UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Dmytro Fishman

Fast approximate max-correlation queries

Master's thesis (30 ECTS)

Supervisor: Konstantin Tretyakov, M.Sc.

Author: .... May "......", 2013

Supervisor: .... May "......", 2013

Approved for defence

Professor: .... May "......", 2013

TARTU 2013

Contents

Abstract

The task of detecting correlated items in high-dimensional datasets is common and important problem for a large variety of applications. To our knowledge this task is nowadays solved using explicit enumeration of all possible pairs of items in the dataset. Considering constantly growing amount of data, a linear scan through all pairs can be a very slow process, taking hours or even days on the researcher's computer.

In this thesis we propose an approximate solution for identifying most correlated pairs of items that produces almost exact results in linear time. The solution is based on nearest neighbor indexing. More precisely, we compared the performance of the coordinate-wise search algorithm, the k-dimensional tree and the random projection tree data structures. We conducted experiments that measured the running time and quality of produced results on simulated data. Our tests proved sufficient accuracy and linear time-complexity for the two lattest methods. For the coordinate-wise search our tests demonstrated quadratic time, which was still much faster than for the brute force approach.

Introduction

Searching for the most correlated items among all records in the large dataset is a common and important data analysis technique used in a variety of applications, such as image and signal processing, recommendation engines, etc. Generally it implies calculation of all pair-wise correlations in the dataset. If the number of items is n, and each item has d different measurements (dimensions) then this requires at least $d \cdot n(n-1)/2$, i.e. $O(d \cdot n^2)$ operations. Considering the constantly growing amount of data, a linear scan of all pairs of items can be a very slow process (taking hours or even days on the researcher's computer).

To our knowledge, there are few documented attempts to efficiently answer max-correlation queries in high-dimensional datasets. Those either apply hardware specific optimizations to speed up the calculation of all pair-wise correlations [1], or use the solution of this problem as an intermediate step for more complex data analysis procedures (e.g. clustering [2]) and thus pay minor attention to the efficiency of the intermediate steps.

In this thesis we first attempt to give a broad overview of the related mathematical terms and proofs used further in the text (Chapter 1).

We show that the problem of searching for the most correlated pair can be successfully reduced to the problem of detecting nearest neighbors in terms of Euclidean distance (Chapter 3).

We provide an overview of the three state-of-the-art nearest neighbor indexing methods in Chapter 2. Our main focus is on two approximate techniques (K-dimensional tree and random projection tree) and one exact (coordinate-wise search).

Finally, in Chapter 4 we evaluate our method in terms of running time and quality of final results. To measure those, we run two different types of tests on simulated data.

Our tests proved sufficient accuracy and linear execution time for both approximate methods. Although coordinate-wise search has a quadratic time-complexity, it still substantially outperforms the brute force method.

In order to show that our results are applicable for the real-world applications, we tested our solution on a dataset containing records related to

methylation values for different genes in different individuals. Experiments prove sufficient accuracy of achieved results and capability of detecting distant genes with highly correlated expression.

Chapter 1

Mathematical Background

In this chapter we introduce basic mathematical notions and terms that are necessary to understand the further material in this thesis.

1.1 Complexity and big-O notation

Complexity is a general way to assess the performance of an algorithm. The complexity of an algorithm is defined by a numerical function $T(n)$ that represents execution time of an algorithm for a given input size n. Note that our goal is to define the time taken by an algorithm without regard to its implementation details. Typically, the running time of an algorithm on the same inputs will vary according to following factors: computational power of the processor, disk speed, implementation language, etc. Thus, by measuring $T(n)$ as the number of elementary "steps" (basic operations) needed to achieve the final results, we can estimate efficiency of a particular algorithm, disregarding the implementation details [3].

Here are some examples of "basic operations":

- one fixed bit-size mathematical operation (+,-,* and /),
- one assignment to a variable,
- one comparison between two values,
- one write of a value.

Let us examine the classical example – addition of two integers. We will be adding these integers, digit by digit (bit by bit), and will consider each such addition as a basic "step" in our computational model. Thus, addition of two n-digit (bit) integers requires n operations according to our model. Therefore, the total execution time is $T(n) = c \cdot n$, where c is the time necessary to perform an addition of two digits. On different computers, addition of two digits may take different time, say c_1 and c_2, thus the addition of two n-digit integers takes $T(n) = c_1 \cdot n$ and $T(n) = c_2 \cdot n$ respectively. It shows that even though for different machines basic "steps" take different amounts of time, the total time $T(n)$ grows linearly as an input size increases.

The main purpose of the complexity analysis is to classify algorithms according to their performances. We will represent the time function $T(n)$ by the big-O notation to express the algorithm running time. For example,

$$T(n) = O(\log n)$$

says that algorithm has a logarithmic time complexity.

Formally, any monotonic function $f(n)$ and $g(n)$ of positive integers, $f(n) = O(g(n))$, if there exist constants c>0 and n_0>0, such that

$$f(n) \leq c \cdot g(n), \tag{1.1}$$

for every

$$n \geq n_0.$$

Intuitively, from (1.1) it follows that function $g(n)$ grows as fast as function $f(n)$ for sufficiently large $n \to \infty$. If $f(n) = O(g(n))$ and $g(n) = O(f(n))$ we say that $f(n) = \Theta(g(n))$.

Figure (1.1) illustrates the $f(n) = O(g(n))$ relation.

Constant time: $\Theta(1)$

Algorithm is said to run in constant time if its execution time does not grow with input size. Examples are accessing an element in array or operation with bits.

Linear time: $\Theta(n)$

Algorithm is said to run in linear time if its running time is directly proportional to the input size, i.e. time grows linearly as input size increases. Examples are summing all elements in an array or linearly searching for one.

Quadratic time: $\Theta(n^2)$

Algorithm is said to run in quadratic time, if its running time grows twice faster than the input size, i.e. doubling the input size results in four times longer running time. Typical examples are bubble and insertion sort.

Figure (1.2) illustrates the complexity classes described above.

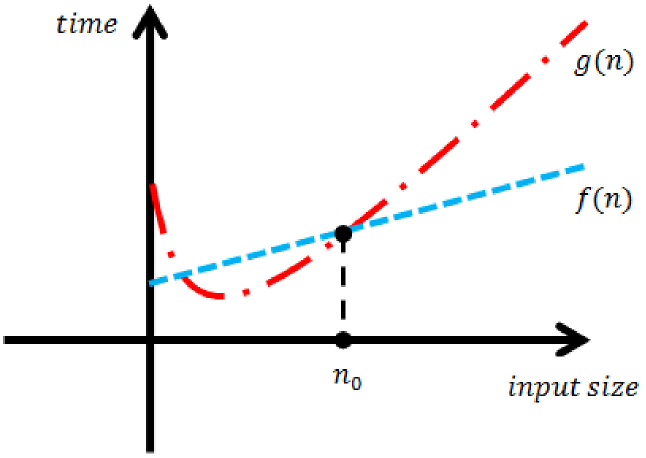

Figure 1.1: Function $g(n)$ grows as fast as function $f(n)$ for sufficiently large $n \to \infty$

1.2 Euclidean space

Let us denote the set of all real numbers by $\mathbb{R}$. The set of all pairs of real numbers by $\mathbb{R}^2$, all triples by $\mathbb{R}^3$, and in general set of all n-tuples by $\mathbb{R}^n$ [4]. Thus,

$$\mathbb{R}^n = \mathbb{R} \times \mathbb{R} \times ... \times \mathbb{R}.$$

Each element in $\mathbb{R}^n$ we shall call a *vector* and denote it by small boldface letter:

$$\mathbf{x} := (x_1, x_2, ..., x_n).$$

We also will regard $(x_1, x_2, ..., x_n)$ as *coordinates* or *dimensions* of a vector $\mathbf{x}$, thus the space $\mathbb{R}^n$, which vector $\mathbf{x}$ belongs to will be referred to as n-dimensional space.

Let us define addition of two elements in $\mathbb{R}^n$ in the following way:

$$\mathbf{x} + \mathbf{y} := (x_1, x_2, ..., x_n) + (y_1, y_2, ..., y_n) = (x_1 + y_1, x_2 + y_2, ..., x_n + y_n),$$

and multiplication by a scalar (real number) $\alpha \in \mathbb{R}$ is:

$$\alpha\mathbf{x} := (\alpha x_1, \alpha x_2, ..., \alpha x_n).$$

Norm and inner product

Define the *norm* or *length* of a vector $\mathbf{x}$ in $\mathbb{R}^n$ as:

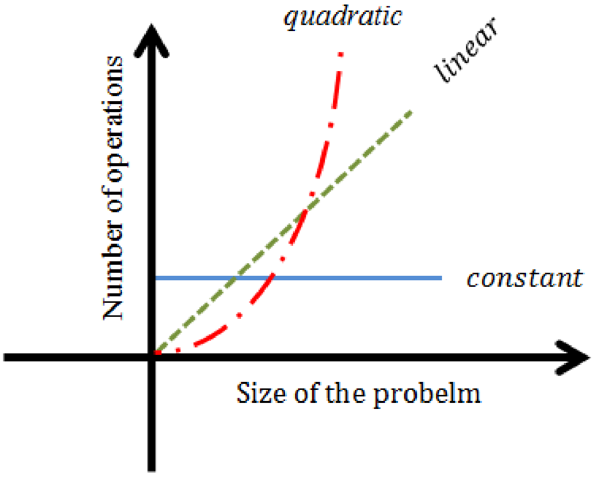

Figure 1.2: Different types of time complexity

$$\|\mathbf{x}\| := \sqrt{\sum_{i=1}^{n} (x_i)^2}. \tag{1.2}$$

For measuring angles in $\mathbb{R}^n$ we define the *inner product* between two vectors $\mathbf{x}$ and $\mathbf{y}$ as follows:

$$\langle \mathbf{x}, \mathbf{y} \rangle := \sum_{i=1}^{n} x_i y_i.$$

The set of vectors in $\mathbb{R}^n$ along with addition, multiplication by a scalar, norm and inner product form an *euclidean space*.

A vector $\mathbf{x}$ in euclidean space is a geometrical object that possess both *magnitude* and *direction*. It can be regarded as an arrow pointing from $(0, 0, ..., 0)$ to $(x_1, x_2, ..., x_n)$. Its magnitude corresponds to its length, its direction to the direction of the arrow. As an example let $\mathbf{x} := (1, 2)$ and $\mathbf{y} := (2, 1)$ in $\mathbb{R}^2$, then Figure (1.3) represents $\mathbb{R}^2$ plane, which contains these two vectors.

