Masterarbeit

Approximate nearest neighbor query methods for large scale structured datasets

Joachim Wolff

24. Mai 2016

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Bearbeitungszeitraum

27. 11. 2015 – 27. 05. 2016

Gutachter

Prof. Dr. Rolf Backofen

Dr. Frank Hutter

Betreuer

Dr. Fabrizio Costa
Milad Miladi

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweiße, bereits für eine andere Prüfung angefertigt wurde.

Freiburg im Breisgau, den 24. Mai 2016

Contents

Zusammenfassung

Die Lösung des Similaritätsproblems der k-nächsten Nachbarn ist mit einem exakten Algorithmus in $O(n^2 * m)$, mit n Instanzen und m Dimensionen gegeben. Die Laufzeit eines solchen Algorithms wird gerade unter dem Aspekt des 'curse of dimensionality' kritisch. Approximative Algorithmen stellen hierbei einen Ausweg dar. In der vorliegenden Masterthesis wird ein Algorithmus implementiert welcher zuerst die Dimensionen der Eingabemenge reduziert und dann in zwei weiteren Schritten Kandidaten für die Nachbarschaft auswählt um mit ihnen die k-nächsten Nachbarn auf den Orginaldaten zu berechnen. Der vorliegende Algorithmus ist spezialisiert auf sehr spärlich besetze und hochdimensionale Datensätze. Des weiteren unterstützt die vorliegende Implementierung moderne Multicore-Prozessoren und bietet die teilweise Berechnung auf der Graphikkarte an. Möglichkeiten zur Reduzierung des Speicherverbrauchs des Algorithmuses werden untersucht und wenn sinnvoll implementiert. Es wird ein Python-Interface welches zu dem von scikit-learns nearest neighbors Implementierung[1] kompatibel ist angeboten.

[1] http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html, accessed: 2016-05-24

Abstract

An exact algorithm to solve the similarity problem of the k-nearest neighbors is given with a runtime of $O(n^2 * m)$ with n instances and m dimensions. Given the curse of dimensionality the runtime of this algorithm is critical, approximate algorithms can be helpful to find a solution. In this master thesis an approximate nearest neighbor search algorithm is implemented. In a first step the dimensions are reduced, followed by two candidate selection rounds on the original dataset to compute the k-nearest neighbors of an instance. The implemented algorithm is specialized to very sparse and very high-dimensional datasets. Methods to reduce the usage of memory are examined and implemented if useful. Multi core-CPU support is provided and parts of the algorithm can run on the graphic card. An interface compatibility to scikit-learns nearest neighbor algorithm[1] python interface is provided.

[1]http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html, accessed: 2016-05-24

1 Introduction

In bioinformatics it is useful to know if molecules, genes, proteins or RNA structures are somehow related, searching here for similarities is used to get a first intuition what could be related and is reducing the search space for a more complex computation. Another usage is to select candidates for a real world test in the laboratory. It is simply a cost factor to test as less as possible in real, a good candidate selection is crucial to save a lot of working time and money. A similarity search is given by a brute force algorithm which runs in $O(n^2 * m)$, with n instances and m dimensions. In bioinformatics it is likely to have very sparse and very high dimensional datasets, there are datasets which are having one million dimensions and 400 to 4000 non-zero features per instance. Working here with dense datasets is today possible, typically the nearest neighbors are computed with a matrix multiplications which leads to a memory usage of $O((n * m))$. The required main memory for example 128 GB costs around 800 to 1000 Euro (May 2016) which is the price of a normal computer and second the computation time is too long. Too much resources are wasted for storing and computing zeros. Storing and computing as a very sparse dataset for the k-nearest neighbor search is very important to overcome these issues, but todays k-nearest neighbor search algorithm are often not specialized to this kind of data structure. To reduce the complexity only an algorithm which is based on approximations is possible, the payoff is to allow a specific error term. In many cases inaccuracies are acceptable because the order of the k-nearest neighbors is not that important, only the fact that these instances are within the k-nearest neighbors matters. Furthermore it is in many cases better to have a solution with a few errors within reasonable time instead of waiting long for the computation.
In this work an approximate approach is used. Given a very sparse and very high dimensional dataset it is likely for an instance to have their neighbors with these instances that share features with it, no matter what the values of this feature are. In the following an algorithm and an implementation specialized on this kind of data structure is developed.

1.1 Task definition: Approximate k-nearest neighbors

Current approximate nearest neighbor search algorithms are often dealing with a lack of support of very high dimensional but very sparse datasets. The benefits of a sparse dataset is that only the non-zero features of every instance need to be stored.

Algorithms like annoy[1] or local sensitive hashing forest[2] accept sparse datasets as an input but they need a lot of memory. Other algorithms like kd-tree or ball-tree do not accept a sparse data matrix as an input.
The higher the dimensions are the more is the runtime of the algorithm depended on the number of dimensions and not as usually assumed by the number of input instances. Bellman [1] was showing the problematic with this issue in his book about dynamic programming in 1957. The solution to this problem is to reduce the dimensions. To reduce them, a random projection can be used and then the reduced dataset is given to the nearest neighbor algorithm. But random projection leads to a decrease in accuracy, see Figure 1.4. Lastly, the nearest neighbor search is a task which can be parallelized perfectly in theory. Every neighbor and every k-nearest neighbor query for an instance can be computed independently. Usually two or four, less common also eight, twelve and sixteen cores are available in a modern processor. In most cases only a single thread is used instead of all threads which are available. Second, if a parallelization is used like in scikit-learns nearest neighbor algorithm it is badly implemented. For example in the most recent version 0.17 of scikit-learn multi core support for the nearest neighbor search was introduced. But on a quad core CPU there is just a little bit more than 20 % speedup between one (1.2 seconds) and four (0.93 seconds) threads instead of the expected speedup of factor 4. Third, general purpose GPU programming (GPGPU) with frameworks like openCL[3] or Nvidia's CUDA[4] with a huge parallelization capability is not used.
This master thesis tries to solve these issues. First, a dimension reduction based on the two hash algorithms 'minimum hash' (MinHash) and 'winner takes it all hash' (WTA-Hash) is implemented. Second, an approximate k-nearest neighbors search based on a candidate selection is computed. The implementation is written in C++ to get optimal speed and parallelization is supported with openMP[5]. Last, for those parts of the algorithms which can be parallelized a GPU support is provided. To reduce the amount of stored memory different methods are evaluated. With the acceptance of less accurate results memory can be saved which makes sense for large datasets which could not be computed otherwise with the available memory of a todays desktop computer (8 - 16 GB). The two implementations are compared with other nearest neighbor search algorithms and it is examined if they can be used as a base for a classification and clustering algorithm. The two dimension reduction methods MinHash and WTA-Hash are compared to each other.
An interface which is compatible with scikit-learns python interface for the nearest neighbor search[6] is provided.

[1]https://github.com/spotify/annoy, accessed: 2016-05-24
[2]http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LSHForest.html, accessed: 2016-05-24
[3]https://www.khronos.org/opencl/, accessed: 2016-05-24
[4]http://www.nvidia.com/cuda, accessed: 2016-05-24
[5]http://openmp.org, accessed: 2016-05-24
[6]http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html, accessed: 2016-05-24

1.2 Related works

In the following different algorithms for the nearest neighbor search are shown and libraries which provide an implementation of (approximate) nearest neighbor search algorithms are mentioned.

1.2.1 Algorithms

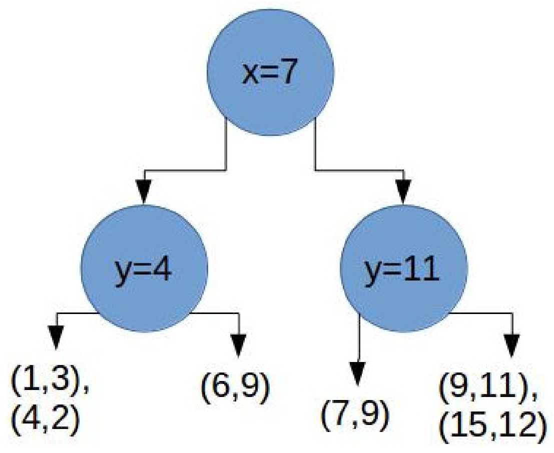
1.2.1.1 Brute force solution

The brute force solution provided by the nearest neighbor algorithm of scikit-learn is computing the nearest neighbors based on different measurements like the euclidean distance or the cosine similarity. The brute force algorithm compares every instance with every other instance and returns in this way an exact solution. The computation can be done by multiple dot product computations. These computations are implemented in the background with BLAS[1], using highly optimized Fortran source code. This makes the algorithm in the case of high dimensions like 4000 very fast.

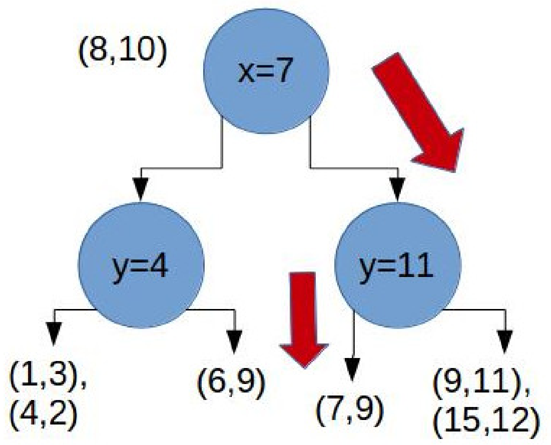
1.2.1.2 Kd-tree

Kd-trees were first introduced by Bentley in 1975 [2] and used by Friedman [3] as a base for the nearest neighbor search. Kd-trees are binary trees and they split the data space by the median of each set for the specific dimension. A kd-tree works as follows: Given a d-dimensional dataset D with the instances $\{x_0, ..., x_n\} \in D$ and $x_i = \{y_0, ..., y_d\}$. The tree is constructed by computing the median for the first dimension and placing every instance which is smaller to the left and every instance which is bigger to the right. For every subtree this is repeated with the next dimension of the subset. In Figure 1.1a the 2-dimensional set $\{(1, 3), (4, 2), (6, 9), (7, 9), (9, 11), (15, 12)\}$ is given. The median for the first instance is 7. According to this value the dataset is split into the two sets: $\{(1, 3), (4, 2), (6, 9)\}$ and $\{(7, 9), (9, 11), (15, 12)\}$. For the second dimension the first set is split at 4 which leads to the two subsets $\{(1, 3), (4, 2)\}$ and $\{(6, 9)\}$; the second set is split at 11 which leads to $\{(7, 9)\}$ and $\{(9, 11), (15, 12)\}$. A nearest neighbor search works as follows: Given some data point x the value of the first dimension is compared to the root node. Based on the decision here the left or the right subtree is evaluated according to the value of the second dimension. This method is an approximate nearest neighbor search method because it can miss some instance. This can happen if a neighbor falls into another subtree. In this case the instance is not seen. In the example in Figure 1.1b the neighbors for instance $(8, 10)$ are searched. It only finds the neighbor $(7, 9)$ but not the also nearest neighbor $(9, 11)$. The kd-tree implementation from scikit-learn is not accepting a sparse matrix as an input.

[1]http://www.netlib.org/blas, http://docs.scipy.org/doc/scipy/reference/linalg.blas.html, accessed: 2016-05-24



(a) Kd-tree for 2-dimensions



(b) Approximate nearest neighbor search path.

Figure 1.1: A kd tree used for approximate nearest neighbor search.

1.2.1.3 Ball tree

Approximate nearest neighbor search based on the data structure 'ball tree' works similar to kd-tree. Instead of splitting by median, the distance to two specified centroids is computed, every instance belongs to one of the centroids and is creating a cluster implicitly. If the distance for an instance is for both centroids equal, one cluster is chosen at random. A node can not be a member in two clusters. For each created cluster two centroids are chosen again and this is repeated recursive until each cluster is containing a specified number of instances or a predefined number of clusters is reached. The search for an approximate nearest neighbor computes recursively to which cluster it belongs and the nearest neighbors are given by the cluster. Again some nearest neighbor instances can be missed in case that an instance belongs to another cluster. The ball-tree implementation from scikit-learn is not accepting a sparse matrix as an input.

1.2.1.4 RPForest

RPForest [4](random projection forest) is using multiple random projection trees which is a variant from the kd-tree [5]. Each tree is recursively split into subsets until each leaf node is responsible for a predefined number of instances. The instances are separated by the cosine angle value for an instance computed to some random hyperplane. As it is usual in binary trees every instance which is smaller as the median angle falls into the left subtree, everything else in the right one. A query is searching in each tree for the corresponding leaf. All instances from here are merged together with the results of all trees. Duplicates are removed and the candidates are sorted by the measurement to the query point.

1.2.1.5 Annoy

Annoy (Approximate Nearest Neighbors Oh Yeah) [6], [7] is an algorithm based on random projections and trees. It was developed by Erik Bernhardsson in 2015 working at that time at spotify. Annoy is designed to search in date sets up to 100 to 1000 dense dimensions. To compute the nearest neighbors it is splitting the set of points into half and is doing this recursively until each set is having k items. Usually k should be around 100. See Figure 1.2. For every set a binary tree is build. To get the nearest neighbors of a point, the right set needs to be chosen and than the binary tree is used for the search. It can happen that a point is next to the border of a set. In this case the neighbor set is also considered. To get better results a priority queue sorted by the minimum margin for the path from the root node is used to search the tree. For every set multiple trees are build. See Figure 1.3. If k items in the trees are found, duplicates are removed and for these instances the distances are computed on the original dataset.

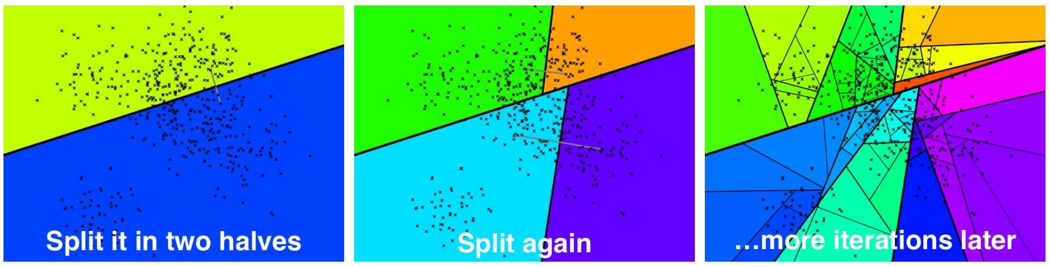
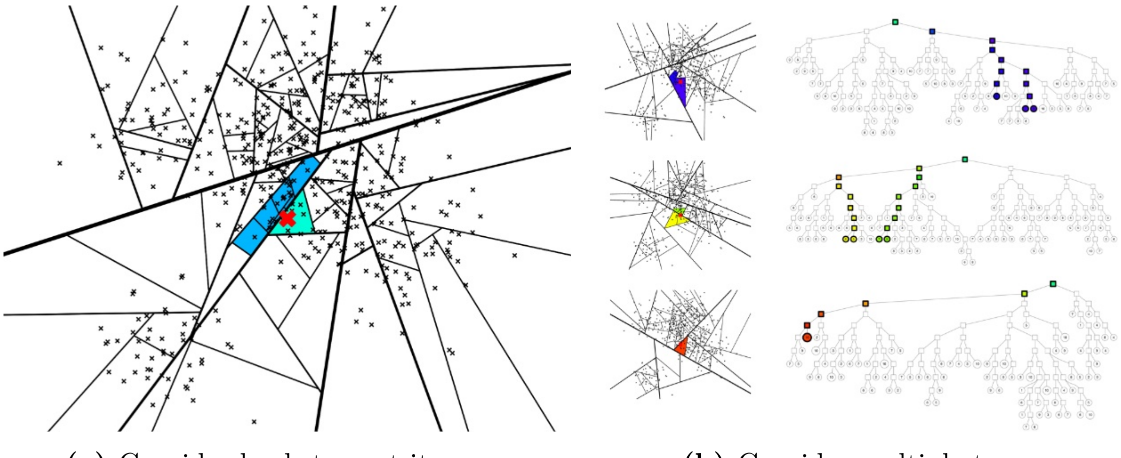


Figure 1.2: Splitting of the dataset used by annoy. Source: [7]



(a) Consider buckets next its own.

(b) Consider multiple trees.

Figure 1.3: Source: [7]

1.2.1.6 Local sensitive hashing forest - LSHF

Local sensitive hashing forest (LSHF) by Bawa et al [8] was published in 2005. LSHF is based on the idea of local sensitive hashing like it was introduced in [9] and [10]. The idea of local sensitive hashing is that similar values are hashed into the same bucket to preserve locality. The probability that two similar values are hashed to the same bucket needs to be higher than the probability of two non similar values. Usually a LSH index is built that one value p is hashed by k hash functions which leads to the label $g(p) = (h_0(p), ..., h_k(p))$. Bawa is introducing a variable length of $g(p)$ instead of a fixed one so that every point gets an unique label. All the labels are than stored in a prefix tree, and each point needs only that many hash functions until its label is unique in the tree. To increase the accuracy of the algorithm multiple trees are constructed which leads to a forest. A query is working in two phases: First all the trees in the forest are searched to get to the leafs with the largest prefix match according to the computed label of the query point. In the second phase all instances from the leafs are collected. Starting at the deepest level of the leaf over all prefix trees the instances are collected level by level until the root node is reached or a predefined number of points is collected.

1.2.2 Libraries providing implementations

1.2.2.1 Flann

Flann (Fast Library for Approximate Nearest Neighbors) [11] is a library which is providing different implementations for a nearest neighbor search. It provides the brute force search and the approximate nearest neighbor search based on kd-trees, kmeans and a mixture of both.

1.2.2.2 Panns

Panns (Python approximate nearest neighbor search) [12] is using a binary tree as a data structure. Every instance is scalar-projected with a random projection. Each node represents the median of all scalar-projections of all included data points. As usual every instance with a value less than this median falls into the left subtree, every other node to the right one. For a query the tree is searched according to the random projections.

1.2.2.3 Nms lib

Nmslib (Non-Metric Space Library) [13] provides different implementations like panns for the nearest neighbor search. It is implemented in C++ and is often using SIMD (Single Instruction Multiple Data) optimizations like SSE. A Python interface is provided.

1.2.2.4 scikit-learn

Scikit-learn [14] is a machine learning library for Python and is providing implementations for the brute-force, LSHForest, BallTree and kd-tree algorithm.

1.2.3 Random projection

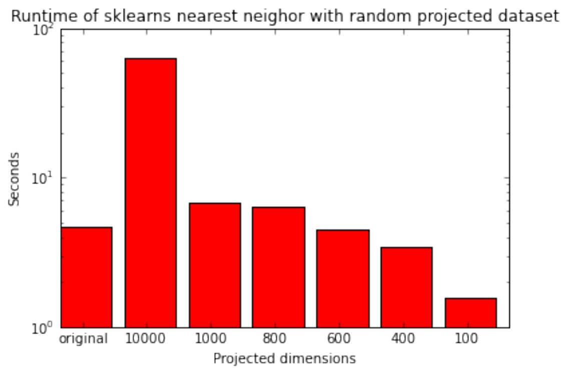
A common used technique to reduce the dimensionality is a random projection. The idea behind random projection is the Johnson-Lindenstrauss lemma [15] projecting the d-dimensional original data space to k-dimensions. It is computed with a simple matrix multiplication:

$$X_{k,n}^{randomprojected} = R_{k,d} * X_{d,n} \tag{1.1}$$

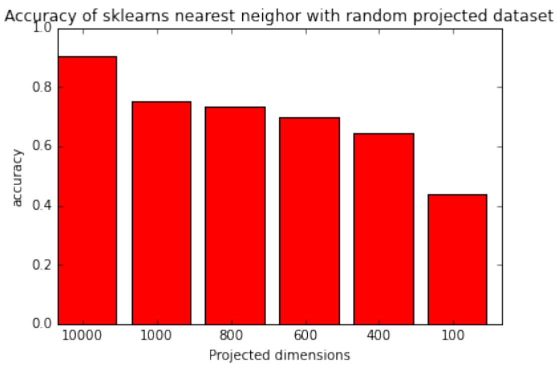
The projection matrix R is original based on Gaussian distribution but Achlioptas [16] has shown that it is enough to use for each entry:

$$r_{i,j} = \sqrt{3} \begin{cases} +1 & \text{with probability } 1/6 \\ 0 & \text{with probability } 2/3 \\ -1 & \text{with probability } 1/6 \end{cases} \tag{1.2}$$

The greater the dimensionality reduction the more accuracy is getting lost. In Figure 1.4b it is shown that if the dataset 'Bursi' (see subsubsection 7.0.1.1) with about 1 million dimensions and 370 non-zero features per instance on average is reduced with the sparse random projection from sklearn the accuracy decreases for example to 90 % with 10000 dimensions and to 73 % for 800 dimensions. Furthermore the runtime is influenced by the projection. The sparse projection tries to achieve that about 30 % are non-zero features, which leads to a runtime of about a minute for the projected dataset with 10000 dimensions compared with a little bit less than five seconds for the original dataset. Also it is interesting to see that the dimensions need to be reduced to 600 dimensions and less to have a faster query than on the original dataset.
The Gaussian random projection in sklearn which would reduce the given sparse dataset to a dense one is not working on a today's normal computer with 8 GB RAM because the software is terminating with error message 'Memory error'.



(a) Runtime for the projected dimensions.



(b) Accuracy for the projected dimensions.

Figure 1.4: Influence of the random projection to the accuracy.

2 Nearest Neighbor Search

The nearest neighbor search is a similarity problem to find the closest points to a given instance:

Definition
Out of a set of n instances $P = \{p_1, ..., p_n\}$ in some metric space X the nearest neighbor search computes the closest instance q to an instance p under some measurement function.

The k-nearest neighbors are:

Definition
Out of a set of n instances $P = \{p_1, ..., p_n\}$ in some metric space X the k-nearest neighbor search computes the closest k-instances $\{q_0, ..., q_k\}$ to an instance p under some measurement function. The set $\{q_0, ..., q_k\}$ is ordered according to the measurement function.

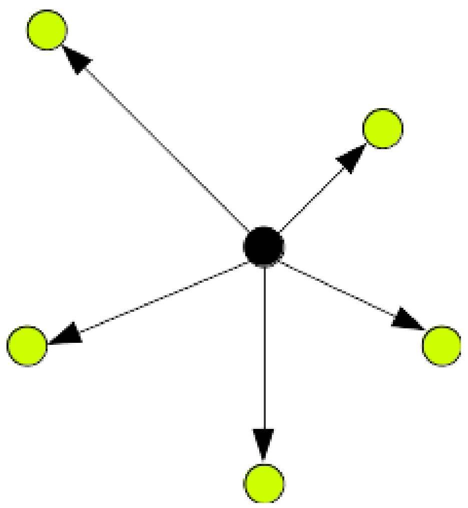


Figure 2.1: 5-nearest neighbors to an instance (black circle).

An exact algorithm is working well with low dimensions, as a distance or similarity measure they usually use the euclidean distance or the cosine similarity. The time complexity is $O(n * m)$ for n instances and m dimensions per query and for all instances it is $O(n^2 * m)$. It is easy to see that for lower dimensions the runtime of a query is not that much influenced by the number of dimensions. If the dimensions are significant larger than the number of instances, it leads to a problem called 'curse of dimensionality'. To fight the curse of dimensionality Indyk and Motwani [10] introduced the approximate nearest neighbor search.

2.1 Approximate Nearest Neighbor Search

The idea behind the approximate nearest neighbor search is to speed up the computation of the nearest neighbors, the exact algorithm which is in $O(n^2)$ can not be improved. The only way to speed up the computation is to allow errors.

Definition
Find a point $p \in P$ that is an ϵ-approximate nearest neighbor of the query q, that $\forall p' \in P$, $d(p,q) \leq (1+\epsilon)d(p'q)$.

This definitions says that instances which are only a factor of ϵ away from the real nearest neighbors can be considered as nearest neighbors. Thankfully this is not too difficult to achieve because the value of ϵ is not restricted to a given range. Approximations can work in different ways. As described in section 1.2 the usual way is to reduce the number of dimensions by a random projection. This does not reduce the $O(n^2)$ runtime but fights the curse of dimensionality. The second approach is to compute k candidates per instance which leads to a runtime of $O(n * k)$. Also both solutions can be combined.

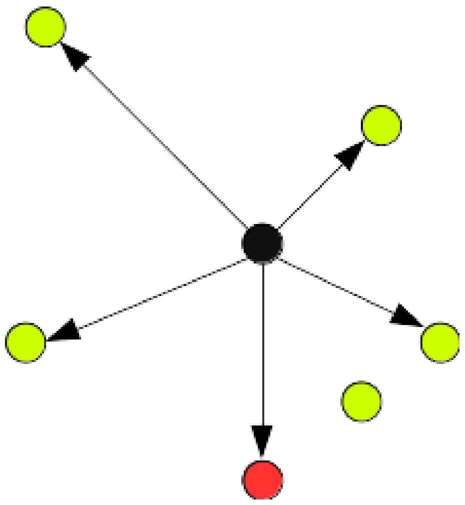


Figure 2.2: Approximate nearest neighbors to an instance (black circle).

3 Estimator induction

To fight the curse of dimensionality two estimators are evaluated, minimum hash (MinHash) [17], [18] and winner takes it all hash (WTA-Hash) [19]. The estimators are used to reduce the dimensions of a dataset to the number of used hash functions. To do this they compute a signature for each instance. After it the signature is inserted to an inverse index; this inverse index is used for a fast prediction of the k-nearest neighbors.

3.1 Curse of dimensionality

The curse of dimensionality was first mentioned by Bellman in 1957 [1]. The well known notion of big O to specify the theoretical runtime of an algorithm focuses on the number of input elements. Bellman showed that for a computation of n instances with m dimensions that the runtime is more depended on the number of dimensions if $m >> n$. For example if $n = 1000$ and $m = 100000$ then a quadratic factor for the runtime of the prediction for the k-nearest neighbors with $O(n^2 * m)$ is not influencing the runtime that drastically in comparison to the influence of the dimensions.

3.2 Minimum Hash

Minimum Hash is a local sensitive hash function introduced by Broder [17]. Local sensitive hash functions are, as the name indicates, hash functions that try to preserve the locality information of the data. Broder addresses the issue of how to compare the content of two documents A and B with each other to find out first, the resemblance $r(A, B)$ ' how likely it is that the documents are roughly the same' and second, 'the containment $c(A, B)$ [...] indicates that A is roughly contained within B'. To achieve this, he reduces the content of a document to canonical tokens which means that no matter how a document is formatted, as long as the content is the same, they are reduced to the same tokens. To compute the resemblance and the containment, Broder reduces the problem to an intersection of sets. To do so, he defines the sets per document as follows:

Definition

A *bag* for a document D contains all *sets* of *subsequences* for the document D with the tokens $S(D, w)$, where w is the shingle size and equals to the size of each subset.

Example:
The document '(a, rose, is, a, rose, is, a, rose)' with w = 4 leads to the bag {(a, rose, is, a), (rose, is, a, rose),(is, a, rose, is),(a, rose, is, a),(rose, is, a, rose)}.

The resemblance is defined as:

$$r_w(A, B) = \frac{\mid S(A, w) \cap S(B, w) \mid}{\mid S(A, w) \cup S(B, w) \mid} \tag{3.1}$$

and the containment as:

$$c_w(A, B) = \frac{\mid S(A, w) \cap S(B, w) \mid}{\mid S(A, w) \mid} \tag{3.2}$$

These definitions implicate the issue that it can happen that different documents A and B would have the a same, but permuted bag, e.g. (a, c, a, b, a) and (a, b, a, c, a). To overcome this, Broder introduces the following:

Definition

Let Ω bet the set of all shingles of size w. Without loss of generality Ω is totally ordered. For fixed parameter s and $W \subseteq \Omega$ $MIN_s(W)$ is:

$$MIN_s(W) = \begin{cases} \text{the set of the smallest s elements in W}, & \text{if } \mid W \mid \geq s; \\ \text{W}, & \text{otherwise} \end{cases} \tag{3.3}$$

and

$$MOD_m(I) = \Big\{ \text{the set of elements of W that are 0 } mod\ m \tag{3.4}$$

Let $g : \Omega \to \mathbb{N}$ and $\pi : \Omega \to \Omega$ be a permutation of Ω and let $M(A) = MIN_s(\pi(S(A, w)))$ and $L(A) = MOD_m(g(\pi(S(A, w))))$.
Now

$$r(A, B) = \frac{\mid MIN_s(M(A) \cup M(B)) \cap M(A) \cap M(B) \mid}{\mid MIN_s(M(A) \cup M(B)) \mid} \tag{3.5}$$

and

$$r(A, B) = \frac{\mid L(A) \cap L(B) \mid}{\mid L(A) \cup L(B) \mid} \tag{3.6}$$

are both values unbiased estimates of the resemblance of A and B; for the containment is

$$c(A, B) = \frac{\mid L(A) \cap L(B) \mid}{\mid L(A) \mid} \tag{3.7}$$

valid.

