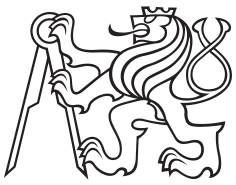
Bachelor Project



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Interactive Nearest Neighbor Search in High Dimensional Data

Prokop Černý

Supervisor: Lasse Blaauwbroek, MSc.
Field of study: Open Informatics
Subfield: Software
May 2019

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Černý** Jméno: **Prokop** Osobní číslo: **466375**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Interaktivní hledání nejbližších sousedů ve vysocedimenzionálních datech

Název bakalářské práce anglicky:

Interactive nearest neighbor search in high dimensional data

Pokyny pro vypracování:

The goal of this thesis is to create a fast approximate neareast neighbor search algorithm in an interactive environment, with the corpus of known data changing dynamically, for the purposes of being used for Coq Theorem Prover. Such algorithms need to strike a balance between speed and prediction accuracy, so evaluation of different algorithms will be an important part of the thesis.
Main objectives of the thesis are to
• Create algorithms and associated datastructures to enable this kind of search.
• Implement such algorithms in OCaml programming language
• Evaluate algorithms with respect to their speed and accuracy

Seznam doporučené literatury:

[1] J .Leskovec, A. Rajaraman, a J.D. Ullman – Mining of Massive Datasets – New York, NY, USA, Cambridge University Press, 2014
[2] Indyk, Piotr, a Rajeev Motwani - Approximate nearest neighbors: towards removing the curse of dimensionality – ACM 1998
[3] Andoni, Alexandr a Piotr Indyk. - Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions – IEEE 2006
[4] Berchtold, Stefan - Fast nearest neighbor search in high-dimensional space - IEEE 1998.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Lasse Blaauwbroek, Český institut informatiky, robotiky a kybernetiky

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **23.01.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **20.09.2020**

Lasse Blaauwbroek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Acknowledgements

I would like to thank my family for their unwavering support they have given me during my studies, my supervisor for all the help and advice he has given me, and last but not least, I would like to thank CTU for being a great *alma mater*.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 24. May 2019

Abstract

Nearest Neighbor search is an important problem in many fields of study. In this thesis we concern ourselves with this problem for a high dimensional dataset in an interactive setting, with frequent insertions and deletions from corpus of known data in which we search for nearest neighbors, specifically for use in automated theorem solvers such as Coq, HOL4 and others.

We explore different similarity measures, to decide which is a best fit for our use case and data, and use techniques such as Minhashing, Locality Sensitive Hashing and LSH Forest to accelerate the search for nearest neighbors, while keeping a reasonable accuracy. We implement our algorithms in OCaml to interface with the Coq theorem prover.

Keywords: nearest neighbor, kNN, search, minhashing, LSH

Supervisor: Lasse Blaauwbroek, MSc.
Czech Institute of Informatics, Robotics, and Cybernetics,
Jugoslávských partyzánů 1580/3,
Praha 6

Abstrakt

Vyhledávání nejbližších sousedů je důležitý problém v mnoha oblastech. V této práci se zaobíráme tímto problémem pro vysocedimenzionální data v interaktivním prostředí, s častým přidáváním a odebíráním z korpusu známých dat ve kterém hledáme nejbliží sousedy, specificky určeno pro použití v automatickém dokazování teorémů v nástrojích jako Coq, HOL4 a podobně

Zkoumáme různé metriky podobností, abysme rozhodli která se nejlépe hodí pro naše účely a datům, a použijeme různé techniky jako Minhashing, Locality Sensitive Hashing (hashování citlivé na rozmístění) a LSH Forest pro zrychlení vyhledávání se současným zachováním dostatečné přesnosti. Naše algoritmy implementujeme v OCamlu aby byly použitelné v Coq theorem proveru.

Klíčová slova: nejbliží sousedé, kNN, vyhledávání, minhashing, LSH

Překlad názvu: Interaktivní vyhledávání nejbližších sousedů ve vysocedimenzionálních datech

Contents

Figures

Tables

Chapter 1

Introduction

k-Nearest neighbors (k-NN) search is a classic computer science problem with applications in many areas, such as computer vision, pattern recognition, clustering and so on. We will be exploring k-NN search techniques to use for datasets in the field of machine assisted theorem proving, meaning searching in datasets with high dimensionality, with a focus on query speed. The use case of k-NN search in theorem proving also needs the ability to dynamically add and remove to the corpus of known data the search works on, as because the proving process is interactive. We will achieve that by requiring that the data structure holding the database used by the k-NN search algorithm will provide these operations. The goal of this thesis is to explore possible techniques to solve this problem in OCaml, so it can easily integrate with the Coq theorem prover.

1.1 Overview of Nearest Neighbor search techniques

The need to search for similar items emerged quickly in computer science. k-Nearest neighbor search has a very wide applicability in many fields, and therefore has a lot of variants, with two main overarching categories emerging. First the exact nearest neighbor search, which always return the true nearest neighbors in a given metric space, and secondly approximate nearest neighbor search, which trades accuracy for speed, memory consumption or both. We shall discuss some of the techniques in the following paragraphs.

1.1.1 Exact methods

Naive search

This is the simplest form of k-NN search, which computes similarities between a query and all points in a database and returns them in descending order according to their similarity. This kind of search is simple to implement, but its query time is quadratic to the size of the database as we compare everything in the database to previous entries and can be practically used only for small datasets. We will use naive search to compare different similarity metrics to select the most appropriate one for our use case.

Partitioning search

These search methods split and partition the search space into smaller chunks to speed up the search and find k nearest neighbors for a given query.

One of the basic algorithms in this category is a k-d tree, originally proposed by Bentley [1]. This algorithm works on k-dimensional data (hence k-d tree) and is a binary tree which on each level splits the search space in two partitions based on a points position in the dimension corresponding to the level in the tree. This algorithm works very well for data with lower dimensional spaces, but its performance degrades in higher dimensions.

Other algorithms for partitioning search include quadtrees and octrees [2], for two and three dimensional spaces respectively, mainly used in computer graphics. Next are R-tree [3] and R*-trees [4], which work by grouping nearby objects together, creating their minimum bounding rectangle. For general metric spaces, an algorithm such as Vantage-point (vp) tree [5] can be used, which on each level of a tree chooses a so-called vantage point and groups points into two groups, those within a specified distance to the vantage point and those which are not.

1.1.2 Approximate methods

Performance of exact methods degrades in high dimensional spaces, because of the curse of dimensionality, as the distance from a query point to its nearest neighbor starts to approach the distance to its farthest neighbor. Therefore techniques were created which relax the criteria to find only approximate nearest neighbors, while gaining speed compared to exact methods.

Approximate partitioning search

Techniques emerged from tree based partitioning search, with the relaxation of allowing approximate neighbors instead of exact ones. Muja and Lowe [6] present an algorithm based on kd-trees, by creating a forest of randomized kd-trees and searching across them in parallel. Another tree-based technique is Rank Cover Tree [7], introduced by Houle and Nett, which distinguishes itself by pruning the search space solely based on the comparison of similarity values to a given query.

Graph based techniques

There are various techniques based on constructing and traversing a graph. Arya and Mount introduce the Randomized Neighborhood Graph [8], which is designed to work on Euclidean spaces. It has the set of stored points in its database as vertices of a graph and then creates cones which have a point from the database as an apex and connects the point to a number of other random points which fall into a given cone.

Another technique by Malkov and Ponomarenko [9], Navigable Small World graph, works by creating a graph in which each object in the algorithm's database corresponds to a vertex. During insertion each point is connected to some number of its nearest neighbors. Querying works by greedily traversing the graph and selecting points based on their distance to the query point. Malkov and Yashunin improve upon this technique, proposing Hierarchical Navigable Small World [10], suggesting a graph structure with shorter traversals compared to his original paper.

Locality Sensitive Hashing

Another direction of research in mitigating the curse of dimensionality for nearest neighbor problems uses hashing. Hashing is typically used to create fast approximate "fingerprints" of data, with as few collisions as possible and with small changes in the input causing large changes in the output. But we can also use hashing the other way, where we want data with distance below a certain threshold (in some metric space) to collide with high probability, while simultaneously data with distance above some threshold to collide with low probability. Groups of hashing functions with this property are called Locality Sensitive Hash families (see section 5.1.1).

Indyk and Motwani propose Locality Sensitive Hashing (LSH) [11] as one possible solution to the nearest neighbor problem using families of locality

sensitive hashing functions. They use these functions to map similar items into buckets, and when searching for nearest neighbors they only consider points from buckets into which a query would hash into.

Bawa, Condie and Ganesan also use LSH families to create a self-tuning similarity search index called LSH Forest [12]. They propose storing the search points in prefix trees (tries), with the path to a point in a trie specified by outputs of functions from a LSH family.

In this thesis we have decided to explore the LSH approach for our solution, as we need to be able to work in an interactive manner and a hashing approach offers speed.

1.2 Problem statement

Our goal is to create a k-nearest neighbor search used as a tool in machine assisted theorem proving. Given the current state of the prover, find most similar states we have recorded in our database and return the actions which were taken in those states, sorted by the similarity of the states to the given query state.

Formal description

To create a k-NN search algorithm, we do not need to have knowledge of the inner working of used theorem prover and its representation of state. As such, the datasets for our search algorithm consist of tuples (fl, a), where fl is a feature list containing features representing a given state, and a is the action taken in the given state. With this knowledge we can now formally describe our problem.

Definition 1.1 (feature list)**.** Let $\mathcal{F}$ be the set of features. We define feature list as a tuple (F, w), where $F \subseteq \mathcal{F}$ is a set of features the list contains, and $w : \mathcal{F} \to \mathbb{N}^+$ is a function returning the count of a feature in a feature list. We shall refer to the set of all feature lists as $\mathcal{L}$ and introduce the following notation for a feature list $x \in \mathcal{L}$, that F_x is the set of features for the given feature list and w_x is the weighing function for the feature list.

Definition 1.2 (database)**.** Let $\mathcal{A}$ be the set of actions. Our problem can be represented as operations on an abstract set D, which is an ordered collection of elements from $\mathcal{L} \times \mathcal{A}$. Each element of D is a tuple of a feature list which represents a state and action taken in that state. Conceptually a database

is a function $D : \mathbb{N} \to (\mathcal{L} \times \mathcal{A})$, which takes a position i and returns the i-th entry in the database. We shall refer to the type of the database as $\mathcal{D}$. Practically, a database is implemented as a more complex datastructure which enables fast kNN queries.

Capability requirements

We will require an $INSERT(D, fl, a) : \big(\mathcal{D} \times \mathcal{L} \times \mathcal{A}\big) \to \mathcal{D}$ operation, for a database D, feature list fl and action a, which will return D', which is D with (fl, a) inserted at its end.

