GraBaSS: Graph Based Subspace Search

Marco Neumann

September 20, 2013

Contents

1 Introduction

The 21th century is the age of information. Every year the amount of new data that is accessible to analysts get higher. More sensor networks get installed, higher resolutions in both, the data dimensions itself and the time, get possible, more communication over the global network is made and new methods for measuring health, environment, social and finance parameters are developed. If this data is used by the right, ethical way, it can have a huge impact on our society, pushing the global development and making new technologies possible. Not just since "Big Data" got the buzzword of the year and thousands of new companies try to sell their products for data analysis, it is clear that this amount of data can only be analyzed using fast computers and clever algorithms.

In the last decades many researchers developed methods to group data [Est+96], predict new data [Qui93] or find anomalies [Bre+00]. But more data does not only mean more data points, it also means more dimensions. And so, many methods getting slow or does not work anymore. This fact is known as curse of dimensionality [Bel57; Bey+99]. When the number of dimensions get higher, dimensions can be grouped together because they are similar or describe disjunct features. This process is also called feature selection and the groups can be called subspaces. If the data is projected into this subspaces, standard analysis methods can be used. So researchers developed methods for feature selection [CFZ99; KMB12]. They offer good results when used with high number of objects and a high number of dimensions and some of them are also proven in a theoretical way, but have one common problem: they are really slow. Having a cubic or higher complexity in the number of dimensions, it is not possible to use them for todays or future data sets. For many of them, it is also not possible to use them in a parallel way, which is highly important today and will get essential in the next years. Another problem are the parameters of the algorithms. If there are too many parameters which interfere with each other and are not intuitive, analysts just choose default, random or dummy parameters and are not able to get a good result. It is also very common to have parameters which ranges that heavily depends on the structure of the input data and not only the data size [CFZ99]. The perfect case would be a few parameters with intersected effects and fixed ranges.

So why is this a problem so relevant, you may ask. Just use a faster computer and more memory or wait a day, a week, a month or even longer. It is important because we are wasting the most important resource we have. It is not water, not energy, not oil, not gold or lithium. It is not knowledge or intelligence. Our most important resource is time and we all are running out of them. I believe that there is a way to get the relevant information faster, just at the moment when you need them. Even when ad-hoc data analysis is not possible today, I believe it is possible in the future. And it will change everything – the way we consume information and media, the art of describing our environment and our society, the way people life and communicate, the methods of research, production, planning and design, even the way we think and decide.

As a first, small step to this future I propose a new faster method for feature selection: Graph Based Subspace Search, or GraBaSS. It may not be as exact or mathematically proven as some of the competitors. But it works fast, parallel, the parameters are easy to choose and the results are intuitive. Depending on the chosen parameters, it can be used for automatic and manual data processing. It is also possible to choose how strong the requirement of similarity is, depending on the data set and the ways the subspaces should used later.

GraBaSS is build on the insight that in the most subspaces all dimensions are similar to each other, which forms a binary relationship. In contrast to other approaches, which often use a bottom up approach to find subspaces and require an enormous amount of time and space, GraBaSS uses this binary relationship to form a graph. This graph gets optimized and the cliques in the resulting graph are forming the subspaces. This work also discusses how to decide if two dimensions are similar and builds a framework around it. This can be used for further research and as possibility to modify GraBaSS for special purposes, e.g. to find subspaces with only linear relationships or where the dimensions are a special transformation of each other.

A special chapter of this thesis discusses the implementation of the algorithm. It explains decisions about programming languages and frameworks and gives useful tips that can be used to implement other methods in a high performance way. The implementation of GraBaSS is provided as Open Source, so that everyone can use it to process its own data and to learn from the implementation. Together with GraBaSS a data backend is provided that stores parsed data like a column storage but gives low level access for good performance. It enables to use the same parsed data set for other tasks like cluster search or outlier detection after doing the subspace analysis. Because dimensions are stored separately, no changes of the storage is required if other tasks are only done in a specific subspace.

2 Designing a high performance algorithm

To handle current and future big datasets a special approach is required. In this chapter I will design a high performance algorithm that is able to handle this task. The main goal is a method that does not beat the complexity of $\mathcal{O}(|D| \cdot |N| \cdot \log |N| + |D|^2 \cdot |N|)$. This complexity is a good upper bound. It combines the bounds of two different tasks. The first one is the preprocessing and structure detection of all objects in all dimensions. $\mathcal{O}(|D| \cdot |N|)$ would be enough to normalize the data, but $\mathcal{O}(|D| \cdot |N| \cdot \log |N|)$ allows some better methods like sorting. The second part is the detection of relations between all dimensions, which could be done in $\mathcal{O}(|D|^2)$. In combination with a non-fixed number a elements, a good method needs $\mathcal{O}(|D|^2 \cdot |N|)$.

To define a algorithm, it is necessary to describe what it should calculate. As stated in the introduction it should extract subspaces from the data set, so a description of what a subspace is would be helpful:

Definition 1 (Subspace)**.** A subspace is a collection of dimensions which are similar to each other. This collection should be maximal. It is possible that one dimension is contained in multiple subspaces or, in other words, that subspaces overlap.

According to the definitions the set of all subspaces does to not partition the set of dimensions. The definition also avoids a description if the relation of dimensions in one subspace are pairwise or can only be described by functions with more than two arguments. For GraBaSS I chose a pairwise definition because it is sufficient for the most real world data sets. As tests in chapter 5 show, I am right with this assumption. Chapter 4 describes when this can be a problem and how to work around it.

2.1 A question of similarity

To search and find subspaces and to use them for further analysis, it is necessary to describe when dimensions should be share the same subspace. In words, a good but very common description is that they should be similar. But this term has different meanings for different applications and in different fields of science. To use the results for analysis, it is important to know which structures or attributes dimensions of the same subspace share. It does not make any sense to use a metric based on covariance for the subspace search algorithm when your outlier detection uses an Manhattan metric to detect high distance points. So, the similarity metric should be chosen wisely. To do so, it should defined:

Definition 2 (Similarity)**.** A similarity $s : D \times D \to [0, 1]$ describes how similar two dimensions are. 0 means "not similar" and 1 means "very similar". It must satisfy the following attributes:

1. Computability: Their has to be an algorithm, two calculate the similarity for two given dimensions and a fixed number of points.
2. Symmetry: For all dimensions $a, b \in D$ it holds that $s(a, b) = s(b, a)$.
3. Identity: For all dimensions $a \in D$ it holds that $s(a, a) = 1$.

The definition also includes the computability. Because this thesis is about high performance feature selection, the complexity of the algorithm is highly important.[1]

A well known similarity is based on the Pearson correlation coefficient. It can be cut to $[0, 1]$, squared or the absolute value can be calculated to get a similarity. The correlation coefficient can be calculated in $\mathcal{O}(|N|)$. For many applications, this type of similarity is sufficient. Because it describes a linear relation, there is an underlying structure which has to exist. Because this structure only has two degrees of freedom, it is very biased.

To handle complex data, a less biased method has to be constructed. One way to do it is to utilize metrics:

Theorem 1. *Given a limited metric $m : D \times D \to [0, \infty)$, $l := \sup_{a,b \in D \times D} m(a, b) < \infty$ a similarity can be constructed in the following way:*

$$s(a, b) = 1 - \frac{m(a, b)}{l} \tag{2.1}$$

When the dimensions of the input set are limited, all p-norms can be used to construct a similarity by calculating the distance of all points in the dataset:

Theorem 2. *Given a p-norm $\| \cdot \|_p$, a similarity can be constructed by using the normalized sum of all distances:*

$$s(a, b) = 1 - \frac{\|\pi_a(N) - \pi_b(N)\|_p}{|N|} \tag{2.2}$$

p-norms can be calculated very efficient but are more biased than the Pearson correlation coefficient, because they are only high if two dimensions are nearly equal in many dimensions.[2] So they are also biased and not usable for many applications.

Another approach to create new similarities is to combine a metric together with a machine learning algorithm. This should be less biased because many machine learning algorithms can produce very flexible results:

Theorem 3. *Given a two dimensions $a, b \in D$, a set of prediction algorithms $P = \{p : \mathbb{R} \to \mathbb{R}\}$ and a machine learning algorithm $l : (\mathbb{R} \to \mathbb{R}) \to P$, a similarity can be formed by choosing two subsets $T, V \subseteq N$, $T \cup V = N$, train the prediction algorithm and validating the results by*

[1] Please notice that big constant factors are also important if they are part of the algorithm. Constant factors should be small if possible.

[2] The number of this dimensions depends on the chosen p

measuring the difference:

$$p_1 = l(\pi_a(T), \pi_b(T)) \tag{2.3}$$
$$p_2 = l(\pi_b(T), \pi_a(T)) \tag{2.4}$$
$$s_1 = 1 - \frac{\|\pi_b(V) - p_1(\pi_a(N))\|}{|V|} \tag{2.5}$$
$$s_2 = 1 - \frac{\|\pi_a(V) - p_2(\pi_b(N))\|}{|V|} \tag{2.6}$$
$$s(a, b) = s_1 \cdot s_2 \tag{2.7}$$

Please choose the training and validation set wisely, especially when you expect outliers in the dataset.

