the MFCC vectors for each snippets will be generated and concatenated with *skipgram*. Skipgram is a technique, where elements in a sequence are concatenated so that the length of the concatenation consists of n consecutive elements, while at most k elements of the sequence are skipped. A more formal definition is given as follows [**guthrie2006closer**]:

Definition 13. *For a given sequence $S = s_1, ..., s_n$ the set of k-skip-n-grams is define as:*

$$\{s_{i_1}, s_{i_2}, ..., s_{i_n} | \sum_{j=1}^{n} i_j - i_{j-1} < k\}$$

Generally speaking, the k-skip-n-gram set of the snippets set is every possible ordered concatenation of n consecutive snippets, while at most k snippets are skipped in between. If for example the set of snippets is given by $S = \{s_1, s_2, s_3, s_4\}$ its 1-skip-2-gram is given by $S_{gram} = \{(s_1, s_2), (s_1, s_3), s_1, s_4), ..., (s_2, s_4), (s_3, s_4)\}$. The idea is to compare two sound excerpt by their skip-gram sets. If sound excerpts are similar, but one of them is distort by noises, they will still have some skip-gram vectors in common.

In the second phase of the process, we apply several Random Hyperplanes to the skip-gram set. By doing so, each element in the skip-gram set is represented by a bit string, where the length of the string is defined by the number of hyperplanes that were applied to each skip-gram.

In the last phase, these bit strings are mapped to a vector as it was described in Section 6.1.1, so that a single vector represents the skip-gram set.

There are several parameters that can be set and will influence how good a vector represents a sound excerpt. Therefore, the test will be executed for different settings of the following parameters:

- amount of MFCC features
- seconds of a single snippet
- k and n for skip-gram generation
- amount of hyperplanes to generate the binary vectors

Section 7 will provide a more detailed analysis of those settings.

6.2 Hyperplane Generation

In some datasets the generated vector of the data records will be of infinite dimension. For example, in the ACM Papers dataset, each index of the vector represents a word and its frequency. However, the amount of words is theoretically unlimited, as any sequence of character can be defined as word. In order to represent any possible word, the dimension of the vector has to be infinite. As a consequence, the angular distance between a hyperplane and those vectors has to be performed in an infinite space, also known as *Hilbert Space* [**akhiezer2013theory**]. The Hilbert Space generalizes the rules for the Euclidean Space, so that the algebraic vector methods are applicable to any finite and infinite dimension. For the Euclidean Space, the angle between a vector $\vec{V} = (v_1, ...v_n)$ and the normal of a hyperplane $\vec{N} = (n_1, ...n_n)$ is defined as

$$\cos\theta = \frac{V \cdot N}{\|V\| \, \|N\|},$$

where $\cdot$ is defined as their dot product:

$$V \cdot N = \sum_i v_i n_i$$

The dot product can be extended to the infinite dimension if this series converges. This is given if both V and N satisfy:

$$\sum n_i^2 < +\infty$$

This case is only satisfied by the vector, since the normal of the hyperplane is of infinite length with an infinite amount of non-zero entries. Thus, its norm $|N| = \sum n_i$ will not be smaller than $+\infty$. While this being true, it is known the entries of $\|N\|$ are limited by the amount of different words in all documents. Moreover, $\|N\|$ is replaced by a constant value. Knowing this, both V and N will satisfy the condition, and therefore the computation of the angle between them can be done as above. Note that this is also possible because we can do the same assumption as we did for f-RHH: Since the normal vector N will be the same for every vector, $\|N\|$ will be constant for every angle computation, and therefore be replaced by a constant value.

7 Computational Results

All experiments were run on a standard desktop computer (Linux Fedora 29, Intel Core i5-3570K CPU @ 3.40GHz and 8GB RAM) and will cover different scenarios. They are explicitly designed to test the performance of the RHH extensions presented in Chapter 5. Furthermore, the results are averaged by 100 samples for the same scenarios and will provide a good empirical overview. The random hyperplanes will be generated based on a seed and shared among all RHH algorithms to ensures that the results are comparable to each other. As key points of this experiments we will focus on the three properties, mentioned in Section 6: (a) query time, (b) space allocation and (c) correctness of the near neighbor set. Measuring these is not a trivial undertaking, because the result depends on many parameters that need to be set. For this reason some of those parameters will be the same for all scenarios. Unless otherwise stated, the forest will consist of 10 trees and the near neighbor search will search for the 5 closest neighbors.

In the first two scenarios, we will validate the performance of each variant for the ACM and BOW dataset. As the datasets differ in dimensionality and data information, each one will proof the performance of the algorithms for different requirements. The results will be discussed independent and connections will be made if necessary. In the third test scenario the classification capability of each RHH variant will be proofed explicitly for the BOW dataset, as this dataset is known to be good for classification tasks. In the end, we will shortly evaluate the presented data model of Section 6.1.3 and discuss if the vector representation of a sound excerpt is reasonable. This will be done by applying the same classification tasks as for the BOW dataset.

All scenarios are structured so that they will test the performances for different tree depths ranging from 1 to 25, while the training and query set will remain the same for each depth. This will allow to observe the behavior of each algorithm for different tree depths and ensure comparability among different tree depths. Figure 18 shows the quality of the query process for (a) p-RHH and (b) f-RHH with different parameters. It can be observed, that increasing these parameters improves the quality of the returned near neighbor set of both extensions, so that each tested tree depth shows increased near neighbor correctness. However this results in the drawback that the amount of nodes to store the tree structure is increased as the following sections will show.