Then we will require a $REWIND(D, n) : \big(\mathcal{D} \times \mathbb{N}\big) \to \mathcal{D}$ operation, which will return D', which is D with n elements removed from its end, undoing the last n insertions.

And finally we need a $SEARCH(D, q) : \big(\mathcal{D} \times \mathcal{L}\big) \to \mathcal{P}(\mathbb{R} \times \mathcal{A})$ (the powerset of $\mathbb{R} \times \mathcal{A}$) operation, for a database D and a query feature list q. We want to find a subset of neighbors $N \subseteq D$, such that for $(fl, a) \in N$ and a given similarity function SIM, the value $SIM(q, fl)$ is as large as possible (find the most similar neighbors), then create a set of possible actions to take $S = \{\big(SIM(q, fl), a\big) \mid (fl, a) \in N\} \subseteq \mathbb{R} \times A$, and return S sorted in descending order according to the computed similarity, as we care most about the best scoring actions.

In addition to providing these operations we will also require that the implementation will be fast, as the quadratic complexity of naive implementation, which has to compare against each entry in the database, is too high for the use case in machine assisted theorem proving.

1.3 Implementation and experiment notes

Implementations as well as the dataset we used can be found on the attached CD. We've implemented algorithms needed for experiments in OCaml, compiled with OCaml 4.07.1 on a 64-bit distribution of Ubuntu 18.04. A guide how to run our experiments is present in a readme file on the attached CD.

All our experiments have been run on a 3.8GHz AMD Ryzen 5 2600 system with 16GB of RAM.

Chapter 2

Dataset analysis

2.1 Technical details

Our dataset is generated from the Coq Standard library [13], which is a collection of basic math theorems. Each selected file from the Standard Library is encoded as an ordered list of tuples of feature list and action (fl, a), with features and actions encoded as integers. The feature lists are explicit representations compared to how we have defined feature lists $\mathcal{L}$ in section 1.2. Instead of the list containing tuples of feature and its count c, it contains the feature repeated c times.

These ordered lists of tuples are then saved into text files, having the tuples (fl, a), represented as '[comma separated list of integers] : integer' on separate lines.

Definition 2.1 (dependencies)**.** Each file can depend on a number of different files. The first line of each file contains a list of files it depends on. In the context of the Coq Standard Library the dependencies represent previously proven lemmas used in a given file. Each dependency can have its own dependencies, but fortunately the dependency graph is a directed acyclic graph and we cannot encounter circular dependencies.

Dependencies for each file are loaded and inserted into the database of the nearest neighbor search prior to starting evaluation of a given file.

Evaluation of the dataset is done by going through each file sequentially, and for each line of (fl, a), we first try to predict a using fl, and save at which index a was predicted in the list of predicted neighbors (if it was present), and then add (fl, a) to the database. After finishing, we create accuracy graphs

based on the distribution of the indexes at which the actions to predict were found.

Using this kind of evaluation, we have found that almost all our algorithms have the same accuracy at the first predictions, starting at around 20%, which was suspicious. After investigating, we have found that our dataset is "previous"-sensitive, in the sense that around 20% of the time, the action to predict is found on the previous line in the input file. This is caused by the lemmas contained in the Coq standard library often having a repeating sequence of the same action.

Because of this we have modified the dataset to include so called flushing, and modified evaluation to not insert new lines into the database immediately after they are encountered.

Definition 2.2 (flushing)**.** The concept of flushing is introduced to separate different lemmas in the source files. Lines containing '#flush' are present in the files at appropriate places. During evaluation new lines should not be added to the database until a flush is encountered, at which moment all lines since the previous flush are added to the database. This is more natural for the way the k-NN search would be used, as usually whole sequences of lines constituting a lemma are inserted into the database at once.

2.2 Statistics

	Total	Mean	Median	StdDev	Min	Max
File count	409	—	—	—	—	—
File length	129027	315,47	81	597,99	0	5041
# of dep files	—	93,5	80	62,58	0	251
# of dep lines	7909079	19337,6	14398	17611,14	0	86296
Distinct actions	27912	—	—	—	—	—
Action frequency	—	4,62	1	82,11	1	7775
Imported act. fq	—	278,82	22	5329,12	1	470415
Feat. list length	—	89,32	49	103,31	1	874

Table 2.1: Dataset statistics

Table 2.1 aggregates interesting statistics about the dataset. Taking only file lengths into consideration, it would seem that we need to create a k-NN search tuned for small databases, but when we look at the number of dependency lines each file imports we see that the mean database size jumps by orders of magnitude.

From action frequencies it would seem that we usually would not be able to predict an action as because median frequency is 1, then at least half of all

actions are unique. But when taking dependencies for each file into account then we see that action frequency increases dramatically.

From the 129027 lines to be evaluated, for 98272 the action to predict for them already exists in the database when not using flushing, which gives us the theoretical best accuracy of $76,164\%$. When using flushing, then the action to predict is present only for 90367 lines, giving us maximum theoretical accuracy of $70,037\%$.

Statistics on feature list lengths give us a general idea on the cardinality of feature lists and will be useful for roughly choosing dimensionality reduction parameters in later chapters.

Chapter 3

Similarity

To be able to search for nearest neighbors we need a way to measure how similar two objects are. This is usually done using similarity measures (functions), which are functions $\mathcal{S} : X \times X \to U$, where X is the space of the objects on which we want to measure similarity and U being the output space of the similarity function, with typically $U = \mathbb{R}$. Similarity functions can usually be thought of as an inverse of distance functions, and many similarity functions reflect that by using the output of their related distance function when calculating similarity. We will explore several different similarity measures which could be applicable to our search space.

3.1 Similarity measures

We want to explore similarity measures for our high dimensional space. For a vector in this space each dimension corresponds to one possible feature, and the value in a given dimension represents the count of that feature.

Feature lists are sparse representations of these vectors. As we have defined in definition 1.1, each feature list is a tuple of (F, w), where $F \in \mathcal{F}$ is the set of features it contains, and w being the function returning counts of a feature in the feature list. When working with feature lists, we can choose to only use the set of features F it contains and discard their weights. We shall refer to doing this as unweighted feature lists. When making use of a feature lists weighing function w we will be referring to this as using weighted feature lists.

Treating all feature lists as unweighted allows us to treat them as sets, which can simplify the calculation of similarity.

For a weighted feature list x, we shall use the notation introduced in definition 1.1, with F_x referring to the set of features of the given feature list, and w_x referring to its weighing function.

Euclid-distance based similarity

One of the most commonly used distance measures is Euclidean distance, with very similar objects having distance approaching zero and becoming more dissimilar as the distance increases. For n-dimensional space it is defined as

$$d^{vector}_{\text{Euclid}}(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{3.1}$$

Calculating the Euclidean distance of unweighted feature lists (sets) a, b be expressed as the size of their symmetric difference.

$$d^{set}_{\text{Euclid}}(a,b) = \sqrt{|a \triangle b|} = \sqrt{|a \cup b| - |a \cap b|} \tag{3.2}$$

The distance of weighted feature lists is calculated in the following way.

$$d^{multiset}_{\text{Euclid}}(a,b) = \sqrt{\sum_{f \in F_a \cup F_b} \big(w_a(f) - w_b(f)\big)^2} \tag{3.3}$$

Note that when this formula is used on unweighted feature lists, the result of $d^{multiset}_{\text{Euclid}}$ equals the result of d^{set}_{Euclid}

Now when we know how to calculate the distance, we need to calculate their similarity. As similarity and distance are in a way an inverse of each other, a rudimentary way to create a distance-based similarity is to use the following formula

$$SIM_{\text{Euclid}}(a,b) = \frac{1}{1 + d(a,b)} \tag{3.4}$$

and just substitute the appropriate distance function for d.

Cosine similarity

The cosine similarity is another widely used similarity measure. Given two vectors it measures the cosine of the angle between them. In n dimensional space it is defined in the following way.

$$SIM^{vector}_{\text{Cosine}}(x,y) = \frac{x \cdot y}{\|x\|\|y\|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}} \tag{3.5}$$

For unweighted feature lists, the similarity can be simplified. The dot product of two vectors in the original vector space, but having only one or zero in each dimension corresponds to the count of positions in which both vectors are one. In set operations this corresponds to taking the size of intersection of two unweighted feature lists. The magnitude of a vector in the original vector space corresponds to the square root of the count of ones in the vector. Given this, the cosine similarity for sets is expressed as follows.

$$SIM^{set}_{\text{Cosine}}(a,b) = \frac{|a \cap b|}{\sqrt{|a||b|}} \tag{3.6}$$

For weighted feature lists (multisets) the formula 3.5 is just slightly modified to work with weights of features instead of iterating over dimensions.

$$SIM^{multiset}_{\text{Cosine}}(a,b) = \frac{\sum_{f \in F_a \cup F_b} w_a(f) w_b(f)}{\sqrt{\sum_{f \in F_a \cup F_b} w_a(f)^2} \sqrt{\sum_{f \in F_a \cup F_b} w_b(f)^2}} \tag{3.7}$$

Jaccard similarity

A very popular similarity measure in data mining is the Jaccard index [14] originally used to study similarity of two regions by the distribution of their plant species.

$$SIM^{set}_{\text{Jaccard}}(a,b) = \frac{|a \cap b|}{|a \cup b|} \tag{3.8}$$

Defined like this it presents itself as a natural way to compare similarity of sets in general, and can be used unchanged when determining similarity of unweighted feature lists. To not lose the additional information held by weighted feature lists we can use the Růžička similarity [15] (also known as generalized Jaccard similarity). It is defined as follows for non-zero vectors $x, y \in \mathbb{R}^n$.

$$SIM^{vector}_{\text{Jaccard}}(x,y) = \frac{\sum_{i=1}^{n} \min(x_i, y_i)}{\sum_{i=1}^{n} \max(x_i, y_i)} \tag{3.9}$$

Reinterpreting this formula for multisets gives us the similarity

$$SIM^{multiset}_{\text{Jaccard}}(a,b) = \frac{\sum_{f \in F_a \cup F_b} \min(w_a(f), w_b(f))}{\sum_{f \in F_a \cup F_b} \max(w_a(f), w_b(f))}. \tag{3.10}$$

TF-IDF based similarity

In addition to just weighing features by the count of their appearances one can also weigh the feature in relation to a corpus of known data. One way to do it is by using Inverse Document Frequency [16], or IDF for short. We use the IDF scheme as proposed by Kaliszyk and Urban [17], where they define IDF for a feature f in a corpus of feature lists D as

$$IDF(f, D) = log\left(\frac{|D|}{|\{l \in D \mid f \in l\}|}\right). \tag{3.11}$$