Heyne and Costa [18] transfered the work from Broder to the subject of the nearest neighbors and dimension reduction. To achieve this, they consider for each instance only the non-zero feature ids of an instance and compute a signature as follows: Using a set of random hash functions $F_i : \mathbb{N} \to \mathbb{N}$. The hash functions have to satisfy: $\forall x_j \neq x_k, f_i(x_j) \neq f_i(x_k)$ and $x_j \neq x_k, P(f_i(x_j) \leq f_i(x_k)) = 1/2$. The minimum hash function or MinHash is defined as

$$h_i(x) = argmin_{x_j \in x} f_i(x_j) \tag{3.8}$$

For every instance the values of n MinHash functions are computed and stored as the signature of this instance. The dimensions of the input data is reduced to the number of used MinHash functions.

The number of features that two instance are having in common is given by the Jaccard similarity: $s(x, y) = (|x \cap y|)/(|x \cup y|)$. MinHash is an unbiased estimator of it:

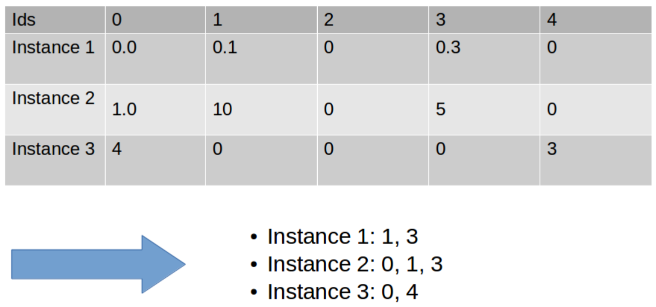
$$P(h_i(x) = h_i(y)) = (|x \cap y|)/(|x \cup y|) = s(x, y) \tag{3.9}$$

This means that the probability that for two instances the hash function with the minimum hash value is the same as the fraction of non-zero features that these two instances are having in common [18]. To decrease the variance of this estimate N independent MinHash functions can be computed. The MinHash algorithm is shown in Algorithmus 1.

```
Data : $m$ hash functions $H$, input vector X with non-zero feature ids
Result : Signature S for input vector X with a size of $m$
for $hash\_function_i$ in $H$ do
    minHashValue = MAX_VALUE;
    argmin = 0;
    for each non-zero feature id $x_j$ in $X$ do
        hashValue = $hash\_function_i(x_j)$;
        if $hashValue < minHashValue$ then
            minHashValue = hashValue;
            argmin = $x_j$;
        end
    end
    $S_i$ = argmin;
end
return S;
```

Algorithmus 1 : MinHash algorithm

Example
Given is the dataset in Figure 3.1. Only the feature ids of the non-zero features are considered, these would be for instance 1: 1 and 3. For every non-zero feature id a hash value is computed and only the minimal hash value and the corresponding feature id is stored, $h(1) = 4$ and $h(3) = 2$. In this case the MinHash value for the first instance would be 3. This is computed for n-hash functions, for every instance. The value per MinHash function is stored in the signature: $\{x_0, ..., x_n\}$



Ids	0	1	2	3	4
Instance 1	0.0	0.1	0	0.3	0
Instance 2	1.0	10	0	5	0
Instance 3	4	0	0	0	3

- Instance 1: 1, 3
- Instance 2: 0, 1, 3
- Instance 3: 0, 4

Figure 3.1: Example data for MinHash

3.3 Winner takes it all hash

The winner-takes-it-all algorithm was developed by Yagnik el al [19]. It is based on the idea of rank correlation measures and satisfies the condition of a local sensitive hash function. It uses m permutations Θ and a window size K. For every permutation θ_i it permutes all the values given by an input vector X, afterwards it considers only the first K values given some ordering and out of these it stores the index id of the highest values as a value for permutation θ_i, called c_i. All c_i are stored in a vector $C_X = [c_0, ..., c_m]$. C_X is the signature. The algorithm is shown in Algorithmus 2.

Example
The example given in Figure 3.2 works the following. For the given permutation $\Theta = \{1, 4, 2, 5, 0, 3\}$ the values are permuted e.g. in (d) the zero position with an 11 is permuted to position index 4 according to the permutation Θ. After this the first k values are considered. In this case just the first $k = 4$ are considered but it could also be sorted according to some other criteria. Out of the first 4 positions the highest value, 13, is searched and the index position, 1, is stored as the winner takes it all hash value.

To use WTA-Hash for a dimension reduction and a candidate selection the definitions from Yagnik et al. needs to be specified more. The permutations of the values are done by hashing, the values are then ordered by the smallest hash value. Out of

```
**Data** : A set of m Permutations $\Theta$, window size $K$, input vector X.
**Result** : Sparse vector of codes $C_X$
for each permutation $\theta_i$ in $\Theta$ do
    a) Permute elements of $X$ according to $\theta_i$ to get $X'$.;
    b) Initialize $i^{th}$ sparse code $c_{x_i}$ to 0.;
    c) Set $c_{x_i}$ to the index of the maximum value in $X'(1...K)$;
    for $j$ = 0 to $K$ -1 do
        if $X'(j) > X'(c_{x_i})$ then
            $c_{x_i} = j$
        end
    end
end
return $C_X = [c_{x_0}, c_{x_1}, ...., c_{x_m}]$. C contains $m$ codes, each taking a value between 0
and $K - 1$.
```

Algorithmus 2 : WTA-Hash algorithm. Source: [19]

the smallest k hash values the index of the hash value for which the original value from the dataset is the biggest is taken as the result.

Example
The original feature ids and associated values are: $1 : 0.3, 5 : 0.7, 9 : 0.1$. Hashing them will produce: $h(1) = 4$, $h(5) = 1$, $h(9) = 3$. With this setting the hash values would be ordered to $1, 3, 9$. If k is set to 2, the two values of $1, 3$ which are the hashed values for the feature ids 5 and 9 and the associated values are: 0.7 and 0.1. The highest value is 0.7, which was for the hash value at index position 1. 1 would be returned as a value for this permutation.

Winner takes it all hashing comes with the problematic nature that it is too less sensitive for the locality information. Only k buckets exist and this leads to the effect that too many instances are in one bucket. To much candidates are selected and the most candidates are not that good distinguishable, they have all more or less the same number of hits. This leads to the effect that for a test run with the algorithm as it was proposed a run time of 2.5 seconds was recorded, compared with 0.6 seconds for MinHash and 0.92 for the brute force algorithm. Second, the accuracy was just 0.55 compared to 0.92 of MinHash. To overcome this issue the algorithm needs to be modified a little bit. Instead of taking the index value of the highest value, the actual value i.e. the hashed feature id is used. With this small change the winner takes it all algorithm runs for the same parameter configuration with 0.7 seconds and an accuracy of 0.88. The modified algorithm is shown in Algorithmus 3.

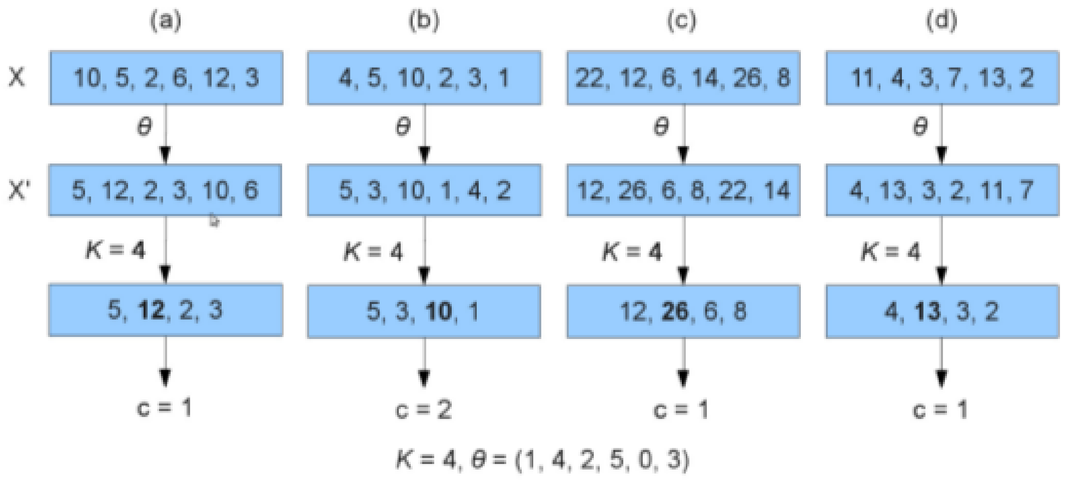


Figure 3.2: Winner takes it all hash example. Source [19]

```
Data : $m$ hash functions, window size $K$, input vector with non-zero feature ids
       X, input vector V with associated values
Result : Vector of codes $C_X$
for each hash function $\theta_i$ in $\Theta$ do
    a) Hash non-zero feature ids $x_j$ of $X$ according to $\theta_i$ to get $X'$.;
    b) Sort hash values by ascending order, keep relation to the original values $v_j$
    in the vector $V'$;
    c) Initialize $i^{th}$ code $c_{x_i}$ to 0.;
    d) Set $c_{x_i}$ to the hash value $X'(j)$ of the maximum original value $V'(j)$;
    for $j$ = 0 to $K$ -1 do
        if $V'(j) > V'(maxIndex)$ then
            $c_{x_i} = X'(j)$;
            $maxIndex = j$;
        end
    end
end
return $C_X = [c_{x_0}, c_{x_1}, ...., c_{x_m}]$.;
C contains $m$ codes, each taking a value between 0 and $MAX\_INT$.;
```

Algorithmus 3 : modified WTA-Hash algorithm.

4 Approximate nearest neighbor search algorithm

The approximate nearest neighbor search algorithm based on the dimension reduction introduced in chapter 3 works in two phases: First the fitting phase: the dimensions are reduced with MinHash or WTA-Hash, then they are inserted in an inverse index. Optional the inverse index is pruned with different methods to save memory. The used measurements are the euclidean distance or cosine similarity. In the second phase, the query phase, the following happens: First the signature of every query instance is computed and every signature is queried against the inverse index. Based on the number of hits per hash function, the instances are ranked. In the fast case of the algorithm, the instances are returned as the nearest neighbors. In the non-fast case two additional rounds are computed: based on the candidate selection from the inverse index, all the neighbors for a given instance are measured on the original dataset. Based on this, the neighbors of the k-neighbors are taken as candidates and are measured on the original dataset again.

4.1 Measurements

The approximate nearest neighbor search is providing two measurements: the euclidean distance as a distance measurement and the cosine similarity as a similarity measurement.

4.1.1 Euclidean distance

As a distance measurement the euclidean distance is used. The euclidean distance is defined for the n-dimensional space as:

$$d(x, y) = \sqrt{\sum_{i=0}^{n} x_i - {x_j}^2} \tag{4.1}$$

For sparse matrices the following definition is having benefits:

$$d(x, y) = \sqrt{dot(x, x) - 2 * dot(x, y) + dot(y, y)} \tag{4.2}$$

And the dot product is defined as:

$$dot(x,y) = \sum_{i=0}^{n} x_i * y_i \tag{4.3}$$

The benefits are that first, $dot(x,x)$ can be precomputed and second, instead of a subtraction and a square root for ever dimension it is reduced to a multiplication of non-zero dimensions.

4.1.2 Cosine similarity

The cosine similarity is used as a similarity measurement and is defined as follows:

$$cos\_sim(x,y) = \frac{\sum_{i=0}^{m} x_i y_i}{\sqrt{\sum_{i=0}^{m} {x_i}^2}\sqrt{\sum_{i=0}^{m} {y_i}^2}} = \frac{dot(x,y)}{\sqrt{dot(x,x)}\sqrt{dot(y,y)}} \tag{4.4}$$

4.2 Estimator

At the fitting process the dimensions of the input dataset are reduced to the number of hash functions. The reduced data per instance is called a signature and is computed with MinHash or WTA-Hash, introduced in chapter 3. The signatures are inserted in the inverse index. During or after the fitting process the inverse index can be pruned. The idea is to save memory and to speed up the computation. These optimizations are explained in detail in chapter 5, the analysis in section 7.2. The computed signatures are stored per instance to avoid the expensive recomputation of a known instance for the queries.

```
Data : Sparse dataset with n dimensions
Result : The inverse index
signatureStorage = list<list<signatures> > ;
inverseIndex = list<map<hashvalues, list<instances> > >;
for every instance do
    signatureStorage[instance] = compute signature;
end
for every signature in signatureStorage do
    for every hashValue in signature do
        insertInInverseIndex(hashValue, instance, hashFunction);
    end
end
```

Algorithmus 4 : Basic fitting algorithm

4.2.1 Runtime analysis

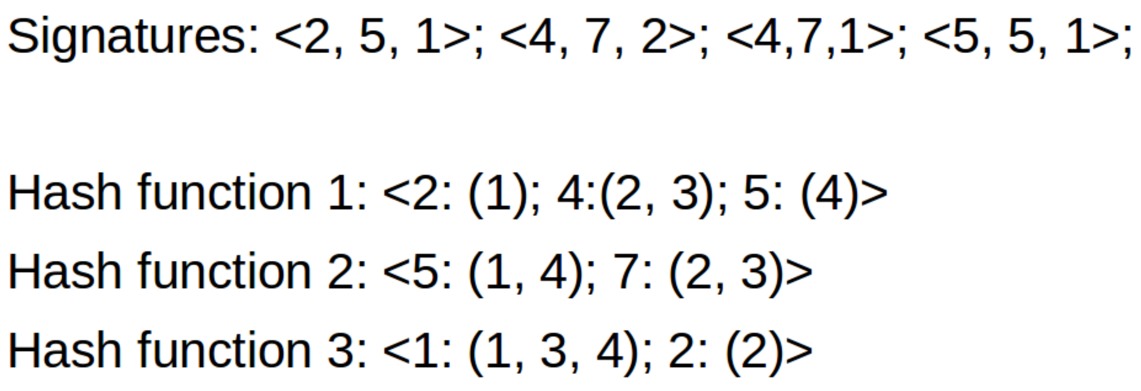
The runtime of the fitting part is given by:

- The computation of the signatures. For each instance n the number of non-zero features m needs to be hashed by h hash functions if MinHash is used. This leads to a runtime of $O(n*m*h)$. For WTA-Hash each instance n with m non-zero features needs to be hashed by h hash functions. Per hash function, in terms of the WTA-Hashing permutations Ω, only the best k-values need to be stored and additional the index with the maximal value is returned as the final value. This needs additional $2 * k$ steps. All together this leads to a runtime of $O(n * m * h * 2 * k)$.
- The values need to be inserted in the inverse index. For n instances and h hash values per signature this needs $O(n * h)$.
- The fitting needs $O(n*m*h+n*h)$ for MinHash and $O(n*m*h*2*k+n*h)$ for WTA-Hash.

4.3 Structure of the inverse index

The inverse index is storing per hash function the created values and from which instance they are coming from. This realized with a map per hash function which is storing as a key the hash value and as a value a vector of the instances.

Example
As it can be seen in Figure 4.1 the computed values are stored per hash function. For the first instance the value 2 is stored in the map for the first hash function and the value 1 for the first instance is associated with it.


Signatures: <2, 5, 1>; <4, 7, 2>; <4,7,1>; <5, 5, 1>;

Hash function 1: <2: (1); 4:(2, 3); 5: (4)>

Hash function 2: <5: (1, 4); 7: (2, 3)>

Hash function 3: <1: (1, 3, 4); 2: (2)>

Figure 4.1: Inserted signatures in the inverse index.

4.3.1 Pruning the inverse index

To save memory and to speed up the computation time the inverse index can be pruned. The details are discussed in section 5.2. To be exemplary for these techniques two ideas are shown as an algorithm. First Algorithmus 5 shows that hash values with less associated instances than a given threshold can be deleted and second Algorithmus 6 shows that hash functions with less entries as a given threshold can be deleted. Both pruning steps are increasing the runtime of the fitting process. The first idea contributes additional $O(h * k)$ with h hash functions and k hash values at maximum to the fitting process and the second $O(h)$.

```
Data : A threshold, the inverse index
Result : The pruned inverse index
for every hash function in the inverse index do
    for every hash value in a hash function do
        if number of instances for hash value < threshold then
            delete hash value;
        end
    end
end
```

Algorithmus 5 : Pruning of hash values

```
Data : A threshold, the inverse index
Result : The pruned inverse index
for every hash function in the inverse index do
    if number of hash values for hash function < threshold then
        delete hash function;
    end
end
```

Algorithmus 6 : Pruning of hash functions

4.4 Prediction phase

The prediction phase is split into two phases: in the first phase candidates are searched in the inverse index and these candidates are taken to compute the real distance respectively similarity on the original dataset. In the second phase the neighbors of neighbors are taken as the candidate set to increase the accuracy level of the algorithm.

4.4.1 Candidate selection I: Values from the inverse index

The first candidate selection round queries against the inverse index which was created during the fitting process. For each instance the signature needs to be computed. If it is a known instance i.e. it was part of the training data, the signature is taken from memory. Each value from the signature is used to check in the inverse index if for the specific hash function a hash value was created at the fitting process. If yes all related instances are taken and stored. After all hash functions were checked, the occurrence of instances is counted. This count is called the number of hits. The instances are sorted after the number of hits in descending order. At this point it is possible to return the found instances, this mode of the algorithm is the 'fast' mode. As it can be seen in chapter 7 it is significant faster but it is less accurate. In the non-fast mode of the algorithm the candidate selection continues with the computation of the real distances respectively similarities on the original dataset. For the computation $k * m$ candidates are considered, k is the number of nearest neighbors that should be returned and m is an excess factor. The excess factor is used that more instances than the k neighbors can be considered. If the last candidate given by the sorting of the number of hits is having the same number of hits as the $k*m+1$ candidate, this candidate is considered too. This extension of the candidate set is continued until the next candidate is having less number of hits than the $k * m$ candidate. The extension is implemented because the instances with the same number of hits can not be distinguished. The computed distances are then sorted by ascending order for the distance measurement respectively in decreasing order for a similarity measurement. If the values would be returned at this point and the second candidate selection would not be computed, the accuracy would be limited to 80% as you can see in Figure 4.2.

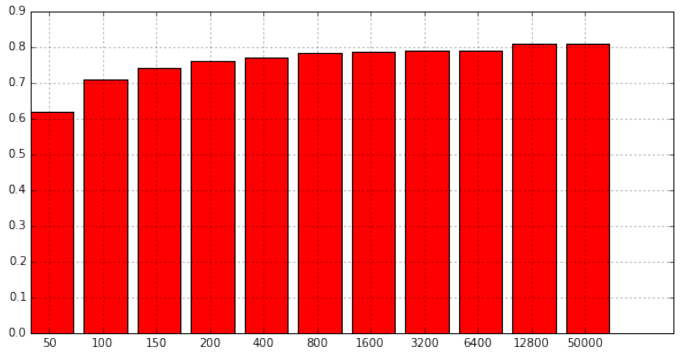
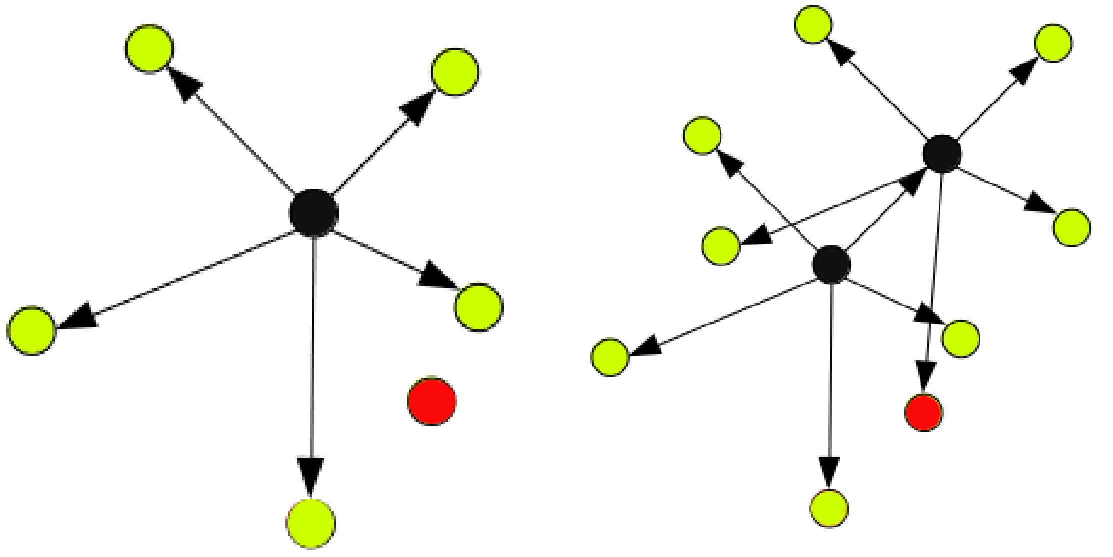


Figure 4.2: Accuracy with n hash functions if returned after the first candidate selection.

4.4.2 Candidate selection II: Neighbors of neighbors

The second candidate selection is based on the idea that not all real nearest neighbors are seen because of the approximate candidate selection, see Figure 4.3a. But maybe a neighbor of an instance is having one of the real neighbors in its candidate set, see Figure 4.3b. The candidate selection II is rebuilding the candidate set for each instance and the members are the computed k-nearest neighbors from the first candidate selection and the $k + m$ candidates from each neighbor. This leads to a candidate set size of $k * (k + m)$ per instance. Given the new candidate set, the real distances respectively the similarity is computed on the original dataset. With this trick a higher accuracy can be reached, accuracy levels of 0.9 to 0.95 are possible in reasonable time as it can be seen in chapter 7. In contrast to the first candidate selection the excess factor is added and not multiplied. This is done to limit the number of neighbors which needs to be computed in the second candidate selection.



(a) Candidate (with out an arrow) is not in the first candidate set.

(b) Candidate is inside the candidate set of a neighbor.

Figure 4.3: Idea of candidate selection II

4.4.3 Runtime analysis

The algorithm shown in Algorithmus 7 needs:

- for n input query instances $O(n*h*m)$ with h hash functions and m non-zero features to compute the signatures if they are not in the signature storage.
- $O(n * h)$ to get the values out of the inverse index and $O(hi + n * c * log_2(c))$ to count hi hits and to sort c unique candidates per instance.
- $O(n * c * m + c * log_2(c))$ to compute and sort the euclidean distance / cosine similarity for n query instances, c candidates per instance and m non-zero features.
- $O(n * c_2 * m + n * c_2 * log_2(c_2))$ to compute and sort the euclidean distance / cosine similarity for the second candidate set and $O(n * k * (k + e))$ to get the second candidate set c_2 with k nearest neighbors and e as the excess factor.

```
Data : Instances to predict the k-nearest neighbors
Result : The k-nearest neighbors of every query instance
for ever instance do
    compute signature;
end
hitsPerInstance = list(set());
for every signature do
    for hash value from hash function do
        if hash value in inverseIndex[hash function] then
            hitsPerInstance(instance) = inverseIndex[hash function][hash value];
        end
    end
    for all instances do
        sort hitsPerInstance in descending order;
    end
    if fast == true then
        return hitsPerInstance;
    end
    for all instances do
        compute for every hitsPerInstance[instance] the euclidean distance or cosine
        similarity and sort desc or asc.
    end
    for all instances do
        collect for every neighbor the nearest neighbors + excessFactor;
    end
    for all instances do
        compute for every new candidate set[instance] the euclidean distance or
        cosine similarity and sort desc or asc.
    end
    return neighbors;
end
```

Algorithmus 7 : Prediction phase.

- All together this leads to a runtime which is mainly dependent on the number of query instances n, the size of the candidate set, the number of used hash functions h and the number of non-zero features m: $O(n * h * m + n * h + n * c * m + n * c_2 * m) \sim O(4 * n * c * m)$.

5 Memory saving techniques

To store as less as possible values is a key factor for a good performance of the approximate nearest neighbor search. The less hash values that need to be considered, the faster a query, the faster the algorithm is. In the following different techniques are introduced and discussed under the two aspects if they achieve to save memory and or to decrease the runtime of a prediction. Except the Bloomier filter which is an alternative to store the inverse index, the methods operate on the built inverse index or during build time.

5.1 Bloomier Filter

The Bloomier Filter is introduced by Chazelle et al in 2004 [20]. It is based on the bloom filter from Bloom published in 1970 [21]. The intend of bloom filters is to save memory by combining the stored elements. A bloom filter is operating on the input elements $I = x_1, ..., x_n$ and storing these values in a $m - bit$-sized vector. Initial all bits are set to 0. To store the values k hash functions are used and each hash function is projection to the range of 0 to m. Every to be stored number is hashed with every hash function and the value of the hash value is taken to set the bit at this index position to 1. If it is already 1 it is not changed. See Figure 5.1. Bloom filters are answering the question if an element is inserted always correct with just one bit operation which makes it fast. An element is in the bloom filter if all computed index positions are set to 1. False positives appear with a probability of $(1-p)^k$ if p is the probability that a bit is 0: $p = (1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$ [22].

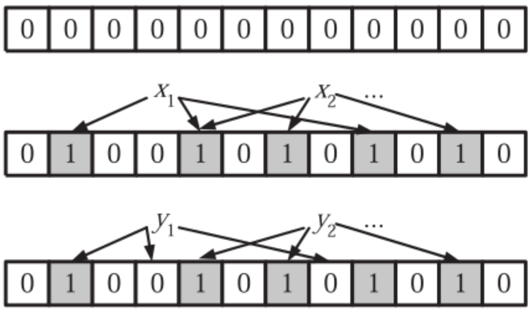


Figure 5.1: Functional principle of a bloom filter. Source: [22]

Bloomier Filters work a little bit different. Instead of just answering the question if a key was seen before, Bloomier filters can store key/value pairs. If a key was seen before, it returns the stored value of this key, otherwise it returns a negative answer. Chazelle et al [20] defines Bloomier filters as follows:

Definition
Given a domain $D = \{0, ..., N-1\}$, a range $R = \{\perp, 1, ..., |R|\}$, a subset $S = \{t_1, ..., t_n\}$ of D, [...] encode the function $f : D \rightarrow R$ such $f(t_i) = v_i$ for $1 \leq i \leq n$ and $f(x) = \perp$ for $x \in D \backslash S$.

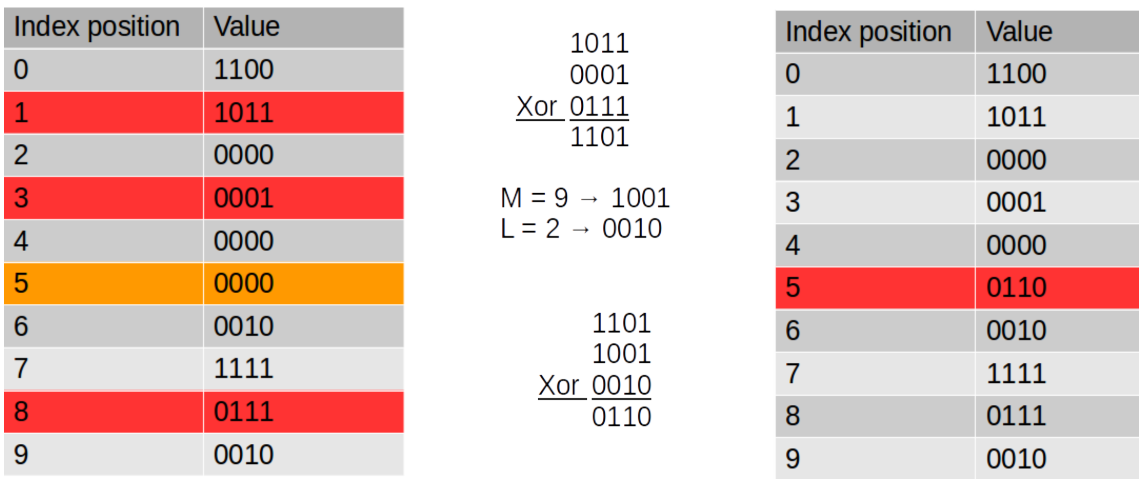
The used data structure is a table with m entries with a size of k each. With a hash function $HASH : D \rightarrow \{1, ..., m\}^k$ random locations can be accessed. To store a value in the table a neighborhood for key t defined by $HASH(t) = (h_1, ..., h_k)$ is computed. These hash values are considered as the neighborhood $N(t)$. The neighborhood is used to store the location of the value. Out of the neighborhood of a key, the minimum element which was not used for any other key before is searched. This element is called a singleton. If there is no such element a new neighborhood needs to be computed. All elements of a neighborhood are defined as used and are not allowed to be changed after it. To set an element the first time the following is computed:

1. The neighborhood $(h_1, ..., h_k, M) = HASH(t)$ with k positions in the table and a mask M.
2. The index position of the singleton $\iota(t) = l$ within the neighborhood.
3. The hash value of the singleton $h_l : L = \tau(t)$.
4. The key is stored in the $Table_1$: $Table_1[L] = ENCODE(l) \oplus M \oplus \bigoplus_{i=1, i \neq l}^{k} Table_1[h_i]$. ENCODE means to get the value l into a m-size bit vector.
5. The associated value v to t is stored in $Table_2[L] = v$.