Since machine learning algorithms allow comparison of dimensions with flexible structure, an important question is how flexible a structure can be to allow a comparison. Mathematics had found a answer to this question long time ago. They defined the term and rules of independence:

Definition 3 (Independence)**.** Two dimensions $X, Y \in D$ are independent, iff:

$$\mathbb{P}(X = z, Y = z) = \mathbb{P}(X = z)\,\mathbb{P}(Y = z), \quad \forall z \in \mathbb{R} \tag{2.8}$$

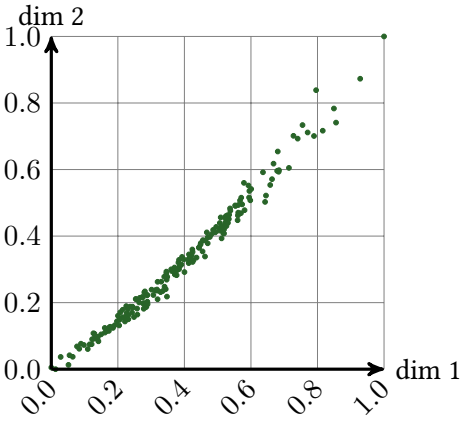
This definition is used by many mathematics worldwide but has one problem. If we assume a limited dataset with real number values and just a little bit random noise, no value in the dimensions will appear twice. According to the definition of independence, all dimensions will be independent in this situation. So it is necessary to extend the Dirac masses to another probability. This can be done by using window functions, which leads to an continuous probability. Another method is binning by using fixed size bins. Because it is possible that the data resolution is different in different parts of the dimension, guessing window sizes or bin widths is not possible every time and can lead to wrong results. To circumvent this problem, I will introduce a method that is known widely but not used by many researcher and analysts: binning with fixed number of objects per bin. To express the effect of this method, I will call it rewrapping.

Rewrapping handles dense data regions very well and ignores non used regions. It is also able to reverse every strict monotone function. This kind of function may occur as part of the measurement of real world data. Figure 2.1 illustrates the process of rewrapping. The number of bins is not a user provided parameter. It is set to $\sqrt{|N|}$ to honor the fact the more data points also describe a more detailed information about the distribution.

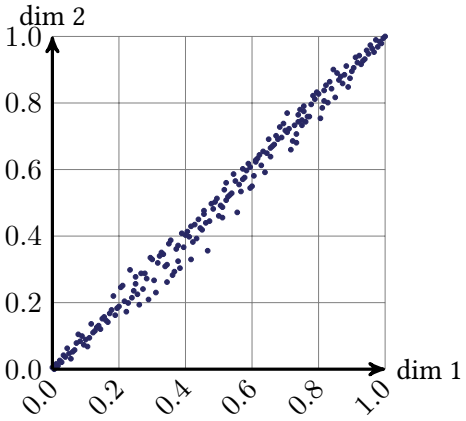
Because real value data can contain discrete values, a small modification is required. Bins are created from the lower to the upper bound of the dataset. If the border between a created and a new bin will split a set of equal values, the entire set will assigned to the lower bin. This can lead to a different bin number and fill size. See algorithm 1 for a complete definition.

To extract a measure of independence from the discrete data set, the mutual information is calculated:

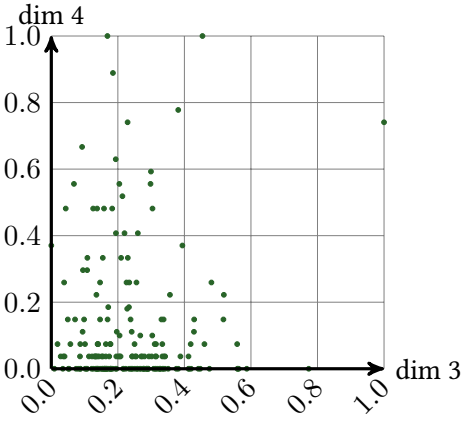
Figure 2.1: Rewrapping



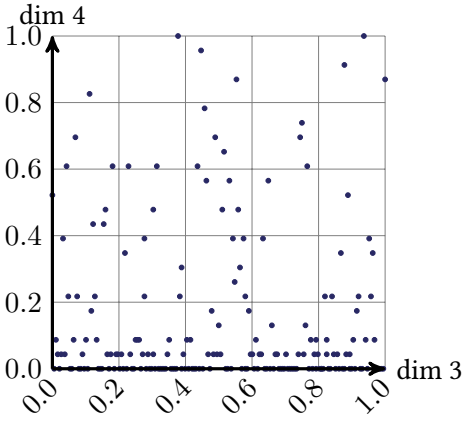
(a) Sample 1: Input data



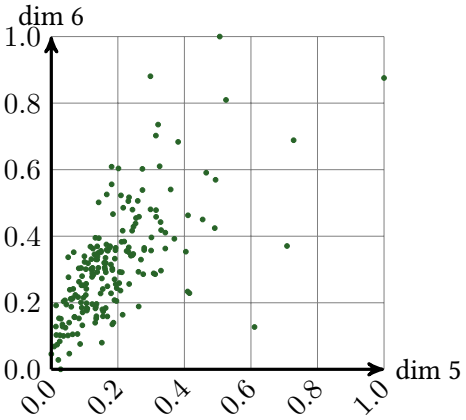
(b) Sample 1: Result of rewrapping



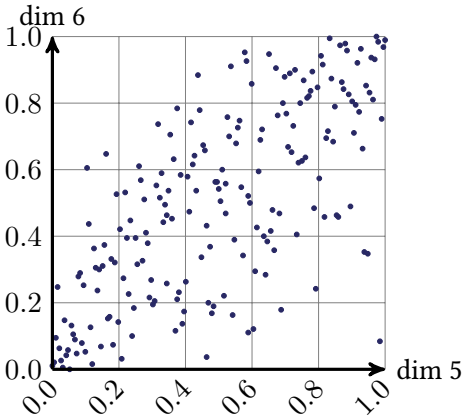
(c) Sample 2: Input data



(d) Sample 2: Result of rewrapping



(e) Sample 3: Input data



(f) Sample 3: Result of rewrapping

Algorithm 1: buildBins

```
Data: Dimension d
Data: Binsize s
Result: Bins B
1  begin
2      B ← ();
3      d_s ← sort(d);
4      x ← undef;
5      b ← ();
6      for x ∈ d_s do
7          if |B| ≥ s ∧ x ≠ l then
8              B ← B+: b;
9              b ← ();
10         end
11         b ← B+: x;
12         l ← x;
13     end
14     if |b| ≠ 0 then
15         B ← B+: b;
16     end
17 end
```

Definition 4 (Mutual Information)**.** For two discrete dimensions X and Y, which are only described by their bins and the probability of each bin, the mutual information $\mathrm{I}(X,Y)$ is calculated by:

$$\mathrm{I}(X,Y) = \mathrm{H}(X) + \mathrm{H}(y) - \mathrm{H}(X,Y) \tag{2.9}$$

To do this, the entropy has to be calculated as followed:

$$\mathrm{H}(X) = -\sum_{x \in X} p(x) \log p(x) \tag{2.10}$$

$$\mathrm{H}(X,Y) = -\sum_{x \in X, y \in Y} p(x,y) \log p(x,y) \tag{2.11}$$

To convert the value of mutual information to a similarity, it has to be normalized by the following method:

Theorem 4 (Normalized Mutual Information)**.** *The normalized mutual information of two dimension X and Y is calculated by the following equation*

$$\mathrm{I}(x,y)_{norm} = \frac{\mathrm{I}(X,Y)}{\min(\mathrm{H}(X), \mathrm{H}(Y))} \tag{2.12}$$

Proof. See [Yao03]. □

Mutual information and independence do not respect permutation of the bins. This can lead to very unintuitive results and makes the calculation heavily depend on the number of bins and their borders. Figure 2.2 shows an example of this problem. The dimensions of the two shown data sets have the same value of mutual information, but when it is clear that the dimensions shown in figure 2.3a are more similar as the dimensions shown in figure 2.3b. To fix this, the information from the absolute Pearson correlation coefficient could be mixed in. This describes if two dimensions are linear correlated and avoids random permutations. The two similarities should be combined in a way that acts like a logic AND. Because both values are real, but have the same range, a limited, non-complete logic can be used:

Definition 5 (Similarity Logic)**.** Similarities can be used as logical values in the following way:

$$true \equiv 1 \tag{2.13}$$

$$false \equiv 0 \tag{2.14}$$

$$\neg a \equiv 1 - a \tag{2.15}$$

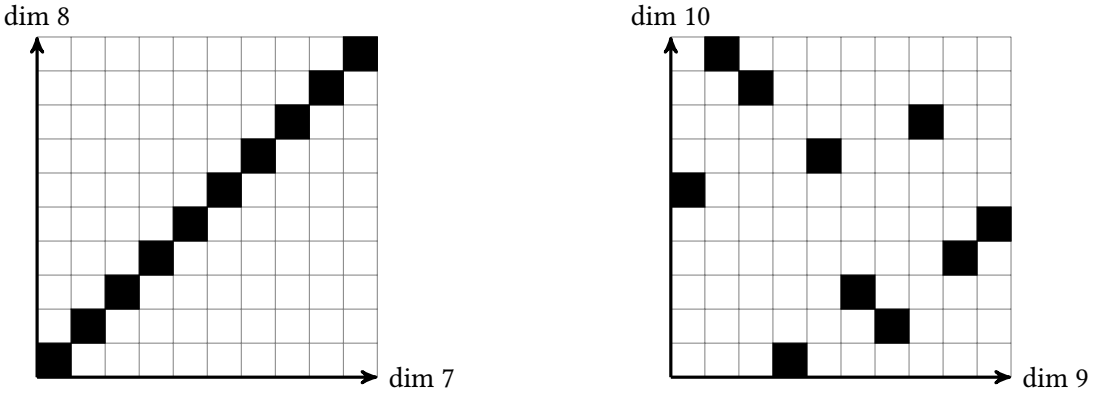
$$a \wedge b \equiv a \cdot b \tag{2.16}$$

$$a \vee b \equiv 1 - (1-a) \cdot (1-b) \tag{2.17}$$

Please note, that this system is not correct in terms of logic, because $a \wedge a = a$, but $a \wedge a \equiv a^2 \neq a$. This fact does not make this system wrong, because it makes sense that the claim of $a \wedge a$ requires a to be stronger than simple claim a alone.