The inner product of two vectors, is related to the angle θ between them as follows:

$$\langle \mathbf{x}, \mathbf{y} \rangle := \|x\| \|y\| \cos\theta,$$

After rearranging:

$$\cos\theta = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\| \|\mathbf{y}\|}. \tag{1.3}$$

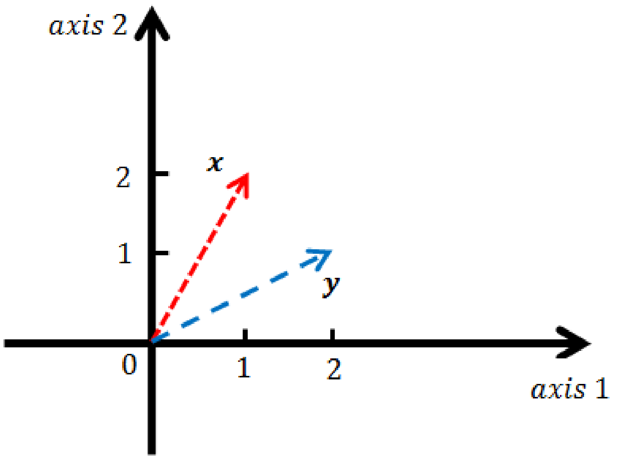

Figure 1.3: Two vectors $\mathbf{x} = (1, 2)$ and $\mathbf{y} = (2, 1)$ in $\mathbb{R}^2$

The distance between points in the euclidean space ("euclidean distance") will be denoted by $d(\mathbf{x}, \mathbf{y})$ and defined as follows:

$$d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\| := \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{1.4}$$

Figure (1.4) shows the particular case of the *euclidean distance* between two vectors $\mathbf{x}$ and $\mathbf{y}$ in $\mathbb{R}^2$ [4].

1.3 Correlation

The correlation of two n-dimensional vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as

$$\operatorname{corr}(\mathbf{x}, \mathbf{y}) = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sigma_x \sigma_y}, \tag{1.5}$$

where

$$\overline{x} = \frac{1}{n}\sum_{i}^{n} x_i,$$

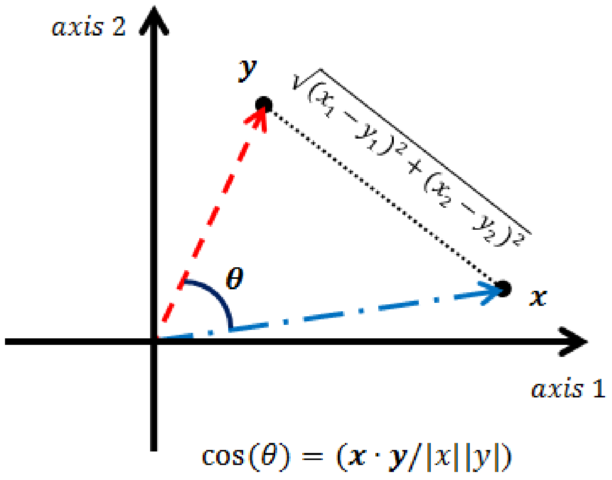
axis 2
y
$\sqrt{(x_1-y_1)^2+(x_2-y_2)^2}$
θ
x
axis 1
$\cos(\theta) = (\boldsymbol{x}\cdot\boldsymbol{y}/|x||y|)$

Figure 1.4: Distance between two points in 2-dimensional *euclidean space* [5]

is the *mean* of x and

$$\sigma_x = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

is the *standard deviation.*

Like euclidean distance, the correlation can be used as a measure of similarity between vectors. For example in bioinformatics, genes with highly correlated expression may often be suspected to have similar functions.

The correlation coefficient as a cosine of an angle between vectors

In the previous section we showed that correlation of two n-dimensional vectors $\mathbf{x} = (x_1, x_2, ..., x_n)$ and $\mathbf{y} = (y_1, y_2, ..., y_n)$ is defined

$$\text{corr}(\mathbf{x}, \mathbf{y}) = \frac{\frac{1}{n}\sum_{i}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\frac{1}{n}\sum_{i}^{n}(x_i - \overline{x})^2}\sqrt{\frac{1}{n}\sum_{i}^{n}(y_i - \overline{y})^2}}.$$

Thus, if we let $\mathbf{x}' = ((x_1 - \overline{x}), (x_2 - \overline{x}), ..., (x_n - \overline{x}))$ and $\mathbf{y}' = ((y_1 - \overline{y}), (y_2 - \overline{y}), ..., (y_n - \overline{y}))$, then

$$\operatorname{corr}(\mathbf{x}, \mathbf{y}) = \frac{\sum\limits_{i=1}^{n} x_i' y_i'}{\sqrt{\sum\limits_{i=1}^{n} (\mathbf{x_i}')^2}\sqrt{\sum\limits_{i=1}^{n} (\mathbf{y_i}')^2}} = \frac{\langle \mathbf{x}', \mathbf{y}' \rangle}{\|\mathbf{x}'\| \cdot \|\mathbf{y}'\|},$$

which, according to (1.3) can be rewritten as follows:

$$\operatorname{corr}(\mathbf{x}, \mathbf{y}) = \cos(\theta), \tag{1.6}$$

where θ is the angle between $\mathbf{x}'$ and $\mathbf{y}'$.

From equation (1.6) it follows that in order to find the correlation between two vectors $\mathbf{x}$ and $\mathbf{y}$ in $\mathbb{R}^n$ it suffices first to find $\mathbf{x}'$ and $\mathbf{y}'$ by subtracting the mean value from each coordinate and then find the cosine of angle between two centered vectors.

Intuitively if $\mathbf{x}$ and $\mathbf{y}$ are two centered n-dimensional vectors in Euclidean space, and they point the same direction, the angle θ between them is 0 and this results in $\cos(0) = 1$, is the highest correlation. When the vectors are pointing opposite directions, then $\cos(180) = -1$, which means the vectors are negatively correlated.

Chapter 2

Nearest Neighbor Indexing

Nearest neighbor indexing is a well known task of preprocessing a dataset of vectors in way that later allows to efficiently answer nearest neighbor queries in terms of Euclidean distance.

One of the first definitions was given probably by Knuth [6] in 1973, since then many algorithms were developed to find a solution to this problem [7, 8, 15, 17, 13, 16]. In our work we rely on three state of the art approaches: coordinate-wise search [12], k-dimensional tree (hereafter KD tree) [10, 11, 17] and random projection tree (hereafter RP tree) [9, 14].

In general the nearest neighbor problem can be defined as follows: given a set of n points $\mathbf{P}$ in a metric space defined over a set $\mathbf{X}$ and distance function F, preprocess $\mathbf{P}$ to efficiently answer queries for finding the closest point $\mathbf{p} \in \mathbf{P}$ for the given point $\mathbf{q}$ in $\mathbf{X}$. In this work we are interested in a particular case when $\mathbf{X}$ is a d-dimensional euclidean space $\mathbb{R}^d$ [7].

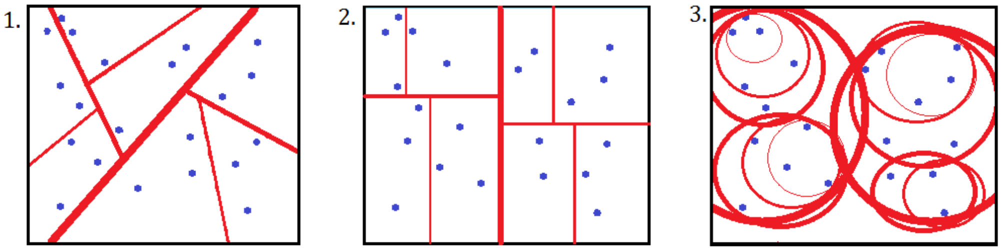

Figure 2.1: Partitioning 2-D dataset with different methods of Nearest Neighbor Indexing. 1. Random Projection tree, 2. K-dimensional tree, 3. Ball tree

2.1 Curse of dimensionality

According to Tsaparas [18], the problem of detecting nearest neighbors for a given query point in a d-dimensional dataset was solved optimally for the case of low dimensions. For example if points lie on a 2-dimensional plane the nearest neighbor can be found in logarithmic time per query and using just $O(n)$ of storage space [19].

Problems arise when the number of dimensions in the dataset starts growing. In this case current knowledge [20, 18, 19, 8, 21] suggests brute force scan through all points to be the only optimal solution, because performance of other methods degrades exponentially in running time and/or storage space according to Sariel Har-Peled [19]. This phenomenon is known as the "curse of dimensionality" [22].

Thus, the problem of detecting an exact nearest neighbor is believed to be unsolvable in linear or even subquadratic time. Instead, allowing algorithms to report approximate results within some reasonable distance from the query point (for example c times distance between $\mathbf{q}$ and closest point $\mathbf{x_i}$) may considerably speed up the running time. Numerous studies show that approximate solutions can be as useful as exact ones for most real-world problems at the same time achievable in linear time [20, 18].

2.2 Methods for nearest neighbor indexing

In our work we rely on three state of the art approaches: coordinate-wise search [12], KD tree [10, 11, 13] and RP tree [9, 14].

The following algorithms consist of two major functions: *indexing* and *querying*. By *indexing* we imply the process of constructing a data structure by assigning values (indexes) to the points in the original dataset. This data structure later allows to efficiently reduce the search space and thus faster answer nearest neighbor queries [23].

2.2.1 Coordinate-wise search

The method by Nene et al. [12], which we refer to as "coordinate-wise search" uses precomputed data structure and binary search to efficiently find points sandwiched between two parallel hyperplanes. The data structure is constructed using the original set of points and is depicted on Figure (2.2). Original dataset is stored as a collection of 1-dimensional arrays such that j^{th} array corresponds to the j^{th} coordinate of the point set.

To find nearest neighbors for a given query point $\mathbf{q} = (q_1, q_2, ..., q_d)$ we need to construct an initial candidate list by selecting points from the dataset

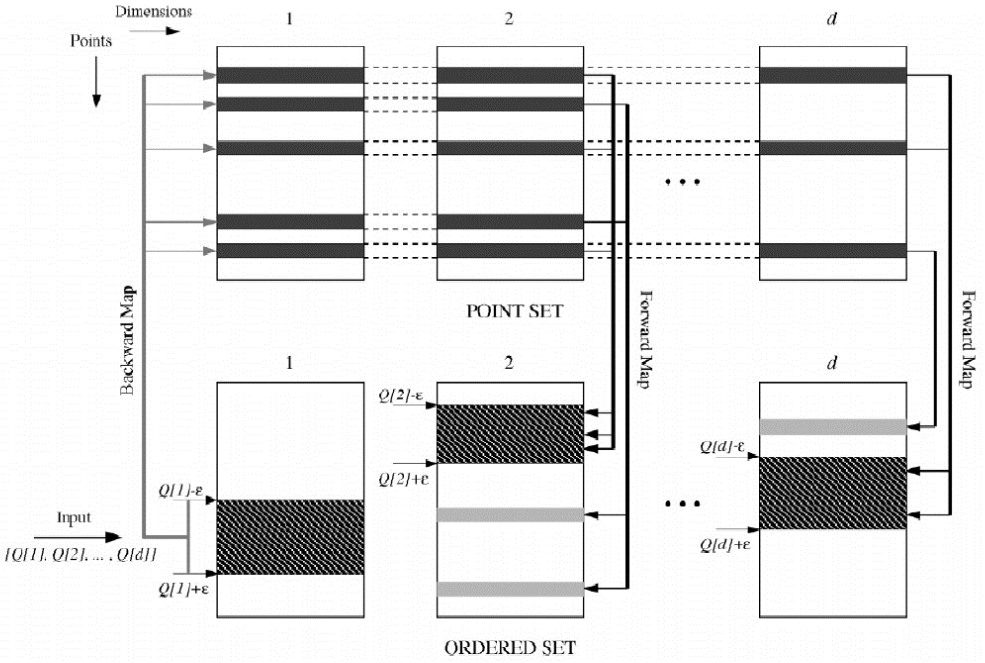

Figure 2.2: Data structure for efficient nearest neighbor queries. Backward and forward maps link together ordered (sorted) and original datasets [12].

whose first coordinates lie within limits $q_1 - \varepsilon$ and $q_1 + \varepsilon$. This is done with the help of two binary searches, one per each limit, providing 1-dimensional arrays are sorted [12].