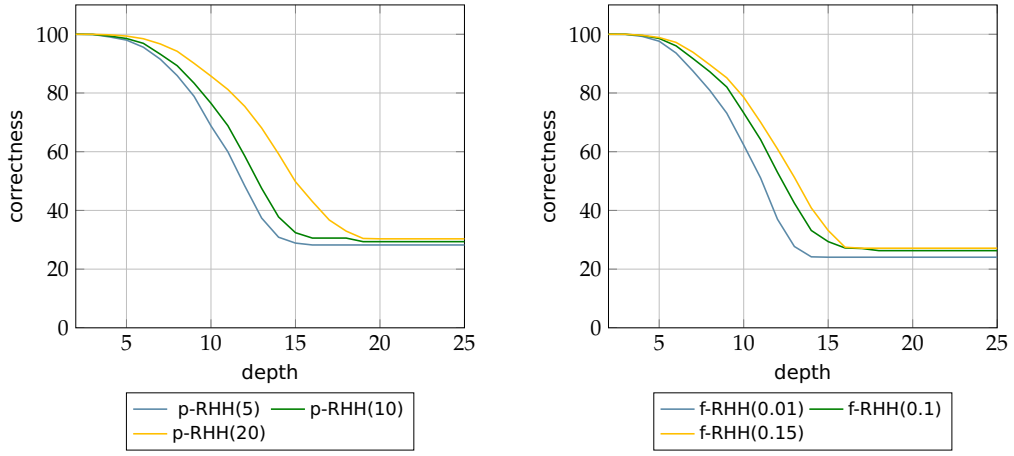
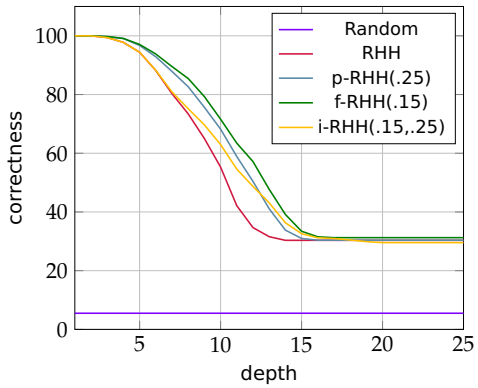


Figure 18: Shows the average correctness of an NNS for (a) p-RHH and (b) f-RHH for tree depths 1-25

7.1 ACM Dataset Scenario

In the first scenario, we built the forests with 2000 training points of the ACM dataset. Note, that in order to find reasonable parameter for f-RHH, p-RHH and i-RHH, we investigated the dataset and computed the distance between random points and hyperplanes. It was observed, that the distance was at least 0 and most 3, while the median was 0.6 and the average 0.7.



d	RHH	p-RHH	f-RHH	i-RHH
1	100	100	100	100
2	99.98	100	100	99.98
3	99.41	99.74	99.74	99.41
		...		
8	73.41	82.66	85.46	75.16
9	65.02	75.65	79.30	69.68
10	55.28	68.21	71.60	62.95
11	42.14	58.88	63.44	54.60
12	34.68	50.42	57.17	48.70
13	31.58	41.11	47.72	43.13
14	30.34	33.82	39.16	36.33
15	30.34	30.98	33.43	32.60
16	30.34	30.38	31.53	31.20
17	30.34	30.37	31.25	30.96
		...		
25	30.34	30.37	31.25	29.56

(b)

Figure 19: Shows the average correctness of an NNS for the ACM dataset for each algorithm and tree depths 1-25

Table 19 shows average correctness for each tree depth. The indecisive parameter for f-RHH was set to 0.15 for f-RHH, the percentage threshold for p-RHH to 10% and for i-RHH these parameters were set to 0.15 and 25%. The first observation we can make is that all approaches perform clearly better than just selecting points randomly from the training set. While the average correctness of random selecting points is 5.5%, the lowest correctness for

all RHH was 29.56%. Although the indecisive parameters for each variant were set very low, means that only smaller portions of the points were hashed as indecisive, we can see that f-RHH and p-RHH perform better as RHH and supports the assumption that both algorithms should have a better correctness as RHH. While for tree depths greater as 16 the difference between them is negligible, we can observe that for depth in between 7 and 15 the difference is clear. For example, for the depth of 12, the correctness of i-RHH it was 57.17%, which is nearly 1.7 times better as the correctness of RHH for the same tree depth.

For i-RHH this observation was not possible, since the correctness for this algorithm is only better or at least as good as RHH for depth until 16, while for the other depths the correctness is slightly lower. This was a bit surprising and contradicts to the assumption we made in Section 5.3. The reason for this could be the limited amount of information that each data record provides. This will force the LSH algorithm to distribute the training points into smaller groups very early so that less points are available to determine the indecisiveness of a hyperplane and RHH becomes more efficient.

For tree depths larger as 16 the correctness of each algorithm does not change anymore. This seems reasonable since at this moment, most points are either hashed into a leaf as single or are sharing a leaf with very similar points, so that the probability of separating them is nearly zero.

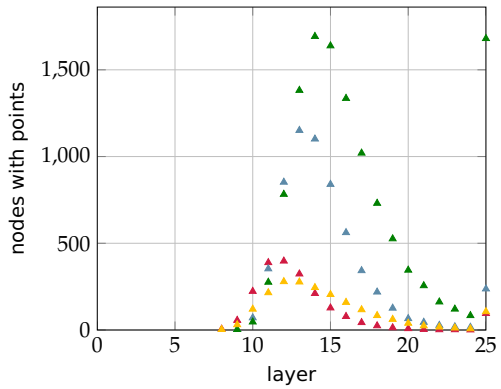
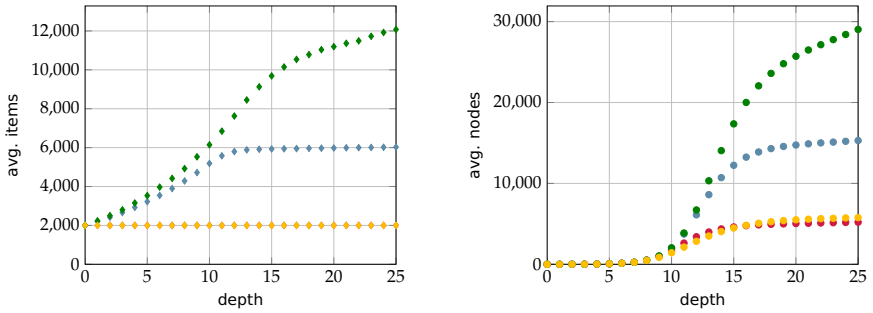


Figure 20: Shows the average points to layer distribution for the tree depth 25

Investigating the item distribution for the tree depth of 25 in particular also approves this assumption. From Figure 20 it can be seen that most points were hashed into leafs between the depth of 11 and 24. Furthermore, the histograms of f-RHH and p-RHH were slightly shifted to the right, when compared with RHH. This is expected and covers with the observation made in [**cochez2017large**] as i-RHH and p-RHH will generate more points for each additional layer, while for RHH the amount of training points will remain the same.