It is now possible to combine different similarities to one final value. As described above, the mutual information is combined with the absolute Pearson correlation coefficient using

Figure 2.2: Permutation of bins



(a) Permutation with high similarity

(b) Permutation with low similarity

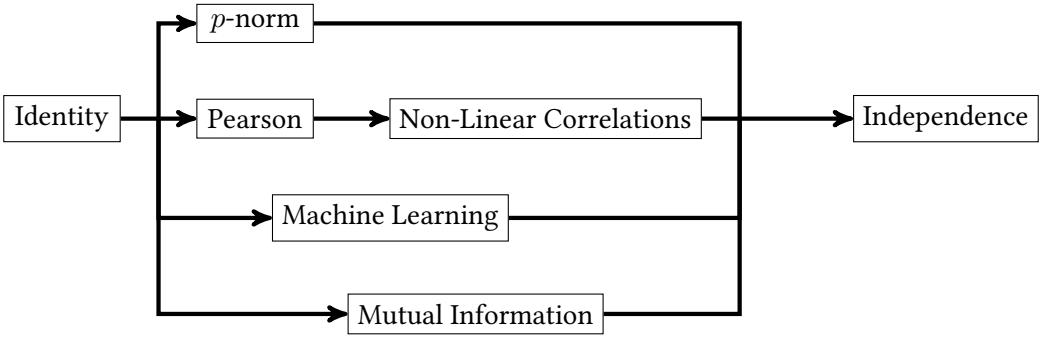


Figure 2.3: Different similarities

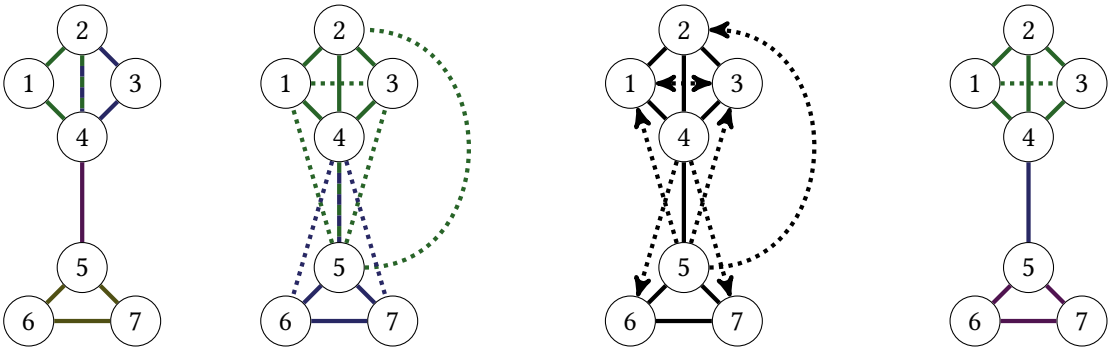
the AND function or, in other words, by calculating the product of them. Figure 2.3 gives an overview about the different similarities and how they work together. There are also stronger and weaker definitions in terms of how the rely on a predefined structure. For example the absolute Pearson correlation coefficient only works well with linear correlation but avoids random permutations, whereas the normalized mutual information is more flexible but ignores permutations. Using this building blocks, it is possible to swap the final similarity when doing further research or combine it with new definitions.

2.2 From similarity to subspace

Based on the combined similarity it is possible to calculate subspaces by calculating the binary relation between all dimensions and find groups that have strong connections. It is important to find this groups in this $|D|^2$ relations highly efficient.

Next, I want to discuss how subspaces are formed. Two dimensions should be contained in the subspaces, when they are similar. This can be expressed by a threshold $p_e \in [0, 1]$, which

Figure 2.4: Different subspaces from different graph preprocessing



(a) Too many subspaces

(b) Simple graph distance

(c) Filtered distance with $\alpha = \frac{2}{3}$

(d) Bidirectional filtered distance with $\alpha = \frac{2}{3}$

the combined similarity should have. By stripping down the similarity matrix to a binary matrix, the dimensions form a graph. This speeds up the calculation, because it allows fast set operations and memory effective management. Because the most data sets have very different dimensions, the number of edges should be low.

The important attribute of subspaces is that all contained dimensions are similar to each other. For graphs this attribute is already described as clique. Because the graph is dense it has a low degeneracy $d \in \mathbb{N}_0$. So, the cliques can be calculated in $\mathcal{O}(d \cdot (|D| - d) \cdot 3^{\frac{d}{3}})$ by using [ELS10]. Because the cliques forms the subspaces, this is the complete algorithm for converting the combined similarity to subspaces.

2.3 Optimizing the graph structure

A core problem of strict clique searchers is the heavy fragmentation if only one edge missing. Figure 2.5a shows an example where the nodes semi-clique $\{1, 2, 3, 4\}$ is split into two cliques because of the absence of the edge $(1, 3)$. For some applications it could be helpful to add this edge to the graph and join the two cliques into one. The first method to do it is the utilization a distance graph:

Definition 6 (Strict Distance Graph)**.** Given a graph $G = (V, E)$ and a distance $n \in \mathbb{N}_0$, the strict distance graph $G^n = (V, E^n)$ is give by:

$$E^0 = \left\{(v, v) \in V^2\right\} \tag{2.18}$$

$$E^i = \left\{(v_0, v_2) \in V^2 \mid \exists v_1 \in V : (v_0, v_1) \in E^{i-1}, (v_1, v_2) \in E\right\}, \quad \forall i > 0 \tag{2.19}$$

Because it does not make any sense to forget about the given edges when calculating higher distances, the definition of the joined distance graph is more useful:

Definition 7 (Joined Distance Graph)**.** Given a graph $G = (V, E)$ and a distance $n \in \mathbb{N}_+$, the joined distance graph $G^{\overline{n}} = (V, E^{\overline{n}})$ is defined as followed:

$$E^{\overline{n}} = \bigcup_{i=1}^{n} E^i \tag{2.20}$$

Using the joined distance graph fixes the problem but does not produce the desired result. As seen in Figure 2.5b, it attaches to many new vertexes to the cliques and make the results unusable. The reason is that it does not count the strength of the connection in the existing graph and so it creates some edges to weakly connected vertexes. A measure for the connection rate between two vertexes can be calculated by the following equation:

Definition 8 (Connection Rate)**.** Given a graph $G = (V, E)$ and two vertexes $v_1, v_2 \in V$, the connection rate $r_G(v_1, v_2) \in [0, 1]$ can be calculated by the relation of connected to all neighbors of v_1:

$$r_G(v_1, v_2) = \frac{|\{v \in V \mid (v_1, v) \in E\} \cap \{v \in V \mid (v, v_2) \in E\}|}{|\{v \in V \mid (v_1, v) \in E\}|} \tag{2.21}$$

Please notice that the connection rate is not symmetric. To honor the connection rate of the vertexes, I use a filtered strict distance graph:

Definition 9 (Filtered Strict Distance Graph)**.** Given a graph $G = (V, E)$, a distance $n \in \mathbb{N}_+$ and a filter rate $\alpha \in [0, 1]$, the filtered strict distance graph $G^n_\alpha = (V, E^n_\alpha)$ is definined as followed:

$$E^n_\alpha = \{(v_1, v_2) \in E^n \mid r_{G^{n-1}}(v_1, v_2) \geq \alpha\} \tag{2.22}$$

The joined filtered graph, that considers all distances up to a given value, is defined analog:

Definition 10 (Filtered Joined Distance Graph)**.** Given a graph $G = (V, E)$a distance $n \in \mathbb{N}_+$ and a filter rate $\alpha \in [0, 1]$, the filtered joined distance graph $G^{\overline{n}}_\alpha = (V, E^{\overline{n}}_\alpha)$ is defined as followed:

$$E^{\overline{n}}_\alpha = \bigcup_{i=1}^{n} E^i_\alpha \tag{2.23}$$

The filtered joined distance graph is not bidirectional. This leads the problem that there are unidirectional edges that are not usable for our result. As Figure 2.5c shows, the weakly connected nodes are connected to the semi-clique whereas the semi-clique is not connected to the satellites. The inter semi-clique edges are bidirectional. Based on this fact, a bidirectional graph can be constructed:

Definition 11 (Bidirectional Joined Filtered Distance Graph)**.** Given a graph $G = (V, E)$, a distance $n \in \mathbb{N}_+$ and a filter rate $\alpha \in [0, 1]$, the bidirectional joined filtered graph $\tilde{G}^{\overline{n}}_\alpha = (V, \tilde{E}^{\overline{n}}_\alpha)$ is defined by the subset of the bidirectional edges of the joined filtered graph:

$$\tilde{E}^{\overline{n}}_\alpha = \left\{(v_1, v_2) \in V^2 \mid (v_1, v_2), (v_2, v_1) \in E^{\overline{n}}_\alpha\right\} \tag{2.24}$$

As shown on Figure 2.5d this method successfully adds inter semi-clique edges to join them to cliques without creating senseless connections or resulting in information lossage. It is important that the graph refinement does not slow down the entire process on big data. In other words, it must not have a higher complexity class. Because the distance parameter n is only used for graph fixing and should not depend on the input size, the following theorem can be shown:

Algorithm 2: extendNeighbors

```
Data: Vertex v
Data: Graph G
Data: Threshold t_n
Result: New neighbors N
1  begin
       /* Search all candidates                                          */
2      C ← {};
3      for w ∈ getNeighbors(v) do
4          C ← C ∪ getNeighbors(w);
5      end
       /* Check all candidates                                           */
6      N ← {};
7      for c ∈ C do
8          if calcConnectionRate(v, c) ≥ t_n ∧ calcConnectionRate(c, v) ≥ t_n then
9              N ← N ∪ {c};
10         end
11     end
12 end
```

Theorem 5. *The graph refinement can be done in $\mathcal{O}(n \cdot |V| \cdot m^2)$ where $n \in \mathbb{N}_+$ is the fixed distance parameter and $m \in \mathbb{N}_0$ is the number of maximum neighbors over all vertexes in the resulting graph.*

Proof. In this proof, I assume that the neighbors for of each vertex are stored in a set, which allows lookup and insert operations in $\mathcal{O}(1)$.