Thus, in order to use binary search (see Algorithm 1) we need to sort the collection of 1-dimensional arrays and store them in an *ordered* set. To preserve the connection between coordinates, we maintain two maps. The *backward* map links coordinates in the ordered set to the corresponding in the original set and $forward$ map that maps coordinates from the original set to the coordinates in ordered one [12]. Note, that both maps are simple integer arrays.

Thus, using backward map we find points that lie in between hyperplanes positioned at $q_1 - \varepsilon$ and $q_1 + \varepsilon$ and add them to the candidate list. Next, we trim the candidate list iterating through $k = 2, 3, ..., d$ as follows. On the iteration k, we check k^{th} coordinate of the every point from the candidate list to confirm if it lies within boundaries $q_k - \varepsilon$ and $q_k + \varepsilon$. Each of this limits is also found using binary search. Points whose k^{th} coordinate lies outside these limits are discarded from the list.

At the end of the last iteration, the candidate list contains points that belong to the hypercube (Figure (2.3)) of side 2ε, centered at $\mathbf{q}$. A linear scan over these points finds the closest point to $\mathbf{q}$. The pseudo code of the

```
**Algorithm:** Binary search
**Input**: sorted array **x**, searched value *key*, *min* and *max* values of an
        array **x**
**Output**: the position of a *key* within **x** or the empty value.
while $min < max$ do
    mid ← median(min, max);
    if $x[mid] < key$ then
        min ← mid + 1;
    end
    else
        max ← mid;
    end
end
if $x[min] = key$ then
    return min;
end
else
    return “query point was not found”
end
```

Algorithm 1: Binary search

coordinate-wise search is given in Algorithm (2).

By choosing ε wisely, the intersection between sets of neighbors can be made small enough, so that the computation is much faster than straightforward exhaustive search. Nevertheless, when the number of points in the initial candidate list is large, then $O(n)$ time needed to iterate over all of them when checking the values of k^{th} coordinates. This results in the overall complexity of $O(n^2)$.

2.2.2 K-dimensional tree

The KD Tree is a binary search tree. It is a data structure, which partitions space hierarchically with hyperplanes placed perpendicular to the coordinate axes [13]. Each node of the tree represents a subset of points and a partitioning of this subset. The *root* node represents the whole set. Each nonterminal node has two successors (children). These are the two subsets obtained by the partitioning specified in the node. Terminal nodes (leaf nodes) are small, mutually exclusive subsets of the data points that collectively form the entire search space [17].

In 1-D case when points are represented by a single coordinate, the par-

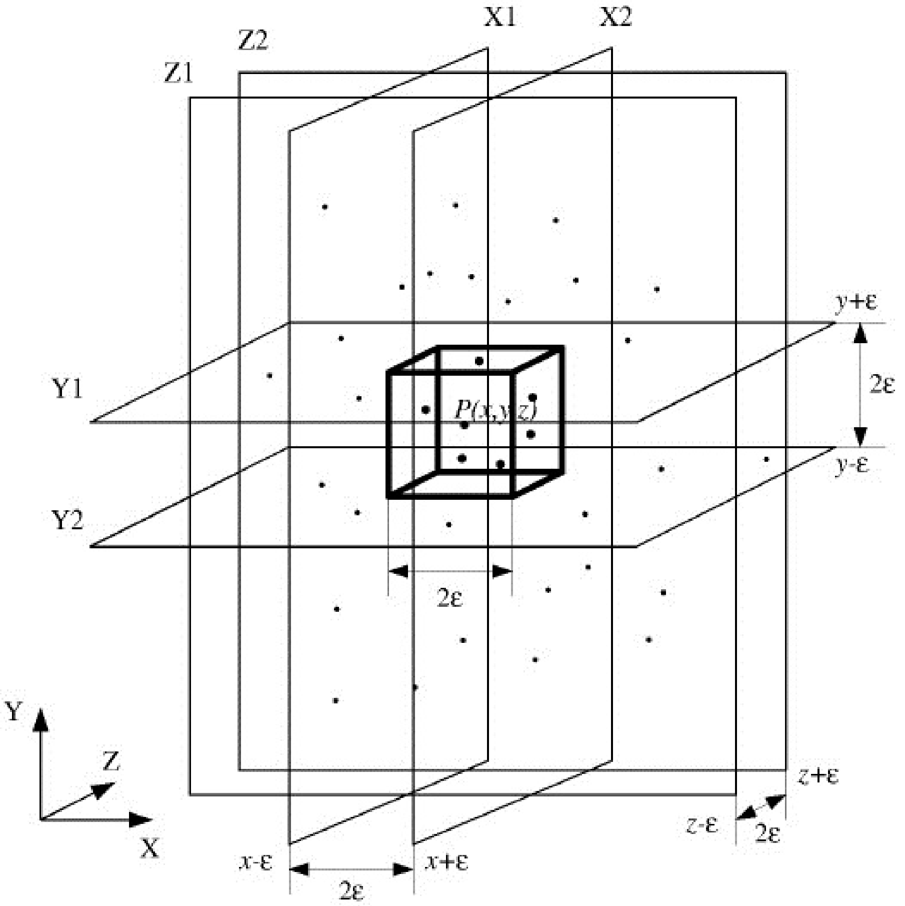

Figure 2.3: Coordinate-wise search efficiently finds points inside a cube of size 2ε around the query point $\mathbf{q}$. The closest point is then found by performing an exhaustive search of all points within the cube using Euclidean distance [12]

tition is defined by a certain value of this coordinate. All the points in the node with coordinates less than a partitioning value are assigned to the left branch of the tree and points with values of the coordinates greater than the partitioning value are added to the right branch. We shall call the partition value *discriminator*.

In k dimensions, each point is represented by k coordinates. Every one of those can serve as a coordinate for partitioning (the splitting index) and thus determine the partitioning value. Originally, the splitting index was chosen depending on the current level of the node in the tree [10].

In this work, we randomly choose the splitting index from the interval $\{1...k\}$. The partitioning value is then chosen to be the median of the array of corresponding coordinates. Both the splitting index and the partitioning

```
Algorithm: Coordinate-wise search
Input: original set P, ordered set O, backward B and forward F
       maps, $\varepsilon$, and query point q.
Output: nearest neighbor index
top ← an upper bound q[1] + $\varepsilon$ for the value of the first coordinates;
bottom ← a lower bound q[1] - $\varepsilon$ for the value of the first coordinate;
m ← top - bottom;
for $j \leftarrow 0$ to $m$ do
    initialize candidateList with points $x_j$ whose $1^{st}$ coordinates lie in
    between top and bottom;
end
for each new coordinate i from d do
    top ← a new upper bound q[i] + $\varepsilon$ for $i^{th}$ coordinate;
    bottom ← a lower bound q[i] - $\varepsilon$ for $i^{th}$ coordinate;
    for every point $x_j$ in candidateList do
        if $i^{th}$ coordinate of $x_j$ lie within limits top and bottom then
            add $x_j$ to candidateList;
        end
        else
            discard $x_j$;
        end
    end
    m ← new size of the candidateList;
end
nearestPoint ← exhaustiveSearch(candidateList,q);
return nearestPoint;
```

Algorithm 2: Coordinate-wise search [12]

value is stored in the node. We also introduce an upper and lower bound for the size of the leaf nodes. That is the number of points stored in the leaf node cannot be less than 2 and greater than 3. Code for the KD tree construction is given in the Algorithm (4).

Let us proceed with a short example of partitioning with the KD tree. Consider a 2-dimensional dataset that consists of seven points with coordinates: (1,2), (4,3), (8,4), (3,5), (1,8), (6,5), (7,7). We start from choosing value for the splitting index. In our example the partitioning can be done by either the first or the second coordinate. Let the first splitting index be 1. Then, we compute the median over the array of 1^{st} coordinates of original points: (1,1,3,4,6,7,8), which is 4. Next, all the points whose 1^{st} coordinate

```
Algorithm: exhaustiveSearch
Input: list of points, query point $\mathbf{q}$
Output: nearestPoint
for every point $\mathbf{x_i}$ from the list do
    calculate currentDistance between $\mathbf{q}$ and $\mathbf{x_i}$;
    if currentDistance <minimumDistance then
        minimumDistance ← currentDistance;
        nearestPoint ← $i$;
    end
end
return nearestPoint;
```

Algorithm 3: Exhaustive search

is lower than 4 are added to the left branch and points with 1^{st} coordinate bigger than 4 are added to the right branch respectively. By now, we defined points the belong to the left child ((1,2),(1,8),(3,5)) and right child ((4,3),(6,5),(7,7),(8,4)) of the root node. Note that left child cannot be split further because sizes of its successors will not satisfy the lower bound restriction (≥ 2). We shall proceed with splitting right child of the root note, by again randomly choosing a splitting index and computing the median value for the corresponding coordinates. If splitting index would be chosen to be 2, then the median for the array (3,4,5,7) is 5. Then, points on the left are (4,3) and (8,4),and points on the right (6,5) and (7,7). Figure (2.4) presents resulting KD tree.

In order to answer nearest neighbor query for the given query point **q** using KD tree data structure, we need first to traverse the tree starting from the root and and moving to either left or the right child depending on the information stored in the splitting nodes. After we found the node which **q** should be assigned to, all the points within this node are checked for the closest in terms of euclidean distance to the query point.

When descending the tree the question arises which child to choose to traverse next, left or right. For that we use splitting index and partitioning value stored in the nonterminal nodes of the tree. First, the value of the coordinate with splitting index from **q** is compared to the partitioning value, if it is bigger then we proceed with choosing right children of the node and left child otherwise.

The search time using KD tree has been proved to be $O(log(n))$ even in case of randomly distributed points [17].

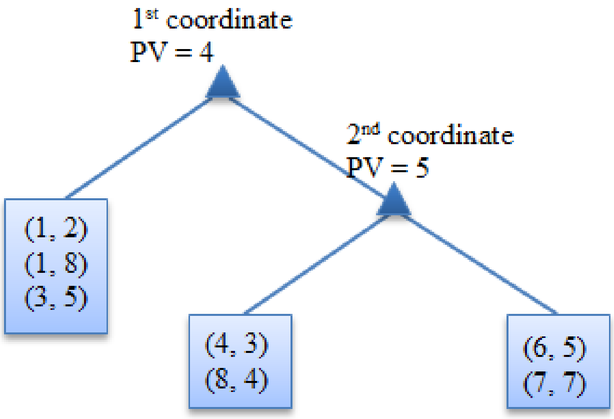

Figure 2.4: Resulting KD tree data structure for the example with seven points in $\mathbb{R}^2$

2.2.3 Random projection tree

Random projection tree (RP tree) uses a similar recursive construction procedure as the KD tree. At the beginning of each new iteration, instead of randomly choosing a splitting index it selects two random points from the given subset and places a *splitting hyperplane* between them. Then, similarly to KD tree, points to the left side of the hyperplane are added to the left branch of tree and points to the right side are added to the right branch, see Algorithm (6).