While most points are distributed on the layers 11 and 24, we can see that there are still many points hashed into a leaf of the bottom most layer. This has the reason that the nodes will be split and branched until only one point is remaining in the node or the maximum tree

depth is reached. In addition, this dataset was mentioned to have some similar data records that will consequentially force the algorithm to hash them to the bottom most layer, as those datapoints will always result in the same leaf. However, the amount of points on layer 25 for f-RHH was nearly as much as for the layer with the most points. The most likely reason to this is that f-RHH will continue to hash points as indecisive, if a node contains more than one item. Therefore, if points are so similar that any of the hyperplanes are unable to separate them, the chances are increased that those points will be hashed as indecisive.



(a) **(b)**

Figure 21: Shows average tree statistics for Figure 19, where (a) shows the average items and (b) average nodes per tree

Figure 21 (a) shows the average points per tree and supports this assumption, since with f-RHH the amount of points per tree is increased with each additional layer. An interesting observation is that for p-RHH the amount of points does not increase much more for the tree depth 12 and higher. If we compute the depth of a binary tree with equally distributed points to nodes for p-RHH as explained in Chapter 5.2, we can see that the result is very close to what we expected, since the depth for this scenario is expected to be 15.

Figure 21 (b) shows the average nodes per tree. It can be seen, that all fuzzy variants compute more nodes as RHH, with f-RHH having roughly 30000 for the depth of 25. Comparing this to the 5205.10 nodes in RHH, the amount is increased by a factor of 6 and not reasonable, when judging in terms of space allocation. For the same depth, the amount of nodes for p-RHH was 15296.80 and if we compute the estimated amount of nodes by $\min\left(k_{prhh}(2000, 0.1, 0), 25\right) = 16384$ we can again see that this is close to our expectations from Section 5.2. Comparing the average nodes of i-RHH with RHH, we can nicely see how indecisive hyperplanes influence the number of nodes for lower tree depths. We can see that the amount of nodes for i-RHH is smaller as for RHH if the tree depth is between 9 and 15. For depths higher as 16, i-RHH surpasses RHH. The reason for this is that i-RHH has defined some hyperplanes as indecisive. For each indecisive hyperplane, the points will remain in the same group and pass to the next layer. This is represented by the lower steepness as for indecisive hyperplanes only one node is generated to distribute the points into the next layer, while RHH is likely to generate two. As the amount of points that will be hashed by a

single node slowly decreases for each additional layer, RHH has more nodes with only one successor, since most points will already be distributed to a single leaf, while for i-RHH this effect is slightly delayed to a deeper tree layer. The effect of i-RHH can also be seen in Figure 20 as RHH have more points distributed between the layer 9 to 12 and i-RHH slowly catch up to distribute the points on the layer above 13.

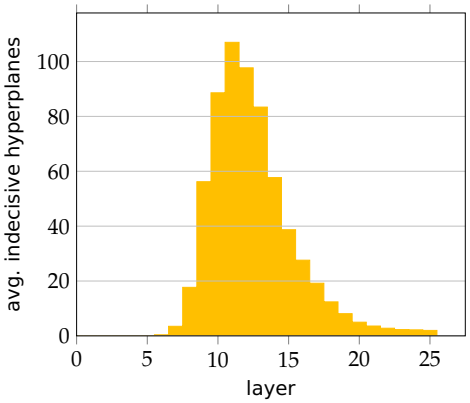
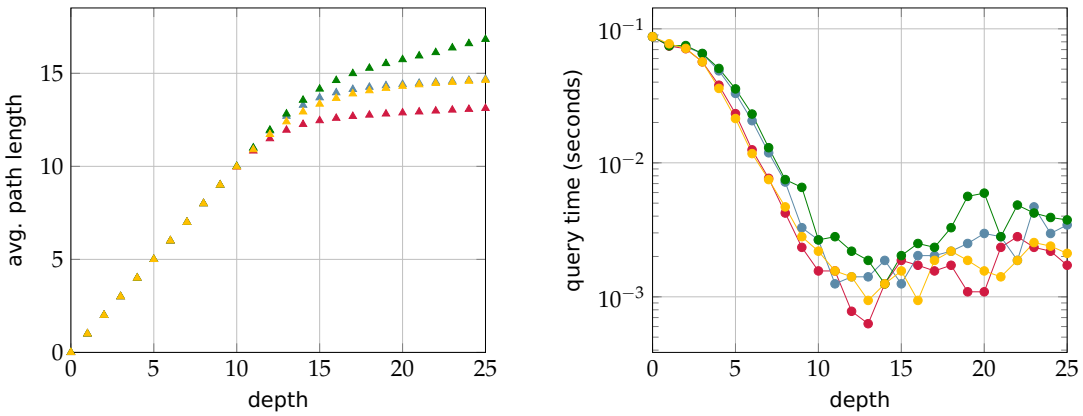


Figure 22: Shows the average number of indecisive planes per layer for the tree depth 25

Figure 22 shows the average number of indecisive planes per layer for tree depth of 25. As one can see, most skips were performed in between 10 and 13. This observation also matches to the previous ones, since from Figure 21 we could assume that most skips were performed in between those layers. Another interesting fact is that when summing up all indecisive planes is approximately equal to the number of nodes that are left after subtracting the ones of RHH and i-RHH. This makes sense since the indecisive planes represents nodes with only one successor, where RHH is more likely to have two successor. Note that we can only assume that RHH has two successor nodes, since this depends on the hyperplane and how the items are distributed into the next layer and thus the difference between the nodes in RHH and i-RHH are not exactly equal to the number of indecisive planes.