To retrieve a value the following is computed:

1. The neighborhood $(h_1, ..., h_k, M) = HASH(t)$ with k positions in the table and a mask M.
2. $l = DECODE(M \oplus \bigoplus_{i=1}^{k} Table_1[h_i])$.
3. If the retrieved l is within the range of $\{0, ..., k\}$ than the value h_l from the neighborhood is taken, $h_l = L$, and $Table_2[L]$ is returned, otherwise there is no value stored.

To change a stored value, the computation is the same like for the retrieving but instead of returning the value, the new value is set. Chazelle et al. is assuming that all values that should be stored are known at the time the Bloomier filter is created. This is not the case if it is used as a data structure for the inverse index. To use it without the knowledge of the to be stored values, the algorithm needs to be changed a little bit. Instead of returning a 'value not known / false' if a value should be set



Index position	Value
0	1100
1	1011
2	0000
3	0001
4	0000
5	0000
6	0010
7	1111
8	0111
9	0010

```
    1011
    0001
Xor 0111
    1101

M = 9 → 1001
L = 2 → 0010

    1101
    1001
Xor 0010
    0110
```

Index position	Value
0	1100
1	1011
2	0000
3	0001
4	0000
5	0110
6	0010
7	1111
8	0111
9	0010

Figure 5.2: Example computation to set a value for the first time in a Bloomier filter.

but a) the retrieved value of l is not within the range of $\{0, ..., k\}$ or the associated index position within the $Table_2$ is empty, the function to set a value the first time is called.

Example
Assume the key t gives the neighborhood $HASH(t) = \{1, 3, 5, 8, 9\}$. Lets assume the value $h_2 = 5$ is the singleton and the mask is $M = 9$. Now $\iota(t) = 2$ and $\tau(t) = 5$. To store the value l the $m = 4$-sized bit vectors in $Table_1$ for the positions 1, 3, 8 need to be xored (Figure 5.2 left and mid up). This result is xor'ed with the mask M which needs to be encoded as a 4-bit vector and with the encoded version of $l = 2$ (Figure 5.2 mid). The result is stored in $Table_1[5]$(Figure 5.2 right), the associated value in $Table_2[5]$. To retrieve the value, the same neighborhood needs to be computed and all the values for the index positions in $Table_1$ are xor'ed. Xoring the values '1011, 0001, 0110, 0111 and 1001' leads to '0010' which is 2. At index position 2 in the neighborhood the value 5 is stored and the value can be retrieved by a lookup at position 5 in $Table_2$.

5.1.1 Usage as the data structure for the inverse index

The intend to use the Bloomier filter as a data structure for the inverse index is to save memory and faster query times. Both intends failed. In the following the approximate nearest neighbor search algorithm was used with first the Bloomier filter as a data structure for the inverse index and second the unordered map implementation from the standard C++11 library. The used algorithm is from an earlier development stage (mid December 2015) and many optimizations in terms of speed are not implemented. The runtime of the algorithm are not comparable to the running times in chapter 7. The memory and time was measured with the ipython notebooks cell magic command %memit and %time. MinHash was used with 400 hash functions and cosine similarity as a measurement, two CPU cores were used on an Intel i5-5200U.

5.1.1.1 Memory usage

The usage of memory with the unordered map was for the fitting about 70 MB, the Bloomier used 230 MB. Why is there such a huge gap between the two data structures? The main reason is how the Bloomier filter is building its tables. For every key it is xor'ing k values. In one of these values the data is written, but all k values are not allowed to be used as singleton after it. With an optimal finding of the singletons $n + k - 1$ values would be needed per Bloomier filter. But the Bloomier filter is computing with hashing k neighbors and out of these one value have to be a singleton. Under the assumption that the used hash function distributes the values uniform, this issue can be reduced to a classical example of stochastics: The coupon collector's problem. How many tries t does it take until all n different coupons are collected if s coupons can be taken at the same time? If the implementation would allow that only the minimal number of values $n+k-1$ is allowed as the domain size, the probability of finding a singleton would become too soon too less. To increase the probability of finding values the size of the domain must be increased. This leads to the issue that many possible positions in the Bloomier filter are empty. If the domain size is too small and no singleton is found in the first try, the algorithm tries to find a singleton with another hash functions. But the information which hash function need to be used to retrieve the values need to be stored which costs additional memory. Second, it is unknown from the beginning how many hash values need to be stored in the Bloomier filter per MinHash function. For every MinHash function in the inverse index the maximum size of values need to be hold. This is significant different from the unordered map. The unordered map can be empty at initialization time and is extended if a new element is inserted. To have this behavior for the Bloomier filter all keys would need to search a new neighborhood after every extension of the space. Nevertheless it would be possible in general but the compressed keys are not stored in the Bloomier filter. To rehash them to find a new neighborhood the original values have to be stored. This contradicts the idea to save memory. Another solution would be to compute first all keys for a Bloomier filter and create the Bloomier filter at the time it is known how many elements should be inserted. The issue with this solution is again the contradiction with the idea to save memory. If the key/value pairs need to be stored and than inserted into the Bloomier filter the peak memory usage would be at least the same as if no Bloomier filter is used at all. Another issue is not solved by the Bloomier filter at all: Not the number of hash values and in terms of the Bloomier filter the keys are the main factor in the memory usage, the associated instances are it. A Bloomier filter would only compress the keys but the associated lists with the instance where a hash values was appearing would still have the same size. In a worst case scenario all instances would have a different hash value per hash function and every list would have just one element. But this distribution is not the case. For example in Figure 7.30 it can be seen that there are more than 10^5 hash values with a list size of one, but the majority is having more than one instance per hash value. This leads to the realization that even if the Bloomier filter would not have all the problematic issues mentioned above, the compression of the keys would not have a major influence to the needed memory. Furthermore for an implementation it is not possible to use random m-bit size vectors per Bloomier filter position. The smallest possible data type in C++ is for example a char with 8-bits, all values of m would need to be round up to the next multiple of 8.

5.1.1.2 Fitting and Query Times

The fitting time for the unordered map was 2.29 seconds, the Bloomier filter used 3.12 seconds. The query time was 0.94 seconds for the unordered map and 1.78 seconds with the Bloomier filter. The computation of the Bloomier filter version takes longer because per access to an element k values needs to be hashed to get the correct index values for the xor operation, and then these k values needs to be xor'ed. Now the result can be taken to return the value out of the second table. This is different from the unordered map implementation. Here it is enough to access the element in constant time and this is returning the associated values.

5.2 Memory saving techniques

5.2.1 Pruning of hash functions

The idea behind removing complete hash functions is that some hash function do not influence the result significant. First the can store values which are more or less the same than in the other hash functions which makes a hash function redundant. Second a hash value can contain for the hash values only instances which will not influence the order of the instances per hit or third simply contribute only instances which are not considered in any other hash function. To remove complete hash functions from the inverse index the following is done: After the the inverse index is built, the number of hash values per hash function are counted. After it, hash functions containing less then n instances are removed. How big a hash function should be can be user defined or every hash function which less than the mean + standard deviation are removed.

5.2.2 Pruning of hash values

Pruning hash values out of a hash function is done to remove hash values which do not contain at least n instances. These hash values usually do not contribute much to the number of hits per instance but appear often.

5.2.3 Prune during the fitting process

The idea to prune the inverse index during the fitting process is to reduced the amount of maximal memory usage. If the inverse index is pruned after the fitting process to a smaller size the maximal amount of used memory is the same.

5.2.4 Store hash values with least significant n-bits equals zero

This is one of the most effective techniques to reduce the the amount of stored values. Only hash values with the least n significant bits equals to 0 are inserted in the inverse index. This method helps that the amount of used memory is not growing that much.

5.2.5 Compressing the signature

Compressing signatures means to combine the hash values of multiple hash functions to one. There are three parameters for this technique: the number of hash functions n, the block size m and the shingle size k. The block size is a multiplication factor for the number of hash functions, this means n * m hash functions are computed. After this the number of hash values are reduced by the factor of k shingles. It means that k consecutive hash values are combined to one. This leads to a signature size of $n * m/k$.

6 Implementation

The implementation of the nearest neighbor search was designed to have a compatible interface to scikit-learns brute force nearest neighbor search algorithm. The first idea was to write the algorithm completely in Python. But it turned out that Python is way to slow to write a serious alternative. Furthermore Python is behaving different as expected and needed. The used hash function from Thomas Wang[1] uses bit shifting. In C/C++ a left bit shift will throw away the highest significant bit all others shift to left and the least significant bit is 0. Python is having an unlimited precision for long integers[2]. This leads to the effect if a normal 32-bit integer is left shifted that there are not 32-bits after the shift but 33-bit in case of a range of one. In practice this means that the data type is not the right one anymore and it needs to be reassigned. This slows down the computation drastically and second the result of the hashing is different. Because of this behavior the algorithm is implemented with C/C++ and an interface for Python is provided. This works the following: the data is given to the Python interface and is parsed to C with Pythons C-API[3]. Within C/C++ the approximate nearest neighbors are computed and are parsed back to Python.

6.1 Interface

There are four python interfaces: MinHash, MinHashClassifier, WtaHash, WtaHashClassifier and MinHashClustering

6.1.1 MinHash

A MinHash object is created as shown in Figure 6.1. First the algorithms needs to be imported from the bioinf_learn package (line 2). After it, the objects needs to be created and the parameters listed in Figure 6.1.1 are possible (line 7), the dataset needs to be fitted (line 8) and than the k-nearest neighbors can be computed (line 10).

[1]Integer Hash Functions, 1997 / 2007 https://gist.github.com/badboy/6267743, accessed: 2016-05-24

[2]https://docs.python.org/2/library/stdtypes.html#numeric-types-int-float-long-complex, accessed: 2016-05-24

[3]https://docs.python.org/2/c-api/index.html, accessed: 2016-05-24

```
from bioinf_learn.util import create_dataset
from bioinf_learn.neighbors import MinHash
# create a sparse dataset with 5 centroids, 100 instances, 1000 features, 1% non-zero features and a noise of 20%
dataset, _ = create_dataset(seed=1, number_of_centroids=5, number_of_instances=100,  number_of_features=1000,
                         size_of_dataset=10, density=0.01,fraction_of_density=0.2)
# fit the dataset
n_neighbors_minHash = MinHash(n_neighbors = 4)
n_neighbors_minHash.fit(dataset)
# get the n_nearest neighbors with the approximate algorithm
print n_neighbors_minHash.kneighbors(fast=True)
# get the n_nearest neighbors with the exact algorithm
print n_neighbors_minHash.kneighbors()
```

Figure 6.1: Trivial example for MinHash

The following parameters can be used for the initialization of the object:

- n_neighbors = 5: This parameter defines how many k-nearest neighbors should be searched by default. This value is used if no other value is defined by the call of the k-neighbors-function.
- radius = 1.0: This parameter defines the range within an instance should be considered for the nearest neighbor search in case of a range search with the function radius_neighbors.
- fast=False: This parameter defines if the fast version of the algorithm should be used or the more accurate but slower one. See chapter 7 for results.
- number_of_hash_functions=400: This parameter defines how many hash functions should be used for MinHash. The number of used hash functions is equal to the number of reduced dimensions. Default value is 400. For this value it is recommended that it is not less than 150 and not more than 1000. Starting with about 1000 hash functions the results converge usually. With less than 150 hash functions too less hits are computed in the inverse index and bad or even worse no result can be computed.
- max_bin_size = 50: This value defines how many instances a hash value for a hash function should have at maximum in the inverse index. The idea here is that too popular values do not contribute anything to the result.
- minimal_blocks_in_common = 1: This parameter defines how many hits a value should have at least after the inverse index was queried to be considered in the further computation.
- shingle_size = 4: This parameter defines how many contiguous values of a signature should be merged to one. Not used if the parameter 'shingle' is set to '0'.
- excess_factor = 5: This parameter defines the factor how many instances should be considered after the querying of the inverse index. For example if the 5-nearest neighbors are searched, for each instance 5 * excess_factor many values are considered and used as candidates. In the second candidate selection round for each instance k*(k+excess_factor) candidates are considered.
- similarity=False: This parameter defines if the cosine similarity (true) or the euclidean distance (false) should be used as a measurement.

- number_of_cores=None: Here it can be defined how many cores by openMP should be used. If set to 'None' all available cores are used.
- chunk_size=None This parameter defines how many elements per thread from openMP should be taken. If it is set to 'None' the default behavior of openMP which is splitting the to be computed elements equally to the threads is used.
- prune_inverse_index=-1: This parameter defines if and how many hash values from the inverse index should be pruned. This means for example if its set to 10 that all hash values with less than 10 instances are deleted from the inverse index. If this value is set to '-1' it is deactivated.
- prune_inverse_index_after_instance=-1.0: This parameter defines if the inverse index should be pruned after x% of the fitted data or not (value -1 or 0.) If it is for example 0.1 it means that the inverse index should be pruned after 10%, 20%...90% of the data. This value should not be greater than 1.
- remove_hash_function_with_less_entries_as=-1: This parameter defines if a hash function with less hash values than n should be deleted. If it is set to '-1' it is deactivated. If it is set to '0' all hash functions with less hash values as the mean + standard deviation over all hash functions are removed.
- block_size = 5: This parameter defines how many additional MinHash values should be computed for the signature. If k hash functions are used, a block_size of m and a shingle_size of n, the size of the signature will be $(k * m)/n$. This parameter is ignored if the parameter 'shingle' is set to '0'.
- shingle=0: This parameter defines if MinHash values in the signature should be merge together (set it to 1) or not (set it to 0). The parameters 'block_size' and 'shingle_size' are influencing the size of the signature.
- store_value_with_least_significant_bit=0: This parameter defines that only hash values which are having the n-least significant bits 0 are stored in the inverse index. All other values are thrown away. If it is set 0 it is deactivated.
- cpu_gpu_load_balancing=0: This parameter defines if the parts of the computation should be supported by the GPU (1) or CPU only (0).
- gpu_hashing=0: If the hashing should be computed on the GPU (1) or not (0).
- speed_optimized=None: This parameter defines if values for the parameters above computed with a hyper-parameter optimization should be used. The goal here was to reach at least an accuracy level of 0.7 and to speed up the computation. If this parameter is used, all other parameters are ignored. Can not be used with the parameter 'accuracy_optimized' at the same time.
- accuracy_optimized=None: This parameter defines if values for the parameters above computed with a hyper-parameter optimization should be used. The goal here was to reach at least an accuracy level of 0.9. If this parameter

is used, all other parameters are ignored. Can not be used with the parameter 'speed_optimized' at the same time.

The following functions are provided, the documentation to each function can be read in the Python doc strings.

- fit(X, y=None)
- kneighbors(X=None, n_neighbors=None, return_distance=True, fast=None, similarity=None)
- kneighbors_graph(X=None, n_neighbors=None, mode='connectivity', fast=None, symmetric=True, similarity=None)
- radius_neighbors(X=None, radius=None, return_distance=None, fast=None, similarity=None)
- radius_neighbors_graph(X=None, radius=None, mode='connectivity', fast=None, symmetric=True, similarity=None)
- fit_kneighbors(X, n_neighbors=None, return_distance=True, fast=None, similarity=None)
- fit_kneighbor_graph(X, n_neighbors=None, mode='connectivity', fast=None, symmetric=True, similarity=None)
- fit_radius_neighbors(X, radius=None, return_distance=None, fast=None, similarity=None)
- fit_radius_neighbors_graph(X, radius=None, mode='connectivity', fast=None, symmetric=True, similarity=None)

6.1.2 MinHashClassifier

The initialization parameters are the same as for MinHash subsection 6.1.1.

- fit(X, y)
- partial_fit(X, y)
- kneighbors(X = None, n_neighbors = None, return_distance = True, fast=None)
- kneighbors_graph(X=None, n_neighbors=None, mode='connectivity', fast=None)
- predict(X, n_neighbors=None, fast=None, similarity=None)
- predict_proba(X, n_neighbors=None, fast=None, similarity=None)
- score(X, y , sample_weight=None)

6.1.3 WtaHash

WtaHash uses the winner takes it all hashing for the dimension reduction. All other things are equal to MinHash, except that the hashing on the GPU is not provided. The object creation interface is having one additional parameter:

- rangeK_wta=10: The first k elements that should be used to select the min value after the permutation.

6.1.4 WtaHashClassifier

The WtaHashClassifier interface is equal to the MinHashClassifier except the one parameter for the initialization of the objection mentioned in the interface description of WtaHash in subsection 6.1.3.

6.1.5 MinHashClustering

MinHashClustering takes two objects as an input: The MinHash object and an object for a clustering algorithm. It precomputes the nearest neighbors with MinHash and uses the result as input for the clustering algorithm. The clustering algorithm needs to support the parameter 'precomputed' as input.

6.2 GPU support

To speed up the computation parts of the algorithm can be computed on the GPU. As a framework NVIDIAs CUDA was chosen, which means the GPU support is only available on computers with a NVIDA graphic card. Given the programming model and C as the only programming language the computation of the hash values for MinHash and the computation of the euclidean distances respectively the cosine similarity can be done on the GPU. Important here is that the graphic cards needs to have enough memory otherwise the computation will fail. To program CUDA and as a source of knowledge for the next section the following books were used: 'Cuda by Example' from Sanders and Kandrot [23] and 'CUDA Programming' from Cook [24].

6.2.1 Programming model

To program software for the graphic card is fundamental different than classical, serial CPU programming. A function which is running on the GPU is called a 'kernel' and can be called from the 'host' (the CPU), the graphic card is called 'device'. A kernel is called by defining a number of blocks and for each block a number of threads. Not more than 65536 threads can run in parallel. This parallelism is a pseudo-parallelism only 32 threads per instruction unit can be executed at once, these 32 threads are called a 'wrap'. But the context switch is very fast. Nvidia is using for CUDA the programming model 'single instruction multiple data' (SIMD) which means that all threads from one wrap are executing the same line of code. This is significant different from the CPU. If there are multiple threads on the CPU these threads are executed independently from each other in the sense that each thread can be at a different part of the instruction execution. If i.e. there is an if-else construct this construct would be executed by two threads independently from each other. If one thread goes to the if branch and the other one to the else, both would be executing the specific part of the source code and would terminate. On the GPU this works different. If there would be an if-else construct and for the half of the threads the if condition would be true and for the other half not, still all threads would execute first the if and then the else branch. But the instruction would not be applied on the data for the threads that the if would be false for. This comes because each instruction is applied not to one thread, it is applied to a wrap. To prevent branching is crucial to get a speedup with the GPUs. In theory all blocks and threads can run at the same time but in practice 32 threads per instruction unit run at the same time. This leads to the programming model that always 32, or a multiple of it, threads should be used per block, the data structures should be organized that always the number of threads many elements can be read at once and no boundary checking needs to be performed. It is faster if a few threads are calculating with neutral elements instead to do a boundary checking. To get optimal speed it is necessary that data which should be assigned to consecutive threads is stored consecutive in the memory. Per read from the memory 128-bits can be read, if e.g. one integer with 32-bit is read at random, 3/4 of the bandwidth is not used. But if 4 consecutive threads want to access 4 integers which are stored consecutive in memory, only one read is necessary and the values are distributed to the threads. GPU programming is always a speedup factor if one operation can be applied to many data points without checking any index values e.g. the multiplication of two dense matrices. The multiplication of a sparse matrix is not that fast because the index values need to be checked.

6.3 Installation

To install the software download it from:

`https://github.com/joachimwolff/minHashNearestNeighbors`

and run `python setup.py`. There are additional parameters:

- `--user`: To install it in an user only context
- `--noopenmp`: If the software should not be installed with openMP support.

The installer assumes under Ubuntu that openMP is installed. If this is not the case or openMP should simply not be used set this parameter.

- `--openmp`: Under MAC OS X it is assumed that openMP is not installed by default. If openMP support should be forced, set this parameter
- `--nocuda`: If the installer registers that CUDA is installed on the system, it is compiling the related classes. With this parameter it can be forced that the CUDA classes are not compiled and installed.

6.4 Class diagram

The class diagram shown in Figure 6.2 gives a short overview how the software is implemented. There is a Python interface and from there the data is parsed to C++. Within C++ the computation of nearest neighbors is done. The parts which are providing source which is necessary for the GPU implementation are labeled with the suffix 'Cuda'.

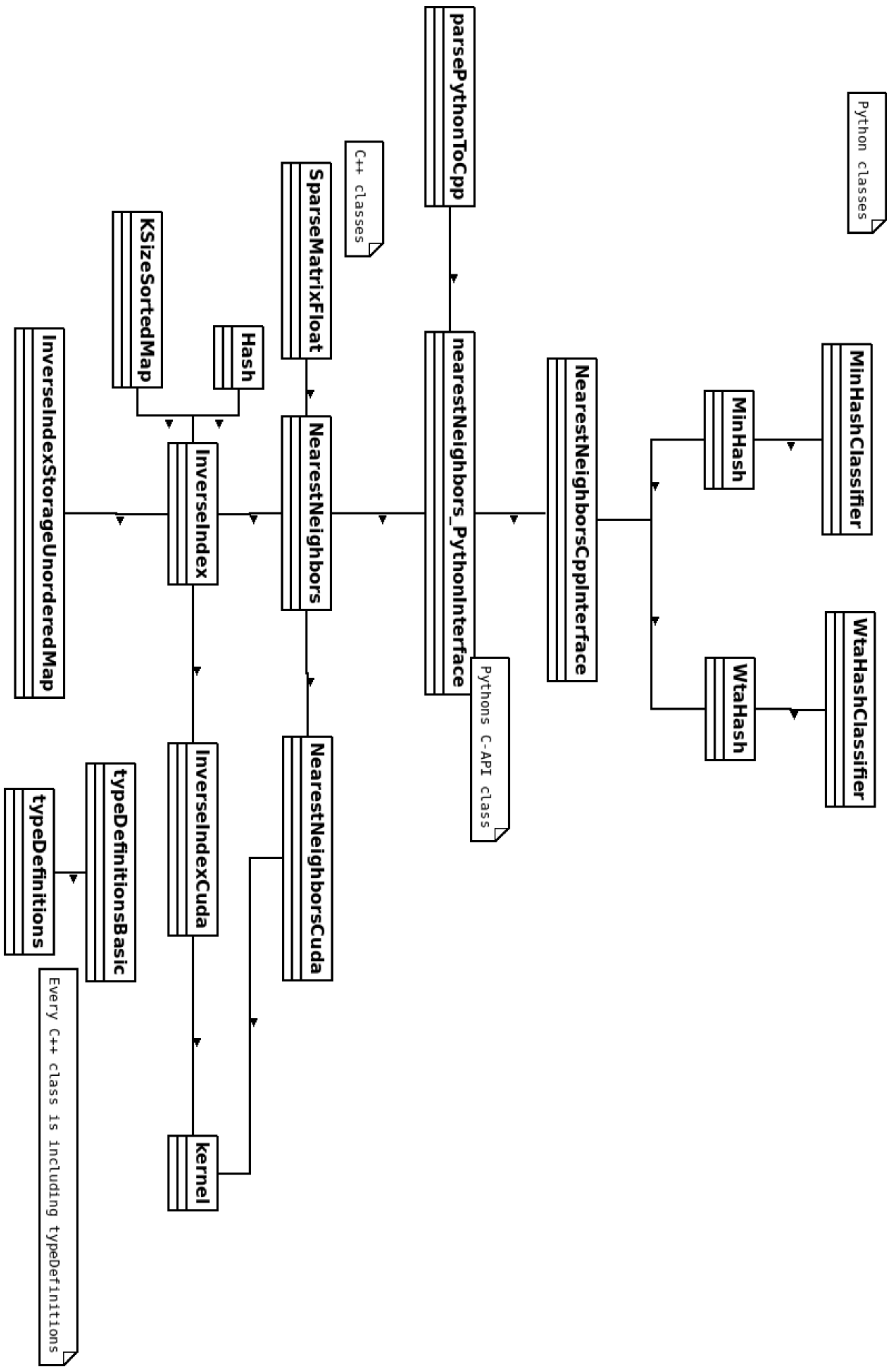


Figure 6.2: The relationship between the different classes.

7 Results

The MinHash and WTA-Hash algorithm are tested on two datasets, Bursi and RNA. Different settings to optimize the speed, accuracy or the memory usage are tested. After this the MinHash algorithm is compared to the brute force algorithm and two other approximate k-nearest neighbor search algorithms: Local sensitive hashing forest and annoy. Furthermore it is examined how the MinHash algorithm and the brute force solution are scaling if up to 64 CPU threads were used and how big the benefit of a todays high-end GPU is. Additional the size of the inverse index is investigated and how big the influence of the different memory saving techniques that were introduced in chapter 5 is. WTA-Hash is not considered because as it can be seen in subsection 7.1.1 or in subsection 7.1.2 that the performance for the accuracy and the prediction runtime are comparable.

7.0.1 Datasets

7.0.1.1 Bursi

Bursi[1] is a dataset that classifies 4337 molecular structures into 2401 mutagens and 1936 non-mutagens. For further information see [25]. Bursi has 1048577 dimensions with 373 non-zero features per instance on average.

7.0.1.2 RNA

The RNA dataset is downloaded from RFAM[26]. RFAM is a database of RNA families, 20 RNA families are evaluated:

'RF00004','RF00005','RF00015','RF00020','RF00026','RF00169',
'RF00380','RF00386','RF01051','RF01055','RF01234','RF01699',
'RF01701','RF01705','RF01731','RF01734','RF01745','RF01750',
'RF01942','RF01998','RF02005','RF02012','RF02034'

These families are used as an input for RNAfold [27] to compute the secondary structure of the RNA. EDeN[2] encodes these secondary structures as a graph and is using subgraphs to transfer the data to a sparse matrix. For further information

[1]http://www.bioinf.uni-freiburg.de/∼costa/bursi.gspan, accessed: 2016-05-24
[2]https://github.com/fabriziocosta/EDeN, accessed: 2016-05-24

see [18]. The parameter setting was 'n_max=50, complexity=3, nbits=16'. This creates a dataset which has 1150 instances, 65537 dimensions and 3670 non-zero features per instance on average.

7.0.2 Used machines

The following results are computed on three computers:

- Intel Core i5-5200U@2.20GHz (2 cores, 4 threads), Nvidia GeForce 920M with 2 GB RAM and 384 CUDA cores, 8 GB RAM. This computer is referred to as 'the slow' computer in the following.
- Intel Core i5-6600@3.30GHz (4 cores, 4 threads), Nvidia GeForce GTX 750Ti with 2 GB RAM and 640 CUDA cores, 16 GB RAM. This computer is referred to as 'the average' computer in the following.
- 2 x Intel Xeon E5-2698 v3@2.3GHz (2x16 cores, 2x32 threads), Nvidia GTX 980Ti with 6 GB RAM and 2816 CUDA cores, 250 GB RAM. This computer is referred to as 'the high-end' computer in the following.

All systems were running with Ubuntu 14.04 and were using Python 2.7, CUDA 7.5, g++ 4.8.4 and ipython 4.0.0.