The graph will generated from the graph with the distance $n - 1$ where a distance of 1 is the input graph. Every of this rounds contains three steps and is done for every vertex in the graph. The first step is the listing of all neighbor candidates without taking connection rates into account and joining them into one set. For each vertex, this takes $\mathcal{O}(m^2)$ operations and results in $\mathcal{O}(m)$ candidates. Step two is the calculation of the connectivity, which can be done in $\mathcal{O}(m)$ for every candidate. For all candidates, it takes $\mathcal{O}(m^2)$ operations. The last step is the filtering according to the connectivity and the build-up of the new neighborhood-set, which takes $\mathcal{O}(m)$ operations. All steps together take $\mathcal{O}(m^2)$ operations for each vertex, which is $\mathcal{O}(|V| \cdot m^2)$ in total. Because this process is repeated n times, the total complexity is $\mathcal{O}(n \cdot |V| \cdot m^2)$. □

Algorithm 3: refineGraph

```
Data: Graph G
Data: Distance d
Data: Threshold t_n
Result: Refined Graph G_r
1  begin
2      G_r ← G;
3      for i=2 to d do
4          G_n ← ();
5          for v ∈ G_r do
6              G_n ← G_n+: extendNeighbors(v, G_r, t_n);
7          end
8          G_r ← G_n;
9      end
10 end
```

Assuming, that the number of maximum neighbors is small and especially lower than $|D|$, the graph fixing process does not have a performance impact. Tests prove this assumption, see section 5.4 for details.

2.4 Algorithm Summary

After discussing all parts of the algorithm, I will now wrap up the entire workflow and explain some optimizations. The main method is shown in algorithm 4. It uses the methods explained earlier. Please note that no method uses a global shared state. This functional attribute helps when it comes to parallelization of the algorithm.

First of all, some values that only depend on one dimension are pre-calculated, this are the mean, the variance and the standard deviation of the dimensions. This step can parallelized at two points. Because the result for one dimension only depends on this dimension itself, the values for multiple dimensions can be calculated independently. The mean and the variance are sums. This sum up can be parallelized can by multiple methods, because all sub sums are independent.

After this, the discrete bins are created. Because bins are only build on top of the information of one dimension, multiple dimensions can handled in parallel too. The creation of the bins requires that the data is sorted by the actual dimension. This sorting can also be done by a parallel algorithm, e.g. merge sort.

Now, the initial graph is build. This is done by calculating the absolute Pearson covariance coefficient and the normalized mutual information. Then, the product is calculated and an graph edge between this two dimensions is created, if this value is at least t_e. I suggest to parallelize the inner loop and leaf the outer one as it is. Depending on the size of the data

Algorithm 4: calcSubspaces

Data: Dataset D
Data: Threshold t_e
Data: Graph distance d
Data: Threshold of the graph connection rate t_n
Result: Subspaces S

begin
  /* Pre calculate aggregates */
  $D_m \leftarrow$ `calcMeanValues(`D`)`;
  $D_v \leftarrow$ `calcVarValues(`D, D_m`)`;
  $D_s \leftarrow$ `calcStddevValues(`D_v`)`;
  /* Build grid */
  $D_g \leftarrow ()$;
  for $d \in D$ **do**
    $D_g \leftarrow D_g +:$ `buildBins(`$d, \sqrt{|D|}$`)`;
  end
  /* Build graph */
  for $(d_1, m_1, s_1, g_1) \in (D, D_m, D_s, D_g)$ **do**
    $n \leftarrow \{\}$;
    for $(d_2, m_2, s_2, g_2) \in (D, D_m, D_s, D_g) \setminus \{(d_1, m_1, s_1, g_1)\}$ **do**
      $x_1 \leftarrow \frac{\texttt{coVar}(d_1, d_2, m_1, m_2)}{s_1 \cdot s_2}$;
      $x_2 \leftarrow$ `dimSimilarity(`g_1, g_2`)`;
      if $x_1 \cdot x_2 \geq t_e$ **then**
        $n \leftarrow n \cup \{d_2\}$;
      end
    end
    $G \leftarrow G +: n$;
  end
  /* Refine graph */
  $G_r \leftarrow$ `refineGraph(`G, d, t_n`)`;
  /* Search cliques (which are our subspaces) */
  $S \leftarrow$ `searchCliques(`G_r`)`;
end

set, this minimizes the number of page-in and page-out operations.[3] Because the Pearson covariance coefficient and the value of mutual information are symmetric, it is only required to calculated the one half of the adjacency matrix.

Before calculating the cliques, the graph is refined by adding new edges. The performance trick by using the symmetry of the adjacency matrix works here too and the loops can be parallelized in the same way as done in the graph generation process.

Finally, the cliques are searched by [ELS10]. Because the algorithm has multiple loops with independent recursions in them, it can be parallelized at this points.

2.5 Choosing parameters

Choosing the right parameters is important to get usable results. This choice also depends on the problem you want to solve or the information you want to extract from the data. The usage of the data by automatic analysis methods targets high information volume and low error rates. Manual, human driven analysis and exploration requires short, readable and non-redundant reports. In this case, a higher error rate is accepted to simplify this task. The GraBaSS has 3 parameters, t_e, t_n and d which should be used as followed.

The first parameter is $t_e \in [0, 1]$, which is the threshold that describes the lower bound of the similarity of an edge of the graph. If the similarity of two dimensions is at least t_e, they are assumed to be similar. This results in an edge in the graph structure and is used by the clique searcher. I suggest to choose 0.4 as start parameter. If the algorithm finds too small subspaces, decrease the parameter. If the results contains only a few but big subspaces, the parameter is too low.

The second parameter $t_n \in [0, 1]$ and the third parameter $d \in [0, \infty]$ should be used together. d describes the maximum graph distance for the graph fixing process and t_n the threshold of the amount of graph common neighbors in this distance that is required to create a new edge. I suggest to choose $t_n = 0.75$ as a start parameter and $d = 0$, which deactivates the graph fixing. Please note that $d = 0$ and $d = 1$ are leading to the same results, because a distance of 1 is already given by the graph. If the results of GraBaSS contains to many similar subspaces with only a few different members, d should be set to 2. The choice of t_n depends on the number of different subspace members. I suggest a value above 0.6, because smaller values lead to a lose of to many information. If this still not help to improve the results, try to increase d by 1. If you get only a few but very big subspaces, d is too high or t_n is too low. Please note that applying the graph fixing always leads to a lose of information. In some cases, especially when manual data exploration is intended, it could be helpful to use it to get some more readable results.

[3] Do not hope, that an entire dimension can be hold by the cache, so cache locality is not an argument.

3 Implementation

For the researcher who want only get a described complexity it is sufficient to use a favorite programming language or DBMS. But if you want to handle big data before getting to old, it is important to choose the right tools and think about some implementation details. In this chapter I want to describe and explain my choices and would like to give you some impressions about the implementation process. Some but not all strategies can also be used for other algorithms and may be used for some other work too. I will finish the chapter with some ideas about future improvements.

3.1 Choosing a programming language

To express the designed algorithm and let the computer do the job it is necessary to write some source code. There are dozens different programming languages out there and every language has its own pros and cons.

A simple and very comfortable choice is a language with a good build-in runtime library.[1] Python or MATLAB is a good choice. Please notice that SQL or other database bound languages are not very portable and suffer when it comes to debugging and flexibility. If the language provided a embedded mathematic system like MATLAB, it is easy to map the theoretical terms to the language. Matrix support makes it easy to handle big amounts of data. The key feature of the language should be syntax that does not produce much boiler plate. This is one point where script languages are very good, but languages like Java are not. C++ provides a short syntax for many things, but suffers like many other compiled languages when you want a comfortable environment. Typesafety is another point that can be useful because it allows you to tackle many bugs before you waste your time for runtime tests.

The next point is a good library support for things that the chosen language does not provide. Matrix operations require a special library, especially if you want to invert or decompose them. High precision floating point operations and big integer calculation needs a different library. Parsing, code management or special IO operations is another topic. Also notice that I don't mean the standard library. A well tested and documented library is required. For example NumPy if you use Python.

In addition to the things you as the author and primary user of the implementation finds useful are the problems your team or the consumers need. An exotic language[2] may be very cool but if nobody can read your code, your research result doesn't help someone. This does not mean that you have to stuck in old good Java or COBOL, but it means that you have to think

[1] What "good" means in this context depends on the algorithm that you want to implement and the operations you need.

[2] Don't know what I mean? Go and find something about APL

about the innovation you want to use. Also think about portability, future development and safety of the language. Safety is important if you don't want to risk the data of your institution or the consumers. You may think that the implementation is only for internal tests but if your results are good it can be the case that your code will be published or used by other persons. Even if low level languages can be insecure when doing unchecked memory operations, also interpreted languages have many weaknesses.