Figure 23 (a) shows the average path length for a point. Note that this is equal to the average fingerprint length for a single point, since each edge represents one value of the fingerprint. As we can see, the lowest length for all tested tree depths is achieved by RHH, where the maximum length for depth 25 is roughly 13. An interesting observation is that although p-RHH increases the number of points, its fingerprint length is the same as for i-RHH. This is more likely related to the fact that i-RHH hashes the items deeper into the tree, and therefore the fingerprint length becomes equal to the ones of p-RHH.

The average query time per tree depth is depicted in Figure 23 (b). During the query execution it was prioritized that there was no other process running which could falsify the search time. However, it still is possible that some unrelated computations were performed by the operating system, so that for low query times the results are not exactly representative. Under this consideration, we can see that RHH and i-RHH have the best query time



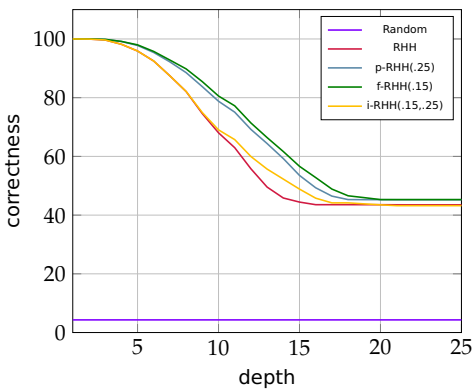
(a) **(b)**

Figure 23: Shows the average path length (a) and query time (b) per depth

until the tree depth of 6, while RHH becomes faster for trees with the maximum depth of 7 or higher. For the maximum tested tree depth, the performance of f-RHH and p-RHH was nearly the same and twice as slow as RHH. This is reasonable, as both algorithm increases the probability of similar points being hashed to the same leaf. As a result, the size of similar points in a leaf will be larger and worsen the query time, as the similarity to the query point has to be computed for each point of those points.

7.2 BOW Dataset Scenario

For this scenario, the forests were constructed by feeding them with 8000 training points of the BOW dataset. The parameters remained the same, since observations showed that the average distance of an random point to the hyperplane was approximately the same as for the ACM dataset.



(a)

d	RHH	p-RHH	f-RHH	i-RHH
1	100	100	100	100
2	99.96	100	100	99.96
3	99.66	99.86	99.86	99.66
		...		
8	82.10	88.53	89.81	82.12
9	74.57	83.75	85.46	74.93
10	68.04	78.79	80.57	69.04
11	62.97	75.07	77.23	65.72
12	55.78	69.20	71.39	59.96
13	49.55	64.38	66.41	55.68
14	45.83	59.27	61.66	52.27
15	44.44	53.52	56.62	48.86
16	43.55	49.31	52.79	45.77
17	43.55	46.47	48.87	44.17
		...		
25	43.51	45.24	45.29	43.22

(b)

Figure 24: Shows the average correctness of an NNS for the BOW dataset for each algorithm and tree depths 1-25

The correctness that is obtained by randomly selecting points from the training set was approximately 4.31% and outperformed by each of the tested RHH algorithms. Interesting to observe is that when comparing the results to the ones of the ACM dataset scenario, the average correctness of randomly selecting a point is lower, while the the average correctness for all tested algorithms is higher as for their counterparts. Beside that, it can be assumed that the training points are hashed much deeper into the tree. An indicator for this is that correctness steadily decreases until the tree depth of 19, while in the previous scenario, the correctness stopped to decrease after the the depth of 16. We can see that for the maximum tree depth the correctness for all algorithms is between 40.70% and 45.24%, while for the previous scenario, the correctness was between 29.56% and 31.25%. This is approximately a difference of 15% and can be explained by the larger training set size. Another reason for this is that the data records of this training set provide much more information, so that their vectors representation have more non-zero entries. This improves the near neighbor search, such that the near neighbor set to a query is less likely to contain points with low a similarity score.

Another observation that can be made is that f-RHH and p-RHH perform better, while i-RHH is slightly worse than RHH. In addition, the correctness for f-RHH was slightly better as for p-RHH. This is quite similar to what could be observed in Figure 19 and also supports the assumption that was made in 5.4, since both algorithm can outperform RHH in terms of near neighbor correctness. Comparing the i-RHH and RHH correctness with each other, we can see that RHH outperforms i-RHH for higher tree depths. This contradicts to the assumption that the reason for the bad performance of i-RHH in the ACM dataset scenario was related to the limited amount of information provided by the data records. However, it is possible that the amount of information provided by this data records is still not enough, so that i-RHH can still not work as expected.

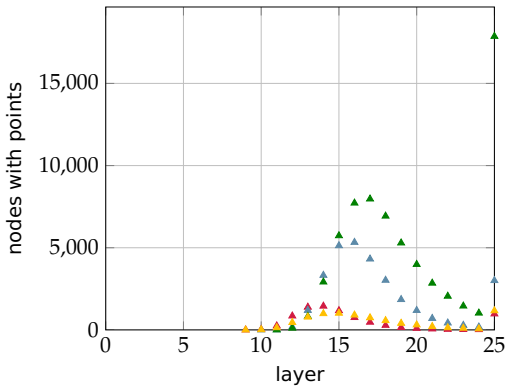


Figure 25: Shows the average points to layer distribution for the tree depth 25

Figure 25 shows the point distribution for the tree depth 25. It can be seen, that the histograms of p-RHH and f-RHH are again slightly skewed to the right and most points are dis-

tributed in between the layers 14 and 21. If we compare the amount of points distributed to the last layer, we can nicely see the how hashing points as indecisive for f-RHH increasing the amount of points that are hashed to the last layer. This was already observed for the ACM dataset, but the effect becomes more visible for the BOW dataset, for several reasons. Firstly, the size of the training set for this scenario is increased, and thus the amount of indecisive hashed points is likely to be higher. Secondly, this dataset contains more data points with a high similarity. This will increase the number of nodes that store more than one point, as those points are likely to be in the same leaf. As a result, they will be hashed to the bottom most layer, while for each layer there is a chance that f-RHH will hash them as indecisive. If so, the probability that f-RHH will hash more than one point as indecisive is more likely as hashing only one of them as such. Therefore, the same set of similar points can be in multiple leafs, while each time they are hashed to the bottom most layer.