Formally, let $\mathbf{z}$ and $\mathbf{y}$ be uniformly chosen points from the original set of points. A hyperplane placed between points is defined as the set

$$\{\mathbf{x} : \langle \mathbf{w}, (\mathbf{x} - \mathbf{p}) \rangle = 0\} \tag{2.1}$$

where

$$\mathbf{w} = \mathbf{z} - \mathbf{y},$$
$$\mathbf{p} = \frac{\mathbf{z} + \mathbf{y}}{2}.$$

The hyperplane splits the points in the dataset in two - points $\mathbf{x}$ for which $\langle \mathbf{w}, (\mathbf{x} - \mathbf{p}) \rangle > 0$ go to the left branch, and those for which $\langle \mathbf{w}, (\mathbf{x} - \mathbf{p}) \rangle < 0$ –

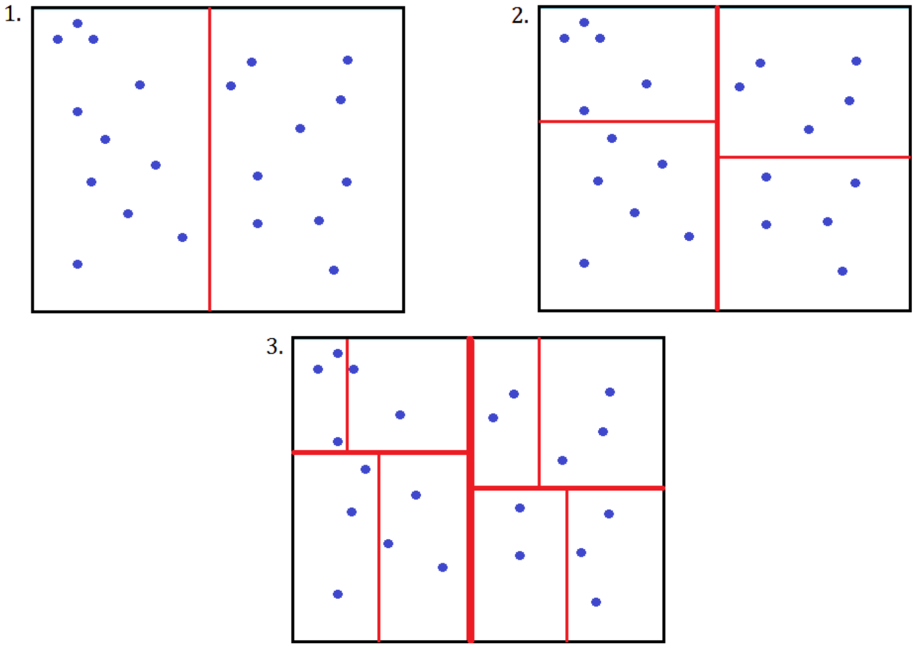

Figure 2.5: The partitioning of a 2-D dataset using KD tree. Line thickness denotes partitioning order (thicker lines were partition first)

to the right.

It is worth mentioning that the KD tree in some sense uses a similar mechanism when splitting points. Let j be the index of randomly chosen coordinate from the set $\{1..d\}$. Then the splitting hyperplane can be defined as:

$$\{\mathbf{x} : \langle \mathbf{w}, \mathbf{x} \rangle = m\},$$

where $\mathbf{w}$ is a vector of weights that in case of KD tree has zeros at all positions except the jth, and m is the partitioning value.

```
Algorithm: KDTreeIndexing
Input: set of points $\mathbf{x}$
Output: instance of the KD tree
if the number of input points < limit then
    return new KD tree node;
end
else
    sIndex ← random(1,d);
    m ← calculateMedian;
    for each point $x_i$ from $\mathbf{x}$ do
        if $x_i[sIndex] > m$ then
            leftBranch.add($x_i$);
        end
        else
            rightBranch.add($x_i$);
        end
    end
    new node KD tree();
    newNode.median = m;
    newNode.splitInx = sIndex;
    newNode.leftBranch ← KDTreeIndexing(leftBranch);
    newNode.rightBranch ← KDTreeIndexing(rightBranch);
end
return node;
```

Algorithm 4: KD tree indexing

```
Algorithm: KD tree query
Input: kd tree node and query point q
Output: kd tree node containing query point
if both child nodes are empty then
    return node;
end
else
    splittingIndex = node.splitInx;
    if q[splittingIndex] > node.median then
        return searchKDtree(node.rightBranch);
    end
    else
        return searchKDtree(node.leftBranch);
    end
end
```

Algorithm 5: KD tree query

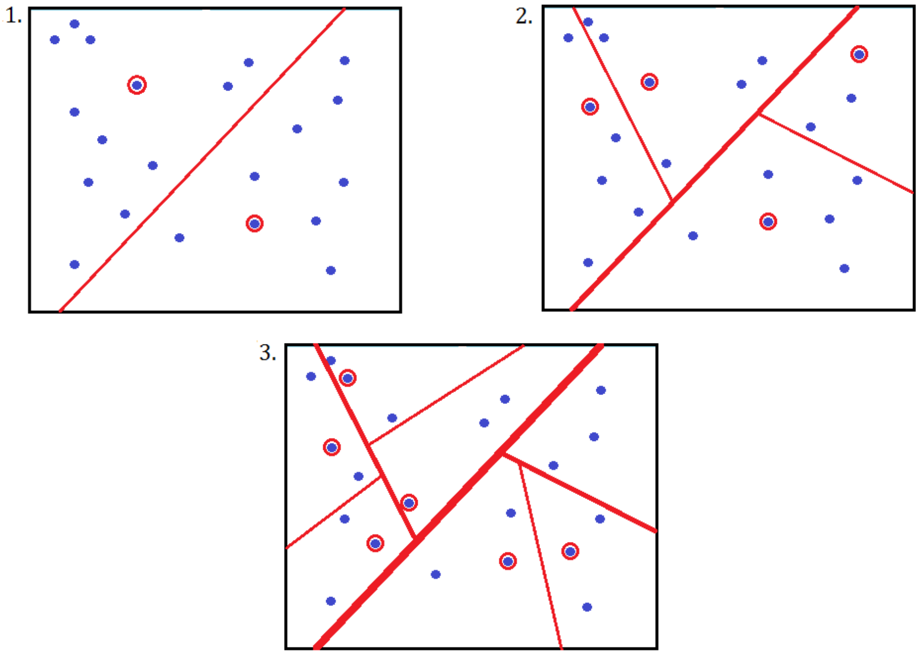

Figure 2.6: The partitioning of a 2-D dataset using RP tree. Points surround by red circles correspond to the randomly chosen points based on which new hyperplanes where placed

```
Algorithm: RP tree indexing
Input: set of original points.
Output: RP tree node
if the number of input points < limit then
    return new RP tree node;
end
else
    using coordinates of the original points createHyperplane();
    new node();
    store median and vector of weights in the node;
    for each point in the original dataset do
        if classifyPoint >0 then
            assign point to the right branch of the tree;
        end
        else
            assign point to the left branch of the tree;
        end
    end
    call function RPTreeIndexing for the left branch of the node;
    call function RPTreeIndexing for the right branch of the node;
end
return RP node;
```

Algorithm 6: Function RPTreeIndexing

```
Algorithm: createHyperplane
Input: set of points
Output: vector of weights $\mathbf{w}$
$\mathbf{z}$,$\mathbf{y}$ $\leftarrow$ two random distinct points from the input dataset;
$\mathbf{w} \leftarrow \mathbf{z}$ - $\mathbf{y}$;
$\mathbf{p} \leftarrow \frac{\mathbf{z}+\mathbf{y}}{2}$;
w[0] $\leftarrow \mathbf{p}$;
return $\mathbf{w}$;
```

Algorithm 7: Function createHyperplane

```
Algorithm: RP tree query
Input: RP tree node, query point q.
Output: RP tree node containing query point q.
if both child nodes are empty then
    return node;
end
else
    if classifyPoint(q,node) then
        return searchRPtree(node.rightBranch);
    end
    else
        return searchRPtree(node.leftBranch);
    end
end
```

Algorithm 8: RP tree query

```
Algorithm: classifyPoint
Input: Vector of weights w from the node, query point q
Output: true in case node is on the right from the splitting
        hyperplane or false otherwise
s ← $\sum w_i q_i - w_0$;
return (s$\geq$ 0);
```

Algorithm 9: Function classifyPoint

Chapter 3

The Max-correlation Problem

The problem of detecting most correlated pairs in large multi-dimensional dataset is very common for a large variety of real-world applications such as image recognition, signal processing and bioinformatics. Figure (3.1) shows a partial view of the Promoterome Matrix by Gu et al. [24] that consists of 9 242 target genes (rows) and 333 target factors (columns). Thus, to find the most correlated pair among those genes we need to perform at least 9 242 · (9 242 - 1)/2 · 333, which is approximately 14 billions of operations. The process of calculating all pair-wise correlations for this dataset may take hours or even days on the researcher's computer.

3.1 Reduction to nearest neighbor search

In this work we propose to improve the time needed to compute the most correlated pair(s), by reducing the problem to the task of finding closest points in Euclidean space.

The correlation of two vectors $\mathbf{x}$ and $\mathbf{y}$ can be computed by first standardizing these vectors and then taking their inner product.

$$\operatorname{corr}(\mathbf{x}, \mathbf{y}) = \langle \tilde{\mathbf{x}}, \tilde{\mathbf{y}} \rangle \,, \tag{3.1}$$

where

$$\begin{aligned} \tilde{\mathbf{x}} &= (\mathbf{x} - \overline{x})/\sigma_x, \\ \tilde{\mathbf{y}} &= (\mathbf{y} - \overline{y})/\sigma_y, \end{aligned}$$

are standardized vectors.