For a fair comparison between the CPU and the GPU implementation it is important that the hardware is more or less on a same level. To compare e.g. a good old Pentium 4 CPU with a high-end Titan X GPU would be unfair. It is obvious that processors and graphic cards are very different, their architecture is optimized for different tasks and objective measures like FLOPS can not help to compare; i.e. the slightly more powerful i5-6600K (instead of the used i5-6600) is having around 174 GFLOPS[1] and the 750Ti 1305 GFLOPS[2] for single floating point precision. But graphic cards are only faster if the specific task fits good into the programming model. With this in mind the conclusion is that the only fair comparison measure is how much the hardware costs. At the time this thesis is written, May 2016, the used i5-6600 CPU costs around 200 Euro. The used 750Ti GPU costs around 120 Euro, for around 200 Euro a GTX 960 could be bought. This GPU is having 1024 CUDA cores[3] and reaches 5939 points[4] compared to 3684 points[5] of the 750Ti in the 'G3D Mark'-benchmark. Please have in mind: if equal hardware in terms of

[1]https://www.pugetsystems.com/labs/articles/Skylake-S-i7-6700K-and-i5-6600K-for-compute-maybe-697/, accessed: 2016-05-24

[2]http://www.pcgameshardware.de/Grafikkarten-Grafikkarte-97980/Tests/Geforce-GTX-750-Ti-im-Test-Maxwell-1109814/, accessed: 2016-05-24

[3]http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-960/specifications, accessed: 2016-05-24

[4]http://www.videocardbenchmark.net/gpu.php?gpu=GeForce+GTX+960, accessed: 2016-05-24

[5]http://www.videocardbenchmark.net/gpu.php?gpu=GeForce+GTX+750+Ti, accessed: 2016-05-24

the price would be used, the GPU parts of the average computer should be around 40% faster. The following benchmarks are created with multiple ipython notebooks. The runtime of the algorithms on the bash are usually faster than with the ipython notebook but because all benchmarks have to deal with this issue, it should not matter.

7.0.3 Measurements

7.0.3.1 Accuracy

The accuracy measure is measuring how many instances within a k-neighborhood are equal compared to the brute force computation of scikit-learns nearest neighbors.

Definition
Let X be a k-sized approximate neighborhood, $X = \{x_0, ..., x_k\}$ and Y is the k-sized brute force neighborhood, $Y = \{y_0, ..., y_k\}$. The accuracy is given by:

$$accuracy(X, Y, k) = \frac{|X \cap Y|}{k} \tag{7.1}$$

The accuracy of a n-sized query is given by:

Definition
Let A be the set of k-approximate neighborhoods, $A = \{a_0, ..., a_n\}$ and $|A| = n$; B be the set of k-neighborhoods which are computed by brute force, $B = \{b_0, ..., b_n\}$ and $|B| = n$. The accuracy is:

$$accuracy\_query(A, B, k) = \frac{\sum_{i=0}^{n} accuracy(a_i, b_i, k)}{n} \tag{7.2}$$

This definition is less strict than an itemized accuracy but it is used because it is considered more useful to know that the neighbors within a k-neighborhood are correct than that the ordering and the index position is correct too. In the case of an itemized accuracy a shift of the positions by just one would lead to 0.0 accuracy. The error term is given by:

Definition

$$error = 1 - accuracy\_query(A, B, k) \tag{7.3}$$

7.0.3.2 Speed

To measure the speed of an algorithm the runtime is stopped in python. For example if the speed of a query should be tested, the time of the system with pythons 'time.time()' is stopped before and after the call. The difference is taken as the runtime in seconds.

7.0.3.3 Memory

To measure the memory usage, two approaches were used. To measure the memory usage of the inverse index all size_t's within the inverse index's are counted and this number is multiplied by 8 because 8 bytes are used for a size_t in C++. This leads to the memory usage in bytes. The overhead which is appearing for the usage of vectors or maps is ignored. The second measure is how much memory an execution is requesting from the operating system at maximum. It is measured with '/usr/bin/time -v ' and here the parameter 'Maximum resident set size' is listed.

7.0.3.4 Score

The score as a measurement can be used to balance the error term, the speed and the memory usage. The most important factor is the error; memory and time are log'ed to make sure only significant changes are taken into account. The memory usage is here equal to the size of the inverse index. Memory and time can be weighted by the parameters α and β to control the influence of each parameter.

$$score = \begin{cases} \text{error} + log_{10}(\text{memory})\ \alpha + log_{10}(\text{time})\ \beta & \\ \text{penalty score} & \text{if error} >= \text{min error level} \end{cases} \tag{7.4}$$

7.1 Approximate nearest neighbor search

MinHash and WTA-Hash are used with a parameter setting which was computed with a hyper-parameter optimization using the Python software hyperopt[1]. In the following the MinHash and WTA-Hash algorithm are optimized to achieve the highest possible accuracy independent from the runtime. After it, the optimization tries to achieve the best score which means that the highest accuracy should be reached, but also the runtime should be minimized and the size of the inverse index should be as small as possible. The next section is examining if optimizations which were computed for one dataset can be transfered to another one, for this MinHash is compared to competitive algorithms. Furthermore the scalability for multi core computing of MinHash and the brute force algorithm is observed, MinHash and WTA-Hash are then compared to a brute force solution which is operating on a sparse random projected dataset. Last but not least the memory usage of the algorithms is measured and compared.

[1]http://jaberg.github.io/hyperopt/, accessed: 2016-05-24

7.1.1 Best accuracy

To compute the best accuracy value the hyper-parameter optimization optimized only the accuracy value and ignored the runtime and memory usage. The optimization was computed for the 10-nearest neighbors using first the Bursi dataset (see subsubsection 7.0.1.1) and after it the RNA dataset, see subsubsection 7.0.1.2.

7.1.1.1 Bursi dataset

The best accuracy for MinHash and WTA-Hash for the Bursi dataset was reached with the parameters listed in Table 7.1. In Figure 7.1 the comparison of the accuracy with the different versions of MinHash and WTA-Hash is shown. The fast version of the MinHash algorithm reaches almost 0.6 accuracy (red bar), WTA-Hash around 0.5 (blue bar). The non-fast version of MinHash reaches 0.964 accuracy (green + cyan bar) and WTA-Hash 0.923 (purple + yellow bar). There is a slightly difference in the accuracy of the CPU versions compared to the GPU versions. Reasons for this are that first, floating precision leads to slightly different rounding on the different devices which can influence the distance and with this the ordering of the instances. Second, it is also possible that this difference is caused by bugs in the implementation. In Figure 7.2 the runtime of a query is shown. As a query the nearest neighbors of all data points were searched, the computed signatures from the fitting could be taken from memory. It is interesting to see that both, MinHash (second red bar) and WTA-Hash (blue bar) are outperforming the brute force algorithm with their fast versions. This is independent what hardware is used. For the non-fast case it depends on which hardware the algorithm is running. If a slow GPU like the used 920M is used, the computation time increases in comparison to the CPU version (see Figure 7.2a green vs cyan bar). But if average hardware like the 750Ti is used, the performance is almost the same, see Figure 7.2b green vs cyan bar). The fitting time is shown in Figure 7.3. It is obvious that the GPU version of MinHash benefits from the high parallelism and reduces the computation time drastically (red vs. cyan bar). This is independent from the used hardware. The fitting time of WTA-Hash is much longer as the MinHash fitting time. This is caused by the more complex computation especially the ordering of the lowest hash values and after it the search for the hash value with the highest original value. These steps are not necessary for MinHash, it is enough to get the minimum hash value. This is one reason why there is no GPU version of the WTA-Hash, the mentioned steps do not fit well into the programming model of a GPU.
To achieve a high accuracy many hash functions are necessary and only hash values with less instances as 1 should be removed. At the same time only a quarter (MinHash) or the half (WTA-Hash) of the created hash values are stored. A high excess factor of 14 for both algorithms shows that a few instances which are necessary to achieve a high accuracy are not even close in the k-neighborhood after the first candidate selection on the inverse index. An excess factor of 14 means that for a

k-neighborhood $k * 14$ instances are taken as candidates. As a base for this benchmark the 10-nearest neighbors were searched, per instance 140 candidates for the first candidate selection are considered. This is usually equal to consider all found candidates, the average number of candidates after the inverse index was searched is about 80.

Parameter	MinHash	WTA-Hash
number_of_hash_functions	903	916
max_bin_size	49	46
shingle_size	2	1
excess_factor	14	14
prune_inverse_index	1	0
prune_inverse_index_after_instance	0.0	0.5
remove_hash_function_with_less_entries_as	0	0
shingle	1	1
block_size	4	3
store_value_with_least_significant_bit	2	1
rangeK_wta	-	16

Table 7.1: Parameter setting for the best accuracy on Bursi.

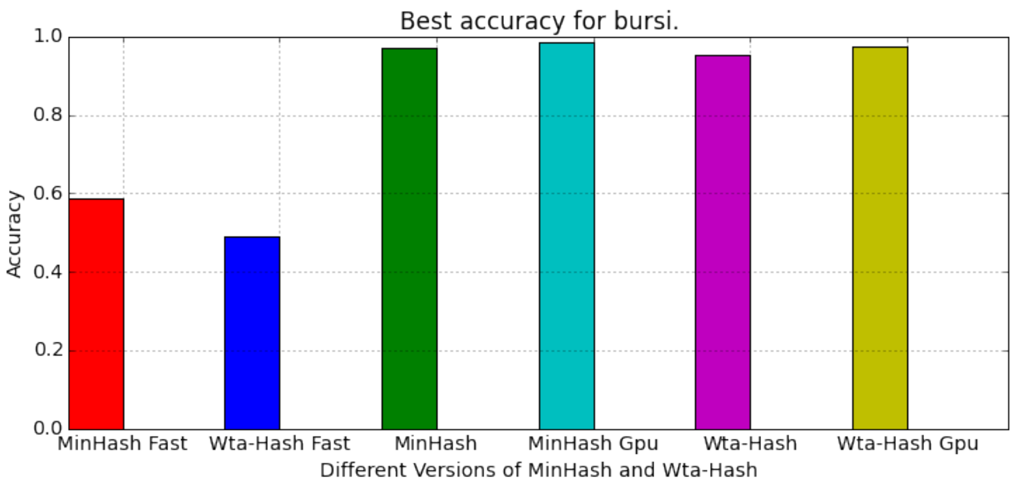
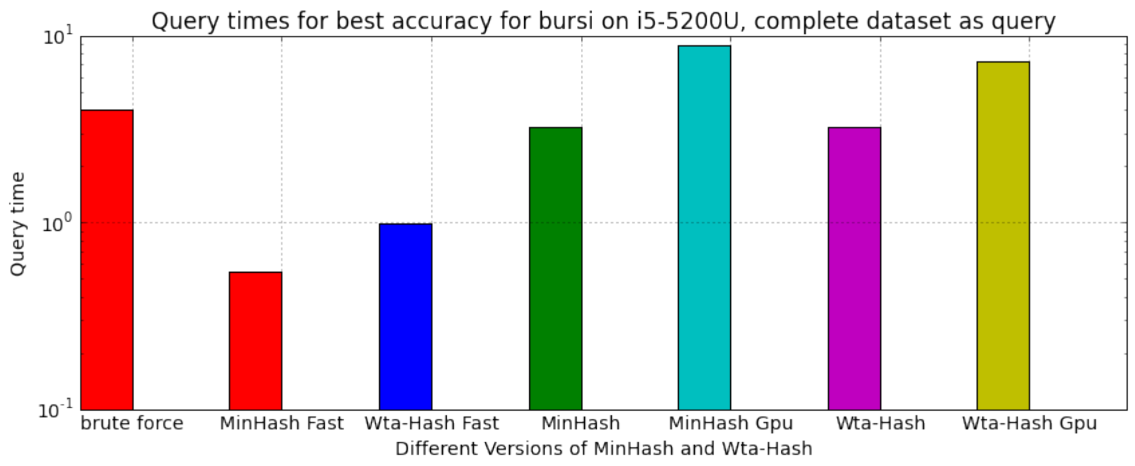
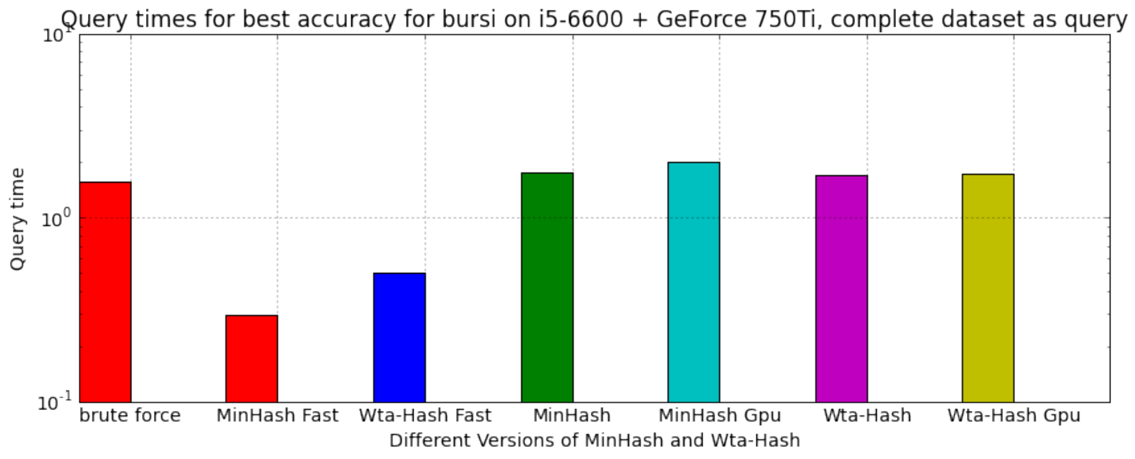


Figure 7.1: Best accuracy on Bursi for MinHash and WTA-Hash.

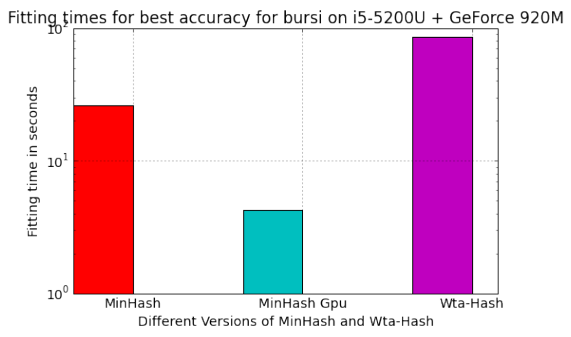


(a) Slow computer

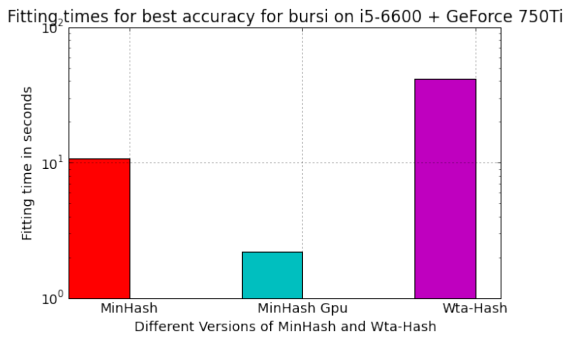


(b) Average computer

Figure 7.2: Query time on Bursi dataset for best accuracy.



(a) Slow computer



(b) Average computer

Figure 7.3: Fitting time on Bursi dataset for best accuracy.

7.1.1.2 RNA dataset

On the RNA dataset the parameters for MinHash are looking similar, see Table 7.2, the same high excess factor for WTA-Hash is used. Interesting is that for WTA-Hash the merging of hash values (shingle) is deactivated. The maximal accuracy with 0.83 is less; WTA-Hash is achieving an accuracy of 0.84. The influence of the GPU fitting phase is higher than for the Bursi dataset. About one magnitude less fitting time on the slow and average computer shows that the fitting task fits perfectly to the GPU programming model, see Figure 7.5 red vs cyan bar. For the queries it is the first time that the GPU version is faster as the CPU version, see Figure 7.6b green vs cyan and purple vs yellow bar. If a faster GPU like the one from the high-end computer could be used the benefit should be higher. But it needs to be considered that the GPU version on the slow computer is much slower than the CPU version, see Figure 7.6a e.g. green vs cyan bar for MinHash. Furthermore it should to be noticed that the brute force implementation is faster as the MinHash and WTA-Hash implementation in the non-fast case.

Parameter	MinHash	WTA-Hash
number_of_hash_functions	828	739
max_bin_size	51	30
shingle_size	2	3
excess_factor	11	14
prune_inverse_index	0	1
prune_inverse_index_after_instance	0.0	0.5
remove_hash_function_with_less_entries_as	0	0
shingle	1	0
block_size	4	1
store_value_with_least_significant_bit	1	2
rangeK_wta	-	23

Table 7.2: Parameter setting for the best accuracy on RNA dataset.

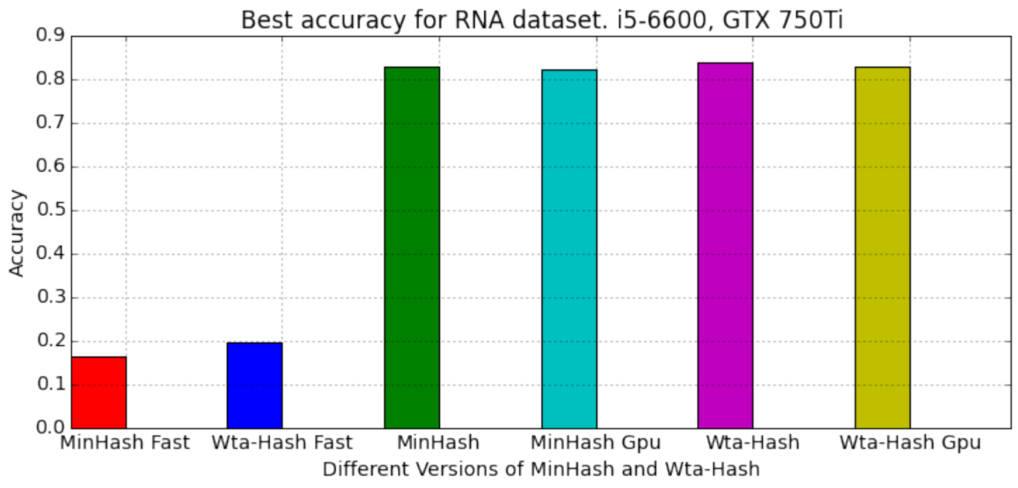
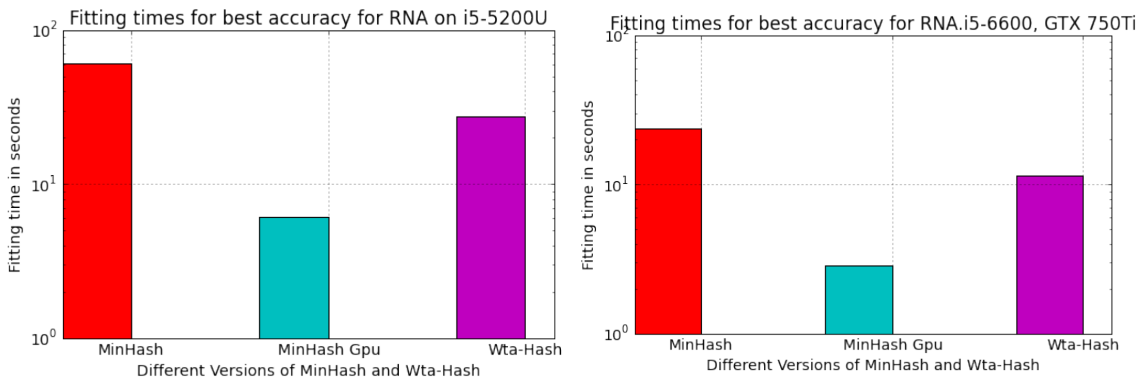
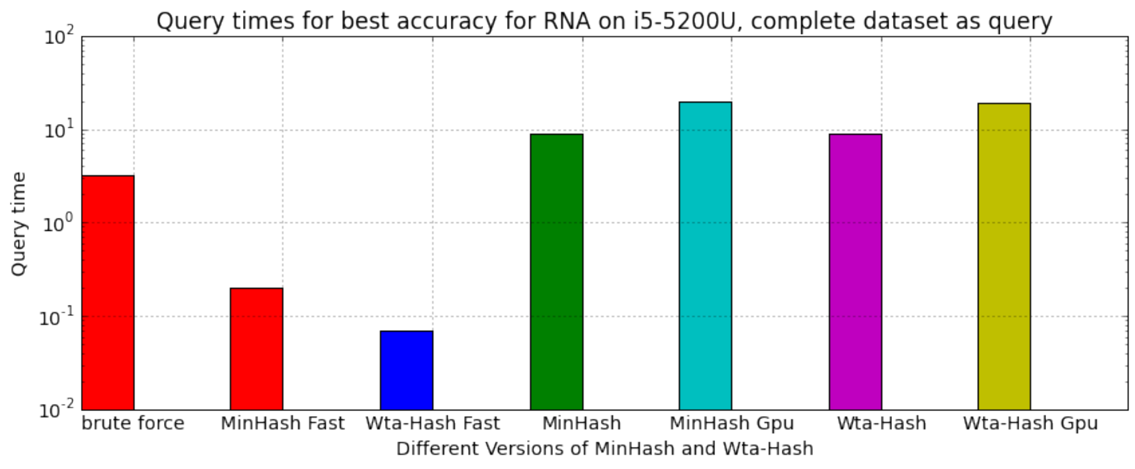


Figure 7.4: Best accuracy on RNA for MinHash and WTA-Hash.

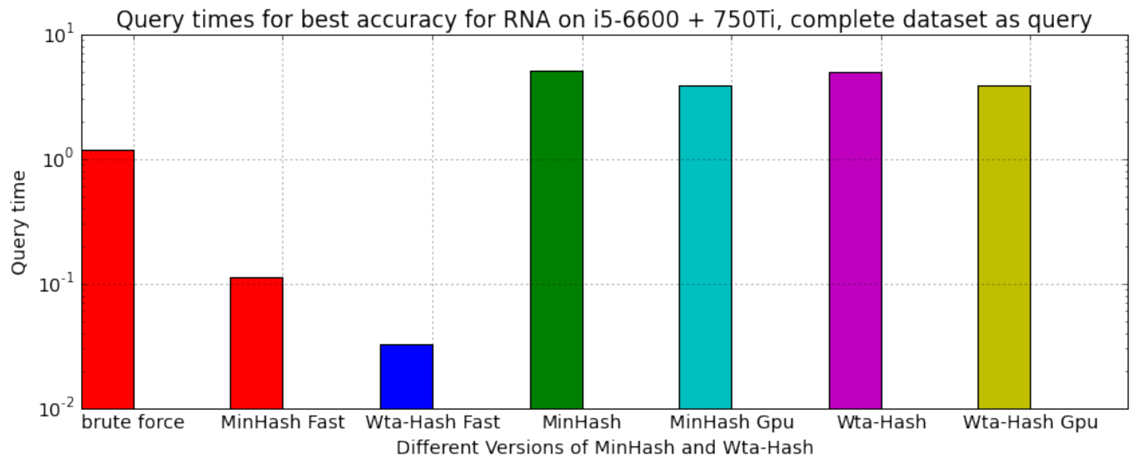


(a) Slow computer **(b)** Average computer

Figure 7.5: Fitting time on RNA dataset for best accuracy.



(a) Slow computer



(b) Average computer

Figure 7.6: Query time on RNA dataset for best accuracy.

7.1.1.3 Conclusion

To achieve a high accuracy level it needs a large number of hash functions, it is a benefit to activate the shingling but on the same time the hash values with a size less than n should not be pruned or only for a value of 1. For non-zero feature ids of 300 to 400 on average per instance like it is on Bursi the MinHash and WTA-Hash implementations are on the same level with the brute force algorithm. But if the situation is like on the RNA dataset with 4000 non-zero feature ids, the brute force implementation is performing significant better. The usage of the GPU is a benefit for the fitting, for the query part it depends which GPU is used and how fast the CPU is. As it can be seen later in Figure 7.17 not only a high-end GPU is necessary to outperform the brute force implementation the CPU needs a high single thread speed too. This is caused by the fact that some parts in the MinHash and WTA-

Hash algorithm of the GPU version are implemented as a single thread version, e.g. if the computed values on the GPU are transfered back and need to be parsed to the data format which is used on the CPU side. As it can be seen in Figure 7.1 vs Figure 7.4 the accuracy for the RNA dataset is lower (0.97 to 0.83). This should be caused by the way MinHash and WTA-Hash are designed. Both compute a signature out of the given non-zero feature ids of an instance. If two instances are sharing the same non-zero feature ids the signature will look the same and the instances are not distinguishable for the algorithm. If the shared non-zero feature ids of two instances are not the same but still are sharing the most ones, it is likely that the signatures will look more or less equal and again, the instances are hard to distinguish by the algorithm. Bursi is having 373 non-zero features on average in a bit more than a million dimensions which leads to a sparsity of 0.00035; the RNA dataset is having 3670 non-zero features on average in 65537 dimensions which leads to a sparsity of 0.056. It is for the RNA dataset more likely that instances can not be distinguish by the algorithm than it is for the Bursi dataset. The more sparse a dataset is the better are MinHash and WTA-Hash performing. But be careful: if one instance is sharing more or less no feature ids with other instances, MinHash and WTA-Hash can not find any neighbors.

7.1.1.4 Take home message

For a high accuracy level a lot hash functions should be used and not too much values should be removed out of the inverse index. The accuracy level that can be reached is depended on the dataset, the sparser it is, the better MinHash and WTA-Hash are performing.

7.1.2 Best score

In this section it is examined how the parameters look if the hyper parameter optimization for MinHash and WTA-Hash optimizes for the best score. The score is defined in Equation 7.4. As it was in the best accuracy section, subsection 7.1.1, the optimization is computed for the 10-nearest neighbors first on Bursi and second on the RNA dataset.

7.1.2.1 Bursi dataset

The hyper parameter optimization for MinHash and WTA-Hash was executed with $\alpha = 0.1$ $\beta = 0.23$. The resulting parameters are listed in Table 7.3. MinHash and WTA-Hash achieving the same accuracy level of about 0.92, the fast version is with less than 0.4 around 20% less accurate in comparison to the highest accuracy, see Figure 7.7 vs. Figure 7.1. The fitting support of the GPU is a benefit (Figure 7.8) for MinHash but the computation of the prediction takes longer with GPU support on the slow computer (Figure 7.9a green vs. cyan bar), on the average computer it is a minimal benefit to use the CPU. The parameters for the shingle_size and block_size can be ignored because shingle is set to 0 and deactivated. With the computed parameter setting a score of 0.54 and an accuracy of 0.92 for MinHash and a score of 0.73 with an accuracy of 0.91 for WTA-Hash is reached. The higher score for WTA-Hash is caused by the higher memory usage. As it can be seen in Table 7.3 MinHash is pruning all hash values with less instances as 11, WTA-Hash removes only the values with a size less than 6. Second reason is that MinHash is storing only 1/8 of the created hash values, WTA-Hash stores 1/2. Mentionable is that the accuracy in the fast case for MinHash and WTA-Hash differs not much although MinHash is storing less values in the inverse index.