This formula assumes that feature f is present in some feature list from L. This gives us a measure of importance of a feature in relation to the whole database of known data. This is typically combined with a feature weight for the feature list it originates from, in the form of Term Frequency (TF), defined as follows for a feature f from feature list a

$$TF(f, a) = \frac{w_a(f)}{\sum_{l \in F_a} w_a(l)} \tag{3.12}$$

Note that for unweighted feature lists the TF term is constant for all features it contains. We combine these two terms into a TFIDF weigh for a feature f in a feature list a in a document corpus L.

$$TFIDF(f, a, L) = TF(f, a) \cdot IDF(f, L) \tag{3.13}$$

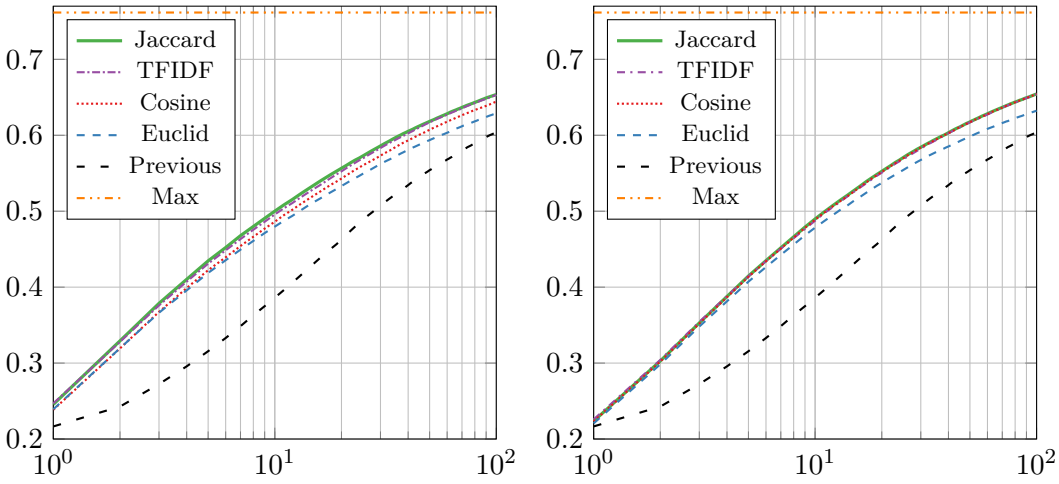
Now we can create a similarity using TFIDF weighing over a corpus of known feature lists L. We will use the generalized Jaccard similarity (eq. 3.10) as the base, but we will weigh the features by their TFIDF scores instead of using raw appearance counts.

$$SIM^L_{TFIDF}(a, b, L) = \frac{\sum_{f \in F_a \cup F_b} \min\left(TFIDF(f, a, L), TFIDF(f, b, L)\right)}{\sum_{f \in F_a \cup F_b} \max\left(TFIDF(f, a, L), TFIDF(f, b, L)\right)} \tag{3.14}$$

3.2 Experiments

We will compare the above mentioned similarity measures using a naive exhaustive search on the dataset, to get the clearest picture of which metric is suitable for our data.

We have to decide whether to use weighted feature lists or if to simplify and reduce weighted feature lists to unweighted ones to treat them as sets, as weighing does bring additional computational costs.



(a) : Weighted feature lists **(b) :** Unweighted feature lists

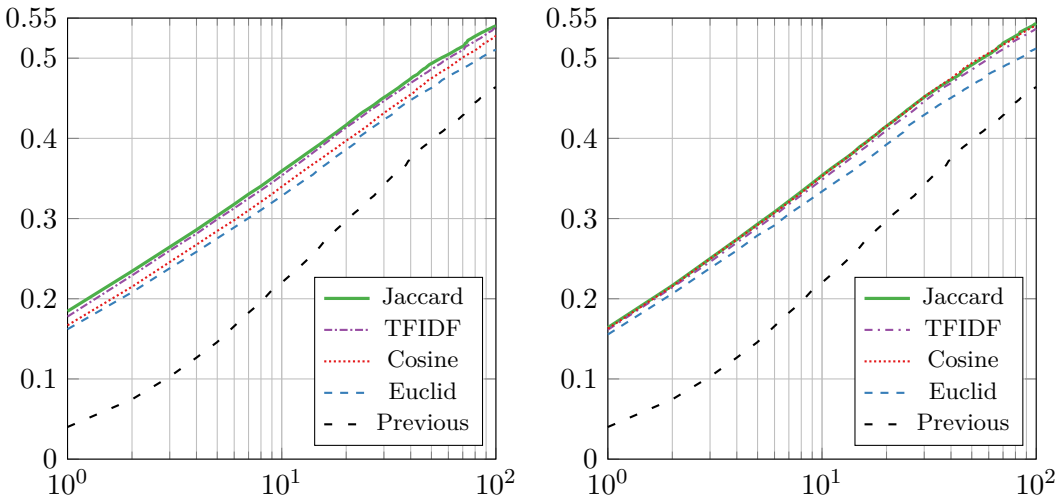
Figure 3.1: Cumulative accuracies of different measures on original dataset

In fig. 3.1 we can see the cumulative accuracies of tested similarity measures, as well as the maximum theoretical accuracy of 76,2% (see section 2.2). The values on the x-axis are the position in the returned list of neighbors, and values on the y-axis showing the percentage of cases when the sought for action was returned at or before that position.

From both fig. 3.1a and fig. 3.1b we see that Jaccard similarity performs best, with TFIDF in a very close second place when working with weighted feature lists, and Jaccard, TFIDF and Cosine performing basically identically for unweighted feature lists. Euclid performs the worst out of these techniques, which was expected as Euclidean distance stops being a meaningful measure in high dimensional spaces as explored by Aggarwal [18].

Of important note is the fifth included line, dubbed Previous in fig. 3.1, which shows the prediction accuracy when we returned the whole database in reverse insertion order as the list of nearest neighbors, e.g. the front of the returned list was the last inserted entry and so on.

As we have mentioned is section 2.2, this is caused by the lemmas often containing a sequence of the same action repeated, and is also the reason why almost the metrics perform almost identically at the first positions. This discovery has led us to the conclusion that we need a better dataset. To achieve this, we have introduced flushing (definition 2.2), to meaningfully separate the different lemmas in the input data. We shall refer to this dataset as the flush dataset.



(a) : Weighted feature lists

(b) : Unweighted feature lists

Figure 3.2: Cumulative accuracies of different measures on flush dataset

In fig. 3.2 we can see the results of different measures on the flush dataset. The maximum theoretical accuracy for this dataset is 70%. Although the overall accuracy drops, we can see that we now have a more interesting dataset as the Previous predictor's accuracy has dropped to around 4%, compared to 22% when not using flushing.

The different measures also separated a bit more, especially at first positions as we can see in table 3.1. The separation is greatest for weighted feature lists, although their accuracy order has remained the same. A surprising discovery is that TFIDF does not improve the accuracy, as would be expected, since it brings additional information when comparing two feature lists, in the form of the IDF term which takes the whole database so far into account.

We can also see that using weighted feature lists does bring a slight improvement in prediction power and will be taken into account in the following chapters when speeding up our k-NN search algorithm.

Jaccard similarity, and especially generalized Jaccard appears to be the best choice of similarity measure for our data, and for this reason we have decided to focus on creating algorithms which quickly approximate these similarities in the following chapters.

	Dataset			
	Non Flushing		Flushing	
	acc(1)	Runtime	acc(1)	Runtime
Jaccard	22,36%	4h07m58s	16,41%	4h06m34s
TFIDF	22,59%	20h36m59s	16,13%	20h22m25s
Cosine	22,31%	4h12m02s	16,27%	4h14m32s
Euclid	22,06%	4h14m54s	15,54%	4h10m10s
Weighted Jaccard	24,56%	10h36m23s	18,44%	10h37m16s
Weighted TFIDF	24,69%	20h43m50s	17,77%	20h27m28s
Weighted Cosine	23,85%	10h21m55s	16,69%	10h39m44s
Weighted Euclid	23,92%	9h56m11s	16,20%	10h19m00s

acc(1) is the accuracy at position 1

Table 3.1: Overview of accuracies at first position and timings of different measures

Chapter 4

Dimensionality reduction

The first step to speed up the k-NN search is to reduce the dimensionality of our data, as its dimension is $|\mathcal{F}|$, where $\mathcal{F}$ is the universal set of features (see section 1.2). One such technique is Minhashing proposed by Broder [19].

4.1 Minhashing

As we have decided in section 3.2, Jaccard similarity is the best similarity measure for our dataset, and we want to compute it faster. Minhashing is a technique which enables both dimensionality reduction as well as efficient computation of approximate Jaccard similarity, as computing set operations can get expensive on large sets.

4.1.1 Description

The basic idea of minhashing is to probabilistically approximate the similarity of two sets. Suppose we have a universal set of features U, a function h mapping elements of set U to distinct integers and a random permutation p of U. For two sets $A, B \subseteq U$, if we permute each set by permutation p, and find the minimum value of h for the permutation of each set, then the probability that the sets produced the same minimum after this process is the same as their Jaccard similarity.

Definition 4.1 (minhash function)**.** Suppose we have a set U, a hashing function $h : U \to \mathbb{Z}$ and a random permutating function $p : U \to U$. Given

this, a minhash function $h^p_{min} : \mathcal{P}(U) \to \mathbb{Z}$ for a subset $S \subseteq U$ is defined as

$$h^p_{min}(S) = \min\{h(p(s)) \mid s \in S\}$$

If we can map all elements of our set U to distinct integers, then the probability of a collision of two subsets of U under a minhashing function is exactly equal to Jaccard similarity of the subsets.

Theorem 4.2. *Assuming h is injective then*

$$\forall A, B \subseteq U\colon \mathbb{P}[h^p_{min}(A) = h^p_{min}(B)] = SIM^{set}_{Jaccard}(A, B)$$

Proof. Given a set U with cardinality n, each subset $S \subseteq U$ can be represented as a vector $s \in \{0, 1\}^n$, with each dimension corresponding to an element from U, and having 1 if the subset contains the given element and 0 if it does not.

Without loss of generality we can assume that h for each $u \in U$ gives us the index of the dimension that represents element u in this vector representation. Now for $S \subseteq U$, let s^p be the vector representation of $p(S)$. Then $h^p_{min}(S)$ is the smallest index i of s^p such that $s^p_i = 1$.