You may ask where this all have to do with high performance. The simple answer is: nothing. They make your life easier, but do not provide a fast implementation. So the most important point for my work is performance. And this kills the most high level languages, because they do not provide a fine grained control about data allocation, movement and copying. Even the object overhead of languages like Java and naive C++ is too big. I had to strip down the calculation core to a simple low level core. But I do not want to loose all high level features. In my eyes, there is currently only one languages that provides this schizophrenia: handcrafted and well written C++. I will give you some rules about hints about the language specific methods in the next section.

3.2 Special C++ tips

C++ can be used in many different ways, depending on your background, the interfaces you want to use or have to provide and the attributes your program should have. As most other programming languages out there, C++ is available in different standards and the compiler and STL support differs. One of the best collections of tips for good programming style is [Mey05]. Furthermore I used the new C++11 standard which brings makes many code sections easier to read and more C++ stylish rather than C based. It also avoids many boiler blade an manual memory management. Templates may be the most hated but most also most loved and most essential features of C++. I used them when I find it useful. Some calculations are written in C style because in some cases, the object overhead was to high. Because of the usage of new C++11 features, the resulting code only compiles with newer versions of GCC and Clang. Microsoft and Intel compilers are not supported at the time of the writing.

3.3 Avoiding reinventing the wheel

The STL already provides useful tools that makes programmers life much easier. But it does not provide high level thread operations like fork and join, system operations like memory mapped IO and fast a fast parser library. C++ has a special collection of well designed and reviewed libraries that are note yet a part of the STL: The Boost C++ Libraries. They provide two of the three libraries I was looking for – the Boost Interprocess library, which implements memory mapped IO and the Spirit V2 library which is a special set of function and templates for parser writing. For high level multi core operations I use another well known library: the TBB, which is designed and published by Intel. All libraries are open source and free which enables researcher to use them for their work.

3.4 The low level data backend

In the field of data processing and data mining, many algorithms where build on top of existing server driven databases. This has many benefits. In many cases, the data that should be analyzed already exists in this databases in a normalized form and is supplied by fresh data. Because of the nature of feature selection, column stores are than row oriented stores. They provide a cache local access to data of different dimensions and reduces the processing overhead. The main drawback of the most server driven data stores is the performance when using complex algorithms.[3] Another solution is using an embedded database like SQLite, but they also have too much overhead when accessing raw data. Please consider that the database also provides too many operations that are not needed for this kind of data processing. Only append and read operations are required, no indices, no aggregations, no write, modify and delete.

This brings me to the question why I need a backend for data storage. In the current situation, most researchers don't have systems with enough main memory to hold the complete data. So it is important that they have a backend that can easily page out data to the disc or SSD. This should also be combined with low overhead. So sequential loading on demand is not an option. Another problem is that the application programmer cannot easily determine, when the system runs out of memory, because the OS swaps out data on its own discretion. An old but very wise advice of OS designers is, that the kernel always has better information about the entire system than every user program. So why not let the kernel do the job? So I decided to just use plain arrays for column representation and memory map them from a file. So the kernel can load them or page them out if necessary. For better management and better append operations, the columns are defined into fixed size segments. The segments and the column metadata like size and name are managed by the Boost Interprocess Library, which uses a trees to manage a string to pointer mapping.

As shown in chapter 2 the algorithm needs a graph structure. An append only graph can mapped to a column store by utilizing two columns and encode each vertex as an unique, continuous, unsigned integer. One column is called the neighbor column and stores the appended list of all neighbors of all vertexes. The other column is called the index column stores the split points of the neighbor column. To get back the neighbors of an vertex v, just output the neighbor column from the index that is written in the in index column at position v to the index that is written at position $v + 1$.

Maps for pre-calcuated metadata can also be mapped to column stores by storing key value tuples and rebuilding the hash index when loading the map. This does not have performance impact because the algorithm only need a small fixed number of metadata per dimension.

[3] This may change in the next few years when in-memory databases become more prominent and affordable.

4 How to beat the system

GraBaSS only uses the relationship between two dimensions to build up a graph and find cliques which forms the subspaces. It is possible, that this pairwise approach does not find higher order dependencies. In this case, there are subspaces that are not found by GraBaSS or subspaces that do not reach their maximum size. The pairwise approach was not only chosen for performance but also because it is sufficient for the most data set. In this chapter, I will show how a higher order data set could look like, why this kind is not very common and how to handle them.

4.1 Constructing a special dataset

In mathematical terms, higher order dependencies exists because of the following fact:

Theorem 6. *Given three dimensions X, Y and Z, which satisfy the following requirement:*

$$\begin{aligned} \mathbb{P}(X = x, Y = y) &= \mathbb{P}(X = x)\,\mathbb{P}(Y = y), & \forall (x, y) \in X \times Y \\ \mathbb{P}(Y = y, Z = z) &= \mathbb{P}(Y = y)\,\mathbb{P}(Z = z), & \forall (y, z) \in Y \times Z \\ \mathbb{P}(Z = z, X = x) &= \mathbb{P}(Z = z)\,\mathbb{P}(X = x), & \forall (z, x) \in Z \times X \end{aligned} \tag{4.1}$$

there can cases where $\exists (x, y, z) \in X \times Y \times Z$:

$$\mathbb{P}(X = x, Y = y, Z = z) \neq \mathbb{P}(X = x)\,\mathbb{P}(Y = y)\,\mathbb{P}(Z = z) \tag{4.2}$$

So condition 4.1 is not sufficient for independence of the given dimensions.

Proof. See example described below. □

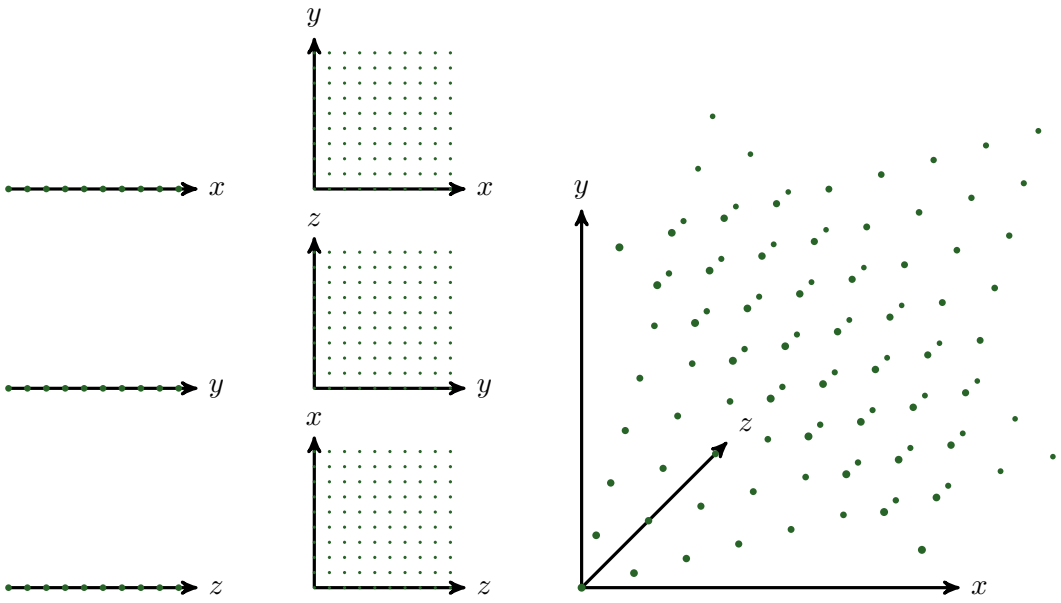
To illustrate the situation explained in theorem 6, consider the following a dataset, that is made of many random points $(x, y, z) \in [0, 1)^3$, that satisfy the following constraint:

$$(x + y + z) \bmod 1 = 0 \tag{4.3}$$

The points are uniform distributed if you project them on one or two dimensions. But if you calculate the three dimensional distribution, the points do not show a uniform distribution. (see Figure 4.1). Because the generated points have two degrees of freedom, the two dimensional distribution test will fail. It is also possible to generate datasets with even more degrees of freedom using the same technique.[1]

[1] You may not be able to imagine datasets with more than three dimensions. For 4 dimensions, you find some help: `http://crepererum.github.io/brain4D/#constructed`

Figure 4.1: Situation described in Equation 4.3



(a) 1D views (b) 2D views (c) 3D view

The problem occurs every time, when the dataset contains a subspace with $n \in \mathbb{N}_+$ Dimensions but $m \in \{1, \ldots, n-1\}$ degrees of freedom. Furthermore the dependency of each dimension on the degrees of freedom has to be equal, so no relation is visible in two dimensional projections. Especially the last condition is uncommon when looking at real world data sets.

4.2 Handling this issue

For data sets which has dimensions described in theorem 6, I will propose a simple method to handle them. To get a good performance on high dimensional datasets, it is not possible to check all higher order dependencies. Because the described problem occurs when the degrees of freedom are not aligned with the dimensions, a transformation of the dataset to the degrees of freedom would help. PCA is a well known method which finds such transformations. As shown in [SP07], a PCA can be computed in $O(|D|^2 h + |D|^2 |N|)$ where h is the number of degrees of freedom. If the dependencies of the dimension are formed by a near-linear process, PCA will find them. The result of the transformation can be used as input for GraBaSS. The isolated subspaces and the transformation found by PCA together explain the relation between dimensions. If there is any prior knowledge about the data set, it can be used to find such problematic dimensions. Note, that PCA can also be applied to subspaces before and after using GraBaSS, where the pre-analysis subspaces can be extracted by using expert knowledge.