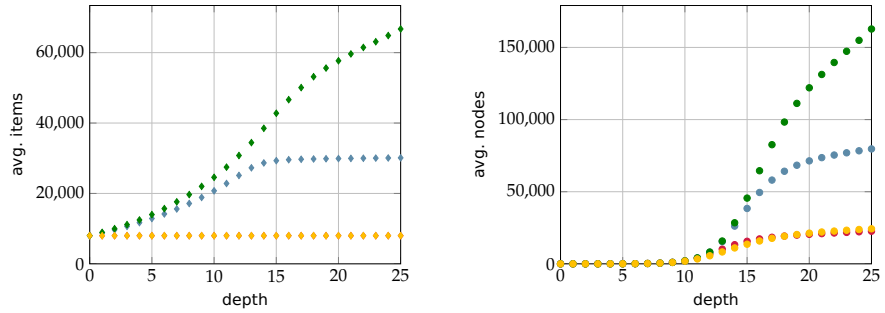


Figure 26: Shows average tree statistics for Figure 24, where (a) shows the average items and (b) average nodes per tree

Investigating the average points per tree as depicted in Figure 26 (a) we can make the same observation as for Figure 21 (a), as the amount of points for f-RHH is steadily increasing. Furthermore, we can see, that the proposed upper bound for generated points by p-RHH is also true for this dataset, as for tree depths larger as 15 most points are already distributed until layer 15. Given the maximal amount of necessary nodes to store those points as $\min\left(k_{prhh}(8000, 0.1, 0), 25\right) = 131071$ we can see that p-RHH is clearly under the proposed upper bound, as for this scenario only 79500 nodes were necessary to store the tree.

If we compare i-RHH to RHH, we can again observe a similar behavior as for the ACM dataset. The amount of nodes generated by RHH is higher in between the maximum depth of 9 and and 19, while for larger tree depths i-RHH surpasses RHH in terms of node generation.

Figure 27 shows the average indecisive hyperplanes per layer for the tree depth being 25. It can be seen that most indecisive hyperplanes are set for nodes between layer 12 to 16, while most of them are on layer 13. If we compare Figure 27 and 22 we can notice that for the BOW dataset the algorithm performed 4 times more skips as for ACM. If we consider that the size of the training set was also increased by the factor of 4, it seems like the amount of

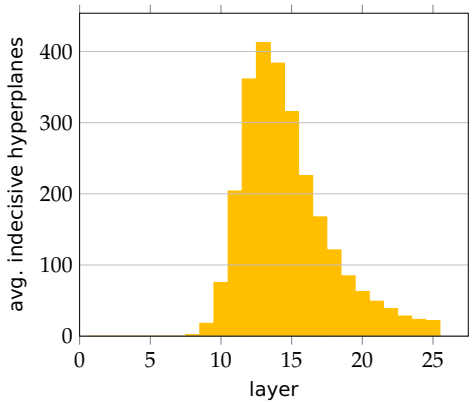


Figure 27: Shows the average number of indecisive planes per layer for the maximum depth of 25

indecisive planes correlates to the size of the training set. The reason to this could be that on layer 13 most points are grouped to smaller chunks, so that nodes of deeper layers will have just a few points to hash. If this amounts becomes lower as 4 it is not possible to enable the percentage threshold of 25% and set the hyperplane as indecisive. This is also represented by the steepness in Figure 27 as the amount of indecisive hyperplanes decreases steadily for each additional layer.

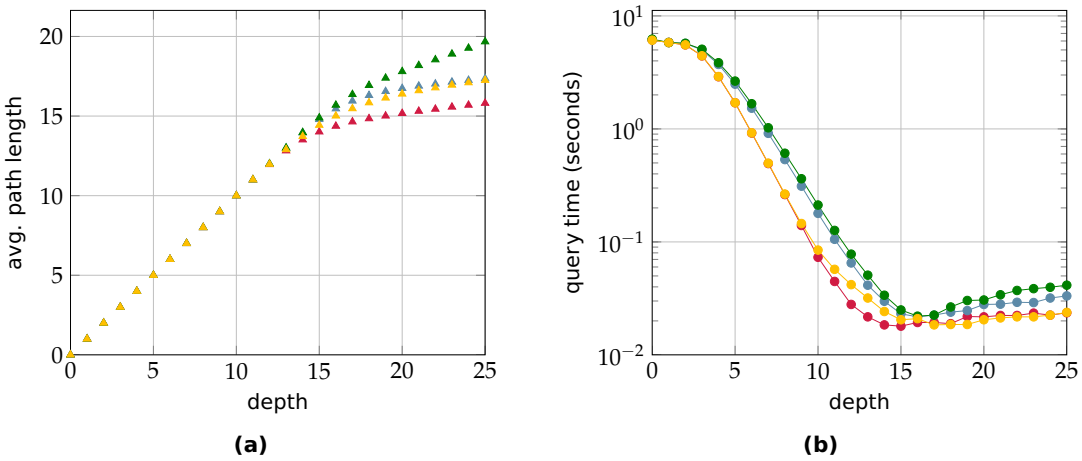


Figure 28: Shows the average path length (a) and query time (b) per depth

Figure 28 (a) shows the average path length per maximum tree depth. Again, this is very similar to what we observed for the ACM dataset. As long as all points are hashed down to the bottom most layer, the average path length matches with the maximum depth of the tree. This can be observed until layer 12 is reached, since the steepness of the average path length for RHH slowly starts to decrease. This is also the case for the other RHH algorithms, although the steepness is not decreased as much as for RHH. From Figure 28 (b) we can nicely see how the average path lengths influences the query time. Until the the maximum tree depth of 15 all algorithm suffer from the huge amount of near neighbor candidate, since