Given $A, B \subseteq U$ and vector representations of their permutations a^p, b^p, the tuples (a^p_i, b^p_i) can have either both elements 0, only one element 1 or both being 1. Let x be the number of tuples $(1, 1)$, and y be the number of tuples with 1 in only one element. We can see that $x = |A \cap B|$ and $(x + y) = |A \cup B|$. Given this we can see that $SIM^{set}_{Jaccard}(A, B) = \frac{x}{x+y}$.

To determine the probability that $h^p_{min}(A) = h^p_{min}(B)$, we need to go through the tuples (a^p_i, b^p_i) sequentially. If we first encounter one of tuples $(1, 0)$ or $(0, 1)$ at position i, then the set with value 1 gets minhash value i. For the other set we have so far only encountered 0 in its vector up to and including position i. This means that it has a 1 at a greater position than i, and therefore the minhash value will be greater than i. Therefore if this kind of tuple is encountered first, then $h^p_{min}(A) \neq h^p_{min}(B)$.

The probability of encountering a tuple $(1, 1)$ before either tuple $(1, 0)$ or $(0, 1)$ is $\frac{x}{x+y}$. But if the first tuple encountered during sequential walk except for $(0, 0)$ is $(1, 1)$, then $h^p_{min}(A) = h^p_{min}(B)$. From this we can see that

$$P[h^p_{min}(A) = h^p_{min}(B)] = \frac{x}{x + y} = SIM^{set}_{Jaccard}(A, B).$$

□

The random variable J which is one when $h^p_{min}(A) = h^p_{min}(B)$ and zero otherwise is an estimator of Jaccard similarity of A and B, although not a terribly useful one, as it only takes on one of two values. To approximate the

similarity with greater accuracy we construct more random variables in the same way, using different permutations and average them.

Definition 4.3 (minhash signature)**.** Given a family of k permutating functions $p_1, p_2, ..., p_k : U \to U$, and a hashing function $h : U \to \mathbb{Z}$, a minhash signature sig of a set $S \subseteq U$ is $sig(S) = \left(h^{p_1}_{min}(S), ..., h^{p_k}_{min}(S)\right)$.

For minhash signatures of two sets, let c be the count of positions in which the signatures have the same value. The value $\frac{c}{k}$ is then an estimate of Jaccard similarity using minhashing.

Definition 4.4 (minhash similarity)**.** Let sig be a function assigning subsets $S \subseteq U$ their minhash signature of length k. Then the minhash similarity is defined as

$$SIM_{minhash}(A, B) = \frac{1}{k} \sum_{i=1}^{k} \left(\begin{cases} 1, & \text{if } sig(A)_i = sig(B)_i \\ 0, & \text{otherwise} \end{cases} \right)$$

The minhash similarity estimates the Jaccard similarity with more accuracy the more k increases.

4.1.2 Computing minhashes

To actually compute minhash signatures on a computer, we need to make a few simplifications. We have so far described minhashing to work on arbitrary sets, which we permute and then pick the minimum value after applying a hashing function mapping the set elements to integers. Computing explicit permutations of sets is an expensive operation, but if we first map our elements using a hashing function $\dot{h} : U \to \mathbb{Z}$ to integers, we can then simulate permutations by using special hashing functions, as suggested by Leskovec, Rajaraman and Ullman [20].

One kind of hashing functions $\tilde{h} : \mathbb{Z}_m \to \mathbb{Z}_c$ which could serve the purpose of simulating permutations is suggested by Carter [21], in the form of

$$\tilde{h}(x) = (ax + b) \mod c \tag{4.1}$$

Where $a, b \in \mathbb{Z}_m$, $a \neq 0$ and c being a prime satisfying $c \geqslant m$.

Let $S \subseteq U$ be the set we want to create a minhash signature of. First, we create a new set $\hat{S} = \{\dot{h}(s) \mid s \in S\}$, which is S mapped to integers. Minhash of such a set is computed in almost the same way as our original minhash, except that $\tilde{h}$ itself is responsible for simulating a permutation, and as the

output of this function is already an integer we do not need to map it again to pick a minimum (as is done in definition 4.1).

$$h_{min}^{\tilde{h}}(S) = \min\{\tilde{h}(s) \mid s \in \hat{S}\} \tag{4.2}$$

Minhash signature is then created using a family of k distinct hashing functions $\tilde{h}_1, ..., \tilde{h}_k$ simulating permutations instead of explicit permutations. We can then use minhash similarity on these signatures. (definition 4.4).

4.1.3 Working with multisets

Minhashing as we have described so far only estimates the Jaccard similarity (eq. 3.8). Haveliwala, Gionis and Indyk [22] describe a way to extend it to estimate the generalized Jaccard similarity (eq. 3.10).

Definition 4.5 (augmented set)**.** Let $M = (S, w)$ be a multiset, where $S \subseteq U$ is the set of elements it contains and $w : U \to \mathbb{N}^+$ is a function returning the count of an element from U in the multiset M. An augmented set is then $S' = \{(s, i) \mid i = 1, ..., w(s) \wedge s \in S\}$.

In this new augmented set, each original element $s \in S$ is quantized into $w(s)$ distinct tuples. We can now use minhashing as defined on sets, but using it on the augmented set $S' \subseteq U \times \mathbb{N}$. When calculating the minhash similarity (definition 4.4), the collision probability for an element $s \in S$ with weight $w(s)$ is appropriately increased by it being quantized into $w(s)$ distinct tuples in S', and therefore the minhash similarity on augmented sets approximates the generalized Jaccard similarity on multisets.

4.2 Experiments

We need to select the minhash signature length k to use, e.g. choose the dimension we reduce to. As mentioned in section 2.2, we will focus on signature lengths around the mean and median feature list lengths in table 2.1. We will focus on these as we should not lose much information on average during the reduction.

The goal is to select a minhash signature length such that the order of predicted actions is almost the same as the order of actions as predicted by using Jaccard. As we are also most interested in the first predictions, with the later ones becoming less important for us the later they appear, we want

to make the effect of the first action predicted by Jaccard being shuffled when using minhashing to have a greater impact than when later actions get shuffled.

The signature length experiments are set up as follows. During evaluation, a search algorithm returns a list of tuples (s, a), where $s \in \mathbb{R}$ is the similarity to query and $a \in A$ is the action. We shall cap the length of these lists at the first 300 entries which had the highest similarity s to a query. We will sort the list by the similarity and then we will discard the similarities from the tuples, so we are left with a list of actions to take for a given line, ordered by similarity to a query. We shall refer to one such a list as j, when produced by an algorithm using generalized Jaccard similarity, and m^l when produced by an algorithm using minhash signatures of length l.

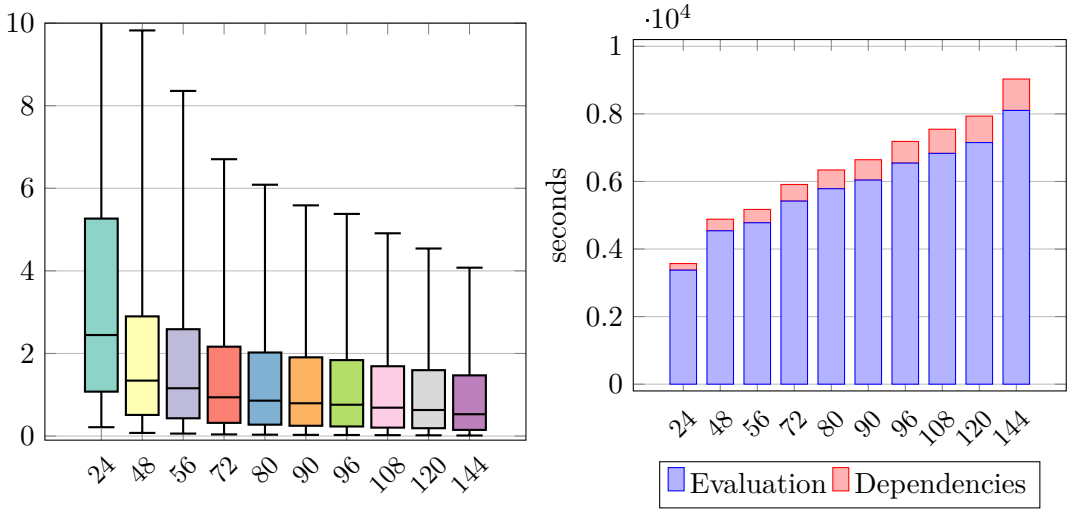
We want to give a score to a signature length by how different the order of returned actions is compared to Jaccard. To create such a scoring function, we first need to define a helping function $pos : A \to \mathbb{N}$

$$pos_x(a) = \begin{cases} \text{position of a in list x,} & \text{if present} \\ 301, & \text{otherwise} \end{cases} \tag{4.3}$$

Using this we will use the following formula for a Jaccard list j, and minhash list m^l to create a score when evaluating one line in an input file.

$$score(j, m^l) = \sum_{i=1}^{300} \frac{|i - pos_{m^l}(j_i)|}{2^{i-1}} \tag{4.4}$$

The nominator is the absolute distance of positions of an action in the two lists, while the denominator is a normalizing term. This is because we care most about the head of the returned list, e.g. the first predictions, and we care exponentially less about how shuffled the actions are at later positions.



(a) : Signature lengths

(b) : Runtime

Figure 4.1: Difference scores of minhash signature lengths to Jaccard

In the boxplot in fig. 4.1a we can see the distribution of scores for a given signature length and in fig. 4.1b we can see the running times for each signature length in seconds, with a split for how long it took to insert all dependencies for all files prior to starting evaluation, and how long the evaluation itself took.

We see that about from signature length of 80, we are starting to see diminishing returns on the accuracy scores, with linearly increasing runtime to the signature length. From this we deem that minhashing signature length above 80 is desirable to sufficiently approximate Jaccard. Higher signature lengths result in a much longer runtime compared to the accuracy we gain, although the runtime of using signature length 144 at 2h30m is still much lower than runtime of generalized Jaccard at 10h36m.

Chapter 5

Approximate k-nearest neighbor search

With minhashing we can reduce the dimensionality of the search space, but we are still left with having to compare all points in a database to find the nearest neighbors. If we can accept approximate answers, we can significantly improve query times by not even considering points with low similarity to our query. We will explore two such techniques.

5.1 Locality Sensitive Hashing

In this section we present a technique proposed by Indyk and Motwani [11], which works by using Locality Sensitive Hash families to group data into buckets based on their similarity and improving query time by only considering data from relevant buckets.