5 High-Dimensional Datasets and Tests

High dimensional datasets are not a very common research topics. In this chapter I will describe some data sources and possible applications of the described algorithm. I also show some interpretations of subspaces in different datasets and some failed approaches of finding them.

This chapter will also compare GraBaSS against ENCLUS, which is described in [CFZ99]. The main parameters of ENCLUS are named as stated in the paper, so they are ϵ and ω, but the binning parameter is ξ, and describes the number of bins. So the parameter $\triangle$ of the ENCLUS paper for a dimension X is $\triangle = \frac{\max X - \min X}{\xi}$. Both algorithms use the same implementation techniques and data backend described in chapter 3. They are compiled using Clang 3.3 using `-O3` and `-ffast-math` flags. Using this optimization, Clang is able to to vectorize some parts of the implementation. This means, that loops can be replaced by a variant that uses SIMD operations, which speeds up the compiled executable. The used TBB has version 4.0.

To get, isolate and pre-process the data I've used some scripts. Because I think open research should not only contain open publications but also open data sources, you can find the scripts and some notes about them online.[1] Should you have some questions, find a bug or want to contribute, feel free to use the issue tracker or file a pull request.

5.1 Architecture and Climate

This data set is a time series that is used to determine, when windows of a building should be opened and when they should closed. To archive this, different metrics are measured at different locations of the builds, e.g.:

- light intensity from different directions
- wind force and direction
- outer and inner air temperature
- temperature of the building itself
- energy consumption of lighting, ventilation, air conditioner, different consumers in the rooms
- CO2 concentration
- status of the manual sun protection system and manual opened windows

[1] `https://github.com/crepererum/GSD`

The data set is only available for institute members, so I cannot provide a source. The original data set contains 434 dimensions and $43\,896$ objects, where the dimensions are the different metrics and the objects are the different points in time. Because the data set contains missing values, it is preprocessed. Dimensions are removed when they contain more then $10\,\%$ missing values. After this step, all objects that still have some cells with missing values are also removed. This cuts the data set down to 271 columns and $35\,400$ objects.

GraBaSS gets invoked with $t_e = 0.15$, $t_n = 0.65$ and $d = 3$. It tends to group metrics of subsystems together, that have some logical dependency, e.g.:

- volume of supply air, volume of used air, energy consumption for both systems
- air temperature in different parts of the building
- temperature of different parts of building itself
- temperature of some parts of the building itself and temperature of some rooms
- CO2 concentration at some measure station and temperature of some parts of the building
- number of persons in a room, manual window state and volume of supply air and used air

The algorithm also produces some less logical results, as the CO2 concentration at some points and the temperature of some parts of the building, but this cases are very rare, so they might be anomalies. The graph fixing works pretty well and reduced the number of similar subspaces.

ENCLUS has a problem with this data set, because it contains some outliers. Some dimensions contains only small data from 0.1 to 0.2, but some measures are described by values of 100 and more. Because ENCLUS uses an equi-width descretizer and calculates entropy values on this discrete values, it tries to join all dimensions with this kind of outliers. It also destroys some useful subspaces, if you are able to get some results. Because of the size of the data set, the number of affected dimensions and the size of the subspace is also large. This increases calculation of ENCLUS to calculate the results, so it does not finish within hours.

5.2 Drug Database

A source of high dimensional data is the list of ingredients of drugs. They collect important information about common combination of substances because one ingredient requires another, e.g. alcohol is a solvent for many chemicals. Common combinations can also be formed by combined substances that are used in drugs because manufactures produce them as a base for their product or because they act as a reseller for premixed drugs. The data set also contains information about chemicals that are never used together because they react with each other or because it does not make sense in terms of usability.

I hoped to find some public drug data bases containing drugs registered in Germany or in the EU. But this was not the case. Either they were only usable via a very limited interface or it

would cost me a enormous amount of money to buy a license. Since the open data movement in USA has a very long tradition, it was easy to find an American drug database, that is easy to parse and provides a huge data collection.[2] The entire data set contains $24\,094$ drugs and 4139 possible ingredients. Figure 5.1 visualizes a spectrum of this dataset, where rows act as drugs and image columns describe if a specific drug contains a specific substance.[3] The most common ingredients in this data are:

- water
- glycerin
- titanium dioxide
- propylene glycol
- dimethicone

The most common way to handle this data set would be the usage of a pattern miner, because the data table has only binary entries. I decided to use this data set nevertheless, because the dimensions have very good descriptions and I am able to explain subspaces.

To convert this data to a legal input data set, it is processed as followed: The only preprocessing that is done is to reduce the ingredients to their lower case text because the data set contains the same ingredients written in different combinations of lower and upper case letters. After the case normalization build a list of all ingredients in all drugs and bring them into a fixed order. Every ingredient forms one dimension. Then build a binary vector for every drug in the database where 1 describes that a ingredient is contained in the drug and 0 if this is not the case. I provide a parser and converter for this kind of data.

GraBaSS is used with $t_e = 0.25$, $t_n = 0.65$ and $d = 3$. It extracts many subspaces, that describe rare combinations, that are used in some natural products. The listed substances are also natural, e.g. leafs, flowers and roots. Because the ingredients are so uncommon, they form subspaces when only used together only a few times. To handle this issue, the ingredients are sorted according to the number of drugs that uses them and the upper quarter is picked. For this reduces data set, GraBaSS finds subspaces like this:

- alanine; arginine; glycine; methionine; phenylalanine; proline; thrionine
- citronellal; d&c orange no. 5; d&c red no. 6, 7, 21, 30 and 36
- barium, calcium, ethylene, iron, isopropylparaben, sulfate ion
- avobenzone; homosalate; octisalate; octocrylene; oxybenzone

ENCLUS extracted very small subspaces. When increasing ϵ, the algorithm does not finish within hours because the data set is too large. So I got no usable results.

[2] `http://dailymed.nlm.nih.gov/dailymed/downloadLabels.cfm`

[3] A full-resolution and monitor friendly version can be found at `http://studwww.ira.uni-karlsruhe.de/~s_mneuma/download/drugs.jpeg`

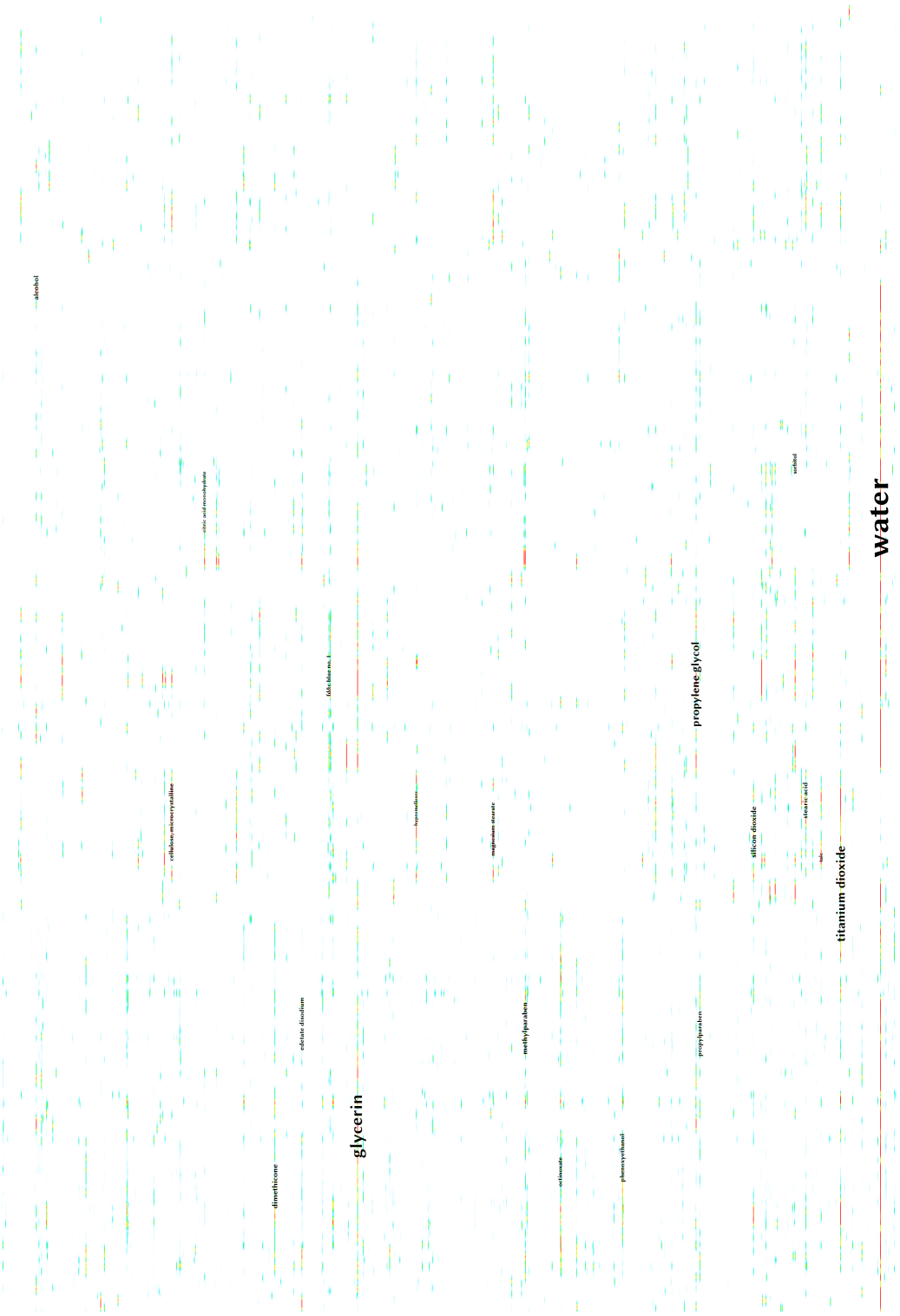


Figure 5.1: drugs spectrum

Table 5.1: Test results

Data set	GraBaSS	ENCLUS	PCA	Random	Fullspace
pendigits	**0.77**	0.52	0.43	0.55	0.55
ozone	**0.66**	0.53	0.48	0.51	0.51
musk	**0.74**	0.60	0.52	0.63	0.44