The Euclidean distance between $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ is thus equal to:

$$\|\tilde{\mathbf{x}} - \tilde{\mathbf{y}}\|^2 = \|\tilde{\mathbf{x}}\|^2 + \|\tilde{\mathbf{y}}\|^2 - 2\langle \tilde{\mathbf{x}}, \tilde{\mathbf{y}} \rangle = 2d - 2\langle \tilde{\mathbf{x}}, \tilde{\mathbf{y}} \rangle, \tag{3.2}$$

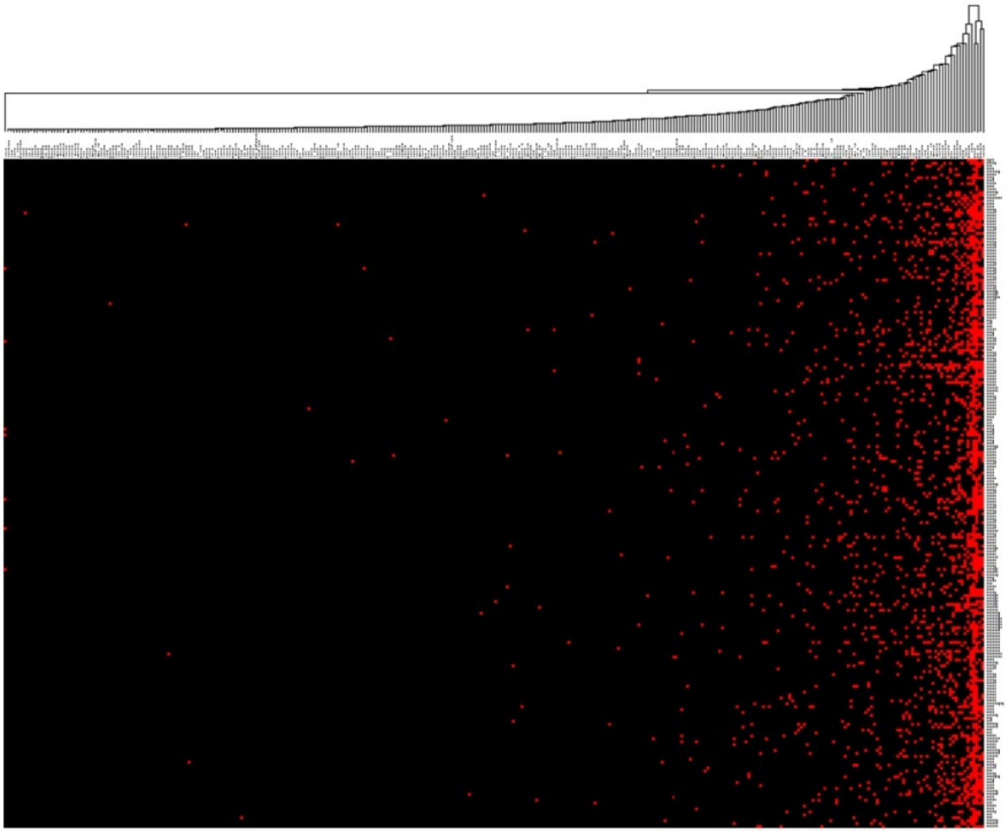

Figure 3.1: Partial view of the Promoterome Matrix (P-matrix) with 9,242 target genes in rows and 333 target factors in columns [24].

where $\|\tilde{\mathbf{x}}\|^2 = \|\tilde{\mathbf{y}}\|^2 = d$ due to standardization and $2\langle\tilde{\mathbf{x}}, \tilde{\mathbf{y}}\rangle$ equals to $2 \cdot \text{corr}(\mathbf{x}, \mathbf{y})$ according to equation (3.1) .

From equation (3.2) it follows, that in order to find $\mathbf{x}$ and $\mathbf{y}$ with the maximum correlation among a set of n points it suffices first to standardize all vectors in the dataset and then find $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ with the smallest possible Euclidean distance in between (see Algorithm (10)). Thus, the problem of searching for the most correlated pair reduces to the task of finding two closest neighbors.

```
Algorithm: Max-correlation
Input: index and query datasets
Output: indexes of two most correlated points
for every point $\mathbf{x}$ and $\mathbf{y}$ in query and index datasets do
    $\tilde{\mathbf{x}} = (\mathbf{x} - \overline{x})/\sigma_x$;
    $\tilde{\mathbf{y}} = (\mathbf{y} - \overline{y})/\sigma_y$;
end
index $\{\tilde{\mathbf{x}}_\mathbf{i}\}_i$ using a nearest neighbor method;
for every point $\tilde{\mathbf{y}}_i$ do
    find nearest neighbor $\tilde{\mathbf{x}}'$ for $\tilde{\mathbf{y}}_i$ in $(\tilde{\mathbf{x}})$;
    currentBest = $\mathrm{corr}(\tilde{\mathbf{x}}', \tilde{\mathbf{y}}_i)$;
    if currentBest $>$ max then
        max $\leftarrow$ currentBest;
        bestPair $\leftarrow \tilde{\mathbf{x}}', \tilde{\mathbf{y}}_i$
    end
end
return bestPair
```

Algorithm 10: Max-correlation for two datasets

Chapter 4

Experimental Evaluation

We divided our tests into two parts. In the first part we measured the running time performance of our solution for all algorithms with the respect to both growth of dimensionality and number of points in the dataset. In the second part, we run experiments that captured the quality and thus reliability of our solution. To show this, we computed and plotted the accuracy of our approximate algorithms with the respect to the exact solution previously found by the the linear scan and further comparison of all possible pairs of points(hereafter brute force approach). Therefore, we calculated the ratio between exact correlated pair and results found by KD tree and RP tree. Finally, we plotted counts for number of times our algorithms found intentionally inserted pair with the highest correlation in the dataset.

All tests were run on the laptop computer (Samsung R590, Intel Core i3 2,27 GHz CPU processor and 4GB RAM).

4.1 Time evaluation

Data points for our tests were generated from the uniform distribution $U(0, 100)$. First, we measured the running time for the brute force approach in comparison with coordinate-wise search on datasets with gradually increasing number of points from 2000 to 32000 (Figure (4.1)). Hence, even though execution time for both algorithms is clearly quadratic, coordinate-wise search in our implementation is at least 2 times faster than straightforward the brute force approach. Figure (4.2) introduces analogous comparison between KD tree and RP tree performance time for the same datasets, it follows that both algorithms are linear in running time.

Secondly, we tested how the performance of our approach depends on number of dimensions of the dataset. As a result, Figure (4.3) shows that the performance of both our approximate algorithms stays linear. Alternatively

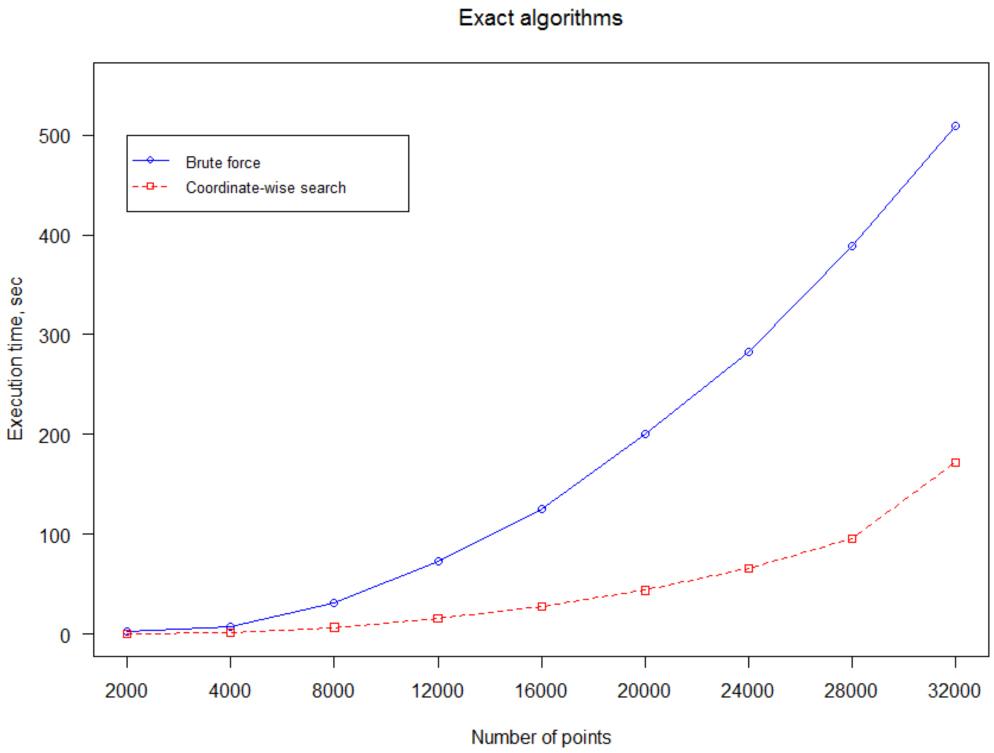

Figure 4.1: Execution time in seconds with the respect to the number of points for Brute force and coordinate-wise search

Figure (4.3) presents results of this test for the exact algorithms.

To summarize the impact of size of a dataset on overall performance of our solution, we run tests on different datasets both in terms of number of dimensions and number of data points. Two figures were obtained, one per each method (Figure (4.5) and Figure (4.6)). Each of those figures contain 9 curves, that represent datasets with different number of dimensions (from 20 to 100). Each of those datasets had size ranging from 1000 to 32000 data points. Figure (4.5) represents results of the test for KD tree and Figure (4.6) for RP tree. It can be underlined that running time of RP tree method in our implementation rises faster than in KD tree, but still remains linear. We did not run these tests for the brute force and coordinate-wise search due to very long execution time.

4.2 Quality evaluation

Another important issue that we studied in this work is quality our of approximate results. To observe how accurate our answers are, we first used

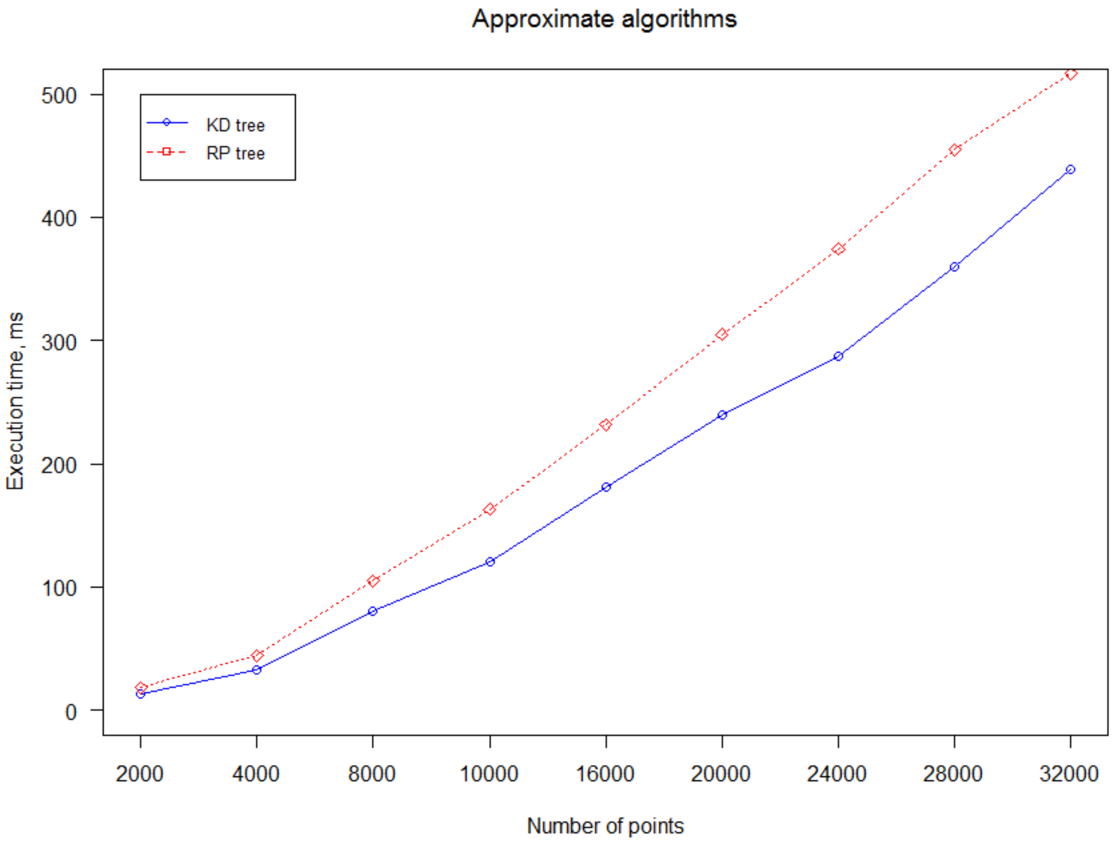

Figure 4.2: Execution time in milliseconds with the respect to the number of points for KD tree and RP tree

the brute force algorithm to find exact most correlated pair in nine randomly generated datasets with fixed number of points $n = 10000$ and number of dimensions d randing from 20 to 100. For each dataset we were were dividing approximate results of KD and RP tree by the exact solution previously found by brute force.