5.1.1 Locality Sensitive Hash families

Definition 5.1 (locality sensitive hash family)**.** A hash family $\mathcal{H} = \{h : X \to U\}$ is called (d_1, d_2, p_1, p_2)-sensitive for input space X and distance function D, if for $\forall x, y \in X, \quad \forall h \in \mathcal{H}$ the following holds true

$$D(x, y) \leqslant d_1 \Rightarrow P[h(x) = h(y)] \geqslant p_1 \tag{5.1}$$

$$D(x, y) > d_2 \Rightarrow P[h(x) = h(y)] \leqslant p_2 \tag{5.2}$$

This means that if we have two points which have smaller distance than d_1, we want the probability of them colliding while using a function from $\mathcal{H}$ to be greater or equal to p_1 and also if they have distance greater than d_2, we want their collision probability to be smaller or equal to p_2.

We want to create a LSH schema to use with minhashing techniques as introduced in the previous chapter. As explored by Andoni [23], this is possible as minhash functions do form a LSH-family.

Lemma 5.2. *A family of minhash functions (see definition 4.1) $\mathcal{H}$ is $(d_1, d_2, 1-d_1, 1-d_2)$-sensitive for Jaccard distance $D_J(a,b) = 1 - SIM^{set}_{Jaccard}(a,b)$*

Proof. Suppose we have two points p, q with Jaccard distance $D_J(p,q) \leqslant d_1$. As per theorem 4.2, the probability

$$P[h(p) = h(q)] = SIM^{set}_{Jaccard}(p,q) \geqslant 1 - d_1$$

for all $h \in \mathcal{H}$. This satisfies condition 5.1. The proof for condition 5.2 is analogous. □

Amplifying a LS-family

We may find that a given family doesn't satisfy our needs because it is not accurate enough. Leskovec [20] describes two ways how a given LSH-family can be used to create a new family with different properties using functions from the original family.

Definition 5.3 (AND-construction)**.** Given a (d_1, d_2, p_1, p_2)-sensitive family $\mathcal{F}$ we can create a $(d_1, d_2, (p_1)^r, (p_2)^r)$-sensitive family $\mathcal{G}$ using an AND-construction for a fixed parameter r. Using a function $t : \mathcal{G} \to \mathcal{F}^r$ we associate each function in $\mathcal{G}$ with a tuple of r functions $(f_1, ..., f_r)$, where each f_i is a distinct function from F. We say that

$$\forall g \in \mathcal{G} : (g(p) = g(q) \iff \forall f \in t(g) : f(p) = f(q))$$

In other words, for items to collide under $g \in \mathcal{G}$, they must collide for all functions f_i from $t(g)$.

Definition 5.4 (OR-construction)**.** Along the vein of the previous definition, given a (d_1, d_2, p_1, p_2)-sensitive family $\mathcal{F}$ we can create a $(d_1, d_2, 1-(1-p_1)^b, 1-(1-p_2)^b)$-sensitive family $\mathcal{G}$ using an OR-construction for a fixed parameter b. Using a function $t : \mathcal{G} \to \mathcal{F}^b$ we associate each function in $\mathcal{G}$ with a tuple of b functions $(f_1, ..., f_b)$, where each f_i is a distinct function from F. We say that

$$\forall g \in \mathcal{G} : (g(p) = g(q) \iff \exists f \in t(g) : f(p) = f(q))$$

For items to collide under $g \in \mathcal{G}$ we only require that the items collide under at least one function $f_i \in t(g)$ instead of requiring they collide under all of them as when using AND-construction.

These constructions can then be combined to create a LSH-family of functions with the desired collision probabilities.

Minhash signatures as LS-families

For a set $S \subseteq U$, its minhash signature $s = (m_1, ..., m_k)$ of length k, where each element m_i is the result of applying the i-th minhashing function $h \in \mathcal{H}$ on set S. We want to create a LSH-family for finding similar signatures. This can be represented by splitting the minhash signature s into b r-tuples, where $br = k$ as illustrated

$$(m_1, ..., m_k) \rightarrow \big((m_1, ..., m_r), (m_{r+1}, ..., m_{2r}), ..., (m_{(b-1)r}, ..., m_{br})\big) \quad (5.3)$$

When searching for similar signatures, we shall only consider signatures which exactly match in at least one r-tuple.

This represents doing a b-OR construction on a r-AND construction on the original (d_1, d_2, p_1, p_2)-sensitive hash family $\mathcal{H}$. After this construction we have a $(d_1, d_2, 1-(1-(p_1)^r)^b, 1-(1-(p_2)^r)^b)$-sensitive family $\mathcal{H}'$, consisting of a single hash function h_{LSH}. When searching for nearest neighbors for a query signature q, we shall only consider signatures p for which $h_{\text{LSH}}(q) = h_{\text{LSH}}(p)$.

5.1.2 Implementation

In addition to keeping a database D of tuples (fl, a) in their insertion order as described in section 1.2 we need to create a data structure to efficiently store the tuples to enable locality sensitive searching on minhash signatures of their feature lists as described above.

Given parameters b and r and a minhash signature s of length k, with $br = k$, it is necessary to create a structure to hold database entries based on the b r-tuples their signatures decompose into. We shall refer to the r-tuples as bands. This can be achieved by having b hashtables $t_i, i = 1, ..., b$, with table t_i mapping a band to a bucket of database entries which have the same r-tuple as their i-th band in their signature.

Inserting the tuple (fl, a) into the i-th hashtable T, with the i-th band B of fl's minhash signature is done like this

Algorithm 1 Inserting into single hashtable

```
1: function InsertToSingleHashtable(T, B, fl, a)
2:     bucketIdx ← Hash(B) ▷ Hash the band with a generic hash function
3:     T[bucketIdx] ← T[bucketIdx] ∪ {(fl, a)}
```

Insertion into all hashtables is done by generating a minhash signature of the given feature list as described in section 4.1.2, and inserting the entry into i-th hashtable using i-th band in the signature as follows.

Algorithm 2 Insertion into hashtables

```
1: function InsertToHashtables(tables, fl, a)
2:     augmentedSet ← create an augmented set from fl (see def. 4.5)
3:     signature ← generate the minhash signature of augmentedSet
4:     bands ← split signature into b bands (see eq. 5.3)
5:     for i in 0, ..., (b − 1) do
6:         InsertToSingleHashtable(tables[i], bands[i], fl, a)
```

When rewinding (see section 1.2) in addition to removing entries from the tail of the database we also need to remove them from the hashtables. Removal of a tuple (fl, a) from a single hashtable is straightforward.

Algorithm 3 Removal from a single hashtable

```
1: function RemoveFromSingleHashtable(T, B, fl, a)
2:     bucketIdx ← Hash(B)
3:     T[bucketIdx] ← T[bucketIdx]\{(fl, a)}
```

Then when rewinding the database, we need to remove a given entry from hashtables before removing it from the end of the database.

Algorithm 4 Removal from hashtables while rewinding a database D

```
1: function Rewind(D, tables, n)
2:     repeat n times
3:         (fl, a) ← get the last entry in D
4:         augmentedSet ← create an augmented set from fl
5:         signature ← generate the minhash signature of augmentedSet
6:         bands ← split signature into b bands
7:         for i in 0, ..., (b − 1) do
8:             RemoveFromSingleHashtable(tables[i], bands[i], fl, a)
9:         D ← D with the last entry removed
```

Search for nearest neighbors is a two-step operation. First given a query feature list we need to collect only candidate entries, to avoid having to consider all known entries, and then collecting actions to take from collected candidates based on the similarity of a candidate's feature list to the query feature list.

Given a band B of a minhash signature of the query and the respective hashtable T, finding candidates in a table for a given band is done in the following manner.

Algorithm 5 Collecting candidates from a single hashtable

```
1: function CollectFromHashtable(T, B)
2:     bucketIdx ← Hash(B) ▷ Hash the band with a generic hash function
3:     return T[bucketIdx]
```

Finding all candidates for a given query feature list q is then straightforward

Algorithm 6 Collecting all candidates

```
1: function CollectCandidates(tables, q)
2:     augmentedSet ← create an augmented set from q
3:     signature ← generate the minhash signature of augmentedSet
4:     bands ← split signature into b bands
5:     return ⋃_{i=0}^{b−1} CollectFromHashtable(tables[i], bands[i])
```

As per section 1.2, the final search algorithm will collect all candidate neighbors N, rank their actions by a chosen similarity function SIM of their feature list to a query feature list into S, and return them in descending order.

Algorithm 7 Searching for nearest neighbors

```
1: function Search(tables, q)
2:     N ← CollectCandidates(tables, q)
3:     S ← {(SIM(q, fl), a) | (fl, a) ∈ N}
4:     S ← sort S by the computed similarities in descending order
5:     return S
```

5.2 LSH Forest

LSH performance and especially accuracy is directly tied to the choice of parameters b and r, for amount of bands and their width, respectively. The best choice of parameters is dependent on a particular dataset and there is still ongoing research on how to best choose these parameters. Even after good parameters are found, if the size of the database changes dramatically they need to be retuned. Ideally we want to collect a constant amount of neighbors. But since parameters b and r fix the probabilities that an entry will hash to a bucket. As the database grows, the buckets start to contain too many entries, and the database should be rehashed with different parameters b and r to maintain good performance, since the main goal of LSH is to provide fast query times.

Bawa, Condie and Ganesan [12] present a LSH-based algorithm which alleviates both of the mentioned issues. Firstly, it solves the difficulty of finding good parameters, as it is self-tuning for parameter r. Secondly it is designed to collect a constant amount of neighbors to alleviate the issue of collecting too many neighbors, to maintain good performance.

5.2.1 LSH tries

Fundamentally, LSH algorithm works by having a set of buckets of entries and grouping the entries into buckets to by their r-element labels (band). A bucket contains entries with the same label, and the whole algorithm works by constructing several of these bucket sets.

Having r fixed presents a problem, because it affects accuracy based on how many entries are stored in a database. When a database is small, and we do not have many entries, then buckets are generally almost empty. In this case we would want to be less strict and collect even entries which are possibly more dissimilar. On the other hand, when a database is large, then the amount of entries in one bucket would usually be large, and we would wish to collect fewer entries from a given bucket, to improve performance.

A solution to this problem is, instead of using fixed size labels, to introduce a variable length label. It is possible that some entries would be very similar and generate very long labels, which is why we impose a maximum length $k_{\max}$.

Definition 5.5 (Variable length label)**.** Given a family of hashing functions simulating permutations $\mathcal{H}$ and maximum label length $k_{\max}$, we pick a sequence of $k_{\max}$ functions $\langle h^{\tilde{h}_1}_{min}, h^{\tilde{h}_2}_{min}, ..., h^{\tilde{h}_{k\max}}_{min} \rangle, tildeh_i \in \mathcal{H}$. A label of length $x, x \leqslant k_{\max}$ for a set $S \subseteq U$ is then $l(S, x) = \left(h^{\tilde{h}_1}_{min}(S), h^{\tilde{h}_2}_{min}(S), ..., h^{\tilde{h}_x}_{min}(S)\right)$.