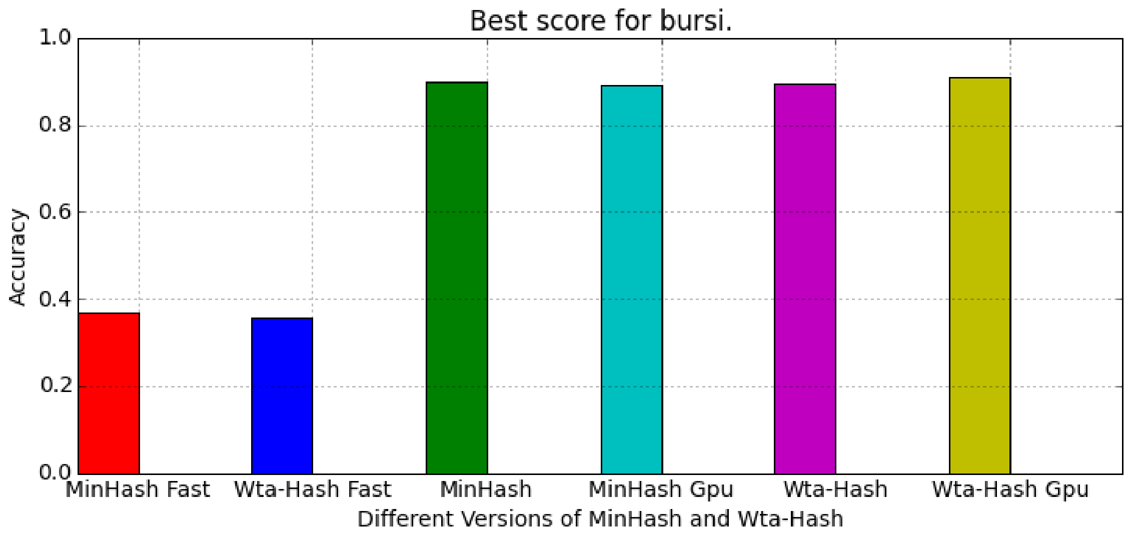
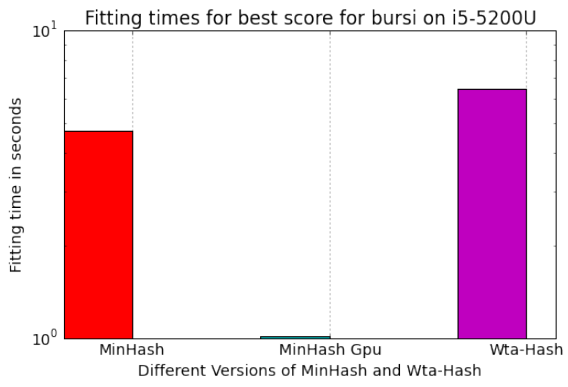
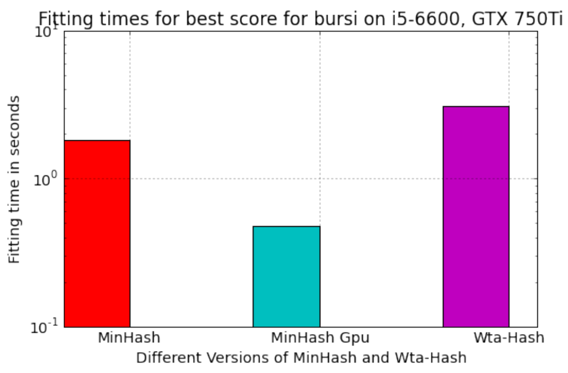


Figure 7.7: Best score on Bursi for MinHash and WTA-Hash.

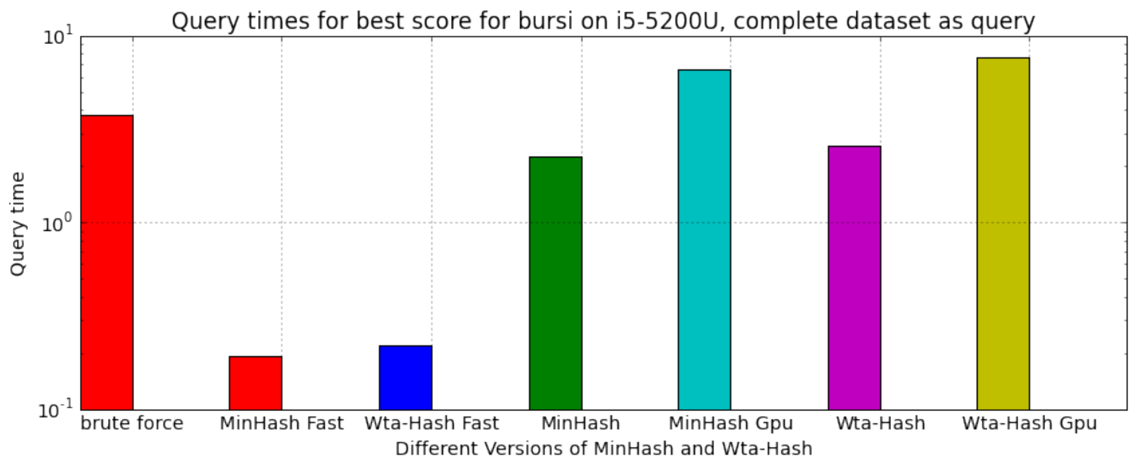


(a) Slow computer

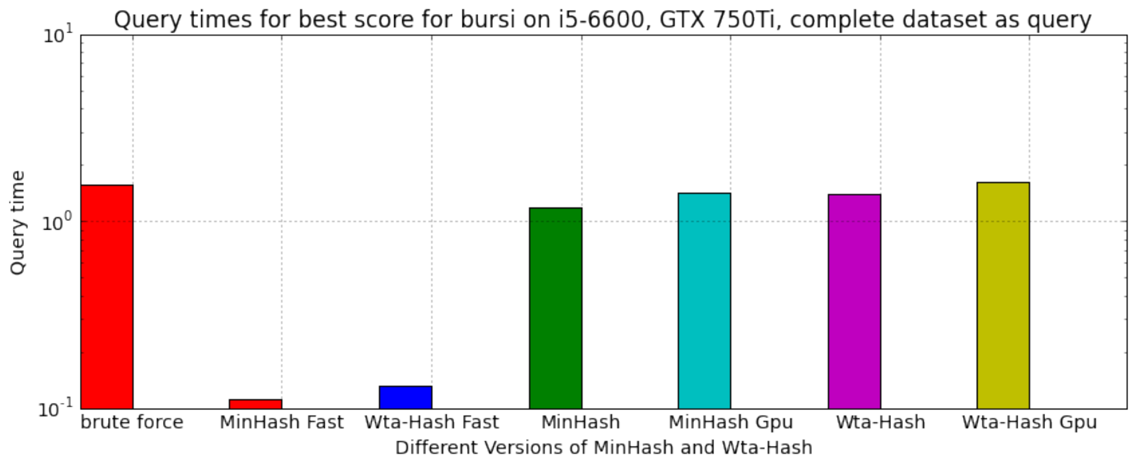


(b) Average computer

Figure 7.8: Fitting time on Bursi dataset for best score.



(a) Slow computer



(b) Average computer

Figure 7.9: Query time on Bursi dataset for best score.

Parameter	MinHash	WTA-Hash
number_of_hash_functions	596	186
max_bin_size	49	87
shingle_size	-	-
excess_factor	11	11
prune_inverse_index	11	6
prune_inverse_index_after_instance	0.5	0.0
remove_hash_function_with_less_entries_as	0	0
shingle	0	0
block_size	-	-
store_value_with_least_significant_bit	3	1
rangeK_wta	-	17

Table 7.3: Parameter setting for the best score on the Bursi dataset.

7.1.2.2 RNA dataset

For the best score on the RNA dataset the hyper-parameter optimization for MinHash and WTA-Hash was running with $\alpha = 0.1$ $\beta = 0.5$. The time was weighted higher because the runtime on the RNA dataset is, compared to the brute force algorithm, worse. To get a parameter configuration which is optimized for speed and not for memory usage was more desirable. MinHash reaches an accuracy of almost 0.8, needs 11529 elements in the inverse index and is having a score of 1.01. WTA-Hash reaches about 0.75 accuracy, needs 47510 elements in the inverse index and is having a score of 1.11. The higher score should be caused by the higher memory usage. WTA-Hash is using more memory because it needs more hash functions, 168 vs. 100 for MinHash and instead of pruning hash values with less instances as 14, it just prunes 2. The query time is on the RNA dataset with GPU support slower if a slow GPU like the one from the slow computer is used. With an average GPU it is slightly faster, see Figure 7.12 purple and yellow bar. The usage of the GPU for the fitting is on the RNA dataset a benefit too. Both algorithms are slower than the brute force implementation of scikit-learn. For both algorithms the merging of hash values is deactivated and it is interesting to see that MinHash is pruning hash values with less size of 14 and at the same time storing only 1/4 of the created hash values.

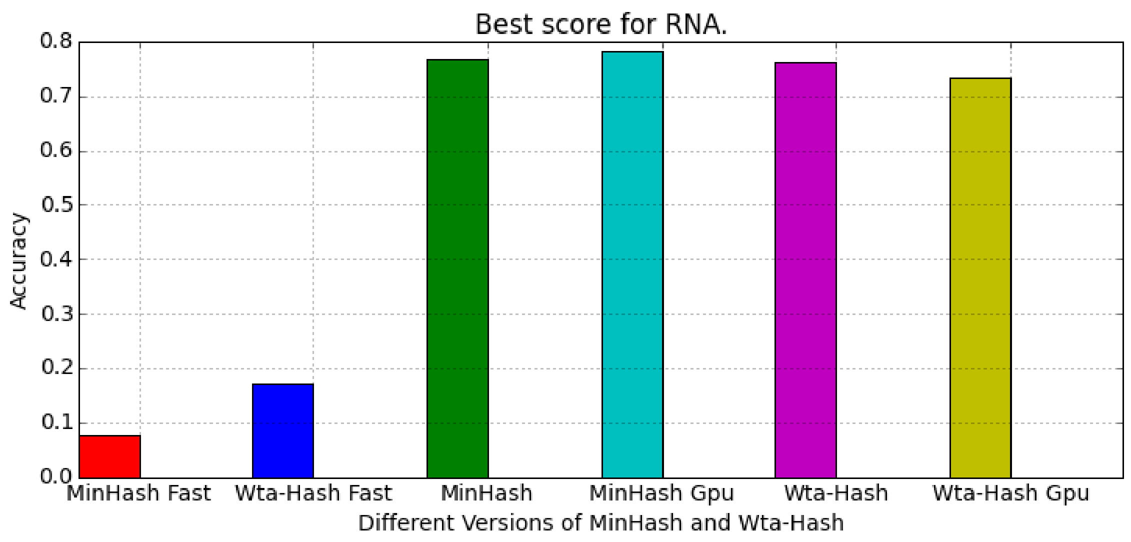
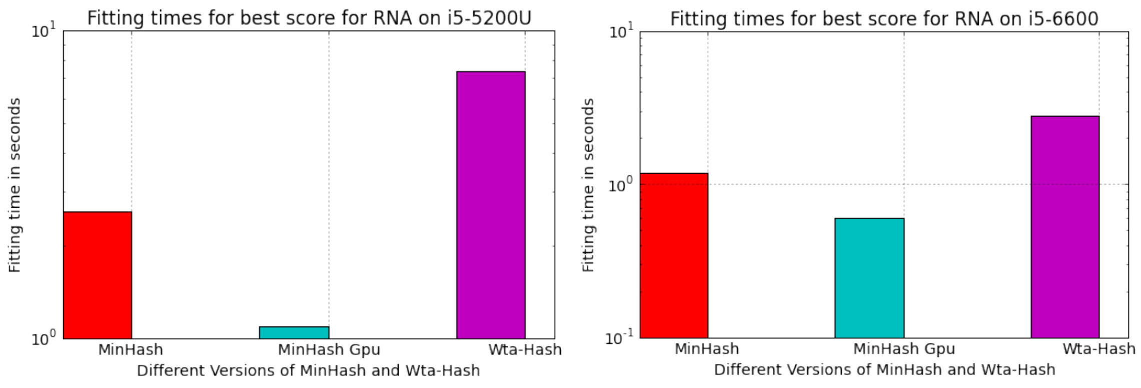
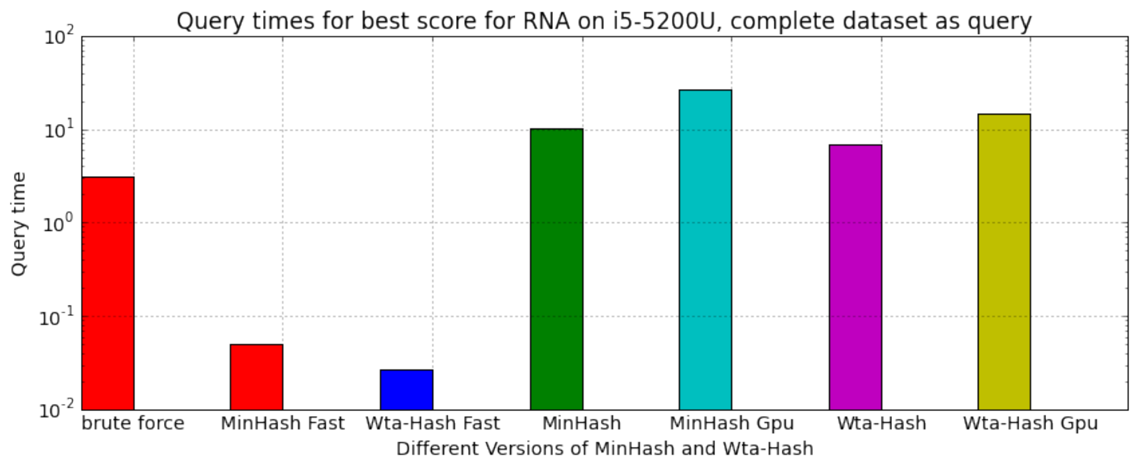


Figure 7.10: Best score on RNA for MinHash and WTA-Hash.

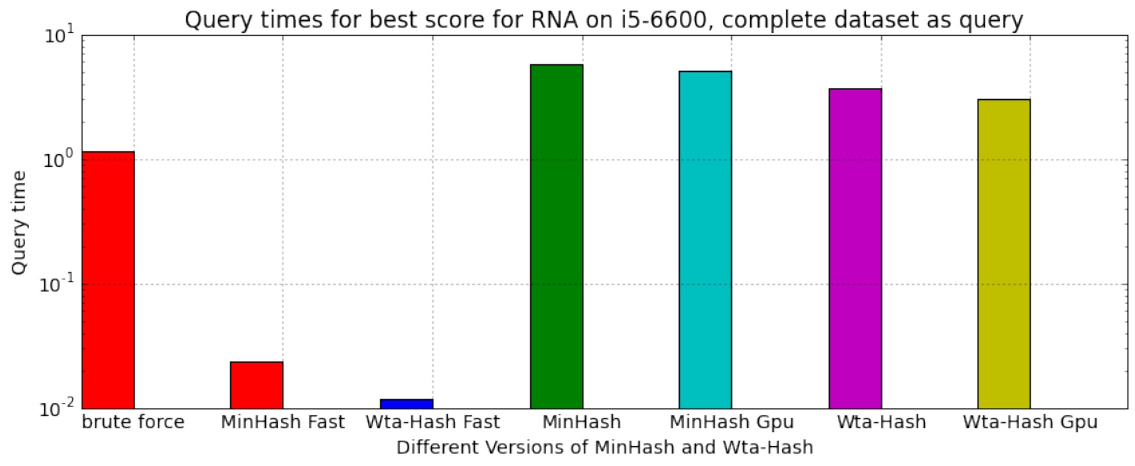


(a) Slow computer **(b)** Average computer

Figure 7.11: Fitting time on RNA dataset for best score.



(a) Slow computer



(b) Average computer

Figure 7.12: Query time on RNA dataset for best score.

Parameter	MinHash	WTA-Hash
number_of_hash_functions	100	168
max_bin_size	90	47
shingle_size	-	-
excess_factor	13	11
prune_inverse_index	14	2
prune_inverse_index_after_instance	0.5	0.5
remove_hash_function_with_less_entries_as	0	0
shingle	0	0
block_size	-	-
store_value_with_least_significant_bit	2	1
rangeK_wta	-	19

Table 7.4: Parameter setting for the best score on the RNA dataset.

7.1.2.3 Conclusion

The best score for both dataset comes only with a small higher error but the fitting time compared to the highest accuracy is about one magnitude (10^1 vs 10^0) faster, see Figure 7.3b and Figure 7.8b. The query time is on the Bursi dataset for the best accuracy slower as the brute force implementation, for the best score the MinHash and WTA-Hash are faster. On the RNA dataset the brute force implementation is for the prediction always faster.

7.1.2.4 Take home message

It is possible to run MinHash and WTA-Hash with not that many hash functions and still getting a high accuracy as it can be seen at the RNA dataset.

7.1.3 Comparing with related algorithms

In this section the algorithms and implementations mentioned in the introduction should be compared to MinHash and WTA-Hash. For not all algorithms was this possible. Algorithms like the kd-tree and ball-tree based solutions from scikit-learn do not accept a sparse data matrix as an input. If called with one an error message is displayed and a fall back mode to the brute force algorithm is executed. A dimension reduction to a dense dataset with the Gaussian random projection[1] from scikit-learn is e.g. for the Bursi dataset not possible because it is simply terminating with the

[1]http://scikit-learn.org/stable/modules/generated/sklearn.random_projection.GaussianRandomProjection.html, accessed: 2016-05-24

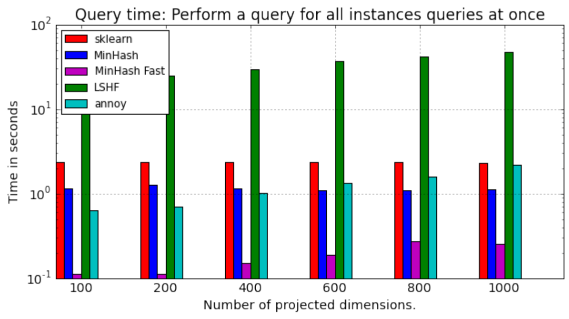
error ’memory error’. Flann and Panns accept sparse matrix inputs in the first place but are failing in the fitting process because they can not handle the data. An algorithm like RPForest can not be used because of its massive memory usage. Even if the input matrix is reduced by a sparse random projection down to 100 dimensions, a configuration with a leaf size of 3 and 5 trees needs more than 16 GB RAM plus 16 GB swap and its terminated by the operating system.
To test the algorithms which accept sparse input matrices but need too much main memory to compute, like annoy or LSHF, the dataset is sparse random projected to values between 100 and 1000 dimensions. These dimensions are equal to the number of used hash functions for MinHash and WTA-Hash. As it can be seen in subsection 7.1.1 MinHash and WTA-Hash perform similar for the prediction, WTA-Hash is significant worse for the fitting. Because of this similar behavior for the prediction runtime and accuracy and the better performance for the fitting process only MinHash is compared to the brute force implementation of scikit-learn, annoy and the local sensitive hashing forest.
The idea to use a hyper-parameter optimization for the parameters mentioned in chapter 6 was designed because the approximate nearest neighbor search is very sensitive to the parameter values. It can happen that one wrong parameter value can make the difference between an accuracy level of 0.95 to just 0.3 or even worse to 0.0. To prevent this, a hyper-parameter optimization was running on the dataset ’bursi’ and in a next step these parameter values are tested on the RNA dataset. With this it is examined if the parameter setting is transferable to other datasets. The optimization was done for the 10-nearest neighbor search. For the hyper-parameter optimization the prediction runtime was used as an objective function as long as the defined accuracy level was reached on the Bursi dataset. In the following the brute force algorithm is always executed on the original dataset, not on a projected one. It appears for a better comparison for every projected dimension again.
Annoy was used with 100 trees, local sensitive hashing forest with 200 trees and 20 estimators.

Parameter	0.95	0.9	0.7
number_of_hash_functions	800	600	200
max_bin_size	45	86	54
shingle_size	-	-	4
excess_factor	10	7	8
prune_inverse_index	14	10	0
prune_inverse_index_after_instance	0.0	1.0	0.0
remove_hash_function_with_less_entries_as	0	0	0
shingle	0	0	1
block_size	-	-	4
store_value_with_least_significant_bit	0	2	1

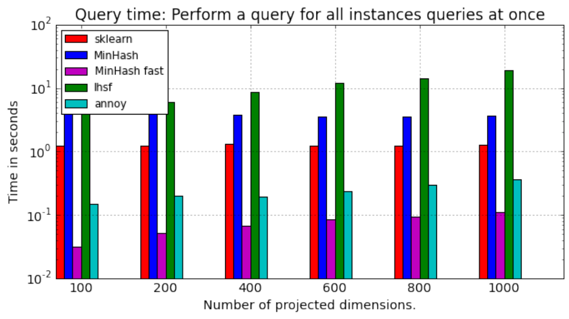
Table 7.5: Parameter setting for the accuracy level 0.95, 0.9, 0.7 with the fastest query time.

7.1.3.1 Accuracy level 0.95

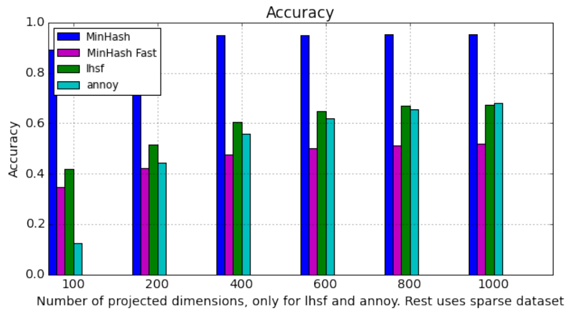
Figure 7.13a shows that MinHash fast is the fastest (purple), the slow version (blue) is slower but faster as the brute force algorithm and with more dimensions faster than annoy. The local sensitive hashing forest is by far the slowest algorithm (green). A look to the other dataset gives us a slightly different picture. For the RNA dataset the MinHash Fast algorithm is still the fastest but the slow version is significant slower than the brute force algorithm and annoy. The issue here is that the brute force algorithm is using the highly optimized library BLAS which is written in Fortran. The fitting time for MinHash (Figure 7.13e and Figure 7.13f) benefits from the GPU (yellow) compared to the non-GPU version (blue); with the GPU support the fitting times are similar to annoy, LSHF is having the fastest fitting time. The brute force algorithm does not do any fitting, it is just setting a reference to the dataset. The accuracy level that was reached for Bursi could not be reached for the RNA dataset but the difference is not that bad if it is compared to the highest possible accuracy that was achieved in subsubsection 7.1.1.2.
A look at the parameter setting, Table 7.5, shows that hash values with less instances as 14 were pruned but at the same time all created hash values are stored. The main difference to the highest possible accuracy is that no shingles were used which is reducing the computation time for the fitting process.



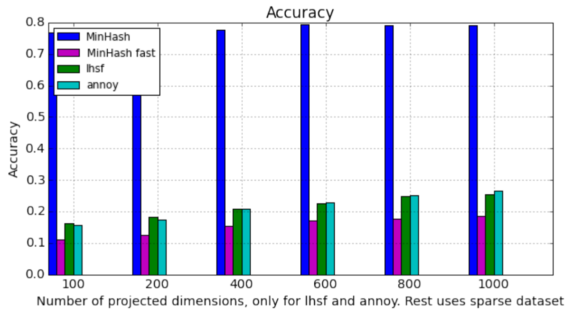
(a) Query time Bursi



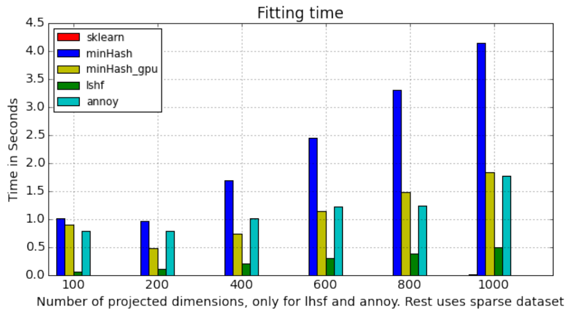
(b) Query time RNA dataset



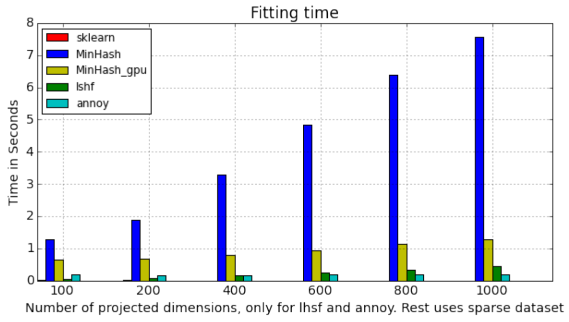
(c) Accuracy Bursi



(d) Accuracy RNA dataset



(e) Query time Bursi



(f) Fitting time RNA dataset

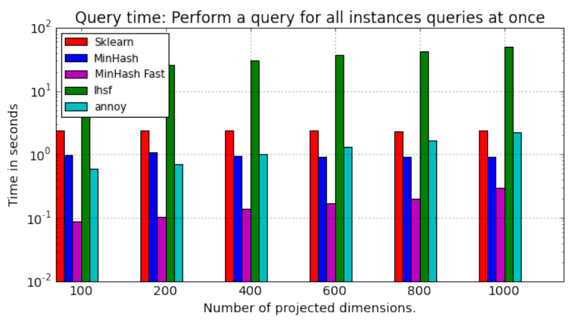
Figure 7.13: Different query times, accuracy and fitting times for the hyper-parameter optimization for an accuracy level of 0.95 for the Bursi dataset.

7.1.3.2 Conclusion

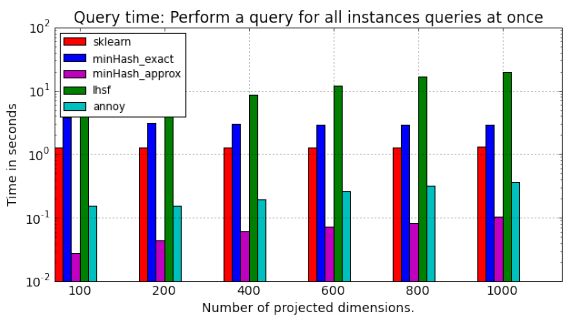
The usage of the GPu leads to comparable fitting times for MinHash, the CPU version needs more time and is growing faster. The runtime for Bursi and RNA dataset are comparable, the growth factor depending on the input data is linear and the increase of the runtime is linear by the number of projected dimensions. The parameter setting is transferable, the accuracies are, compared to the maximal accuracy, on a high level.

7.1.3.3 Accuracy level 0.90

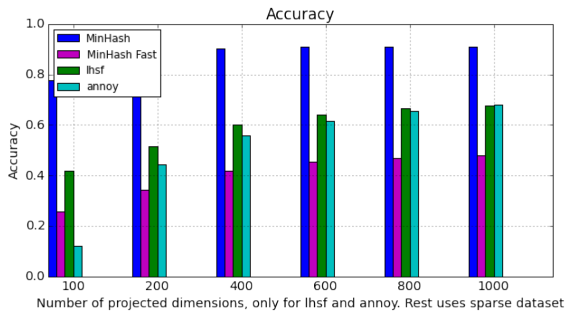
The query time on Bursi is for this accuracy level faster than the brute force implementation and for 400 projected dimensions at the level of annoy Figure 7.14a. LSHF is by far the slowest algorithm but it is more accurate than the fast implementation of MinHash. Annoy and LSHF are at a similar accuracy level. The GPU supported fitting is faster than annoy for Bursi, Figure 7.14e, CPU based fitting is again the slowest. The MinHash algorithm is the most accurate on Bursi and on the RNA dataset, on the RNA dataset accuracy is getting lost, only 0.7 is reached. But this is still a high value compared to LSHF and annoy, Figure 7.14d.



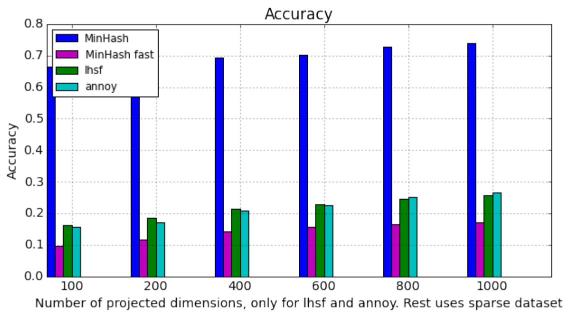
(a) Query time Bursi.



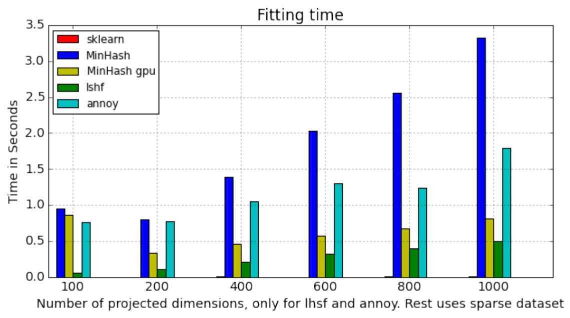
(b) Query time RNA dataset.



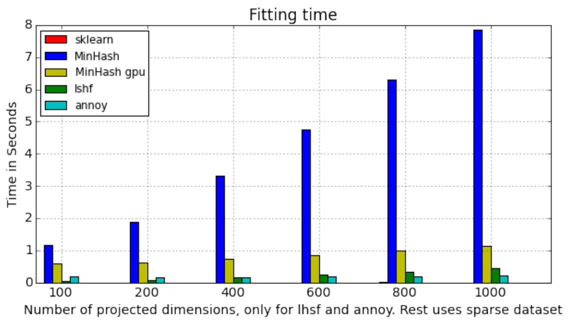
(c) Accuracy Bursi.



(d) Accuracy RNA dataset



(e) Fitting time Bursi.



(f) Fitting time RNA dataset

Figure 7.14: Different query times, accuracy and fitting times for the hyperparameter optimization for an accuracy level of 0.90 for the Bursi dataset.