5.3 Tests with outliers

To test if the subspaces found be GraBaSS are usable for automatic data processing, I compared it to ENCLUS [CFZ99], PCA, random and the fullspace by doing an outlier analysis with LOF [Bre+00]. Because GraBaSS searches subspaces according to the similarity of the dimensions and does not look if the subspaces are good for outlier detection, a special post filtering is required. It should ensure that only subspaces are used for outlier detection which are good for this kind of analysis. I decided to use a very simple approach and filtered out subspaces found by GraBaSS with an entropy below a limit which is named f_{post}. More advanced filtering techniques are left for further research. For the comparison with PCA, the PCA was calculated and $|D|$ subspaces are generated with increasing number of dimensions so that the first subspace only contains the first PCA dimensions, the second the first two and so on. To compare against a random selection of subspaces, 20 subspaces are generated. For each, the number of dimensions is chosen uniformly from 1 to $|D|$ and then the dimensions are picked randomly.

I used the pendigits, ozone and musk data set from the UCI Machine Learning Repository[4]. The LOF values of all subspaces get summed up and the area under curve is calculated. The extraction of the true outliers from the data sets is described below. Table 5.1 shows a sum up of all results.

The pendigits data set[5] was prepared by choosing the smallest class and sampling this class down to create outliers. GraBaSS was invoked with $t_e = 0.003$, $t_n = 0.65$, $d = 2$ and $f_{\text{post}} = 12.5$. The post filter stripped 5 4-dimensional subspaces so it finally found 12 subspaces. I started ENCLUS with $\epsilon = 37$, $\omega = 12.745$ and $\xi = 82$ and I got 8 subspaces.

Of the ozone data set[6] I chose the normal day as outliers because it was the best choice for all all subspace sets except of PCA. GraBaSS got fired up with $t_e = 0.05$, $t_n = 0.65$, $d = 3$ and $f_{\text{post}} = 10.4$ and finished with 18 subspaces. This example shows that the post filter does always removed the smallest subspace, because it removed some 1-dimensional, the only existing 2-dimensional and some 3-dimensional subspaces, but the final result still has some 1-dimensional subspaces left. ENCLUS was started with $\epsilon = 23$, $\omega = 10.6$ and $\xi = 50$ and gave me 10 subspaces.

For the musk data set[7] the musks class is chosen from the clean2 file to be the true outliers. GraBaSS got the following parameters: $t_e = 0.30$, $t_n = 0.32$, $d = 3$ and $f_{\text{post}} = 5$. The post filter removed 8 of 22 1-dimensional subspaces, so LOF got 79 subspaces as input. The filter

[4] https://archive.ics.uci.edu/ml/index.html
[5] https://archive.ics.uci.edu/ml/datasets/Pen-Based+Recognition+of+Handwritten+Digits
[6] https://archive.ics.uci.edu/ml/datasets/Ozone+Level+Detection
[7] https://archive.ics.uci.edu/ml/datasets/Musk+%28Version+2%29

did a pretty good job here because it removed many trash dimensions. I passed ENCLUS the parameters $\epsilon = 17$, $\omega = 8.2$ and $\xi = 81$ and found 329 subspaces.

In all cases it was easier to get good subspaces from GraBaSS as from ENCLUS because ENCLUS tended to explode in complexity when trying to get high dimensional subspaces. This shows that the bottom-up approach the authors chosen for it does not work very well. When looking at the results the small subspaces always look like the have to get merged later but the complexity of the algorithm prevented me from running ENCLUS with such a parameter.

5.4 Scalability

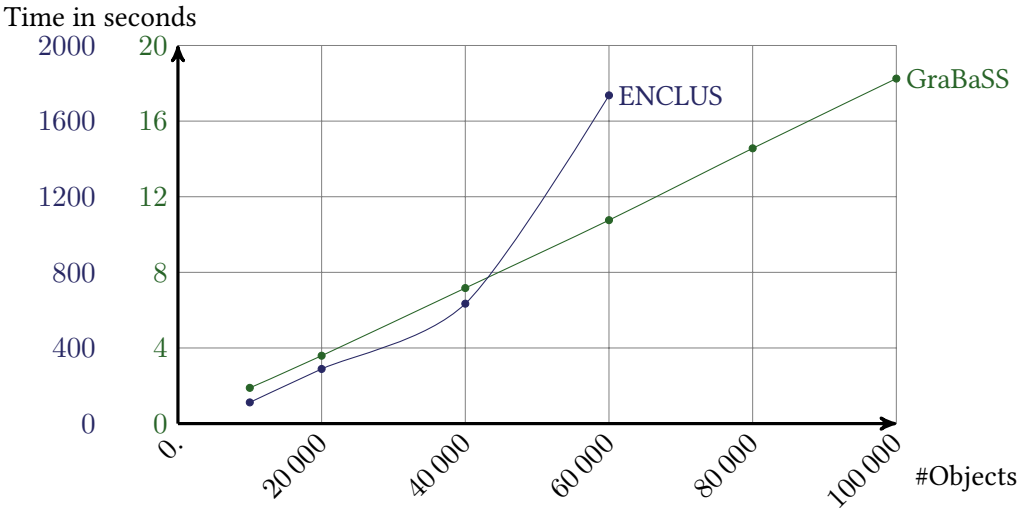
There are two scaling series, one for the number of objects and one for the number of dimensions. The GraBaSS parameters are fixed per each series. Because ENCLUS requires to change the ξ parameter to handle more dense data, it is changed when increasing the number of objects. ENCLUS is also very unstable when changing the number of bins, the number of dimensions or the density of the data. So I tried adjust the parameters to get the approximate same results for each data set of the series. For ENCLUS it was not possible to get subspaces above 4 dimensions because the complexity of the bottom-up approach just explodes when setting the parameters to get so interesting results.

The series with increasing number of objects contains always 100 dimensions whereas the number of objects is set to $10\,000$, $20\,000$, $40\,000$, $60\,000$, $80\,000$ and $100\,000$. GraBaSS scales nearly linear. I expect that this fact may change when using bigger data sets, because the binning requires $\mathcal{O}(|D| \cdot |N| \cdot \log |N|)$ and takes about $40\,\%$ of the entire calculation time. Table 5.3a shows the time profile. Changing the number of objects does not heavily change the time distribution over the parts of the algorithm. For ENCLUS I set ξ to $\sqrt{|N|}$ to satisfy the higher information density. Because increasing the number of bins also increases the complexity when calculating metrics for higher dimensional subspaces, ENCLUS only provides result in acceptable time when running with at most $60\,000$ objects. This problem occurs with every algorithm that tries to measure high dimensional subspaces directly on a discretized data set. It is not possible to set the number of bins for low dimensional subspaces correctly on dense data and avoiding a sparse grid on high dimensional data which also leads to high computation times. Figure 5.3a shows a scaled plot of the durations that GraBaSS and ENCLUS require. I chose a different scaling for the two algorithms to remove constant factors which can occur because of different data structures, non equal implementations[8], different cache friendly or unfriendly layouts or constant factors which are part of the algorithm itself.

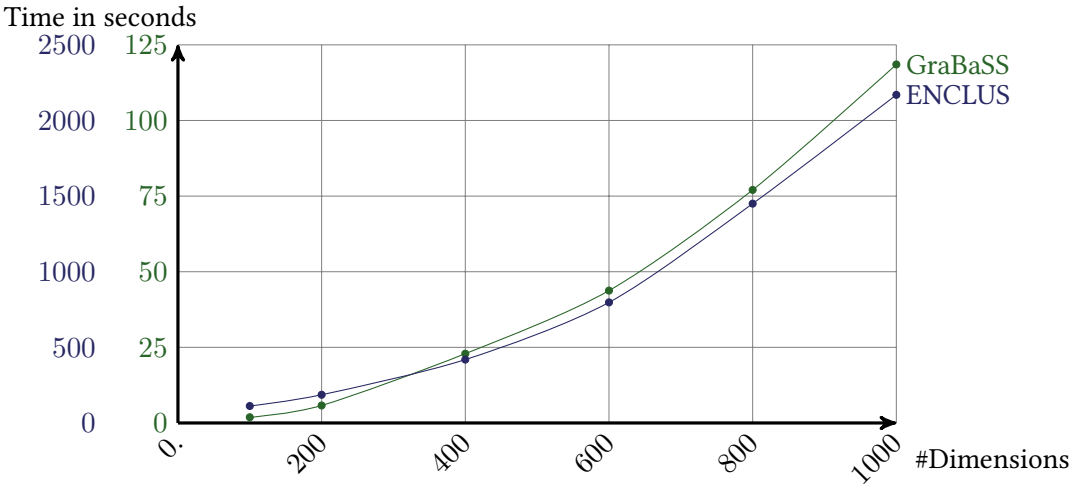
The dimension scaling series contains 100, 200, 400, 600, 800 and 1000 dimensions at a stable objects count of $10\,000$. GraBaSS and ENCLUS scale nearly equal as Figure 5.3b show, but the results are pretty different. GraBaSS can provide higher dimensional subspaces and handles the higher dimensional information. ENCLUS got stuck in low dimensional subspaces because the calculation times increased so heavily when trying to get more interesting results. This is a problem every algorithm that tries to calculate non-linear metrics for higher dimensional subspaces. This are all metrics that are not just a summation over all projected data points,

[8] I have tried to implement both algorithm in the same way but different algorithms also mean different techniques. So I cannot ensure that the two implementations have exactly the same quality.