Definition 5.6 (LSH Trie)**.** LSH trie is a prefix trie constructed over the set of all labels, with its leaves storing database entries. A trie can either be empty, an internal node, or a leaf. Only leaves hold entries.

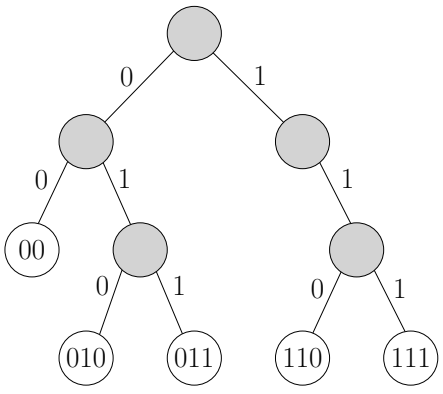


Figure 5.1: An example trie using binary prefixes.

In fig. 5.1 we can see a trie with five leaves. Assuming that the maximum label length is higher than three, then each leaf contains a single entry. We see that the prefixes describe the path to take from the root to the given leaf.

We then construct a LSH Forest by creating b such LSH tries, each with its own associated sequence of minhashing functions used to generate variable length labels for the given trie.

We will need a function to find a child trie c of a trie t for a feature list fl at depth d. This will be achieved by selecting the child using the d-th element in the variable length label $l(fl, d)$, where l is specific to the trie t.

Algorithm 8 Finding a child of a trie

```
1: function GetChild(t, fl, d)
2:     if t is a node then
3:         L ← d-th element in label l(fl, d)        ▷ l is specific to each trie
4:         C ← child of t for prefix element L at depth d
5:         return C
6:     else                                          ▷ Is a leaf or empty trie
7:         return empty trie                         ▷ child of a leaf is undefined
```

Insertion into a forest is done on each tree independently. Insertion of an entry of a feature list and an action (fl, a) into a trie works by descending the trie according to the variable length label generated for fl by the given trie's label function l. If an entry (fl', a') with the same prefix is encountered at depth less than $k_{\max}$, then each entry's prefix is lengthened until they occupy distinct leaves, or a maximum label length $k_{\max}$ is reached, at which point the entries get stored in a list.

Algorithm 9 Inserting into a trie

```
1: function InsertToTrie(t, fl, a, d)                 ▷ Initial d = 1
   ▷ returns a new trie with (fl, a) inserted
2:     if t is empty then
3:         return leaf containing (fl, a)
4:     else if t is node then
5:         child ← GetChild(t, fl, d)
6:         newChild ← InsertToTrie(child, fl, a, (d + 1))
7:         return t with newChild in place of child
8:     else if t is leaf then
9:         if d > k_max then
10:            t ← append (fl, a) to elements in leaf t
11:            return t
12:        else
13:            (fl', a') ← the (only) entry stored in leaf t
14:            temp ← InsertToTrie((empty node), fl', a', d)
15:            return InsertToTrie(temp, fl, a, d)
```

Deletion from a forest is also done independently on each trie. Deleting an entry (fl, a) from a trie works by descending the trie to the leaf where the entry (fl, a) is stored and removing it from the leaf.

When traversing back up to the root, we contract the internal nodes if the removal caused other entries to have unnecessarily long prefixes (as we only use prefixes as short as possible for the stored entries to occupy distinct leaves, or a maximum depth is reached). An internal node n can be deleted if it has only one child c that is a leaf containing a single entry. If this condition is met, then n is "deleted" by replacing it with the child c. We repeat this procedure until this condition is not true.

Algorithm 10 Deleting from a trie

```
1:  function DeleteFromTrie(t, fl, a, d)                    ▷ Initial d = 1
    ▷ returns a new trie with (fl, a) removed
2:      if t is empty then
3:          return t
4:      else if t is node then
5:          child ← GetChild(t, fl, d)
6:          newChild ← DeleteFromTrie(child, fl, a, (d + 1))
7:          t ← t with newChild in place of child
8:          if t has an only child c which is a leaf with a single entry then
9:              return c
10:         else
11:             return t
12:     else if t is leaf then
13:         t ← remove (fl, a) from the leaf t
14:         return t
```

Finding neighbors for a given query feature list q on a LSH Forest is done by descending each trie to a leaf using the prefix that is generated for q for a given trie. Once all tries have been descended, we start a synchronous ascend across all tries at the same levels. Suppose the maximum depth at which a prefix match was found is d_m. We start a synchronous collection of entries across all tries from level d_m moving up to root.

We stop collecting neighbors N after $|N| > K$, where K is a chosen constant. If we did not stop after a certain number, we would collect up to the root of all tries and we would have collected all entries in the database. Note that N includes duplicate entries, because we are collecting across multiple tries.

Algorithm 11 Collecting neighbors from LSH Forest

```
1: function CollectFromForest(tries, q, d)                    ▷ Initial d = 1
2:     if all tries are leaves then
3:         N ← collect all entries stored in tries
4:         return N
5:     else
6:         childTries ← {GetChild(t, q, d) | t is not leaf, t ∈ tries}
7:         N ← CollectFromForest(childTries, q, (d + 1))
8:         if |N| > K then
9:             return N
10:        else
11:            N ← collect all entries from all trees in tries
12:            return N
```

The complete search algorithm is almost the same as algorithm 7, except of having *tables* as the input parameter it has *tries* which is the LSH Forest we will search in, and on line 2 we create a set of neighbors N from the return value of CollectFromForest(*tries*, q, 1).

5.3 Experiments

Both the algorithms in this section work in two phases, first by collecting a set of neighbors N and then by ranking how similar are these found neighbors to a query, with the constraint that the amount of found neighbors $|N| \ll |\mathcal{D}|$, where $\mathcal{D}$ is the whole database.

Both phases employ minhash signatures, finding neighbors employs them in both LSH techniques, and we will be using minhash similarity (definition 4.4) for ranking the found neighbors. It is possible to use the same signature for both phases, but if we do that, then we introduce a new relation between the parameters. Both LSH techniques have parameters b and r, LSH for the amount of bands and width of one band respectively, and LSH forest for the amount of tries and maximum depth of a trie respectively. With a minhash signature length k, there is now a relation $br = k$ (as per eq. 5.3), which dictates the accuracy of the similarity ranking phase of the algorithm.

This is a problem when br results in a short signature length, as when br results in longer signatures it only brings us greater accuracy using minhash similarity. Calculating minhash similarity is an inexpensive operation when minhash signatures are known beforehand. We will use signatures of length br when they result in longer signatures for calculating similarity, as the signatures would be calculated anyway for the finding neighbors phase of the search. But when br drops below a certain threshold, we will use a longer minhash signature for calculating similarity.

We set this threshold to 90, as per the signature length experiments performed in section 4.2. When br drops below this number, we shall compute minhash signatures of length 90. From this signature the first br elements will be used for the finding neighbors phase, and the full signature will be used for ranking similarity.

We introduce another change to minhash signatures for the phase of finding neighbors in the LSH techniques. we will use a hashing function $\hat{h} : \mathbb{Z} \to \{0, 1\}$ and use it to create a new signature from the original, by mapping each element of the original signature to one bit. For LSH forest this simplifies the implementation to use binary prefix tries (fig. 5.1), with a maximum depth of r. We shall do the same for LSH, to be able to easily compare it to LSH Forest. For LSH the reduction means that a signature will have an r-bit bucket label for each of the b bands. This additionally enables us to have finer control of how many buckets there are in each band. Using this bit reduction there are 2^r distinct buckets in one band.

Additionally for LSH forest, for each entry to be inserted, its variable length minhash label for each of its b tries is calculated to the maximum trie depth r straight away as the first step to simplify implementation. This means we are computing minhash signatures of at least length br for each insertion and query.

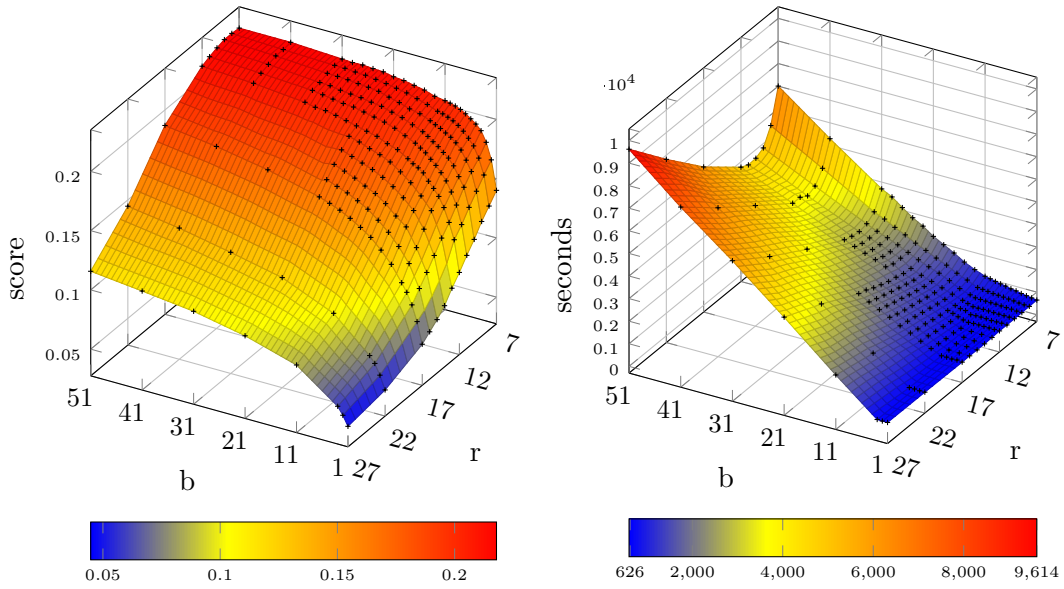
Definition 5.7 (accuracy score)**.** To quickly judge quality of particular parameters we define the following scoring function which will give a single score to a particular run based on the accuracy it achieves. Given an accuracy density function $acc(i) : \mathbb{N} \to [0, 1]$ giving the portion of correct predictions which were at position i in the returned ordered lists of neighbors for a particular run of an algorithm with some parameters. We define the accuracy score in the following way

$$score(acc) = \sum_{i=1}^{\infty} \frac{acc(i)}{2^{i-1}}$$

5.3.1 LSH parameter optimization

We will try to find the best values for parameters b and r for our use case, with respect to their prediction accuracy and also the running time. we will do this by doing a grid-like search. For each amount of bands b $\{1, 11, 21, 31, 41, 51\}$ we will try different widths of bands r $\{7, 12, 17, 22, 27\}$ and score each of the 30 runs using the score from definition 5.7. Given this initial grid search we will get the general shape of the relation of the parameters to overall accuracy, and we will run more experiments to get finer results for particular areas.