7.1.3.4 Conclusion

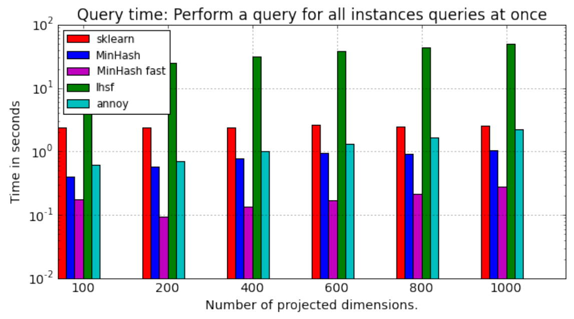
The parameter setting for 0.9 for Bursi can be transfered to the RNA dataset but a bit accuracy is getting lost. Instead of the expected loss of 5%, 10% are lost. The runtime of the brute force implementation is still faster.

7.1.3.5 Accuracy level 0.7

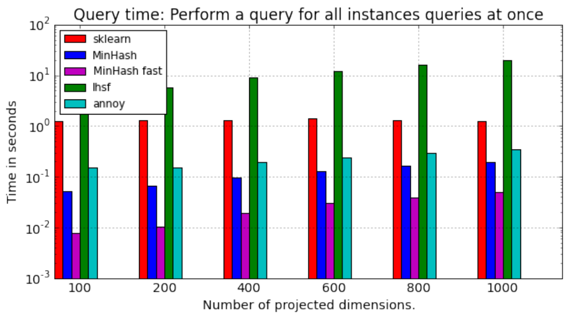
An accuracy level of 0.7 on the Bursi dataset can be achieved with just 200 hash functions, see Figure 7.15c. With this parameter setting MinHash is finally faster as the brute force algorithm for the RNA dataset, Figure 7.15b, but this was paid with a decrease of the accuracy down to 0.15, Figure 7.15d. This is not an acceptable value, MinHash is worse than annoy and LSHF. The main reason for this is that too many values were pruned out of the inverse index, less than 0.1 correct neighbors could be found in the fast case of MinHash. Based on this the exact computation can not achieve more and it is likely that not wrong candidates were found, it is highly likely that no candidates were found. This would explain why the slow case of MinHash is so fast, specially if the runtime of 10^{-1} is compared to the runtime in the 0.95 or 0.9 accuracy level where it was about 10^{1}. It needs to be noticed that the runtime of the fitting is much longer as for the accuracy level of 0.95 or 0.9. This is caused by the activated shingling. For the computed 200 hash functions by the hyper-parameter optimization it is no problem to activate the shingling, but for a higher number of hash functions the runtime is exploding. This is only true for the CPU version. The GPU version is increasing too but compared to the fitting times of the 0.95 and 0.9 level the runtime for e.g. 1000 hash functions is only doubled. The CPU version needs about four times longer.

7.1.3.6 Conclusion for 0.7

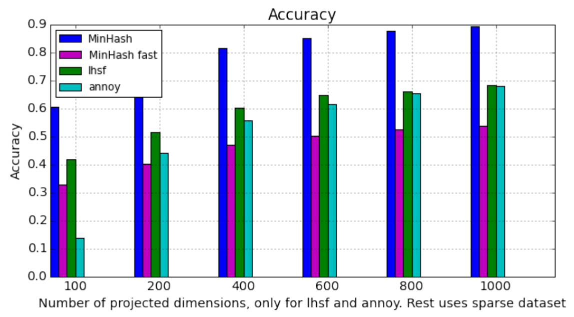
As it was shown here the computed parameters are not always transferable to a different dataset and to a different number of hash functions. Furthermore it can be seen that the influence of the shingle to the runtime of the fitting can be drastically. For a lower number of hash functions shingles should be no problem but for e.g. 1000 hash functions the increase is too high.



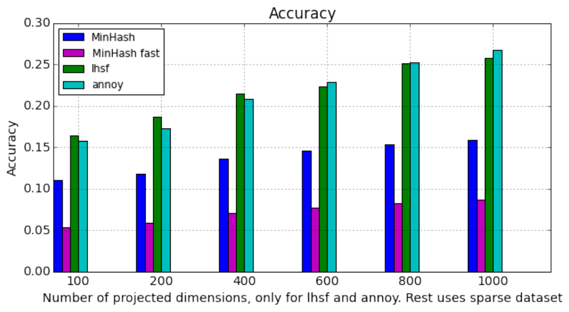
(a) Query time Bursi.



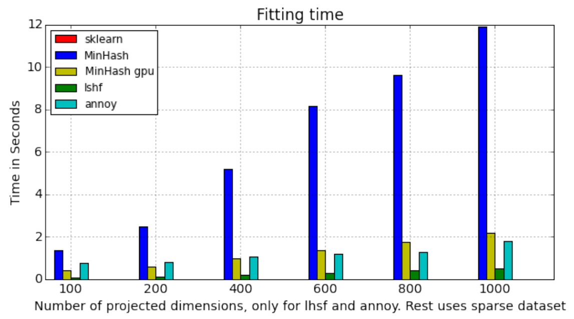
(b) Query time RNA dataset.



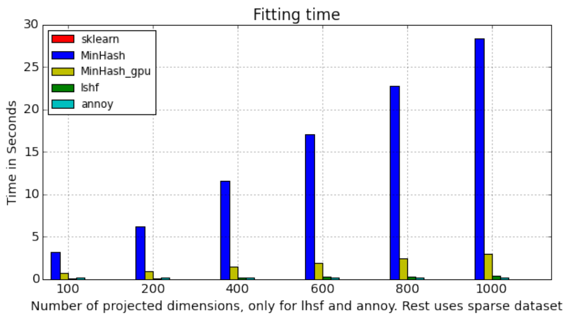
(c) Accuracy Bursi.



(d) Accuracy RNA dataset



(e) Query time Bursi.



(f) Fitting time RNA dataset.

Figure 7.15: Different query times, accuracy and fitting times for the hyper-parameter optimization for an accuracy level of 0.70 for the Bursi dataset.

7.1.3.7 Conclusion for comparison section

A hyper-parameter optimization works great on one dataset but can not always be transfered to a different dataset. MinHash compared to LSHF and annoy is having the best accuracy, if the parameter setting is conservative it is transferable to other datasets. A conservative setting is that many hash functions are used, the excess factor is large and not too many hash values are pruned. If too many values are pruned to reach a faster runtime it can be that the accuracy drops for another dataset too much. For the usage of MinHash it can be said, that it is the best case if a hyper-parameter optimization is applied on a subset of the dataset and with this configuration the algorithm is executed. Another solution would be to go without memory optimizations or apply only obvious ones like to remove hash values with a size of one. To decrease the fitting time and to bring it on a level of LSHF and annoy the GPU version should be used. The low accuracy of annoy and LSHF on the RNA dataset should exposed, this shows how bad a random projection can work.

7.1.4 Scalability

In this section it is examined how the MinHash algorithm scales for the fitting and the prediction phase if more CPU threads are used and if the GPU version is influenced by that.

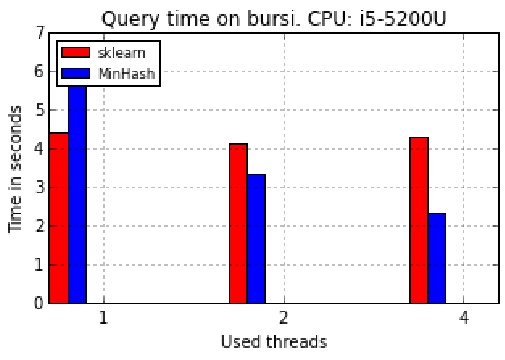
7.1.4.1 Bursi dataset

The MinHash algorithm scales well, the prediction runtime is decreasing by the factor of used CPU cores i.e. Figure 7.16c. It makes a difference if threads can run on a physical available core or if threads using Intel's hyper-threading technique, Figure 7.16c vs. Figure 7.16a and Figure 7.17a with more than 32 threads. The query time for one core is slower as the brute force implementation, for two cores it depends: if the hardware is not that fast i.e. Figure 7.16a it is faster, otherwise it is slower. This behavior is interesting. It should be cause by the fact that the source code of the brute force algorithm is not scaling good for more CPU cores but it profits by a higher instruction number per second of a faster CPU. For 4 threads the MinHash algorithm profits from its good scalability, the brute force algorithm is not increasing its performance as it would be expected. If the GPU is additionally used, it can be seen that slow GPUs slow down the computation Figure 7.16b, the used number of threads does not matter that much. A faster GPU is good to use if a fast CPU is available but not that many cores. On a high-end computer it looks similar: MinHash scales linear from 1 to 16 threads, after it is still getting faster and it is the fastest with 64 threads but the improvement from 32 threads to 64 is only minimal, see Figure 7.17a. For the brute force implementation it is looking different. First it is scaling only a bit, from 8 to 16 threads the performance is identical. If more than 16 threads are used, the performance decreases and is the worst with 64 threads. For 64 threads it needs almost twice of the time as the single thread execution. The benefits of using a high-end graphic card is on Bursi not a big improvement. In comparison to the average 750Ti graphic card, Figure 7.16d, the time is with 1.5 seconds compared to 1.0 not that big; it is 1/3 faster but the performance difference of the two graphic cards is more than 1/3. An explanation is that the bottleneck is the used CPU. The high-end Xeon CPU is having way more cores than the average i5-6600 but the performance per single thread is worse (Benchmark single thread:

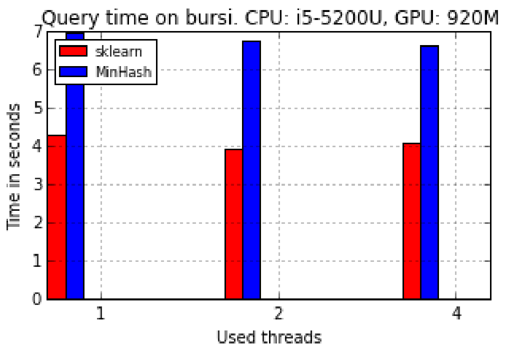
Xeon: 1925 points[1] vs i5-6600: 2105 points[2]).
For the fitting it is even more obvious as for the query: if the number of threads matches the number of physical cores, the time for the fitting reduces as it is expected by the number of threads, see Figure 7.18c. If hyper-threading is used there is still a speed up but only a little one, see Figure 7.18a and Figure 7.19a. The usage of the GPU brings a major improvement for the fitting: the fittings times are decreasing drastically, i.e. instead of about 9 seconds it just needs 1.6 seconds on the slow computer, see Figure 7.18a and Figure 7.18b. The number of used CPU threads is influencing the fitting time only if it goes from one used thread to two, but the usage of more cores brings more or less nothing. On a high-end computer the influence of the GPU is not that high. The fitting on the average computer, Figure 7.16d is with 0.4 seconds faster than the high-end computer, Figure 7.19b, with 0.6 seconds. The reason is, like it was already diagnosed for the prediction, the slower single thread speed of the CPU. It is obvious that the main part of the fitting is caused by the hash computation. On both graphic cards this is done very fast and the time difference is then caused by the CPU.

[1]https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+E5-2698+v3+%40+2.30GHz, accessed: 2016-05-24

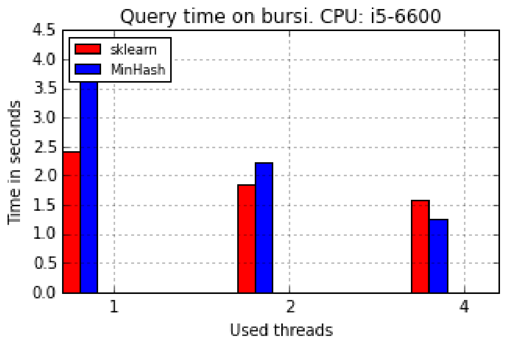
[2]https://www.cpubenchmark.net/cpu.php?cpu=Intel+Core+i5-6600+%40+3.30GHz, accessed: 2016-05-24



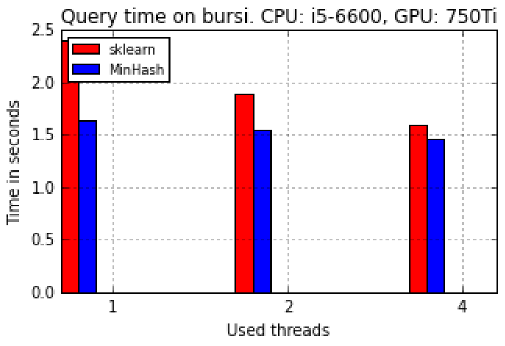
(a) Slow computer



(b) Slow computer

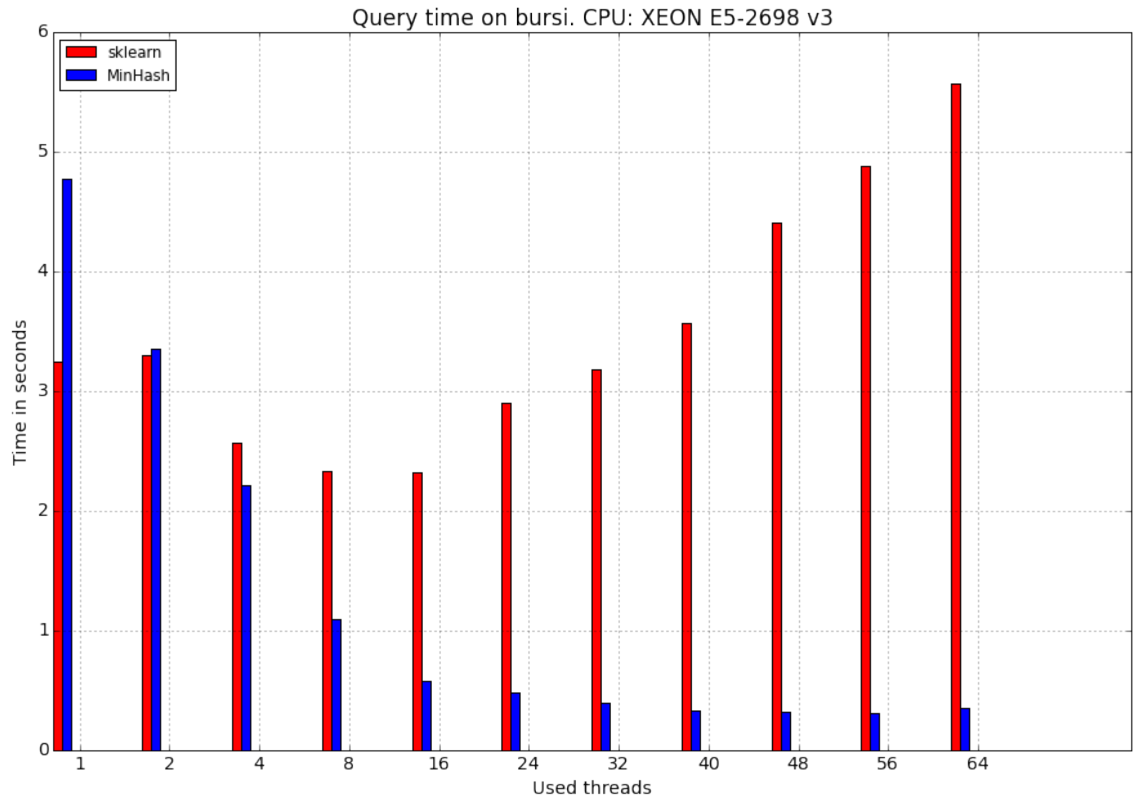


(c) Average computer

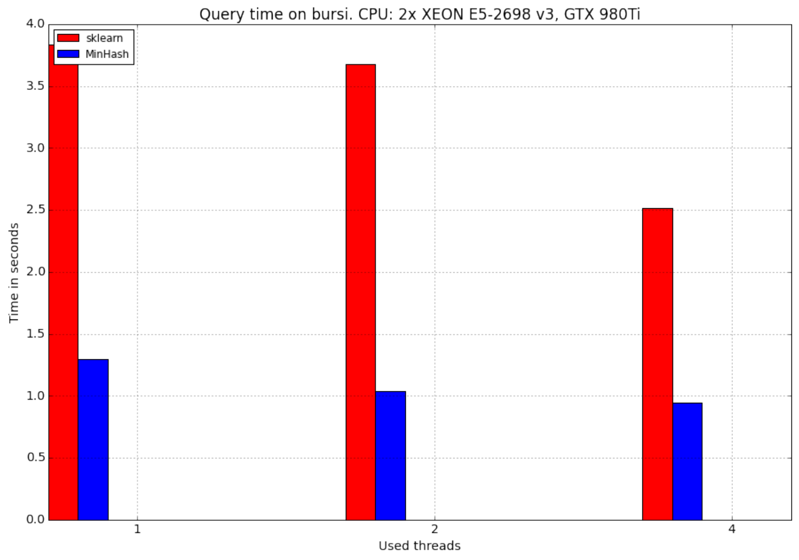


(d) Average computer

Figure 7.16: Query time on Bursi dataset for different number of used CPU threads. Slow and average computer.

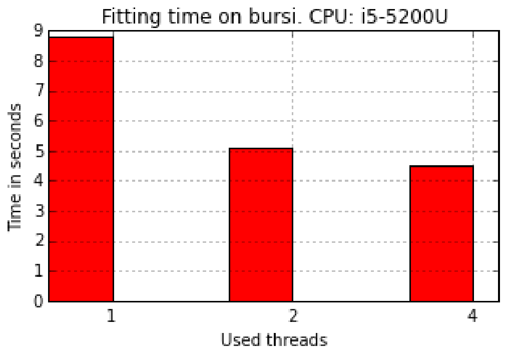


(a) High-end computer

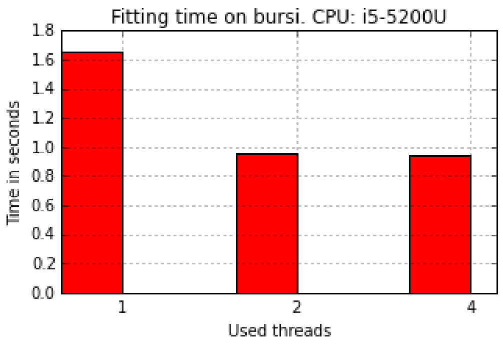


(b) High-end computer

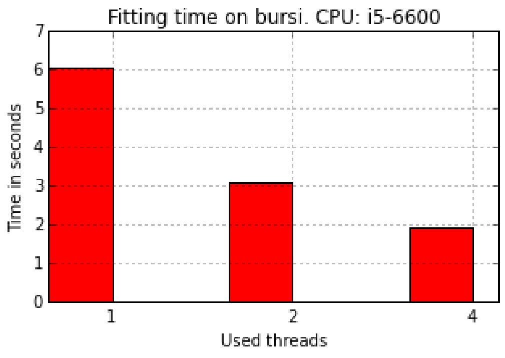
Figure 7.17: Query time on Bursi dataset for different number of used CPU threads on a high-end computer.



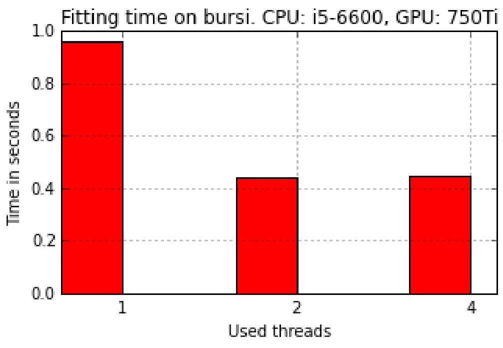
(a) Slow computer



(b) Slow computer



(c) Average computer

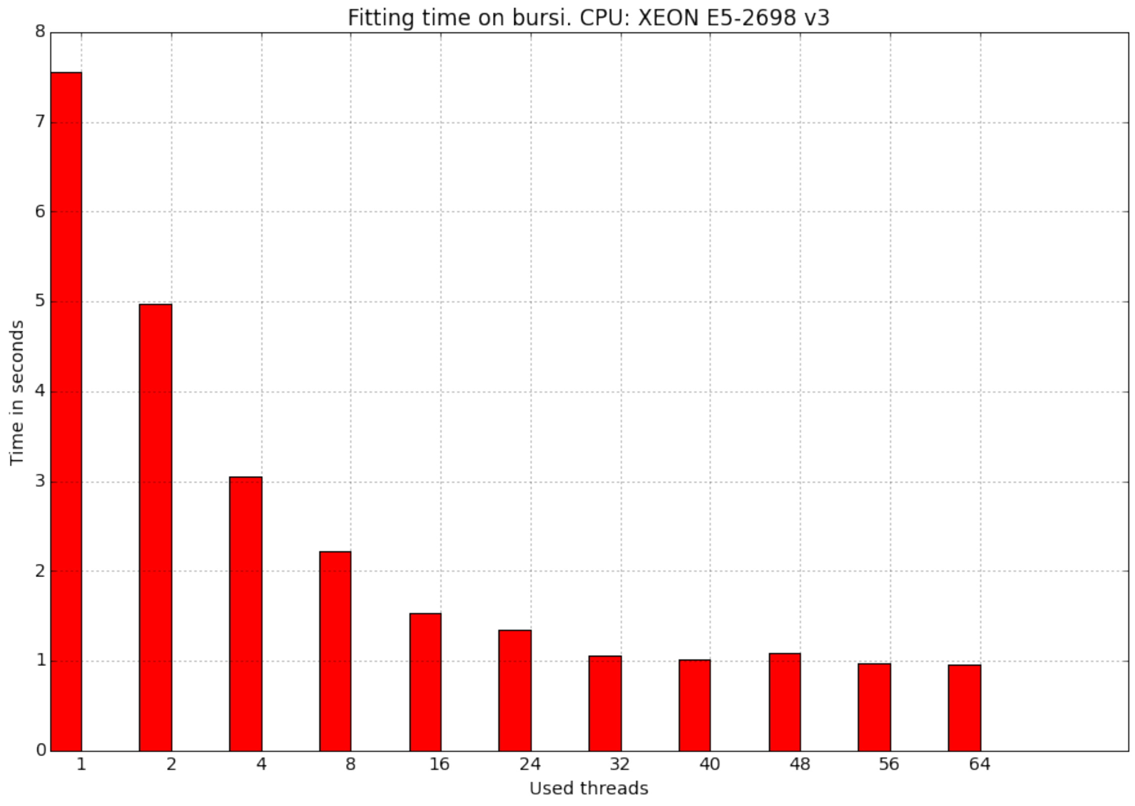


(d) Average computer

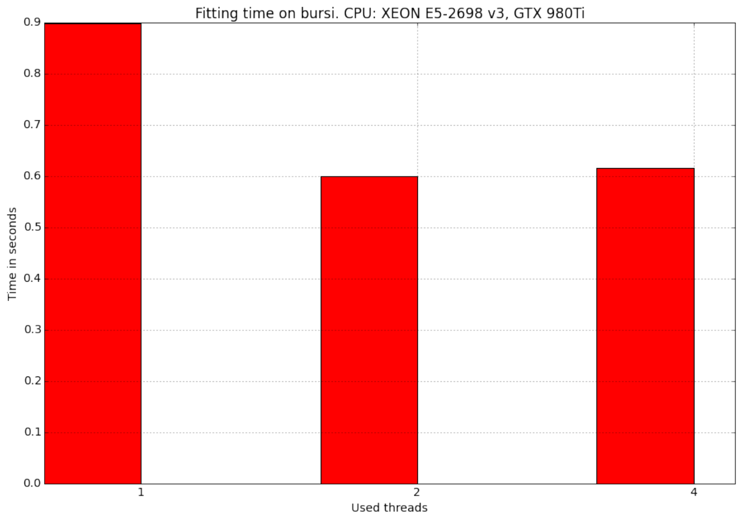
Figure 7.18: Fitting time on Bursi dataset for different number of used CPU threads on a slow and average computer.

7.1.4.2 RNA dataset

The advantage of a high-end GPU can be seen better on the RNA dataset than on Bursi. In Figure 7.20d the query time for the average computer is shown with about 10 seconds if 4 threads are used, in comparison to the high-end computer, Figure 7.21b, where it is less than 5 seconds. It would be interesting to see what would happen if the average CPU with the higher single core speed would be combined with the high-end GPU. On the other hand Figure 7.20b shows the bad influence of a low-end GPU, the runtime is about 10 times slower. The advantage to use more threads for the query and for the fitting can be seen in Figure 7.21a, Figure 7.20a and Figure 7.20c; for the fitting: Figure 7.23a, Figure 7.22a and Figure 7.22c. The brute force implementation is showing the same effect for a higher thread number as it was on Bursi, the hyperthreading of a CPU brings again only a small benefit.

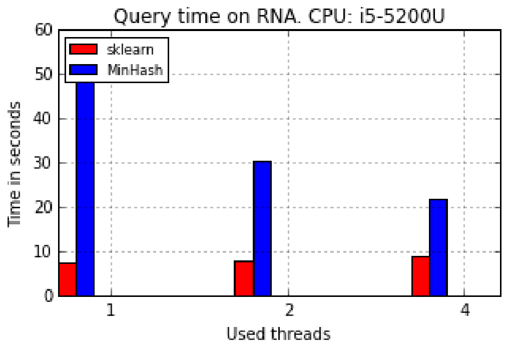


(a) High-end computer, CPU only.

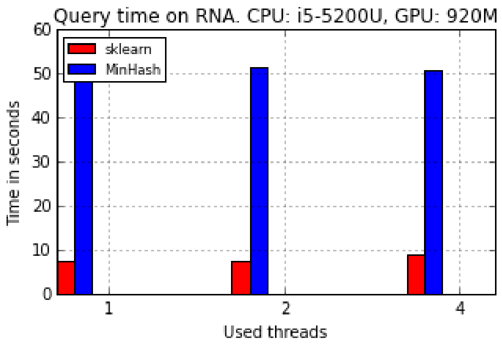


(b) High-end computer, with GPU.

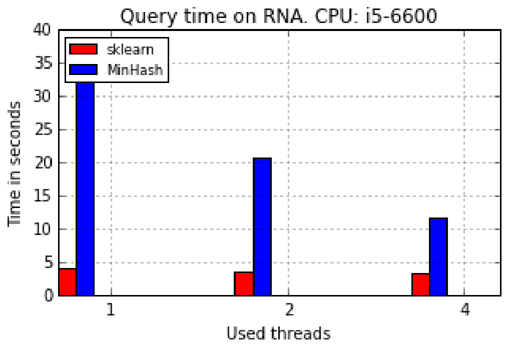
Figure 7.19: Fitting time on Bursi dataset for different number of used CPU threads.



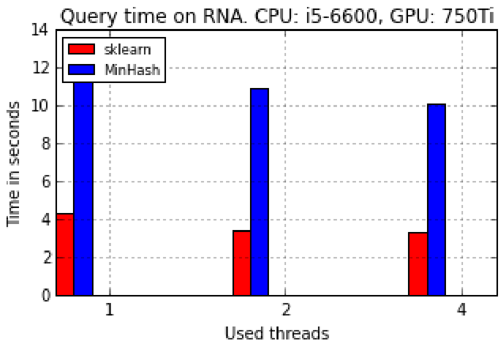
(a) Slow computer



(b) Slow computer.