Figure 5.2: Scalability comparison



(a) Scale number of objects at 100 dimensions



(b) Scale number of dimensions at 10 000 objects

Table 5.2: Time profile

#Objects	10 000	20 000	40 000	60 000	80 000	100 000
Binning	39 %	40 %	41 %	40 %	41 %	41 %
Pre-calulation			< 1 %			
Graph build-up	59 %	60 %	59 %	59 %	58 %	58 %
Graph fixing			< 1 %			
Clique searcher			< 1 %			
Total	1.89	3.59	7.17	10.76	14.56	18.25

(a) Synthetic data set with 100 dimensions

#Dimensions	100	200	400	600	800	1000
Binning	39 %	26 %	29 %	18 %	17 %	14 %
Pre-calulation			< 1 %			
Graph build-up	59 %	73 %	70 %	82 %	82 %	86 %
Graph fixing			< 1 %			
Clique searcher			< 1 %			
Total	1.89	5.79	22.91	43.76	77.04	118.52

(b) Synthetic data set with 10 000 objects

Data set	architecture	drugs	musk
Binning	61 %	30 %	40 %
Pre-calulation	< 1 %	3 %	< 1 %
Graph build-up	38 %	67 %	57 %
Graph fixing	< 1 %		
Clique searcher	< 1 %		
Total	20.72	76.65	1.94

(c) Real world data sets

Table 5.3: Scaling of graph processing

t_e	1	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
#Edges	0	1594	10 546	18 866	22 087	22 921
Max. #neighbors	0	53	178	236	257	261
Degeneracy	0	42	112	178	200	208
#Cliques	272	134	2202	6977	13 951	23 557
Graph build-up	48 %	48 %	44 %	45 %	44 %	42 %
Graph fixing	< 1 %	< 1 %	19 %	21 %	24 %	25 %
Clique searcher	< 1 %	< 1 %	< 1 %	3 %	4 %	5 %
Total	17.07	18.19	24.83	29.50	32.02	33.02

so all density based approaches are affected. As table 5.3b shows, the time profile of GraBaSS changes when increasing the number of dimensions. The reason for this is the linear scaling of the binning over the number of dimensions whereas the graph build-up required quadratic time.

To get a more detailed time profile of GraBaSS, I also tested it with the other data sets of this chapter. The details can be found in table 5.3c. The pendigits and ozone data set where not measured because they are to small to get a solid result. Whereas the distribution of binning and graph build-up changes, the actual graph based operations like fixing and clique searching are always very small. So I decided to test how this parts can affect the calculation time. For this, I chose the architecture data set and passed a changing t_e parameter to it to increase the number of edges. The graph distance parameter d was set to 3 and the neighborhood threshold t_n to 0.5 to get some workload to the graph fixing but to avoid a overproduction of edges. Table 5.3 shows the result. The number of edges is measured before doing a graph fixing, the maximum neighbor count and degeneracy are collected after fixing the graph. The time the graph build-up takes increases when increasing the number of edges but the percentage stays nearly the same. The graph fixing is the part which time amount increases the most. The implementation of this part is not parallelized because this part get not stressed too much under normal conditions. As proven in theorem 5 the complexity of this part is $\mathcal{O}(|V| \cdot m^2)$. A parallel version should take a smaller amount but this would only be a constant factor. Surprisingly the amount for the clique finder is always very low. The numbers clearly show that even a very unrealistic number of edges only leads to about $100\,\%$ more calculation time and no explosive behavior. Because of the fact that the graph processing does not break the complexity of the algorithm, the main goal of designing a method which operates in $\mathcal{O}(|D| \cdot |N| \cdot \log|N| + |D|^2 \cdot |N|)$ is reached.

6 Conclusion and further research

Feature selection is an important task when analyzing data. It helps to understand relationships of attributes, speeds up further analysis tasks and handles the curse of dimensionality. Big data sets of today and the next years require an efficient way and parallel algorithms that provide solid results. This can only be done by using new approaches. Usability is another important aspect because it enables analysts to act faster and to get better results. With GraBaSS, I developed one small piece for the data processing of the next years. It is not only a algorithm, but also a framework and an illustration of ideas that can used by other researcher. It outperforms other approaches when in comes to calculation time and provides clearer, in some cases even better results. The implementation uses optimizations to archive high performance and low memory footprint and the developed data backend provides a foundation for other algorithms.

But GraBaSS and its implementation are not perfect. The chosen similarity is not suitable for all tasks, especially when special expectations are made, what means to be similar. It might be also possible to find a better common similarity which works well for the most data sets. The chosen post-filter is very simple. It may be replaced by other metrics e.g. for measuring contrast or filtering according to training data set so the result only contains subspaces where you can find special clusters or outliers very well. GraBaSS does not perform a pre-filtering so that dimensions get pruned before they are compared to others. This can speed up GraBaSS and avoids that the entire chain of feature selection and further processing operates on dimensions that do not contain useful information. Another part, which is not researched very well are incremental algorithms, which do not require a full recalculation when new data points or even new dimensions are added or even removed to the data set. This would enable better real time analysis of data, because incremental clustering would allow to choose t_e interactively or by an automatic algorithm so it produces a good cluster count. Automatic parameter detection might also be possible when t_n and d should be chosen, because most humans are able to visually detect the problem of small quasi-cliques so an algorithm which does something similar could solve the problem for bigger data sets.

The realization in C++ is good but does not use GPUs or other accelerators like FGPAs and is not scalable about multiple computers. To archive this the first one, an OpenCL based implementation and further memory optimization could help. To use this common technology which operates as interface to GPUs, CPUs, FPGAs and other accelerators, the code has to refactored. It has to be split in small tasks and with a low amount of shared read-write memory. To scale over multiple computers it is required to provide a solid foundation of network code, error detection and load balancing. Heterogeneous architectures makes this task even more complicated. The data backend currently does not support any data checks or normalization for multiple architectures. A high performance, in-memory database which provides a generic interface for this kind of algorithm would be an ideal candidate to solve all these problems.

There are many topics left for research especially because data set size increases and real

time information becomes more important in times where computers do not get faster but get more parallel computing units. I believe that this is not my last work in this field and I hope to provide a small piece for better technology that may help other people in a wide range of tasks, today and in future times.

List of Figures

List of Algorithms

List of Symbols

$\lvert A \rvert$	Cardinality of set A
$+:$	Append (Returns first operand with second operand appended)
$\pi_d(x)$	Projection of point $x \in N$ to dimension $d \in D$
$\mathbb{N}_+$	Set of all positive integers, $\{1, 2, 3, \dots\}$
$\mathbb{N}_0$	Set of all non-negative integers, $\{0, 1, 2, \dots\}$
D	Set of all dimensions
E	Set of all edges of a graph
N	Set of all objects
V	Set of all vertices of a graph

Bibliography

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.

[Bey+99] Kevin S. Beyer et al. "When Is 'Nearest Neighbor' Meaningful?" In: *ICDT*. Ed. by Catriel Beeri and Peter Buneman. Vol. 1540. Lecture Notes in Computer Science. Springer, 1999, pp. 217–235. ISBN: 3-540-65452-6.

[Bre+00] Markus M. Breunig et al. "LOF: Identifying Density-Based Local Outliers." In: *SIGMOD Conference*. Ed. by Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein. ACM, 2000, pp. 93–104. ISBN: 1-58113-218-2.

[CFZ99] Chun Hung Cheng, Ada Wai-Chee Fu, and Yi Zhang. "Entropy-based Subspace Clustering for Mining Numerical Data." In: *KDD*. Ed. by Usama M. Fayyad, Surajit Chaudhuri, and David Madigan. ACM, 1999, pp. 84–93. ISBN: 1-58113-143-7.

[ELS10] David Eppstein, Maarten Löffler, and Darren Strash. "Listing All Maximal Cliques in Sparse Graphs in Near-optimal Time". In: *CoRR* abs/1006.5440 (2010).

[Est+96] Martin Ester et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proc. of 2nd International Conference on Knowledge Discovery and*. 1996, pp. 226–231.

[KMB12] Fabian Keller, Emmanuel Müller, and Klemens Böhm. "HiCS: High Contrast Subspaces for Density-Based Outlier Ranking." In: *ICDE*. Ed. by Anastasios Kementsietsidis and Marcos Antonio Vaz Salles. IEEE Computer Society, 2012, pp. 1037–1048. ISBN: 978-0-7685-4747-3.

[Mey05] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005. ISBN: 0321334876.

[Qui93] J. Ross Quinlan. *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 1-55860-238-0.

[SP07] Alok Sharma and Kuldip K. Paliwal. "Fast principal component analysis using fixed-point algorithm." In: *Pattern Recognition Letters* 28.10 (2007), pp. 1151–1155.

[Yao03] Y.Y. Yao. "Information-theoretic measures for knowledge discovery and data mining". In: *Entropy Measures, Maximum Entropy Principle and Emerging Applications*. Ed. by Karmeshu. Springer, 2003, pp. 115–136. ISBN: 978-3-540-00242-0.