The reason we are not starting at lower widths of band than 7 is because



(a) : Scores **(b) :** Runtimes

Figure 5.2: LSH parameter experiments

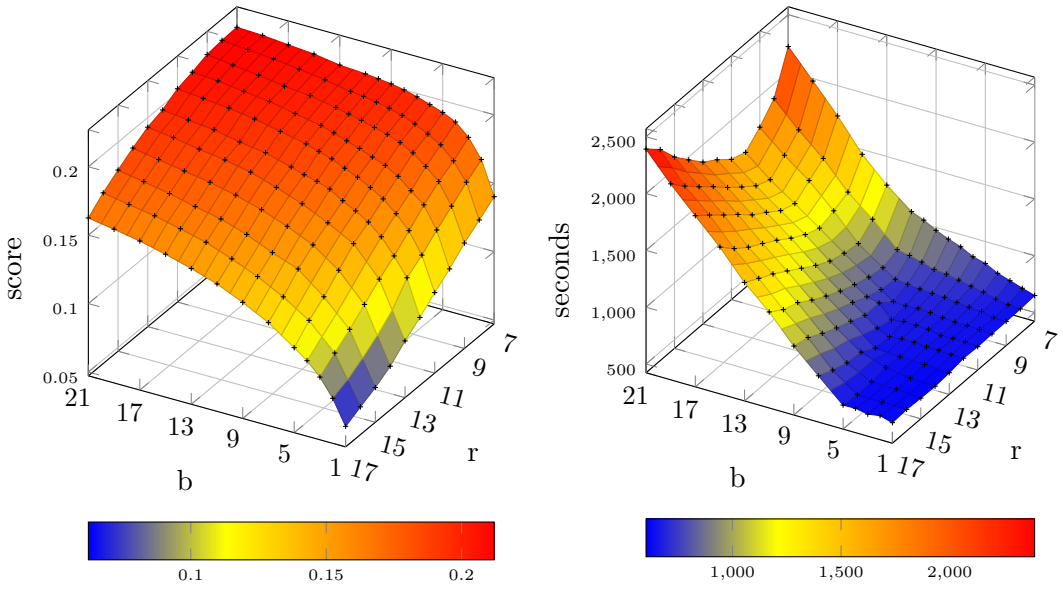
the choice of width directly affects the amount of buckets in one band, which is 2^r. Having narrower bands results in collecting more and more potential neighbors, as the whole database of neighbors is not segmented enough, and performance degrades. By choosing a minimum width of 7 we are choosing to have at least 128 distinct buckets in one band, which we deem a reasonable minimum.

In fig. 5.2 we can see all the accuracy scores for the parameters. Not all possible combinations of parameters have been run. Those that were are noted by black marks in the figure, other values are linearly interpolated.

As presumed, lower values of r (narrower bands) result in higher accuracy, but result in increased runtime by collecting too many neighbors, as can be seen in fig. 5.2b. We also see that having more bands does increase the accuracy, though it starts to plateau as it approaches some maximum accuracy.

Unfortunately, although the accuracy does seem to be convex for different parameters b and r, a maximum is not present in our explored range of parameters. It appears that a maximum will be reached by using a very large number of narrow bands, but such parameters will have a very long runtime, and therefore are irrelevant for us.

We would like to bring attention to fig. 5.2b and talk about the observed runtimes. When looking at the band width r and time plane, we can see that when r gets below 12, the runtime seems to increase quadratically, which can



(a) : Scores **(b) :** Runtimes

Figure 5.3: LSH parameter experiments detail

is explained as we have discussed that we are collecting a lot of neighbors and doing an almost exhaustive search on the dataset. Since exhaustive NN search has quadratic complexity to the size of a database, the runtime grows quadratically the more neighbors we collect even when using LSH.

On the time/width plane in fig. 5.2b with increasing r above 12, as well as on the band count b/time plane, the runtime increases linearly with increasing values for r and b on their respective axes. This is due to the fact that computing minhash signatures is linear to the length of the signature, and our signature lengths are br, except when the signature would be shorter than 90, we compute it to be that long. That is also the reason why the runtimes are almost flat for low parameters b and r.

We will need to pick some parameters which will provide us with good accuracy while maintaining a reasonable runtime. In fig. 5.3 we explore the range of parameters for r between 7 and 17, and b between 1 and 21 in greater detail.

In fig. 5.3b we can clearly see the effect of setting minimum minhashing length for computing similarity to 90, as almost all runtimes for parameters b and r where $br \leqslant 90$ is almost the same. From this we deem that computing minhash signatures is the most expensive operation, and the runtime starts to grow as longer signatures are needed for parameters b and r which result in greater product br.

Values of r from 9 to 12 with b below 7 result in the shortest runtimes,

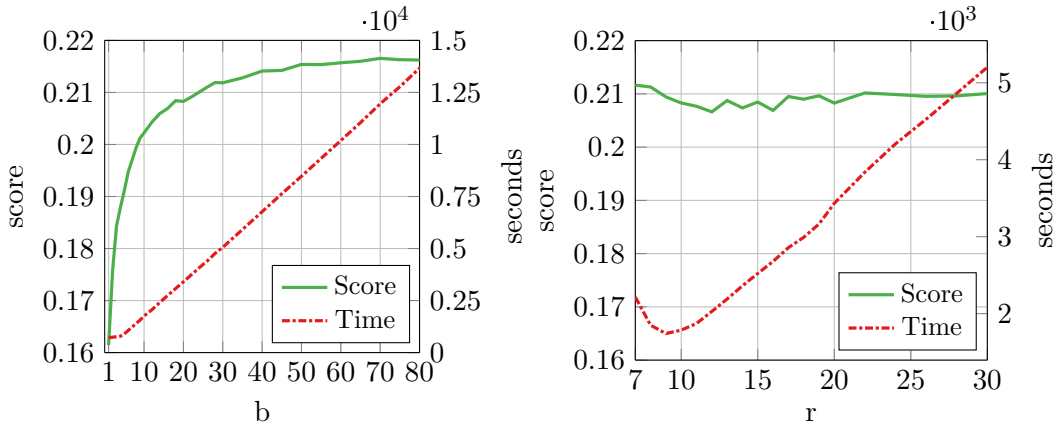
but with lower accuracy scores than we would like. Judging from the graphs it seems that the sweet spot for the tradeoff between accuracy would lie for parameters r between 9 to 11 and b between 9 to 15.

5.3.2 LSH Forest parameter optimization

LSH Forest also employs parameters b and r for the amount of tries and maximum trie depth respectively. It also has parameter K, which is the minimum amount of neighbors it collects before it stops, unless the database does not contain at least K entries. Unlike LSH, we make an argument that parameters b and r are independent, and we do not need to do a grid search similarly as we did for LSH to find good parameters.

As a trie will grow only as deep as it needs to be in order to make sure that all entries are in unique leaves, or a maximum depth has been reached. If the maximum depth r is set sufficiently deep, then an algorithm's accuracy score will almost solely depend on the amount of tries b. We explore this in fig. 5.4a, where we try different amount of tries with a fixed maximum depth $r = 20$. We chose 20, as a binary trie can have up to 2^r leaves. As the maximum amount of lines that will be inserted into the database is 91337 (the sum of maximum file length and maximum amount of dependency lines from table 2.1) is much smaller than 2^{20}, we deem it as a safe choice of maximum depth r to not affect testing of different trie counts b.

Experiments for maximum trie depth r are in fig. 5.4b. As can be seen, we correctly assumed that choosing different maximum depths will have a negligible effect on the overall accuracy of the algorithm, except for low values of r, where the same problem emerges as in LSH. Low maximum depths cause the tries to be shallow, and unable to hold enough distinct leaves, so the search space doesn't get segmented enough and the algorithm collects too many neighbors, potentially even all neighbors in the database. As the depth grows larger it first suffers a slight drop in accuracy, as the database gets segmented enough, but the shallow depth means that most leaves will contain multiple entries. As the depth grows and reaching maximum depth becomes infrequent the accuracy stabilizes, as unlike LSH, the Forest does collect even inexact matches until it has met its desired amount of neighbors.



(a) : Different trie counts for max depth 20 **(b) :** Different max trie depths for trie count 20

Figure 5.4: Scores of different LSH forest parameters

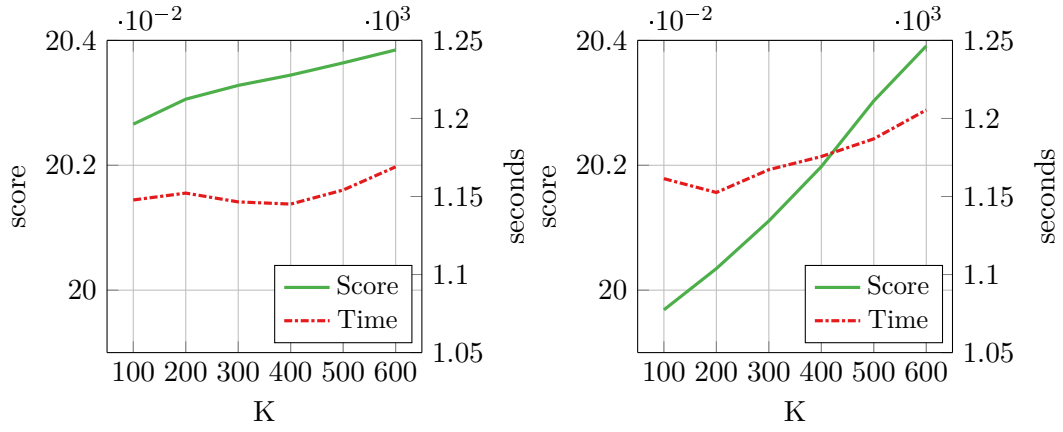
The times we observe in fig. 5.4 suggest that the runtimes for different trie counts and depths b and r will behave similarly as we have seen in fig. 5.2b, with the runtimes being linear to trie count in fig. 5.4a, because as we are computing full minhash signatures for each trie, even if not needed, we have them of length br, and as we have already said, the cost of computing a minhash signature is linear to its length. The time is constant for low trie counts b, because we are always computed signatures of length at least 90.

The same relation applies in fig. 5.4b for higher maximum trie depths r. Only when r becomes small we see a quadratic increase in runtime, which is due to the same reasons as we have alluded to when discussing fig. 5.2b.