Figure (4.7) shows that even though the accuracy of both methods degrades with the number of dimensions, it remains quite high (over 80% accuracy) even for the number of dimensions in the dataset as high as 100.

Next, we experimented with the maximum number of points that are allowed in the leaf node (default is 3). Figures (4.8) and (4.9) illustrate how the accuracy rate was changing with the maximum number of point changing from 3 to 16 and the number of dimensions ranging from 20 to 100 for both hierarchical data structures.

Alternatively, to evaluate and compare the accuracy of both our approximate methods we generated one pair with the precomputed correlation, large

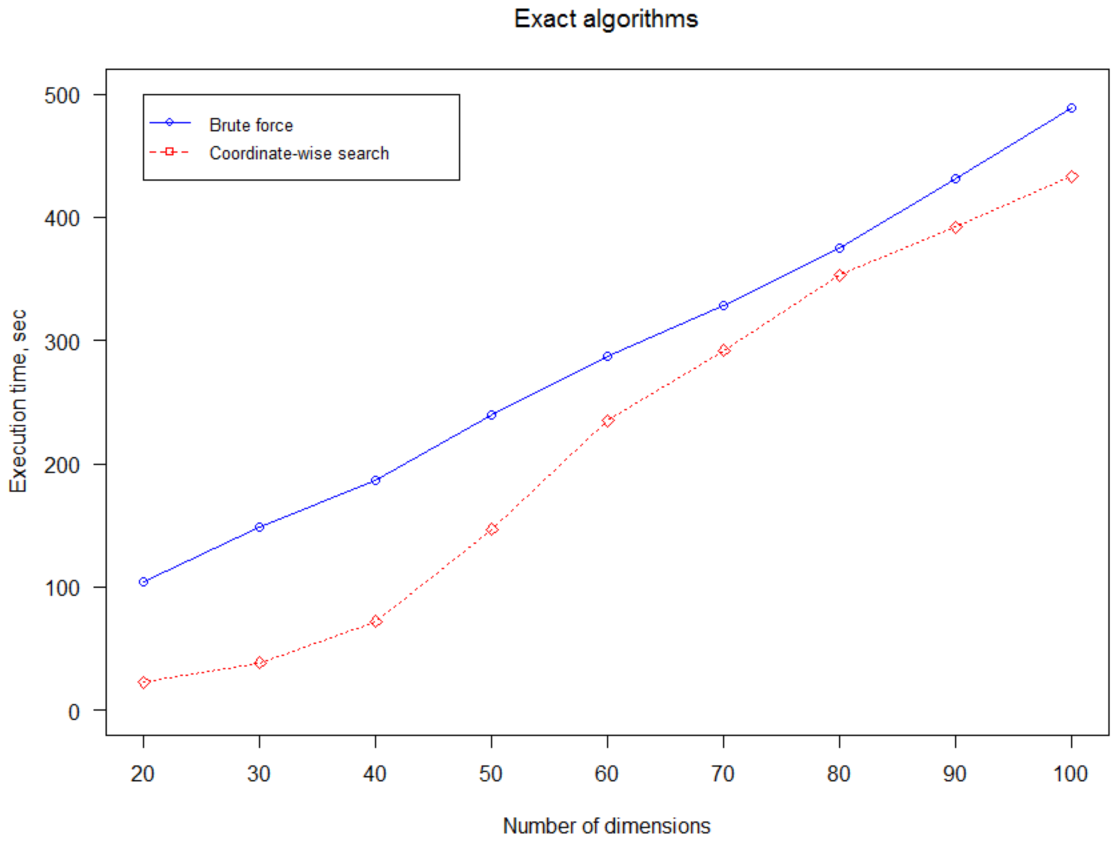

Figure 4.3: Execution time with the respect to the number of dimensions for the brute force and coordinate-wise search

enough to be the highest in among all within tested dataset but not too high so algorithms would be capable of detecting it every time. It was obtained via copying of a random row and adding a noise generated from the normal distribution with mean value 50 and standard deviation 23. This precomputed correlated pair was not changing during the experiment.

We made 25 runs per each method each run giving 20 attempts to find precomputed pair. Thus, $25 \times 20 = 500$ launches. Results of every run we recorded and added to the final table. All the tests were conducted using random dataset with fixed number of rows and columns, 30000 and 40 respectively.

Figure (4.10) presents a distribution of an accuracy for KD tree and RP tree. On the x-axis number of times inserted pair was found per on test (20 attempts - maximum), and the distribution of corresponding counts on the y-axis. From this figure we may conclude that RP tree has higher accuracy in comparison with KD tree data structure. From figure (4.11) it follows that on average execution time of RP tree is longer than for the KD tree.

4.3 Statistical significance

When dealing with very large datasets we face the problem of *multiple testing*, namely when searching for the most correlated pair of vectors in a large dataset, we may obtain very high correlation values just by chance. Figure (4.12) shows the empirical structure of highest correlation values detected in artificially generated datasets with different number of points and dimensions. Each level on this figure corresponds to the best correlation coefficients that our approximate methods found on that random data. Thus, for example, correlation 0.7 obtained on a dataset with a size 300 000×60 is unlikely to be related to a true association.

4.4 Application in bioinformatics

There are many important types of biological data such as methylation data (which contains measurements of methylation values for different genomic positions) or expression data (which measures levels of expression of genes in different individuals). The question of finding co-expressed genes or genes highly correlated to methylation at some sites within one or multiple datasets is of great interest in biology. To illustrate the applicability of our method to this case we tested our solution on a dataset containing methylation values for 463 143 genes in 84 different individuals. KD tree running time was 317 seconds while RP produced results after 1723 seconds. All listed in the results genes had a correlation coefficient from 0.998 to 0.79. From these results we can conclude that correlation coefficients detected by our methods are significantly bigger than values we could expect to obtain by chance as shown in the Figure (4.12). Additionally, we found that some of the listed genes are located far away, one from another, which would not be possible to detect using existing biological tools for the data analysis.

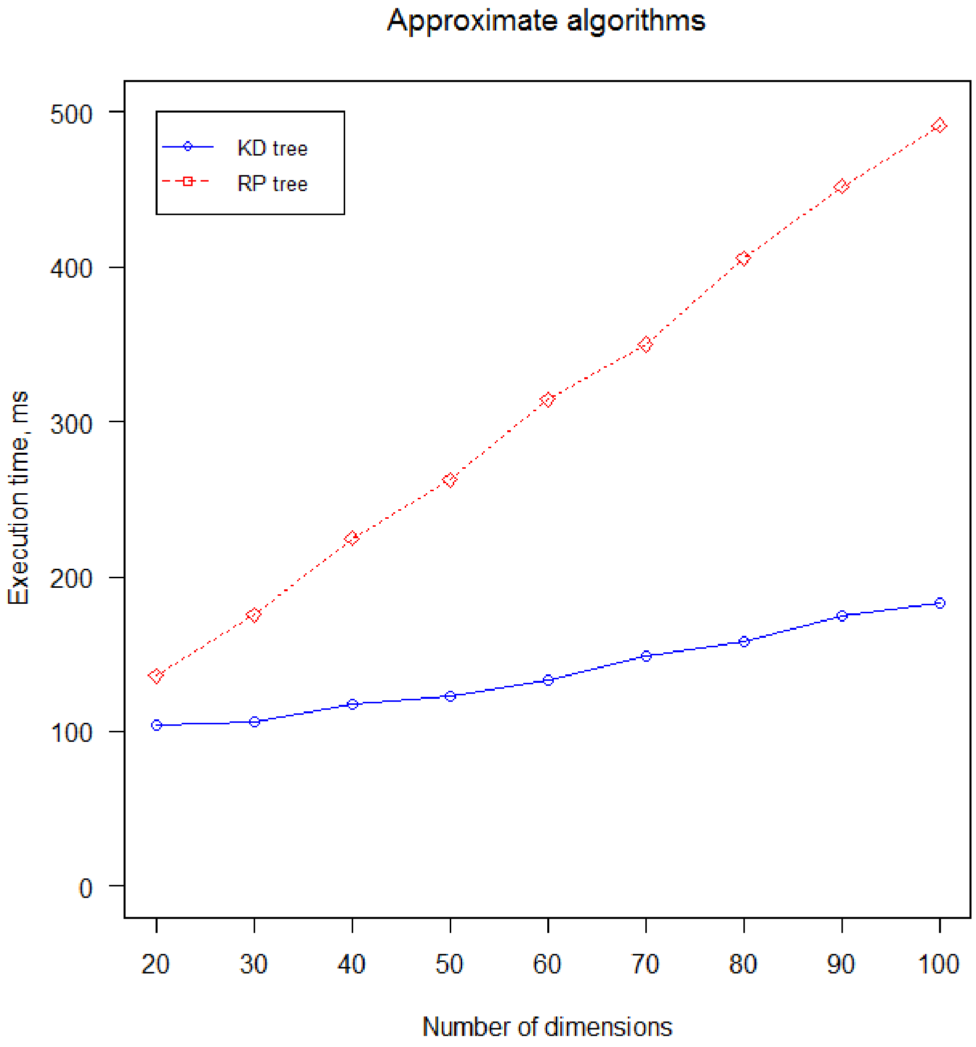

Figure 4.4: Execution time with the respect to the number of dimensions for KD tree and RP tree