for each one its distance to the query point has to be computed. Note that for the depth of 0 this is equal to the exact near neighbor search and is thus of linear time complexity, as all points of the training set will be in the set of near neighbor candidate. With each additional layer, the data complexity decreases and the points in the near neighbor set become fewer. Increasing the maximum tree depth to 16 and above, the query time slowly increases. This indicates that the break even point where hashing the query point to a leaf becomes more time consuming as filtering the set of near neighbor candidates. If we compare the query time for the maximum depth of 25 we can see that the near neighbor search of f-RHH needs nearly twice as much as of RHH, while for p-RHH the the query time is approximately 1.5 times more. Comparing the query time of i-RHH and RHH we can see that the 5-NNS can be performed in approximately the same time.

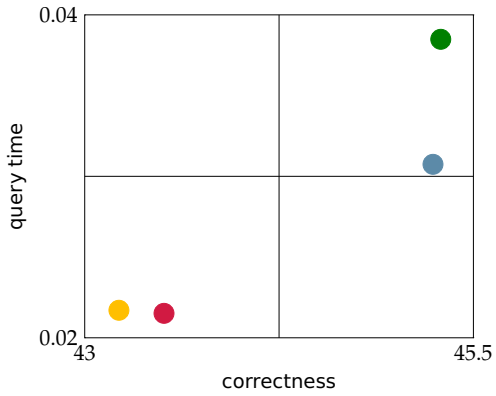


Figure 29: Shows the correctness in relation to the query time for the tree depth 25

The above scatter plot shows a space divide in 4 categories Figure 29 illustrates a scatter plot that categorize the algorithms by the query time and correctness for the tree depth of 25:

1. fast NNS/good NN quality
2. fast NNS/bad NN quality
3. slow NNS/good NN quality
4. slow NNS/bad NN quality

Obviously, the best possible category for any of the variants would be in the lower right quadrant, so that the query is executed fast and the quality of the near neighbors good. As we can see, the RHH algorithms can be grouped into two categories. RHH and i-RHH are representing algorithms that allows querying the near neighbor set fast, while the quality of it is less important. On the other hand, the query time for f-RHH and p-RHH is high, but the quality of the near neighbor search increased.

7.2.1 Classification Tests

This scenario is designed similar to Section 7.2, but aims to test the classification capabilities of each RHH algorithm. More precise, the same training and query set was used, so that each forest was build with 8000 points of the BOW dataset and the near neighbor search executed for 100 query points. This query points were then classified based on the obtained near neighbor set for each algorithm.

Figure 30 shows the class frequency in the training and query set. The training points are selected so that each class is represented approximately equally. Note that the nips collection only consists of 1500 documents and is slightly less present in the training and query set.

The classification of the query was done by following two different approaches: (a) Classify

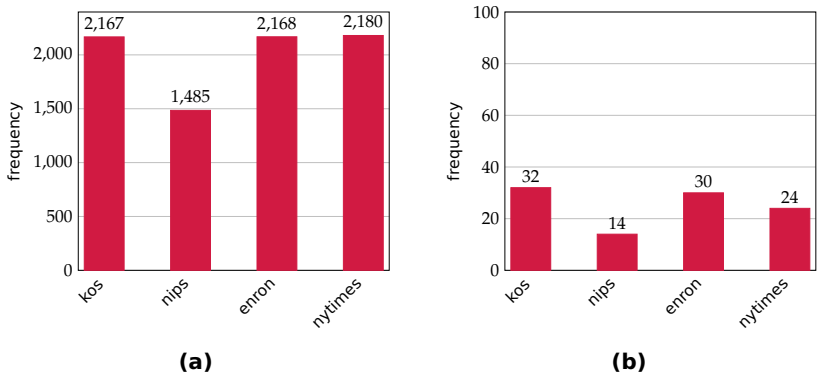


Figure 30: Shows the class frequencies of (a) the training set and (b) the query set

the query by the similarity score of all classes present in the near neighbor search result and (b) Classify the query by the frequency of each class present in the near neighbor search result. In (a), the class for a query is defined by a similarity score. This score is computed by grouping the near neighbor candidates to their classes and summing up their similarity. The query point is then assigned to the class with the highest score. In (b) we simply count the frequency of each class in the near neighbor set and the query point assigned to the most frequent class.

In order to proof if the query item was classified correctly, the computed class is compared to the original class of the query and if both are the same, the classification was successful.

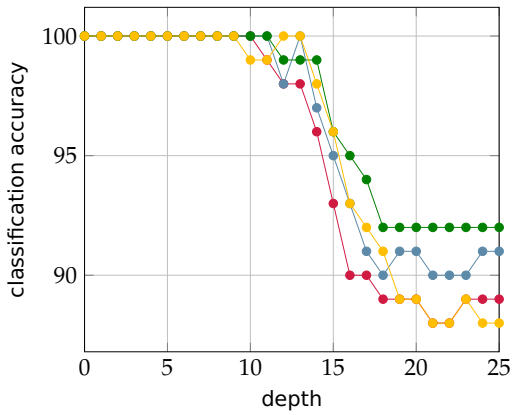


Figure 31: Classification result by score for depth 1-25

Figure 31 shows the outcome of the classification tests when the query is classified by (a). For lower tree depths, all algorithms achieved an accuracy of 100%. This indicates that for the optimal near neighbor set, the proposed classification method works as intended and can classify any given query point correctly. Even if the training space is divided in 256 subspaces, the classification accuracy remain at 100% for all tested algorithms. In between the depths of 12 and 18 the classification accuracy of all RHH algorithm drop steadily. The reason for that is most likely that at this point the training set is already well distributed into

the tree and separated in small groups of points, so that the quality of the near neighbor set is decreased and by that also the classification accuracy.