We need to choose the amount of tries b which has good accuracy with a relatively fast runtime. In fig. 5.4a, at $b = 14$ we are already within a hundredth of maximum observed accuracy score, with a decent running time. As we have seen, the choice of maximum depth is almost inconsequential from an accuracy perspective, and we can therefore choose based on minimum running time, which is achieved for maximum depth r around 9 and 10.

With parameters b and r selected we are left with the choice of how many neighbors we want to collect K. In fig. 5.5 we are experimentally trying different Ks for fixed $b = 14$ and $r = 9$ or $r = 10$. As we see, the amount of neighbors does not affect the accuracy much.

In fig. 5.5a, the choice of K for depth $r = 9$ has almost no effect on accuracy. This is because for tries this shallow, a lot of entries reach maximum depth before they are able to occupy distinct leaves and are therefore inserted into the same leaf at maximum depth. When collecting from leaves containing a high number of entries leads to high number of collected neighbors, and as LSHF checks the amount of how much it has collected after it has done so for



(a) : Maximum trie depth 9

(b) : Maximum trie depth 10

Figure 5.5: Scores of different LSHF neighbor counts K

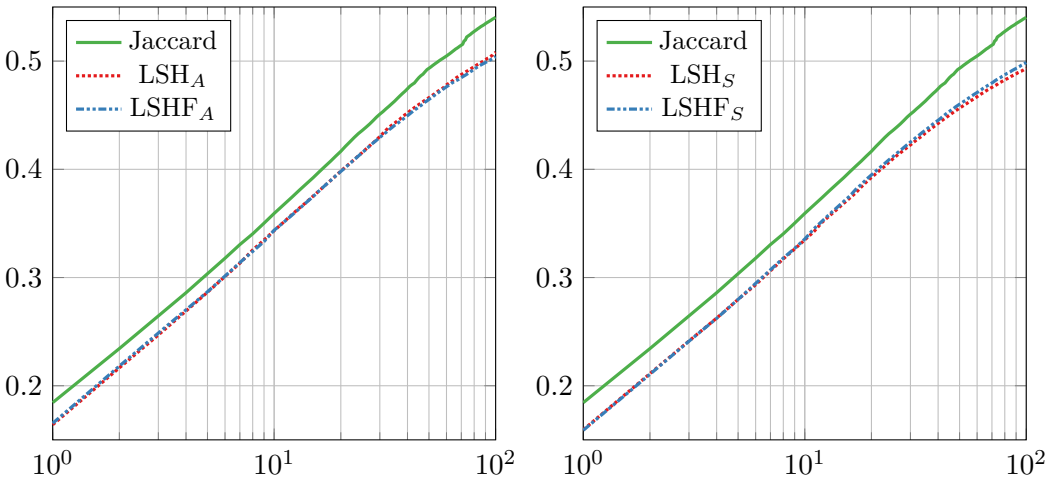
all tries at a given level, then the parameter K is exceeded almost every time.

But as we see in fig. 5.5b, at maximum depth $r = 10$, although the absolute difference in scores is still quite small, we are already seeing a more pronounced effect of different choices of K, as it doesn't get exceeded as often in normal operation. Thus, we conclude that for low settings of r, the choice of K does not matter much, but as the tries get deeper, we should choose higher Ks to collect an adequate amount of neighbors.

5.3.3 Comparisons of LSH, LSHF and Jaccard

We have selected two sets of parameters for LSH and LSH Forest with similar running times with one focusing on speed and the other on accuracy.

- Speed focus
  - LSH_S - LSH with $b = 11$, $r = 9$
  - LSHF_S - LSH Forest with $b = 10$, $r = 9$, $K = 300$
- Accuracy focus
  - LSH_A - LSH with $b = 15$, $r = 9$
  - LSHF_A - LSH Forest with $b = 14$, $r = 10$, $K = 600$



(a) : Accuracy focus **(b) :** Speed focus

Figure 5.6: Comparison of LSH and LSH Forest to generalized Jaccard

As we can see in fig. 5.6 and timing in table 5.1, a well-tuned LSH performs almost identically to LSH Forest at similar running times, but with the downside that we had to search quite extensively to find those parameters. LSH Forest is much more robust in terms of parameter choice, as we have shown that the choice of r does not have a large effect on the algorithm's accuracy, unlike LSH where higher values of r cause a decrease in accuracy. We only need to choose the amount of tries b and collect a sufficient amount of neighbors K to have an algorithm with high accuracy.

In table 5.1, column LSH_S, we see that using LSH we have managed to improve the total running time almost 50-fold compared to doing an exhaustive search using generalized Jaccard similarity. Even more impressive is the almost 300-fold reduction in evaluation time, which is the time it takes to search for queries after all of dependencies of a given file have already

	Jaccard	LSHF_A	LSH_A	LSHF_S	LSH_S
acc(1)	18,44%	16,53%	16,39%	15,89%	15,93%
Runtime	10h37m15s	20m5s	19m35s	13m21s	13m19s
Import	2m10s	17m10s	16m3s	11m5s	11m9s
Eval	10h35m5s	2m55s	3m32s	2m16s	2m10s
QpS	3,39	737,3	608,62	955,76	992,52

acc(1) is the accuracy at position 1
Import is time taken for inserting dependencies
Eval is time taken for evaluation itself
QpS is the average # of Queries per Second during evaluation

Table 5.1: Technique comparison

been inserted into the database. This has been achieved by moving the bulk of time from the actual search phase into the creation of the datastructure enabling us to have such fast query time, and being able to perform almost a 1000 queries per second, instead of slightly over 3 queries per second, which we were able to achieve by exhaustive search.

As we can see in column LSHF_A, if we do not mind dropping our speed to over 700 queries per second we can achieve accuracy of our first prediction within 2% of doing an exhaustive search using LSH Forest. This has the cost of increasing the time required to create the datastructure, but we see this as a good tradeoff, as the amount of dependency lines which are imported usually greatly exceeds the amount of lines which are evaluated (table 2.1).

Chapter 6

Conclusion

We set out to find a way to efficiently find approximate nearest neighbors for searching in a high dimensional dataset, specifically the Coq standard library [13] of mathematical theorems. To achieve that we have explored different similarity measures to find which is best for our dataset. We have tried Jaccard and Cosine similarities and also incorporated TFIDF weighing to see if it gives us more accuracy and have concluded that the generalized Jaccard (Růžička) similarity [15] gives the best accuracy for our dataset. We have also found that additional weighing by means of TFIDF did not give us increased accuracy.

We have explored dimensionality reduction for our data in the form of Minhashing[19], which enables fast computation of the approximate Jaccard similarity by computing minhash signatures (definition 4.3) and experimented with signature length to find a balance between accurately approximating Jaccard and run times. We have found that signature length around 90 offers good accuracy with respect to its runtime.

Finally, we have explored approximate nearest neighbor techniques which enable us to search for approximate nearest neighbors without having to investigate all known data. We have chosen hashing based techniques, Locality Sensitive Hashing as proposed by Indyk and Motwani [11] as a base technique and an improvement in the form of LSH Forest proposed by Bawa, Condie and Ganesan [12].

LSH is a technique enabling creation of a fast datastructure providing fast lookup of approximate neighbors by partitioning data into groups containing other similar items. A fine-tuned LSH provides good accuracy with very short running times compared to exhaustive search, but with the downside that without carefully choosing its parameters, which we have had to find experimentally by running over 200 different configurations, the algorithm's

accuracy and/or performance quickly deteriorates, which is made worse by the parameters being highly dependent on a particular dataset.

LSH Forest is an improvement on LSH, by being self-tuning for accuracy and eliminating the need to search for optimal parameters. It achieves this by creating variable length labels for database entries and using these labels to store entries in prefix trees. We have been able to quickly find LSH Forest parameters to achieve the same accuracy and runtime characteristics as a well-tuned LSH. In future work our LSH Forest implementation could be improved. Currently it calculates the mentioned variable length labels to their maximum potential length, even if it is not needed. Improvement would be to compute them piecewise we could reduce the amount of unnecessary computation and improve performance.

In the end, using LSH and LSH Forest we have been able to improve query times almost 300-fold compared to exhaustive search, with minimally decreasing the accuracy of predictions, and therefore consider that we have successfully achieved our goal of finding a fast k-NN search algorithm for high dimensional data.

Appendix A

Bibliography

[1] J. L. Bentley, "Multidimensional binary search trees in database applications," *IEEE Transactions on Software Engineering*, no. 4, pp. 333–340, 1979.

[2] H. Samet, "An overview of quadtrees, octrees, and related hierarchical data structures," in *Theoretical Foundations of Computer Graphics and CAD*, pp. 51–68, Springer, 1988.

[3] A. Guttman, *R-trees: a dynamic index structure for spatial searching*, vol. 14. ACM, 1984.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *Acm Sigmod Record*, vol. 19, pp. 322–331, Acm, 1990.

[5] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *SODA*, vol. 93, pp. 311–21, 1993.

[6] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[7] M. E. Houle and M. Nett, "Rank-based similarity search: Reducing the dimensional dependence," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 1, pp. 136–150, 2015.

[8] S. Arya and D. M. Mount, "Approximate nearest neighbor queries in fixed dimensions.," in *SODA*, vol. 93, pp. 271–280, 1993.

[9] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.

[10] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[11] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 604–613, ACM, 1998.

[12] M. Bawa, T. Condie, and P. Ganesan, "LSH forest: self-tuning indexes for similarity search," in *Proceedings of the 14th international conference on World Wide Web*, pp. 651–660, ACM, 2005.

[13] "The Coq Standard Library — The Coq Proof Assistant." *Available at* `https://coq.inria.fr/library/`, 2019. [Online; accessed 21-April-2019].

[14] P. Jaccard, "Étude comparative de la distribution florale dans une portion des alpes et des jura," *Bull Soc Vaudoise Sci Nat*, vol. 37, pp. 547–579, 1901.

[15] M. Ružička, "Anwendung mathematisch-statistischer methoden in der geobotanik (synthetische bearbeitung von aufnahmen)," *Biologia, Bratislava*, vol. 13, pp. 647–661, 1958.

[16] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[17] J. U. Cezary Kaliszyk, "Stronger automation for flyspeck by feature weighting and strategy evolution," *PxTP@CADE*, pp. 87–95, 2013.

[18] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," in *International conference on database theory*, pp. 420–434, Springer, 2001.

[19] A. Z. Broder, "On the resemblance and containment of documents," in *Compression and complexity of sequences 1997. proceedings*, pp. 21–29, IEEE, 1997.

[20] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2nd ed., 2014.

[21] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of computer and system sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[22] T. Haveliwala, A. Gionis, and P. Indyk, "Scalable techniques for clustering the web," 2000.

[23] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pp. 459–468, IEEE, 2006.

Appendix B

List of attachments

- CD containing dataset and implementations as described in section 1.3