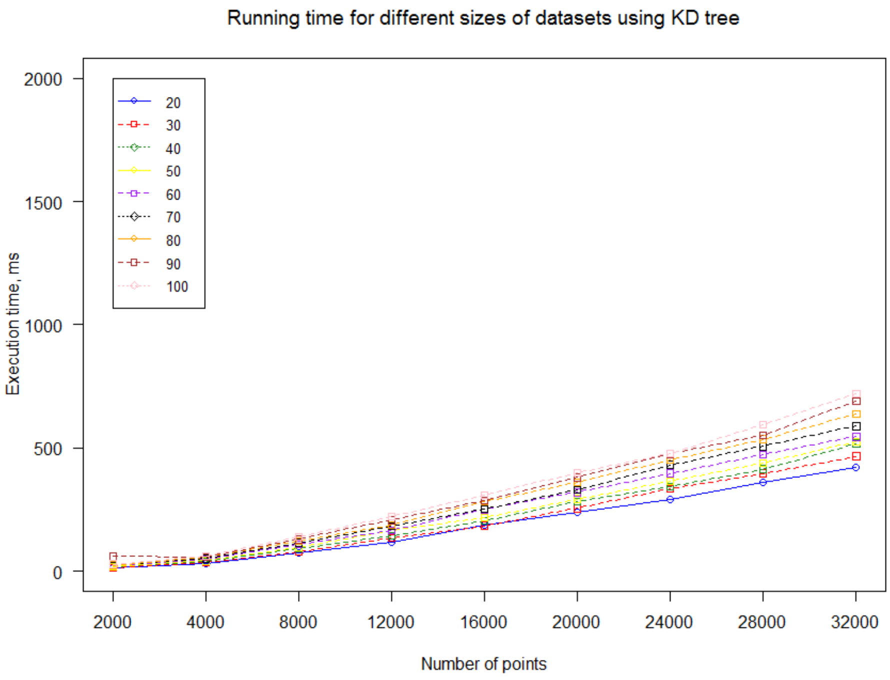

Figure 4.5: Execution time with the respect to the growing number of dimensions for the KD tree

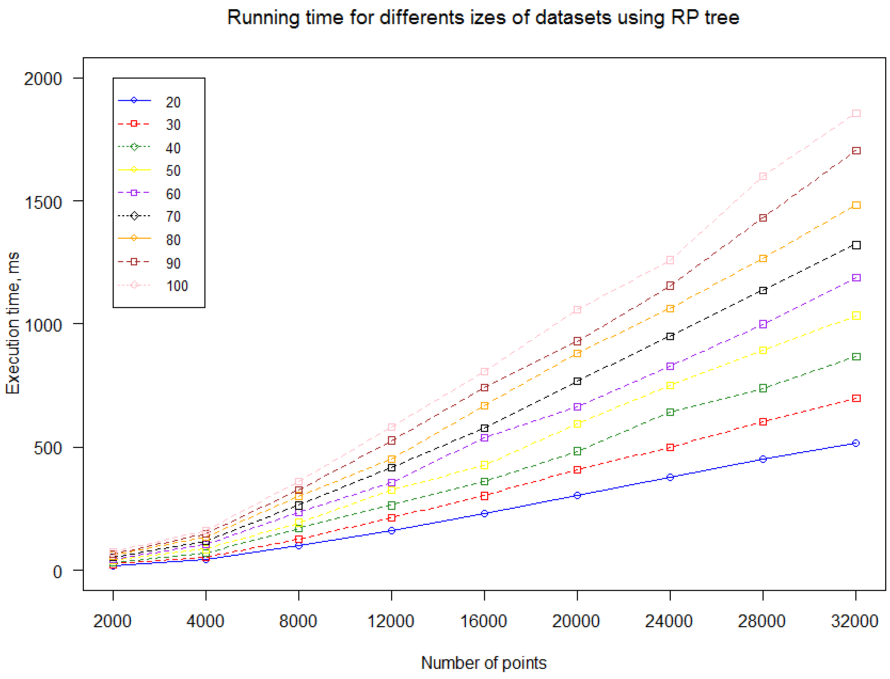

Figure 4.6: Execution time with the respect to the growing number of dimensions for the RP tree

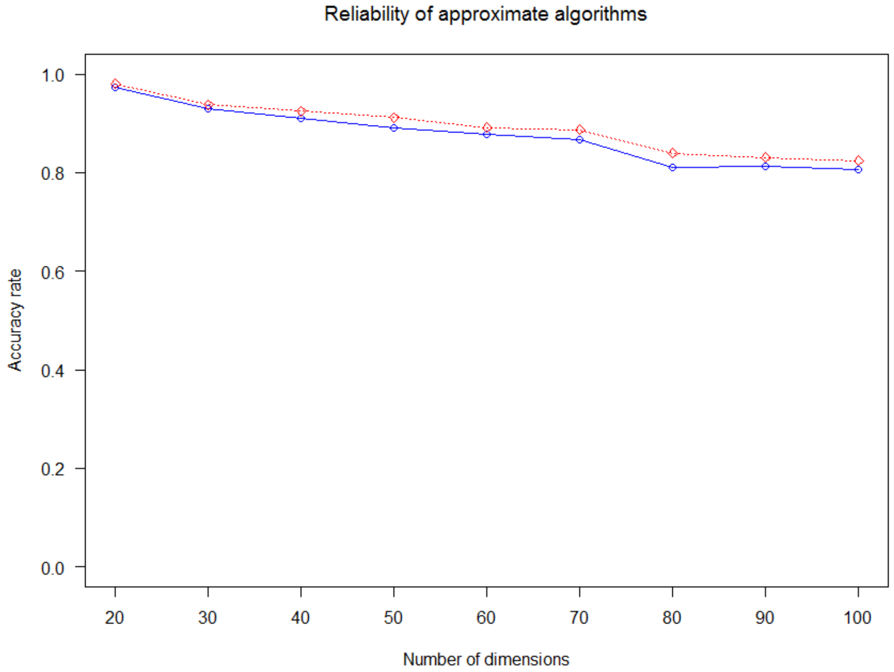

Figure 4.7: Accuracy rate degrades with number of dimensions for both KD and RP trees.

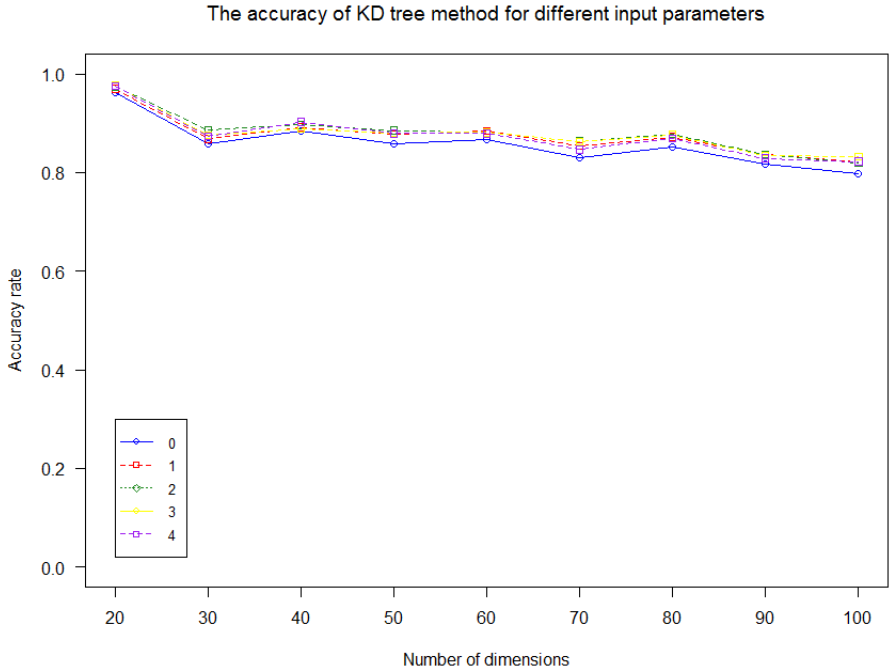

Figure 4.8: The accuracy rate for the KD tree data structure

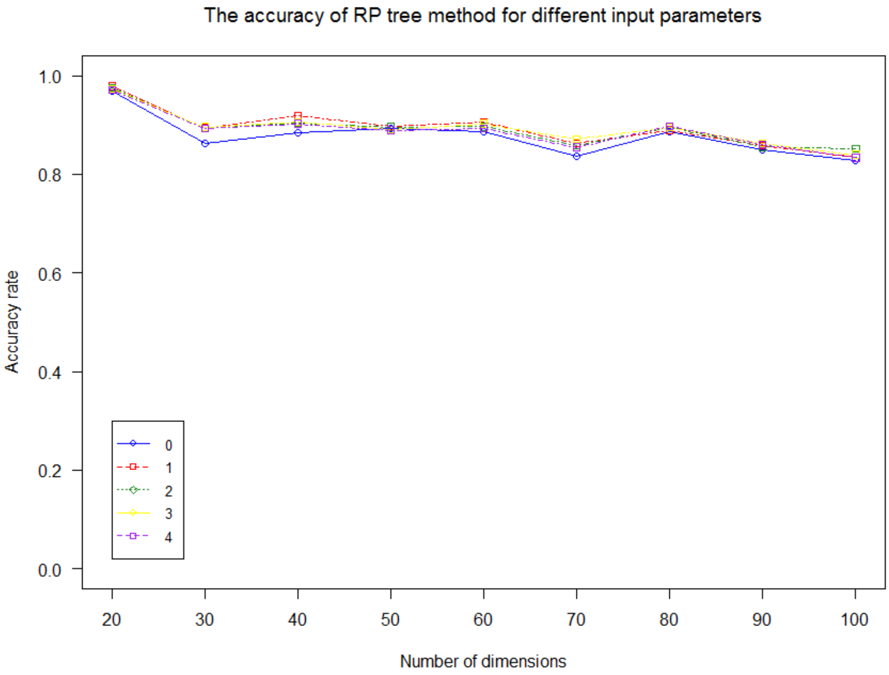

Figure 4.9: The accuracy rate for the RP tree data structure

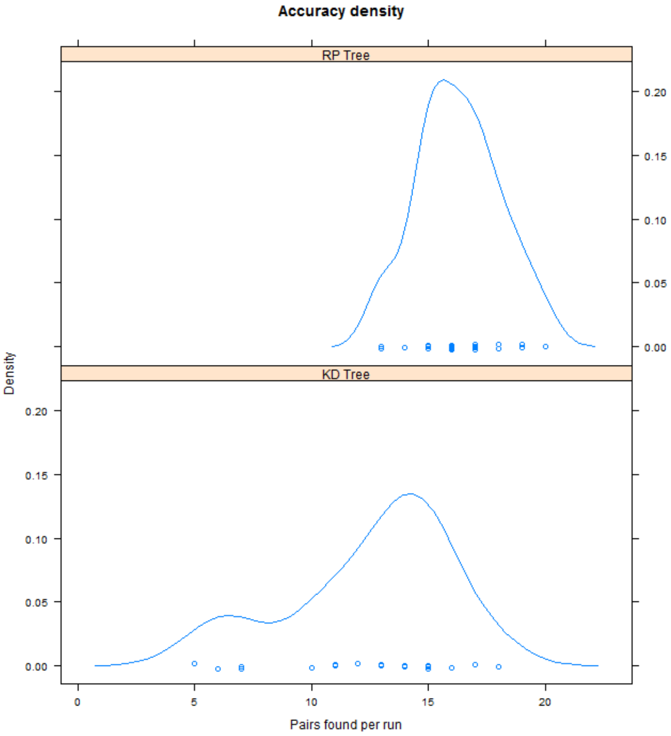

Figure 4.10: Accuracy density.