Investigating the outcome for the tee depths in between 19 and 25 it can be seen that the accuracy of all RHH algorithms stay the same, which indicates that the training points are already distributed into leafs and the query always finds the same set of near neighbors, even if the depths of the trees is increased. The accuracy for the depth 25 ranging from 88% to 92%. This is still a very good accuracy if we consider that for deeper trees the training set is separated into much more subspaces that will allow the classification to be done very fast.

If we rank the algorithms by their classification accuracy, it can be noticed that the ranking is exactly the same as ranking the algorithms by their correctness performance from Section 7.2. This indicates that the improved quality of the near neighbor set that was achieved by using f-RHH and p-RHH correlates to the classification quality of the algorithms. Furthermore, it can be assumed, that the similarity score of the near neighbor is a good indicator to classify the query item.

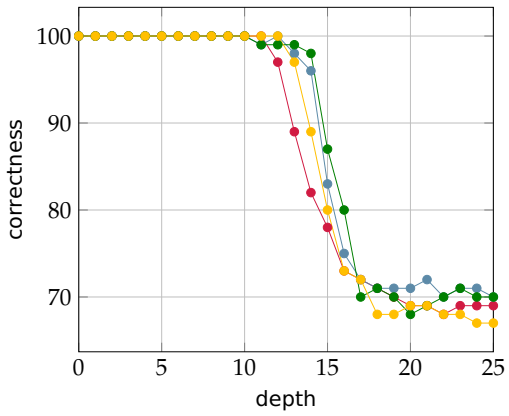


Figure 32: Classification result by frequency for depth 1-25

Figure 32 shows the classification accuracy by the frequency of the near neighbor classes. The classification accuracy for each algorithm evolves very similar to their counter parts from Figure 31 until the tree depth is 11 or higher. A bit surprising is that for this classification approach, the overall accuracy is much worse as for the score approach. In addition, expect i-RHH, all algorithm achieve approximately the same accuracy for deeper trees and the difference between the RHH algorithms is not as big as it was for the score approach. One reason for this could be that although f-RHH and p-RHH increased the quality of the near neighbor set, it still contains more points with different class labels, while points with the same class label have a very high similarity. This would also explains why for the score approach, the classification quality was much better.

7.3 UrbanSounds8k Scenarios

Testing the RHH algorithms with the UrbanSounds8k dataset differs from the previous scenarios, because it is unknown if the proposed data model to generate the vector for a sound excerpt is good or good enough. This made it quite difficult to analyze the outcomes of the RHH algorithms. In addition, many different parameter, like sound snippet length, n-gram and skip size, number of MFCC features and the the dimension of the hyperplanes add more complexity, as each combination of those changes the results significantly. For this reason, different combination were tested for the same scenario and their results compared. The training set was given by 8000 datapoints from the UrbanSounds8k dataset. After loading the datapoints into the LSH Forests, the result of the near neighbor search was averaged over 100 query points. Note that the parameters for f-RHH, p-RHH and f-RHH were the same as in the previous tests.

secs	n-gram	skips	MFCC	planes	∅-nz-entries	∅-rand	RHH	p-RHH	f-RHH	i-RHH
0.2	3	1	10	10	3.9	20.43	97.55	98.17	98.11	97.99
0.2	3	1	10	20	12.67	2.79	79.30	81.06	81.00	80.43
0.2	3	1	20	20	15.64	5.44	70.75	75.80	78.15	71.63
0.2	4	1	20	30	21.09	4.44	63.16	64.78	69.56	63.69
0.2	4	1	20	10	6.3	23.37	96.48	97.29	97.12	96.38
0.2	4	2	20	30	17.03	4.62	67.43	69.13	69.62	65.4
0.3	4	1	20	30	14.37	4.33	61.83	66.05	65.76	60.37

Figure 33: Shows the correctness for different parameter combinations for the UbranSounds8k dataset

The table given by Figure 33 shows how different parameter combinations change the correctness of the NNS. The column "nz-entries" represents the average non-zero entries in the generated vectors. Generally speaking, this number gives an intuition how good the LSH algorithm can distribute the dataset into the data structure, as more non-zero entries increases the chances to recognize two unsimilar data points. This can also be seen from the correctness of the near neighbor search results, as for a low number of non-zero entries, the correctness seems to be unreasonable high. The reason for that is that different types of sound excerpts are represented by nearly the same vector projection. This becomes clear if we compare the generated vectors for two totally different sound excerpts: The values in "vector" and "vector2" representing the index and its value to two different vectors, generated by the parameters in the table. For the first setup, the vectors of both sound excerpts are identically, and therefore will always have the same hash value computed. If we compare the vectors for the second setup it can be seen, that they are different from each others and therefore most likely have different hash values computed. This is not the only case and seems to happen very frequently, when the amount of non-zero entries is low. As a conclusion, we can assume that setups generating more non-zero entries are favorable, as the LSH algorithm can identify similar and unsimilar pairs much better.

```
{
    "name": "106015-5-0-14.wav",
    "class": "engine_idling",
    "vector": {
        "833": 52
    },
    "vector2": {
        "1046923": 52
    }
}
```

```
{
    "name": "162103-0-0-16.wav",
    "class":  "air_conditioner",
    "vector": {
        "833": 52
    },
    "vector2": {
        "982281": 41,
        "981257": 2,
        "982313": 9
    }
}
```


	secs	n-gram	skips	MFCC	planes
vector	0.2	3	1	10	10
vector2	0.2	3	1	20	20

Figure 34: Illustrates vector representations of two sound excerpts with different preprocesss settings

The setup that generates the most non-zero entries is given by ($secs = 0.2$, $n - gram = 3$, $skips = 1$, $MFCC = 20$, $planes = 30$). This is very reasonable if we think about how the different parameters changes the outcome of the generated vector. For example, if the time window for the snippets length is small, the amount of snippets will become higher. This on the other hand will allow to generate more n-gram-skip pairs, and therefore increase the amount of non-zero entries, as each one has to be represented by the vector of a sound excerpt. Choosing smaller n-gram lengths while the skip size is low will have the same effect, as this will result in more permutations of pairs. Increasing the MFCC features and planes will add more accuracy, so that the amount of similar n-gram-skip pairs is reduced and the vector becomes more dense, as each one has to be represented.