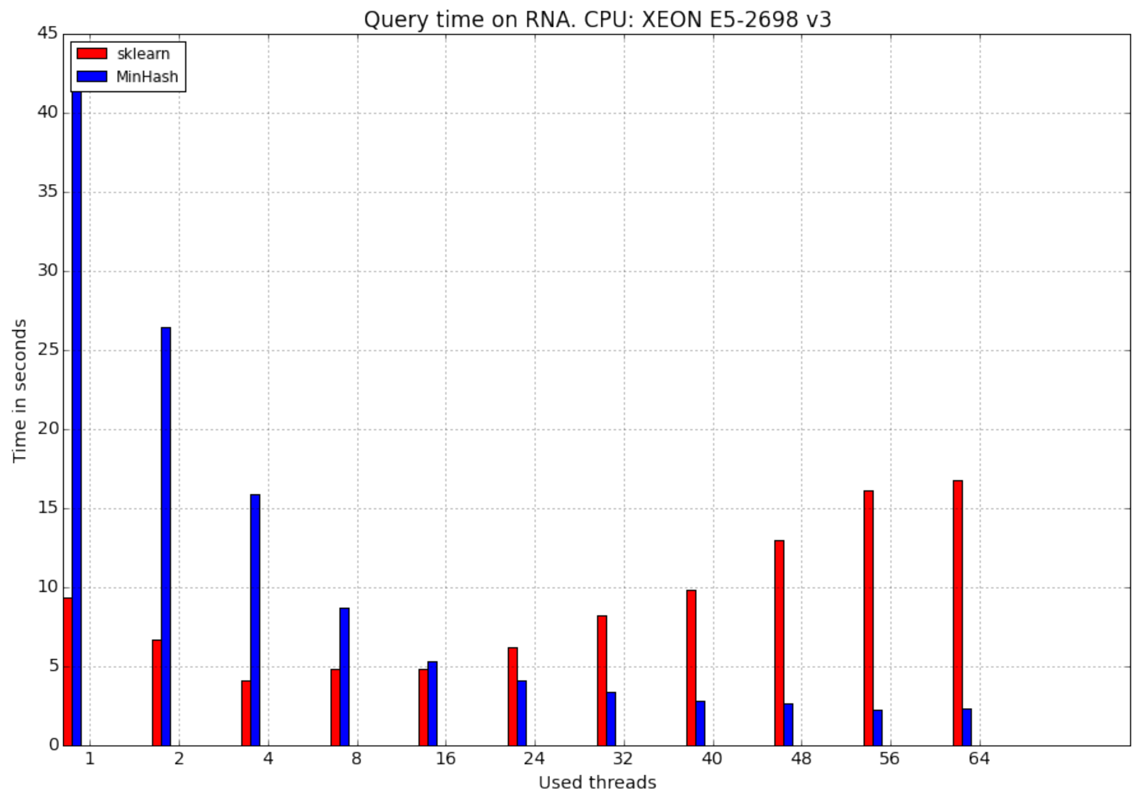


(c) Average computer.

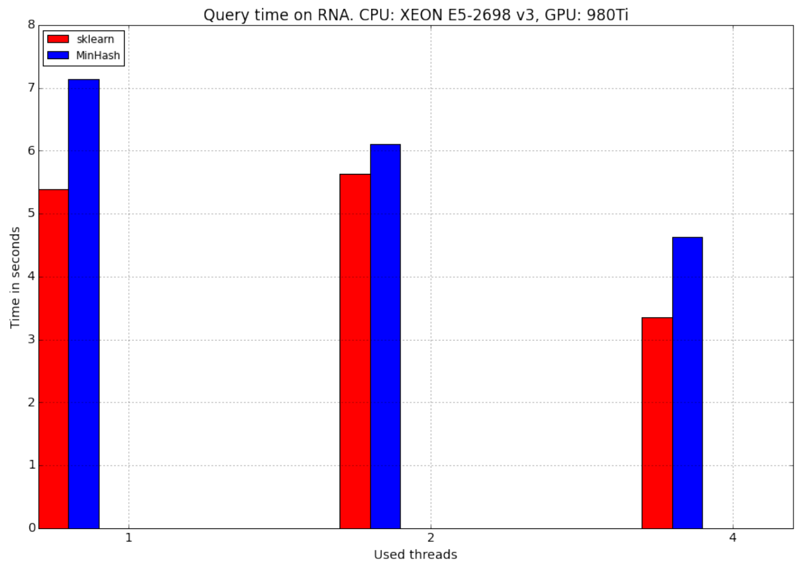


(d) Average computer.

Figure 7.20: Query time on RNA dataset for different number of used CPU threads.

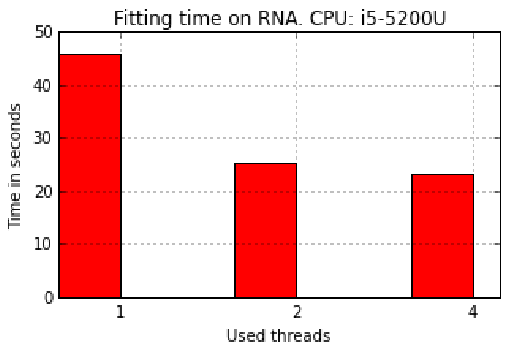


(a) High-end computer, CPU only.

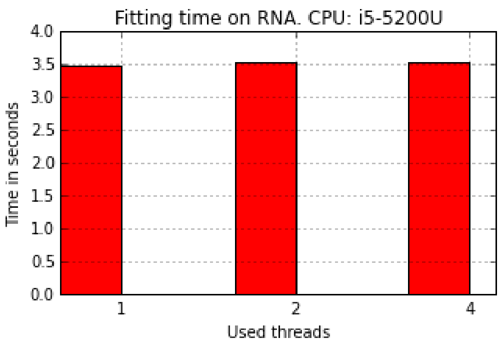


(b) High-end computer, with GPU

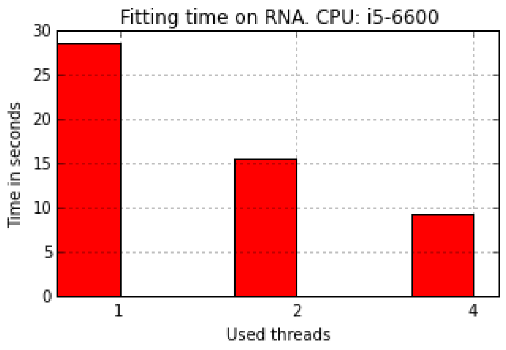
Figure 7.21: Query time on RNA dataset for different number of used CPU threads.



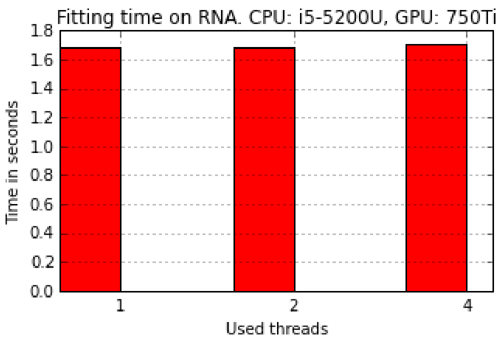
(a) Slow computer.



(b) Slow computer.



(c) Average computer

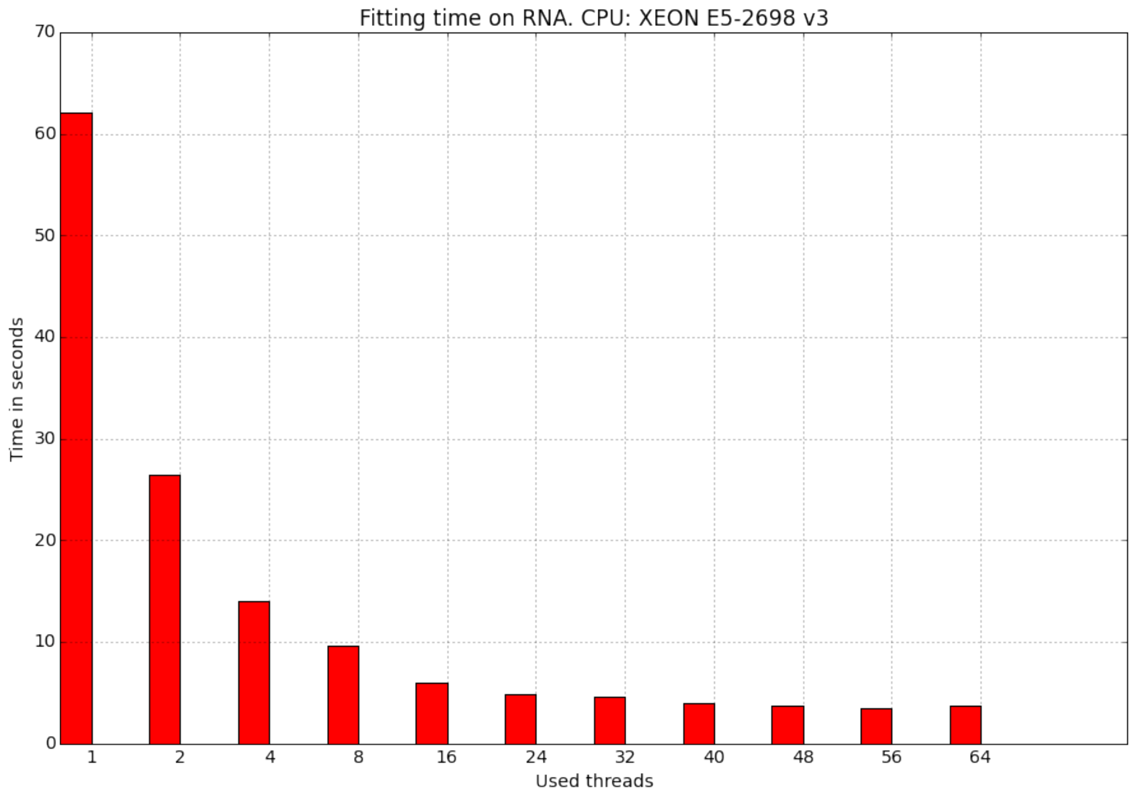


(d) Average computer

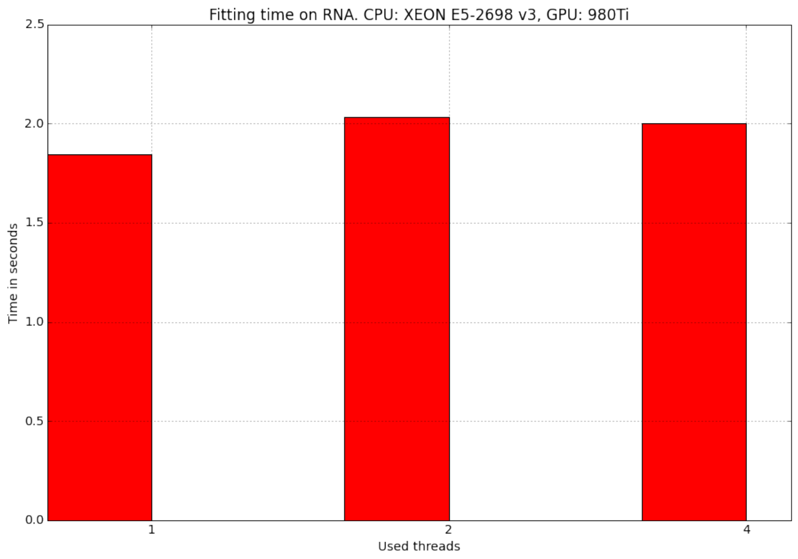
Figure 7.22: Fitting time on RNA dataset for different number of used CPU threads.

7.1.4.3 Conclusion

MinHash is scaling very well with a large number of CPU threads. The brute force implementation is scaling only a little bit and with a higher thread number the runtime is increasing. As it could be shown the influence of a high-end GPU for the fitting process is not that high, here it depends more on the single thread performance of the CPU. To compute the query on a high-end GPU brings a reduction of the runtime by a factor of 2. If a low-end GPU is used, the runtime can increase as it was shown for the slow computer. In this case it is better to use the GPU only for the fitting part. For the fitting it is independent what GPU is used, it is always a benefit. It can be seen that MinHash is scaling linear in respect to the input data size.



(a) High-end computer, CPU only.



(b) High-end computer, with GPU.

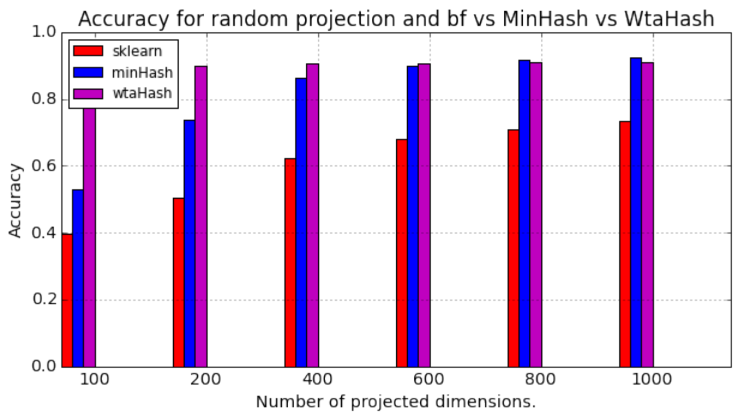
Figure 7.23: Fitting time on RNA dataset for different number of used CPU threads.

7.1.5 Random projection

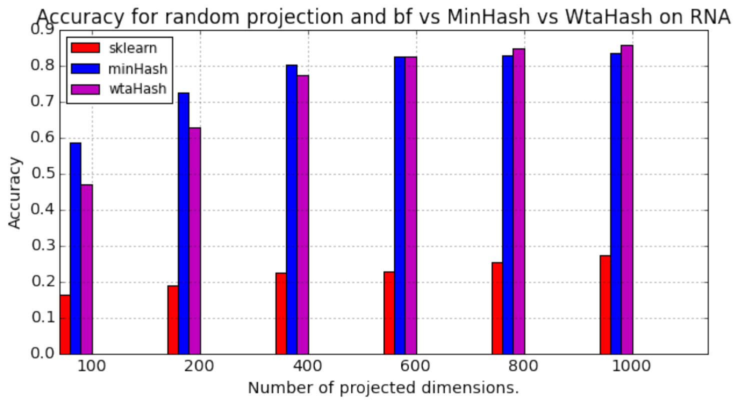
In this section it is examined how a classical brute force algorithm behaves on a random projected dataset in comparison to MinHash and WTA-Hash. MinHash and WTA-Hash are using the number of hash functions to reduce the dimensions. For the random projection the sparse random projection from scikit-learn with default settings was used. In Figure 7.24a it can be seen that the accuracy for the brute force algorithm is significant less than the MinHash and WTA-Hash if the dimensions are reduced to 100 - 1000 dimensions, 0.15 difference to MinHash on average, for 100, 200 and 400 dimensions 0.5 to 0.3 for WTA-Hash. WTA-Hash is accurate even for low dimensions as 100 or 200, MinHash is here less accurate, 0.9 for WTA-Hash vs. 0.5 and 0.7 for MinHash. MinHash can improve its accuracy with an increasing number of projected dimensions and is more accurate as WTA-Hash for 800 and 1000 dimensions. It is also interesting to see that WTA-Hash is not increasing its accuracy with an increasing number of projected dimensions. The disadvantage of WTA-Hash is that it is for less than 600 dimensions slower than the brute force solution and in all cases slower or at the same speed as MinHash, see Figure 7.25a. A look at the RNA dataset shows that the accuracy is just around 0.2 if the brute force algorithm is used (Figure 7.24b) but the query times are much faster than the MinHash and WTA-Hash algorithm, less than 10^0 vs. about 10^1 for MinHash and WTA-Hash.

7.1.5.1 Conclusion

To sparse random project the data and apply on this the brute force algorithm is faster as MinHash and WTA-Hash on the RNA dataset; but the speedup comes with an accuracy level of less than 0.3. On Bursi the brute force algorithm is on an equal runtime level for 200 and 400 dimensions, with 600 dimensions and more it is slower. MinHash and WTA-Hash are at the same time more accurate.

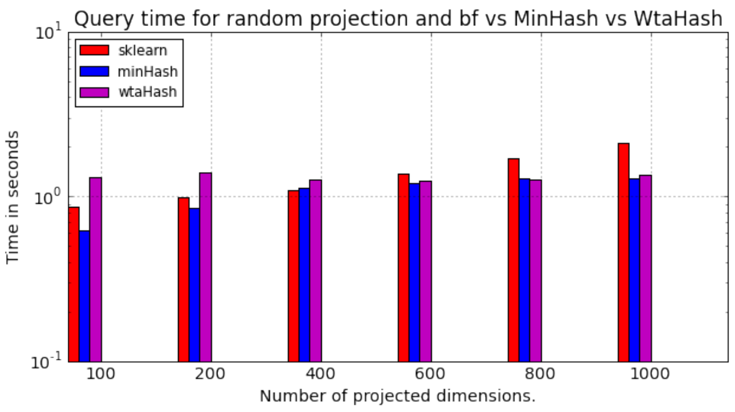


(a) Accuracy on Bursi.

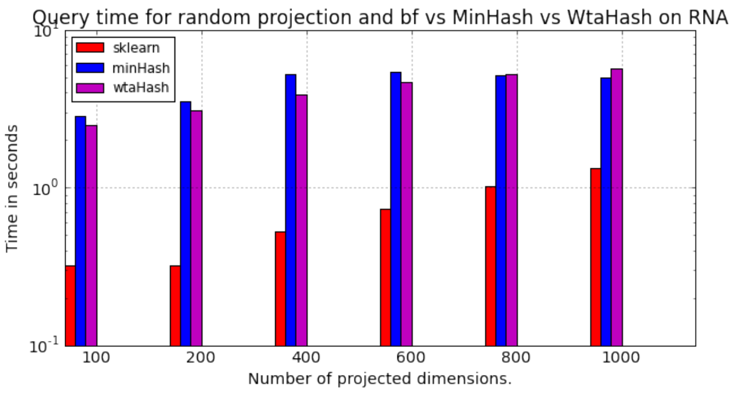


(b) Accuracy on RNA.

Figure 7.24: Accuracy on Bursi and RNA. Using random projection and brute force, running on the average computer.



(a) Query time on Bursi.



(b) Query time on RNA.

Figure 7.25: Query time on Bursi and RNA dataset. Using random projection and brute force, running on the average computer.

7.1.6 Memory usage

The memory usage was one of the key issues to design the MinHash and WTA-Hash algorithm. In Table 7.6 the system memory usage which was measured with '/usr/bin/time -v ' and here the parameter 'Maximum resident set size', is listed. If the random projected dataset is used LSHF, annoy and the brute force algorithm are using less memory, independent from the number of projected dimensions. The higher usage of memory from MinHash and WTA-Hash is caused by its structure: First the data is given to a python interface and the data is hold here in memory. Additional the data is transfered to C++ and is stored here again to have a fast access to the data for the hashing and the exact candidate computation. Also the computed signatures per instance are hold in memory to be faster in the case of a query for all fitted instances. In other words, the original data is hold twice. The impact of the inverse index to the memory size is not that big. If the memory usage of MinHash and WTA-Hash is compared to the competitors when they are applied to the original dataset, it is clear that MinHash and WTA-Hash perform the best.

The brute force algorithm needs about 420 MB, LSHF and annoy unacceptable 5.6 GB respectively 14.6 GB; MinHash in the case of 1000 dimensions 288 MB and WTA-Hash 308 MB. If the higher accuracy of MinHash and WTA-Hash is taken into account the higher usage of memory is acceptable.

Number of Dimensions	100	200	400	800	1000	original
Algorithm						
Brute force	138	139	138	138	139	420
MinHash	239	247	258	278	288	-
WTA-Hash	246	254	267	294	308	-
LSHF	163	170	174	183	195	5.6 GB
annoy	138	139	160	170	187	14.6 GB

Table 7.6: System memory usage in Megabyte (if not defined otherwise) of the different algorithms on the Bursi dataset.

7.1.6.1 Take away message

If no random projection should be done, MinHash and WTA-Hash are using the least memory. If it can be random projected and less accuracy is acceptable, annoy or the brute force algorithm should be used.

7.2 Memory saving techniques

The following analysis is based on the MinHash algorithm which was executed on the Bursi dataset with 400 hash functions. No memory optimizations were activated except the one which was analyzed. In Figure 7.26 the distribution of the number of instances per hash function is shown, over all 324420 values are stored in the inverse index which is with assumed 8 byte per size_t approximately 2.5 MB. In contrast to the used memory which was measured in subsection 7.1.6 it can be seen that only a small piece of the used memory is caused by the inverse index. As it can be seen in Figure 7.26 and Figure 7.27 the distribution for the hash values is for MinHash and WTA-Hash equal; the distribution of the size of the hash functions differ a bit, but both are normal distributed, see Figure 7.28 and Figure 7.29.

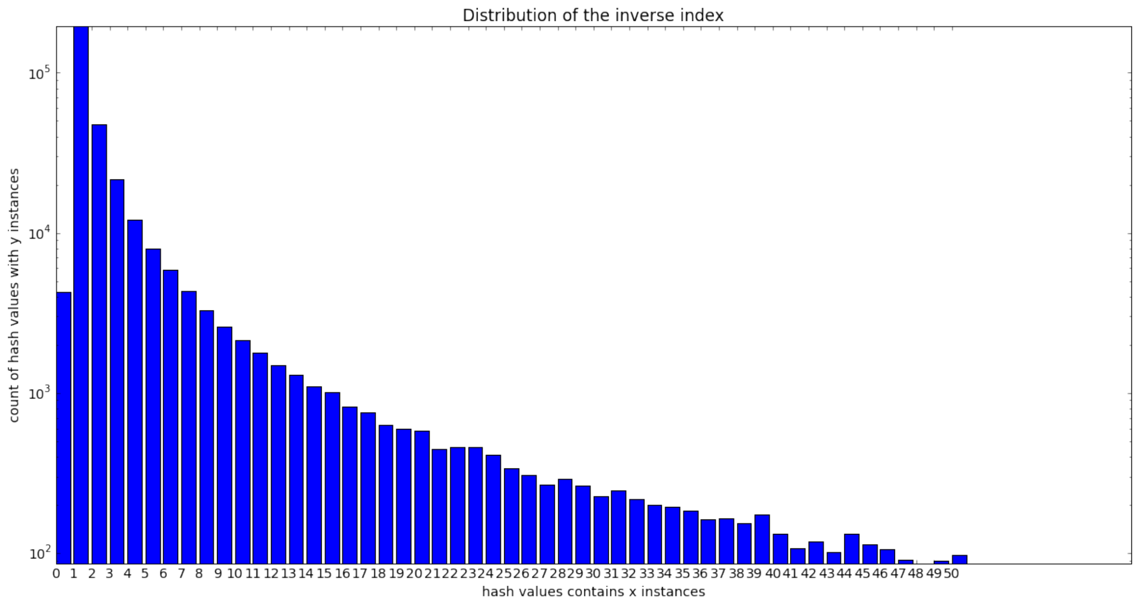


Figure 7.26: Distribution of the number of instances per hash function for MinHash.

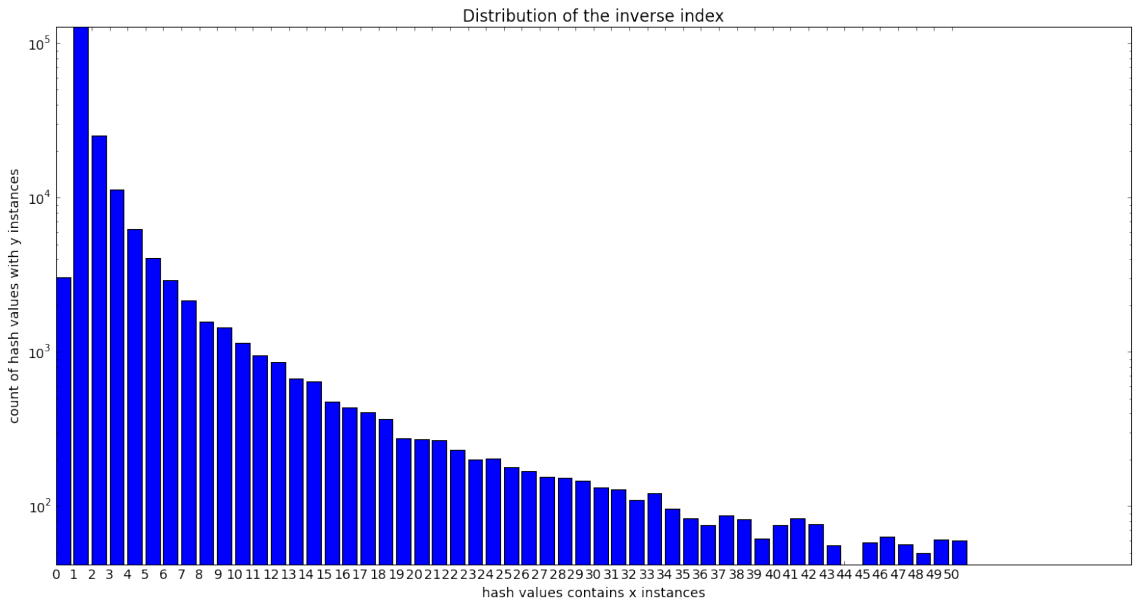


Figure 7.27: Distribution of the number of instances per hash function for WTA-Hash.

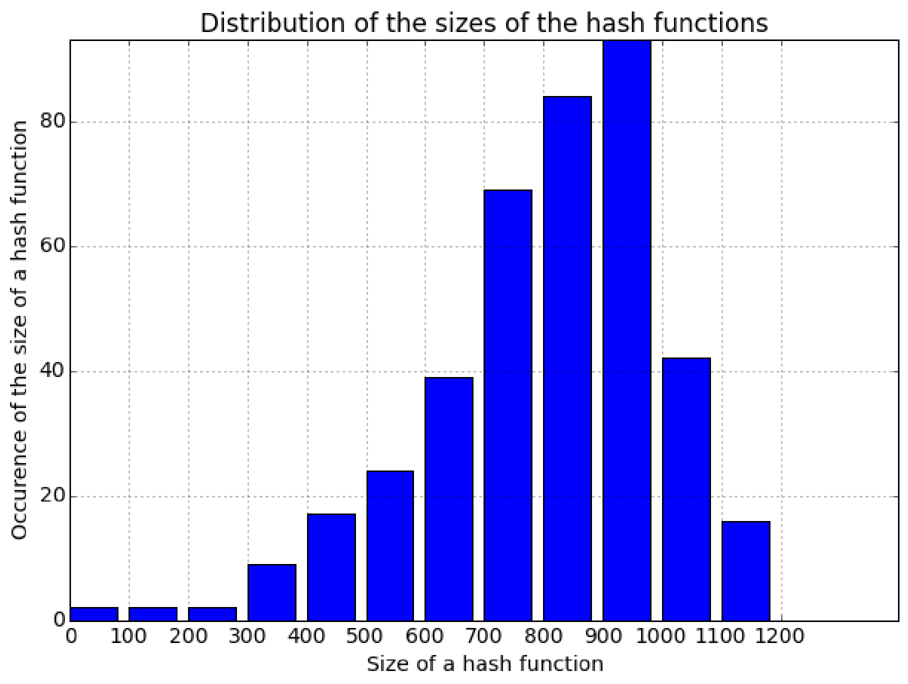


Figure 7.28: Distribution of the size of the hash functions for MinHash.

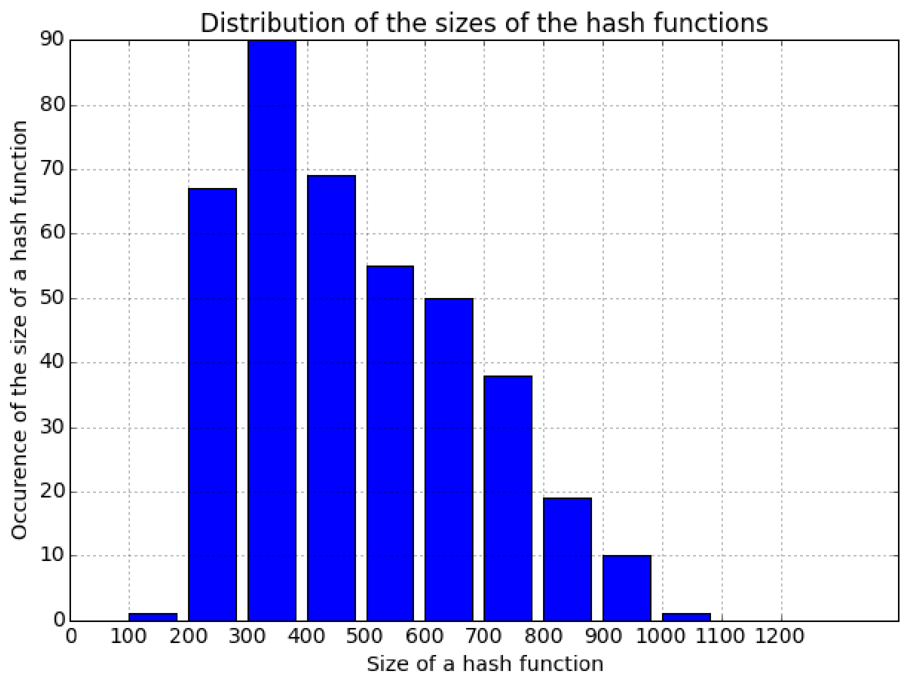


Figure 7.29: Distribution of the size of the hash functions for WTA-Hash.