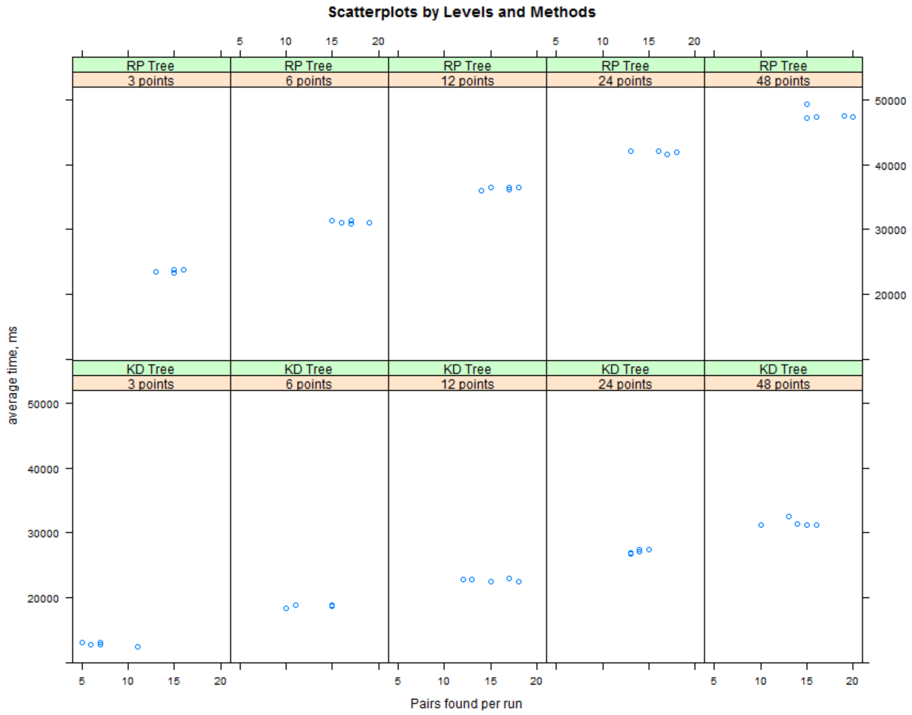

Figure 4.11: Scatterplot of dependency between time, accuracy and maximum number of points in the leaf node

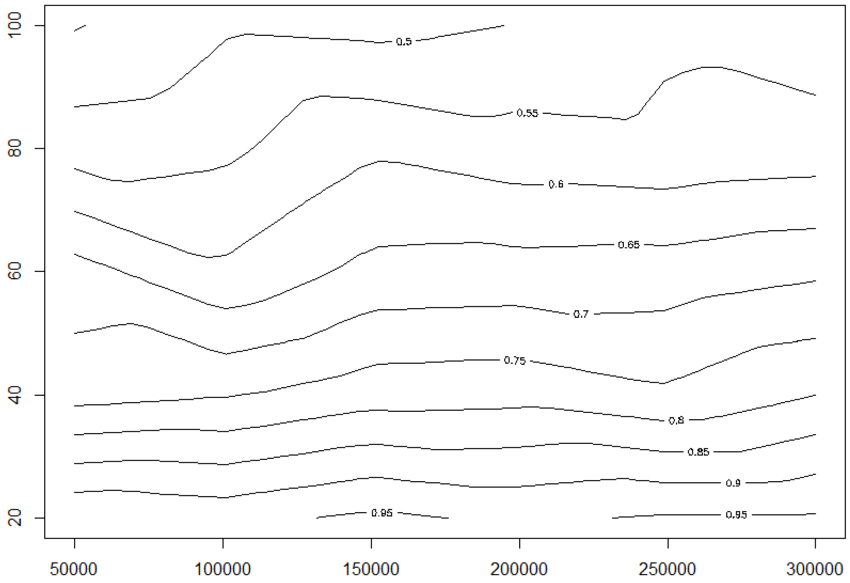

Figure 4.12: Stratified structure of correlation coefficient on random data

Chapter 5

Conclusion

The detection of the most correlated items in large high-dimensional datasets is very important problem for the variety of real-world applications. Nowadays, this task is becoming more and more relevant considering constantly growing volume of the information in the world. To our knowledge, it is currently solved by computing all pair-wise correlations in the dataset, which takes impractically large amount of time. In this thesis we proposed a faster solution for this problem.

We demonstrated that it is possible to improve the time needed to find most correlated pairs. First we standardize all vectors in the dataset and then find the pair with the smallest possible Euclidean distance using nearest neighbor indexing.

Next, we proposed a solution to the original problem that is based on nearest neighbor indexing. In particular, we implemented three state-of-the-art methods: coordinate-wise search (exact), KD tree and RP tree data structures (approximate). All these algorithms start with building a data structure by assigning indexes to the points in a given dataset that later allows to efficiently find nearest neighbors to the query point. In our work we focused mostly on last two approximate methods.

We run two different types of tests on simulated data in order to measure time and quality of the proposed solution. To evaluate its running time we compared performances of all three methods with the one for baseline approach. Both hierarchical data structures showed linear time-complexity for all tests. Although coordinate-wise search has a quadratic time-complexity, it still substantially outperforms the brute force method. In terms of the quality of obtained results tests show that it degrades with the size of the input set for both approximate methods, but nevertheless stays sufficiently high to be useful for the most of the real-world problems.

To demonstrate this, we tested our solution on a dataset containing records related to methylation values of different genes in different individuals. Results show that our approximate methods are capable of detecting pairs of genes with highly correlated expression that belong to distant regions, that was not possible using existing bioinformatical tools.

Kiired ligikaudsed päringud maksimaalse korrelatsiooni leidmiseks

Magistritöö (30 EAP)

Dmytro Fishman

Resümee

Kõige korreleeritumate paaride leidmine suurtes kõrgemõõtmilistes andmestikkes on väga oluline ülesanne, mis leiab kasutust paljudes reaalmaailma rakendustes. Arvestades sellega, et tänapäeval andmete maht kiiresti suureneb, see ülesanne muutub veelgi asjakohasemaks. Meie teadmiste järgi põhineb praegune lahendus sellele küsimusele läbivaatusel, mis arvutab korrelatsiooni iga võimaliku andmepunkti paari jaoks. See lähenemine on liiga aeglane selleks, et kasutada seda praktikas.

Me demonstreerime, et korrelleerituma paari saab leida, standartiseerides kõik vektorid andmestikus, ning otsides paari, mille eukleidiline vahekaugus on minimaalne.

Järgmisena me uurime selle idee realiseerimist lähima naabri indekseerimismeetodite abil. Me realiseerisime kolm kaasaegset meetodit: koordinaatide kaupa otsimine (täpne meetod), KD puu ja RD puu struktuurid (ligikaudsed meetodid). Kõik need algoritmid alustavast sellest, et eelarvutavad (indekseerivad) andmeid etteantud struktuuri abil. See lubab efektiivselt otsida iga punkti lähimat naabrit.

Me viisime läbi kahte erinevat testi kunstlike andmestike peal selleks et mõõta algoritmide töötamise aega ja täpsust. Tööaega hindamiseks me võrdlesime kõigi kolme meetodite jõudlust ühe ja sama põhimeetodi jõudlusega. Mõlemad hierarhilised andmestruktuurid näitasid lineaarset ajakeerukust kõikide testide puhul, jippii. Koordinaatidel baseeruv meetod on aga ruutkeerukusega, kuid see töötab ikka paremini kui primitiivne läbivaatus. Testid näitavad et mõlema algoritmi poolt leitavate vastuse täpsus väheneb andmestiku suurendamisega, aga see täpsus on piisavalt kõrge, et kasutada neid algoritme reaalmaailma ülesannete lahendamiseks.

Bibliography

[1] Abdullah Mueen, Suman Nath, and Jie Liu. Fast approximate correlation for massive time-series data. In *Proceedings of the 2010 international conference on Management of data*, pages 171–182. ACM, 2010.

[2] Meelis Kull, Jaak Vilo, et al. Fast approximate hierarchical clustering using similarity heuristics. *BioData mining*, 1(1):1–14, 2008.

[3] Victor S. Adamchik. *Computer Science - 121.* Computer Science Department, Carnegie Mellon University, Pittsburgh, PA., December 2009.

[4] Konstantin Tretyakov. *A Brief Introduction to Matrix Algebra.* Institute of Computer Science, University of Tartu.

[5] Frank Jones. *Honors Calculus.* Rice University, 2004.

[6] Donald E Knuth. Sorting and searching (the art of computer programming volume 3), 1973.

[7] Jon M Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 599–608. ACM, 1997.

[8] Piotr Indyk. Nearest neighbors in high-dimensional spaces. 2004.

[9] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 537–546. ACM, 2008.

[10] Jon Louis Bentley. Multidimensional binary search trees in database applications. *Software Engineering, IEEE Transactions on*, (4):333–340, 1979.

[11] Peter N Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321. Society for Industrial and Applied Mathematics, 1993.

[12] Sameer A Nene and Shree K Nayar. A simple algorithm for nearest neighbor search in high dimensions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(9):989–1003, 1997.

[13] Rina Panigrahy. Nearest neighbor search using kd-trees. *citeseerx. ist. psu. edu*, 2006.

[14] Hendra Gunadi. Comparing nearest neighbor algorithms in high-dimensional space. 2011.

[15] B. S. Kim and S. B. Park. A fast k nearest neighbor finding algorithm based on the ordered partition. *IEEE Trans Pattern Anal Mach Intell*, 8(6):761–766, Jun 1986.

[16] Andrew W Moore. An intoductory tutorial on kd-trees. *Extract from Andrew Moore's PhD Thesis: Effcient Memory based Learning for Robot Control*, 1991.

[17] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[18] Panayiotis Tsaparas. *Nearest Neighbor Search in Multidimensional Spaces: Depth Oral Report*. University of Toronto, Department of Computer Science, 1999.

[19] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory OF Computing*, 8:321–350, 2012.

[20] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 573–582. Society for Industrial and Applied Mathematics, 1994.

[21] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. pages 331–340, 2009.

[22] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the international conference on very large data bases*, pages 194–205. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS, 1998.

[23] *On effective conceptual indexing and similarity search in text data*, 2001.

[24] Quan Gu, Shivashankar Nagaraj, Nicholas Hudson, Brian Dalrymple, and Antonio Reverter. Genome-wide patterns of promoter sharing and co-expression in bovine skeletal muscle. *BMC genomics*, 12(1):23, 2011.

Appendices

Appendix A. Program code (on a compact disc)

Licence

I, Fishman Dmytro (date of birth: 06.02.1991),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   (a) reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   (b) make available to the public via the university's web environment, including via the DSpace digital archives, as of 20.05.2013 until expiry of the term of validity of the copyright,

   Fast approximate max-correlation queries

   supervised by Konstantin Tretyakov

2. I am aware of the fact that the author retains these rights.

3. This is to certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 20.05.2013