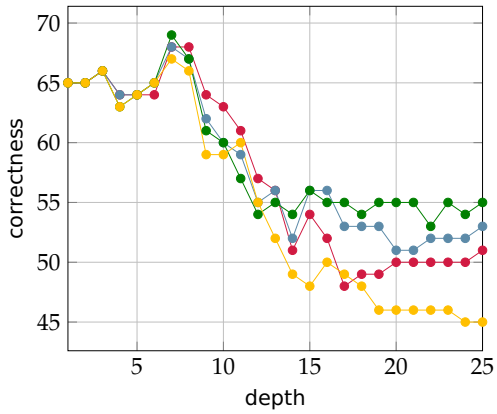


Figure 35: Classification result by score for depth 1-25

Running the same classification tests similarly to the BOW scenario for this particular setup, the classification accuracy by score for the tree depth of 25 was approximately 55% for f-RHH and p-RHH, while for RHH it was 50%. The accuracy computed by the i-RHH algorithm was at 45% and by far the lowest of all.

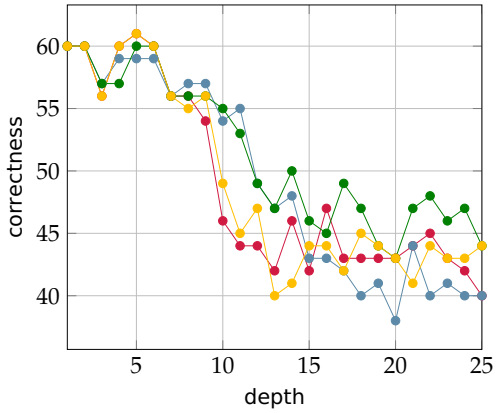


Figure 36: Classification frequency by score for depth 1-25

For the classification by frequency, the accuracy was around 42% for all algorithms. At the first look, both results seem a bit underwhelming as only half of the query items are classified correctly. However, since the amount of available classes are 10, the chances of randomly assigning the correct class label is exactly 10%. Compared to the classification results of f-RHH, the accuracy is increased by 45% which is a huge difference and indicates that the data model for the sound excerpts works partially as intended.

8 Conclusion and Further Work

This thesis presented three extensions to Random Hyperplane Hashing in order to improve the performance LSH algorithm. By running several test scenarios and investigate the outcomes, we found out that f-RHH and p-RHH can outperform the near neighbor correctness of RHH. We also observed that this comes with the drawback of increasing the space complexity of the LSH algorithm, since hashing points as indecisive will expand the trees, and therefore the space that is needed to store them. While the difference for forests with large tree depths was negligible, the gap between the correctness of the fuzzy approaches and normal RHH was noticeable for smaller depths, while the query time was still reasonable fast. In case of i-RHH no improvement was visible. This contradicts to the expectations as it was assumed that selecting better hyperplanes to distribute the training set would increase the chance of two similar points resulting in the same leaf. It would be interesting to see how this algorithm performs for datasets on a larger scale. Another option would be to test the algorithm for datasets that contain more information about each data record, as this allows to distinguish items more precise and could probably improve the performance of i-RHH.

The tests scenarios showed that the proposed upper bound for the amount of indecisive hashed points by p-RHH was as expected. This was a very interesting finding, since it allows to control the trade of correctness and space allocation more precise. This presents a good improvement over f-RHH, as the experiments showed that for this algorithm the amount of indecisive points is only bounded by the maximum tree depth. It would be interesting to see if it is possible to find a better indecisive bound for f-RHH while maintaining the correctness of the algorithm. Furthermore, the space complexity of all proposed extensions can be reduced by using PATRICIA tree, as it can be assumed that most of the nodes are only used to structure the tree and do not store any data points. The reduced tree representation for each algorithm can be different in the number of nodes, as each one distributes the points in a slightly different fashion. Furthermore, these compact tree representations could be compared and see if any noticeable difference to the results of this thesis are observable.

In addition, the classification capabilities of all proposed extensions were tested for two different datasets. We saw that for the BOW dataset, the classification accuracy of f-RHH was at 92% and for p-RHH at 91%. While this dataset was known to be good for classification task, they are still better as expected. Furthermore, the tests indicate that the improved correctness of f-RHH and p-RHH correlates to the classification capability of these algorithms. In the

UrbanSounds8k test scenarios, the accuracy for all algorithms was around 42%. While this seems a bit underwhelming, the tests did not clearly indicate if the proposed data model for this dataset was good enough. This makes it quite difficult to reason about the classification accuracy, as the reasons for this could be ambiguous. Investigating the data model for this dataset further would be the first option in order to validate the results of the classification tests. However, this demands further research that would simply be out of scope of this thesis, as the data model needs to be analyzed in isolation.

As further research, the presented approaches could be combined and see if there is any remarkable performance improvement. As most interesting combination, mixing i-RHH and f-RHH could be investigated and see if the overall correctness can be improved, while the space complexity is reduced, as we still expect that i-RHH should work better as RHH if the dataset contains more information to generate the vectors of each item. Another option is to allow each item to be hashed as indecisive only a specific amount of times. This could help to improve the space complexity of all algorithms, while maintaining the correctness as presented by this thesis.

The results showed that p-RHH and f-RHH represent an alternative to RHH that can improve the near neighbor correctness. This attributes these algorithms to be applied in real scenarios and vindicate further research.

9 List of Figures