7.2.1 Pruning of hash functions

In Figure 7.30 it is shown how much the removing of hash functions is influencing the size of the inverse index. Hash functions with a size less than 500 do not influence the size at all (Figure 7.30), at the size of around 500 the error is decreasing (Figure 7.31). The error is not much influenced until hash functions with a size of 900 to 1000 are removed which means, there are not that many hash functions of this size. A look at Figure 7.30 shows that between a size of 500 and 900 the size of the inverse index is decreasing slow. The size is still bigger than 10^5. If hash functions with a size more than 1000 are removed, the size of the inverse index is decreasing to less than 10^5. This indicates that the size of hash functions is normal distributed; Figure 7.28 confirms this assumption. Removing hash functions with a size less than 800 does not influence the error, removing of bigger hash functions than 1000 influence the error term exponential (blue line). A hyper-parameter optimization in respect to the score gives with $\alpha = 0.5$ and $\beta = 0.1$ a score of 0.702 and suggests to prune at a size of 1013.

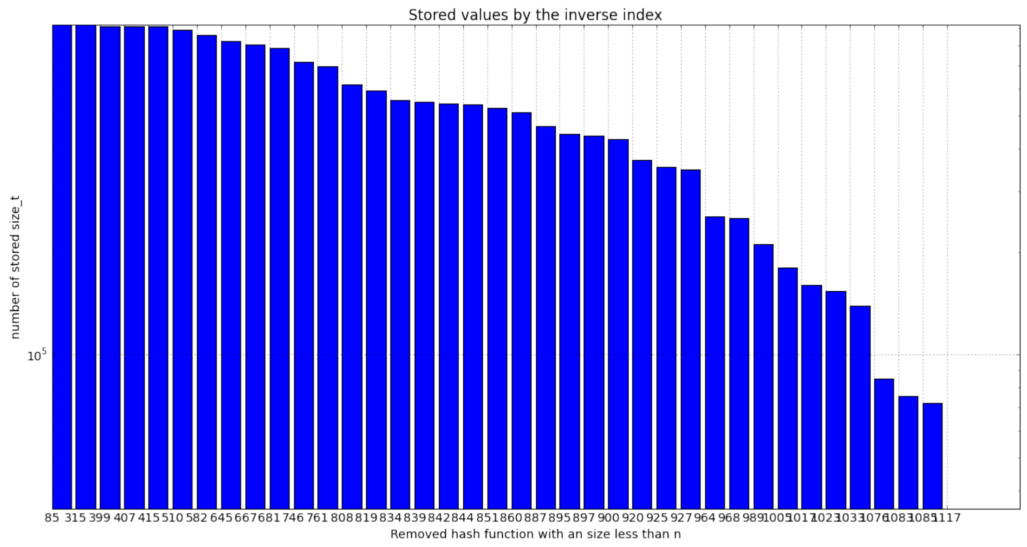


Figure 7.30: Size of the inverse index after pruning hash functions with a size less than n.

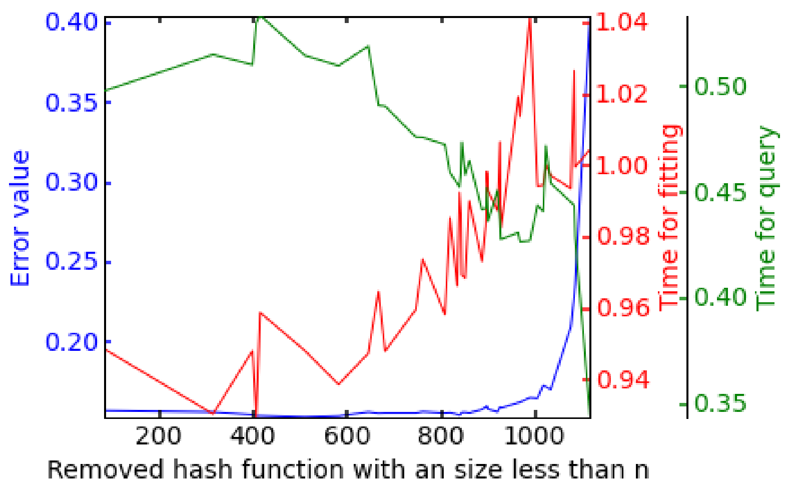


Figure 7.31: Influence of pruning the hash functions and the hyper-parameter optimization.

7.2.2 Pruning hash values

As you can see in Figure 7.26 there are more than 10^3 hash values with no entries and more than 10^5 with just one. Hash values with no entries appear because if the number of instances a hash value is assigned to becomes too large the assigned instances are deleted. To mark this hash value for future insertions as 'full' it is left empty and not deleted. In Figure 7.32 it is shown for 400 hash values and a maximum of 50 instances per hash value how much the removing of hash values with less than n entries is influencing the size of the inverse index. In Figure 7.33 it is shown how much the removing influences the fitting time, the query time and the error rate. For hash values less than ten the error can more or less ignored. For more than ten the error is increasing linear until about 35 - 40, after it it is increasing exponential. The hyper-parameter optimization with $\alpha = 0.5$ and $\beta = 0.1$ is getting as a minimum a score of 1.69 for pruning all hash values with less or equal 10 instance.

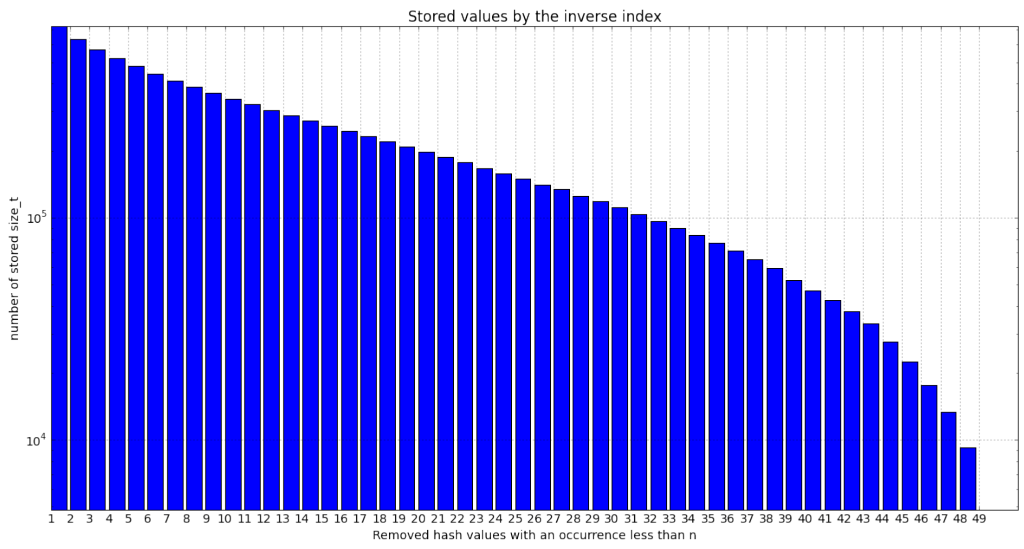


Figure 7.32: Number of elements in the inverse index after pruning hash values with a size less than n.

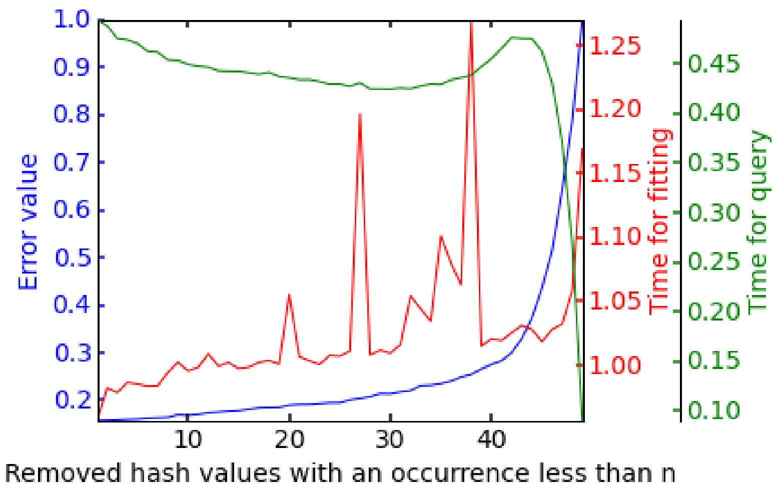
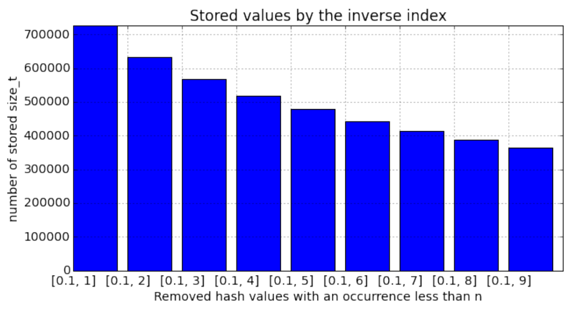


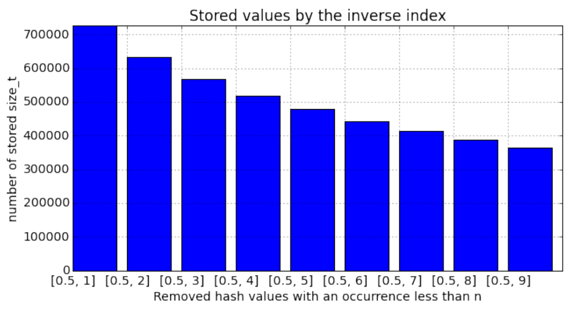
Figure 7.33: Error, fitting time and query time.

7.2.3 Frequency of pruning

To prune after a given number of elements should avoid that the peak usage of the memory is getting too high. It comes with the risk that some hash values are blocked too soon and the accuracy is decreasing. As it can be seen in Figure 7.34a the influence of the pruning is the biggest if every 10% the data is pruned, but if too many values are pruned it can happen that not enough values are stored in the inverse index. To prune after 10% of the data leads to fast prediction times (see Figure 7.35a), the less values and with this the less instances are stored in the inverse index, the less candidates can be selected for the exact computation. If the values are pruned after 50% and then after 100% the decrease in the inverse index is similar but the query times are higher, see Figure 7.35b. Also the fitting time is longer, the reason is that pruning one time more values costs more time than pruning more often less values. Not only the pruning costs more time in the second case, also the increasing size of vectors and maps leads to reallocations. A hyper-parameter optimization in respect to the score gives with $\alpha = 0.5$ and $\beta = 0.1$ a score of 0.883 and suggests to prune every 10% with 7.

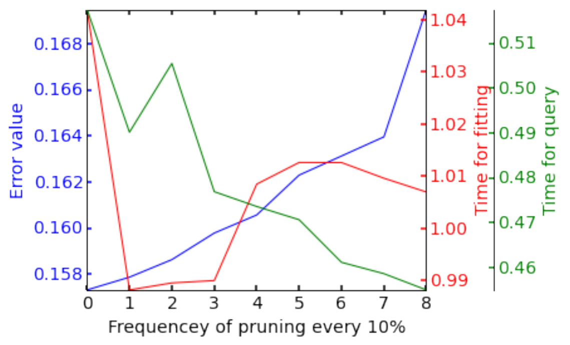


(a) Pruning every 10%.

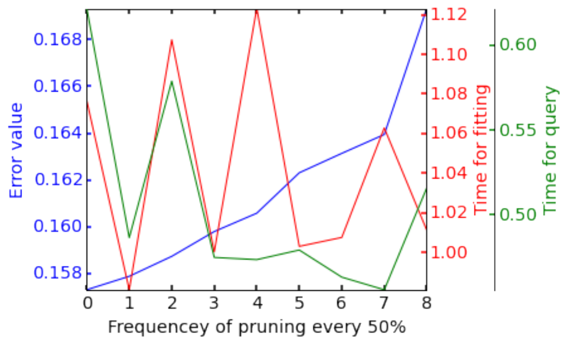


(b) Pruning every 50%.

Figure 7.34: Size of the inverse index if pruned with n hash values after k%.



(a) Pruning every 10%.

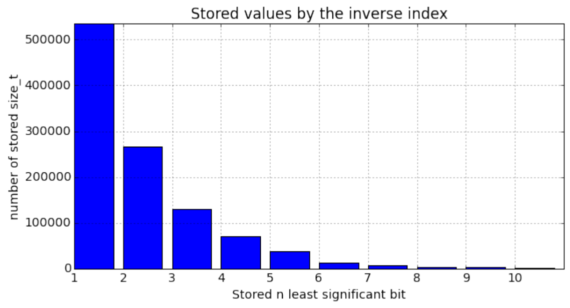


(b) Pruning every 50%.

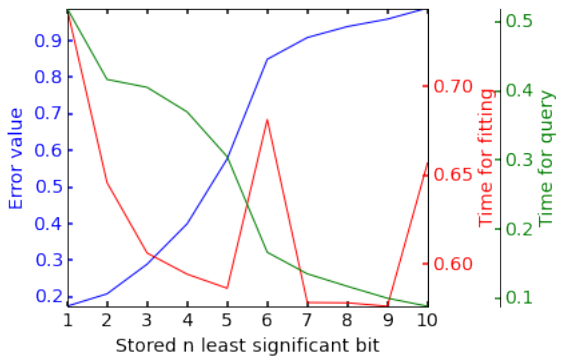
Figure 7.35: Fitting and query time, error rate for pruning during the fitting.

7.2.4 Storing only hash values with least significant bit equal 0

To store only hash values in the inverse index if the least n significant bits are equal to 0 is a very good way to decrease the number of stored hash values. As it can be seen in Figure 7.36b it does not affect the accuracy much if only a half, a quarter or an eight of the values are stored, even if only the sixteenth part of the data is stored the error rate is still less than 0.4. A hyper-parameter optimization in respect to the score gives with $\alpha = 0.5$ and $\beta = 0.1$ a score of 1.0 and suggests to store half of the data.



(a) Size of inverse index.

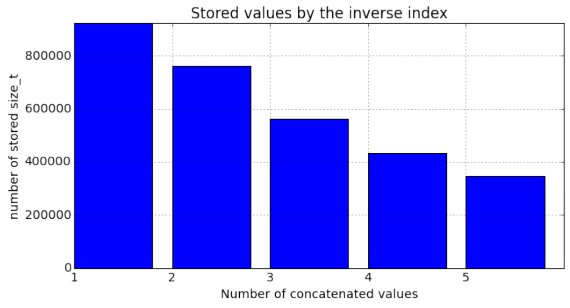


(b) Fitting and query time, error rate

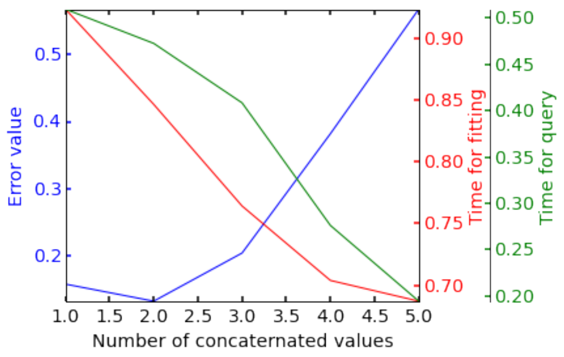
Figure 7.36: Behavior if only values with least n significant bits equal 0 are stored.

7.2.5 Compress the signature

To compress the signature by a factor of $1, ..., 5$ leads to the predictable effect that the size of the inverse index is decreasing linear, e.g. a little less than 800000 values are in the index if 2 values are merged and a little more than 400000 if 4 values are merged together, see Figure 7.37a. In Figure 7.37b it can be seen that merging two values together can have a positive effect to the accuracy, but not more than two values should be merged together. If more values are merged, the error term is getting too fast too big.



(a) Size of inverse index.



(b) Fitting and query time, error rate

Figure 7.37: Behavior if n values are merged together.

7.2.6 Conclusion

The examined techniques are achieving their goals and can save memory. But even more than the memory usage the performance increase is important. For example if every hash value with less instances as 10 is pruned the query time decreases from about 0.55 seconds to 0.45, a decrease of about 20% (Figure 7.33). The same effect can be seen if hash functions with less than the mean + the standard deviation are removed (Figure 7.31) or if the hash values are pruned after 50%(Figure 7.35b), and hash values with less instances as 5 are pruned, 0.63 vs 0.46 seconds query time. It is important to know how theses techniques are influencing each other. In Table 7.5 the parameters for the accuracy level of 0.95, 0.9 and 0.7 on the Bursi dataset are listed. It can be seen that the pruning of hash functions (parameter 'remove_hash_function_with_less_entries_as') is set to 0 which means that all hash functions with a size less than the mean + standard deviation are pruned and hash values with less than 14 respectively 10 are pruned too. To merge values is deactivated, the pruning during the fitting process too. For an accuracy level of 0.7 significant less hash values (200 vs 600 or 800) are used and all the memory saving techniques except the pruning of the hash functions is deactivated. If the parameters are optimized for the best accuracy setting (Table 7.1) the shingle parameter is used. A large number of hash values (903) is used, only hash values with the two least significant bits equal 0 are stored, i.e. only a quarter of the data. At the same time hash values with less instances as 1 are pruned, all hash functions with less entries as the mean plus the standard deviation are pruned. In conclusion, the size of the hash functions is more or less equal distributed and removing hash functions with a size less than the mean plus the standard deviation removes only the outliers. This is the reason why in all hyper-parameter optimizations the result is that these hash functions can be removed.

7.2.7 Recommendations

How many values can be pruned influences the usage of the memory, the runtime and the accuracy of MinHash and the WTA-Hash algorithm. In the following a few recommendations are listed.

- Hash functions with a size less than the mean + standard deviation can usually removed without loosing accuracy. To use it set the specific parameter 'remove_hash_function_with_less_entries_as 'to 0. This parameter can be combined with other parameters with no risk of losing accuracy.
- If the parameters to prune the hash values and to store only 1/x of the data are both activated, use for both only small values like 0, 1 or 2 to prune the hash values and 1 for the storing. If both are used with a large value too many hash values are removed and too less candidates are found in the inverse index. This would lead to a small accuracy.

- If the parameter to store hash values is used alone than do not choose a value bigger as 3, in some cases 4 could be acceptable too. Remember: 3 means that only 1/8 of the created values is stored, for 4 its 1/16.
- If the parameter to prune hash values with less instances as n is used alone, do not prune more than 10 to 15.
- To prevent a high peak memory usage the parameter to prune after x% should be set to 0.5. Especially if it is combined with the other parameters a higher frequence would lead to a decrease of the accuracy. A less frequency would not have a big influence in comparison to the pruning after 100% of the data.

7.3 Classification

In this section it is shown for the RNA dataset how good the prediction of the classifier based on MinHash and WTA-Hash is working. MinHash and WTA-Hash were executed with the parameters for the best score from subsubsection 7.1.2.2. The results are listed in Table 7.7. A look at the predicted probabilities shows that for MinHash are 0.93 of the probabilities are the same as in the brute force implementation. This is a high value but the most of the classes have a probability of 0.0 and an accuracy of e.g. 0.74 for the WTA-Hash fast algorithm where the most probabilities are wrong should relativize this value. The score which measures the mean accuracy on the given test data and labels shows that the MinHash and WTA-Hash are a bit more accurate than the brute force implementation. MinHash, WTA-Hash and the brute force solution are not good estimators to predict the correct labels, 0.6 accuracy is too less.

Algorithm	MinHash	MinHash Fast	WtaHash	WtaHash Fast	Brute force
Predict prob. accu	0.71	0.93	0.74	0.92	-
Score	0.41	0.57	0.47	0.59	0.56

Table 7.7: Accuracy and score of the prediction with MinHashClassifier and WtaHashClassifier in comparison to sklearn's KNeighborsClassifier.

7.4 Clustering

Some clustering algorithms are based on the nearest neighbors and compute them internally as a base for their clustering. Algorithms like the spectral clustering from sklearn having the option that instead of an input data matrix and some measurement a precomputed nearest neighbors graph can be used as an input. The idea is

that the precomputation can be done faster and then the clustering should be faster. With the current version 0.17 of scikit-learn this is not true. The precomputation with some artificial dataset is done fast but somehow the tested spectral clustering needs more time if the precomputed nearest neighbor graph is given compared to the non-precomputed version. Spectral clustering is running in 17.7 seconds and is achieving an adjusted rand score of 0.33. With the precomputed nearest neighbor graph the runtime is 55 seconds with an adjusted rand score of 0.01. The memory usage of the precomputed case is less, 390 MB vs. 560 MB. The situation with DBSCAN is bad in a different way. Without precomputation the runtime is 15 seconds, with precomputation it is 8 seconds. The memory usage with 639 MB vs 86 MB is way better with the precomputation. These two numbers are looking good but the adjusted random score is here the critical issue. DBSCAN is very sensitive to its ϵ-parameter, without the right choice the adjusted random score is 0. In the expectation the adjusted random scores of the clustering with and without the precomputation should be more or less the same if the approximate nearest neighbors compute an accurate neighborhood. But it does not look so. Why this is the case needs to be further investigated.
For the speed difference it looks like that sklearn is using different source code in the implementation for the two cases. If it is using optimized Fortran source code from BLAS like in the brute force nearest neighbors computation and for the precomputed case this is not possible and normal Python or C/C++ code is used the slower runtime in the precomputed case could be explained.

8 Discussion

8.1 Take away messages

- MinHash and WTA-Hash are scaling well on multi core computers, the brute force implementation of scikit-learn does not.
- To compute the hash values for MinHash on the GPU is a benefit.
- For the GPU part it is not only important to have a fast GPU, the CPU single thread speed is crucial too.
- MinHash and WTA-Hash do not need to use a sparse random projected dataset to compute the nearest neighbors. They can operate on the original dataset and are more accurate than the local sensitive hashing forest or annoy. In comparison to these two MinHash and WTA-Hash need less memory.
- For dataset with 300 to 400 non-zero features MinHash and WTA-Hash perform better than the brute force implementation.
- For dataset with 3000 to 4000 non-zero features and the computer has 4 to 8 cores the brute force implementation of scikit-learn is performing better. If 32 cores are available, MinHash should be used.
- The best computer configuration for MinHash would be an 8 core computer with a high-end graphic card. With this setting it should be faster as the brute force implementation in the most cases.
- To reduce the memory usage of the inverse index and to speed up the prediction the half to a quarter of the values can be pruned without losing much accuracy.

The presented work shows that MinHash and WTA-Hash are working great for the approximate k-nearest neighbor search on very sparse and very high dimensional datasets. The most competitive implementations do not work on sparse datasets or perform worse. The local sensitive hashing forest needs longer query times but is fitting very fast. It needs more memory (5 GB) on the original dataset and without a random projection it is not performing well. If projected it needs less memory and the accuracy is about 0.1 higher as the fast version of MinHash but it needs two magnitudes (10^{-1} s to more than 10^{1} s) longer for the query. Compared to the non-fast version it is still slower for lower dimensions but more or less equal for higher ones e.g. like it was shown for the RNA dataset. The query time of Bursi and the RNA dataset combined shows that the local sensitive hashing forest is better in the

fight against the curse of dimensionality, the run times for these two datasets are more or less equal. For further work it would be interesting to examine a similar way for having an index like LSHF. The fitting time should be no problem as long as the GPU is used for this computation. With the CPU this could take too much time. In a best case scenario the accuracy is at a level that one or maybe both steps for the exact computation could be removed. For the local sensitive hashing forest and annoy it is obvious that their lack of accuracy is not caused by a bad approximation. It is highly likely that it is caused by the dimension reduction of the input data. Both algorithms are showing equal results compared to the brute force algorithm operating on the reduced dataset. The biggest issue with local sensitive hashing forest and annoy is that they consume too much memory if they are applied on the original dataset. This is a clear benefit of MinHash and WTA-Hash. They need only the input dataset and additionally they store an inverse index in the memory which is, depending on multiple factors, in the size of a few hundred kilobyte to usually not more than 50 megabyte.
Surprisingly the brute force implementation from scikit-learn shows that there is still room for improvement. The scaling of the algorithm is linear in theory and should be influenced by the curse of dimensionality but the brute force algorithm shows that a highly optimized source code can outperform a theoretical better algorithm. For further work this is one issue that needs to be worked on. The fast case of the MinHash and WTA-Hash algorithm shows that the dimension reduction performs well and the accuracies are not that far away from the brute force solution applied on a reduced dataset.
The usage of the GPU is a big improvement for the fitting phase of the algorithm. The advantages of the GPU architecture can be used in this case perfectly. To compute the dot products it is important to have a fast GPU otherwise the CPU computation will be faster. For further work it would be interesting to see how the usage of CPU extensions like SSE can improve the performance and how a SSE optimized implementation would perform in comparison to a GPU implementation. The performance increase of CPUs were getting slower in the last years but the GPU performance and specially the costs for GPUs are decreasing. The performance of a todays (May 2016) high end graphic card like a Titan X (6.2 TFlops) which costs today around 1100 Euro should be on the same level as the announced upper mid class GPU GTX 1070 (6.5 TFlops) which should cost 400 to 500 US-Dollars[1]. It is likely that the usage and the performance benefit of the GPU will become more important in the near future.

[1]http://www.heise.de/newsticker/meldung/Turbo-Pascal-Nvidia-stellt-GeForce-GTX-1080-und-GeForce-1070-vor-3198470.html, accessed: 2016-05-24

Bibliography

[1] R. Bellman, *Dynamic Programming (Princeton Landmarks in Mathematics and Physics).* republished by Princeton University Press, 1957/2010.

[2] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[3] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[4] "rpforest on github," https://github.com/lyst/rpforest, accessed: 2016-04-15.

[5] S. Dasgupta and Y. Freund, "Random projection trees and low dimensional manifolds," in *Proceedings of the fortieth annual ACM symposium on Theory of computing.* ACM, 2008, pp. 537–546.

[6] "Annoy on Github," https://github.com/spotify/annoy, accessed: 2016-04-15.

[7] "Approximate nearest neighbor methods and vector models – NYC ML meetup 2015," http://www.slideshare.net/erikbern/approximate-nearest-neighbor-methods-and-vector-models-nyc-ml-meetup, accessed: 2016-04-15.

[8] 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.* ACM, 2005, pp. 651–660.

[9] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, vol. 99, no. 6, 1999, pp. 518–529.

[10] 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.* ACM, 1998, pp. 604–613.

[11] "Flann - Fast Library for Approximate Nearest Neighbor search." http://www.cs.ubc.ca/research/flann/, accessed: 2016-04-19.

[12] "Panns on Github," https://github.com/ryanrhymes/panns, accessed: 2016-04-15.

[13] B. Naidan and L. Boytsov, "Non-Metric Space Library Manual," *arXiv preprint arXiv:1508.05470*, 2015.

[14] "Scikit-learn - Machine learning in Python," http://scikit-learn.org/, accessed: 2016-05-02.

[15] W. B. Johnson and J. Lindenstrauss, "Extensions of Lipschitz mappings into a Hilbert space," *Contemporary mathematics*, vol. 26, no. 189-206, p. 1, 1984.

[16] D. Achlioptas, "Database-friendly random projections," in *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* ACM, 2001, pp. 274–281.

[17] A. Z. Broder, "On the resemblance and containment of documents," in *Compression and Complexity of Sequences 1997. Proceedings.* IEEE, 1997, pp. 21–29.

[18] S. Heyne, F. Costa, D. Rose, and R. Backofen, "GraphClust: alignment-free structural clustering of local RNA secondary structures," *Bioinformatics*, vol. 28, no. 12, pp. i224–i232, 2012.

[19] J. Yagnik, D. Strelow, D. A. Ross, and R.-S. Lin, "The power of comparative reasoning." in *ICCV*, D. N. Metaxas, L. Quan, A. Sanfeliu, and L. J. V. Gool, Eds. IEEE Computer Society, 2011, pp. 2431–2438. [Online]. Available: http://dblp.uni-trier.de/db/conf/iccv/iccv2011.html#YagnikSRL11

[20] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: an efficient data structure for static support lookup tables," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics, 2004, pp. 30–39.

[21] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[22] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[23] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 2010.

[24] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs (Applications of Gpu Computing).* Morgan Kaufmann, 2012.

[25] F. Costa and K. De Grave, "Fast neighborhood subgraph pairwise distance kernel," in *Proceedings of the 26th International Conference on Machine Learning.* Omnipress, 2010, pp. 255–262.

[26] E. P. Nawrocki, S. W. Burge, A. Bateman, J. Daub, R. Y. Eberhardt, S. R. Eddy, E. W. Floden, P. P. Gardner, T. A. Jones, J. Tate *et al.*, "Rfam 12.0: updates to the RNA families database," *Nucleic acids research*, p. gku1063, 2014.

[27] R. Denman, "Using RNAFOLD to predict the activity of small catalytic RNAs." *Biotechniques*, vol. 15, no. 6, pp. 1090–1095, 1993.