MASTER

Improved algorithms for clustering ASML event log data
fast hierarchical clustering by a combination of cosine similarity and Fiedler vectors

Slenders, T.P.A.

Award date:
2014

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
Improved algorithms for clustering
ASML event log data
Fast hierarchical clustering by a combination of cosine similarity and
Fiedler vectors

MASTER OF SCIENCE THESIS

For obtaining the degree of Master of Science in Discrete Mathematics
at the department of Mathematics and Computer Science
of Eindhoven University of Technology.

ASML
TU/e

Tom Slenders – 0655386
tom.slenders@gmail.com

Friday 29th August, 2014

Supervisors:
Nikhil Bansal Eindhoven University of Technology
Peter van den Hamer ASML Holding NV
“You can have data without information, but you cannot have information without data.”
- Daniel Keys Moran
Abstract

Machines of ASML sometimes break down, this produces a couple of error messages or so called symptoms for each breakdown. We use cosine similarity to correlate different breakdowns, so called faults, with each other. Then we cluster the faults by recursively splitting each cluster in two. The cut is determined using the bound Cheeger’s inequality provides for the second smallest eigenvector of the normalized Laplacian. We derive a way to calculate this cut efficiently, resulting in an hierarchical clustering algorithm that runs in near-linear time of the size of the input data. We propose a generalization of the hierarchical clustering algorithm that offers versatility for other benchmarks. Then we propose a framework in which this algorithm can be used to remove outliers and create clusterings based on different optimization criteria. Finally we apply these methods to real data of ASML.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>v</td>
</tr>
<tr>
<td>Preface</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xiii</td>
</tr>
<tr>
<td>Nomenclature</td>
<td>xiv</td>
</tr>
<tr>
<td>Glossary</td>
<td>xvi</td>
</tr>
</tbody>
</table>

1 Introduction 1

2 Current approach 2
  2.1 Smart Diagnostics Pipeline, the first three layers 2
  2.2 Smart Diagnostics Diagnoser 3
  2.3 Pareto Analysis 4

3 Our approach 5
  3.1 Translation to mathematics 5
  3.2 Clustering 6
  3.3 Project goal 8
  3.4 Solution outline 8

4 Cosine similarity 10
  4.1 Definitions 10
  4.2 Cosine compared to Jaccard 14
  4.3 Centroids 17
  4.4 Benchmark 19

5 Singular value decomposition and Cheeger’s inequality 22
  5.1 Singular value decomposition 22
  5.2 SVD approximation algorithm 25
  5.3 Laplacian and normalized Laplacian 25
  5.4 Computing second eigenvectors faster 31
Preface

Mathematics can be a mysterious thing. Most people think they do not understand it. Others try to understand it and find its wonders. They get sucked in and help building the magnificent card house that mathematics truly is. Only to find that there is no ceiling. They will be building forever...

After six years of studying mathematics I am still amazed by mathematics. In your first year you get the building blocks that make mathematics and from that you start building. From these building blocks you can make everything, which is something amazing. The most beautiful I find is when two seemingly independent forms of mathematics collide and create something even stronger than before.

This is what I encountered at my amazing internship at ASML. Starting with a measure to correlate things, writing it as elegant linear algebra and its relation with graphs was something I did not expect. While uncovering this method I felt like Archimedes putting his crown in a bath and the only word that came into mind was the word “eureka.”

I could not have done this without the help of some magnificent people.

First of all I would like to thank Peter, who without I could have never completed this project. Always looking at problems with perfect analogies, he gave perfect advice. Also, Nikhil, who often listened to me rambling, only to pinpoint to the exact problems I was having. Furthermore I would also like to thank Cor Hurkens and Michiel Hochstenbach, who, together with Peter and Nikhil, read through this whole thesis.

I want to thank my colleagues, with whom I enjoyed several fun lunches. My family, who gave indefinite support throughout this whole project. Wouter Geelen for this \LaTeX\ style. My friends, everyone at GEWIS and my house mates, who made this whole time period enjoyable also without mathematical content.

Lastly, I want to thank you, for your interest in this work. I hope you enjoy reading it as much as I enjoyed working on it.

Eindhoven, Netherlands

Tom Slenders

Friday 29th August, 2014
We found this gorgeous picture during our research but did not find a proper place to include it. Therefore we show it here as an example of how beautiful mathematics can be.
## List of Figures

2.1 Pipeline for Smart Diagnostics ................................. 3  
3.1 Different clustering types .................................... 7  
4.1 Comparison of Jaccard and cosine similarity for $\gamma = 0.8$ .......... 15  
4.2 Comparison between the minimum and maximum value of the cosine similarity for different values of the Jaccard similarity .......................... 16  
7.1 Overview of the different processing steps the clustering algorithms can go through ........................................ 42  
7.2 Overview of the Tree Pruning step .............................. 46  
7.3 Extensive overview of the processing steps SVD clustering goes through ...................................................... 48  
8.1 Distribution of both symptoms per fault and faults per symptom for both datasets ........................................ 51  
8.2 Distribution of symptoms per fault for different main event types .................. 53  
8.3 Example of barplot visualization ................................ 56  
8.4 Barplot of clustering case 1 clustered by Stacking algorithm and by the closest point method on random faults ........................................ 57  
8.5 Comparison of closest points method on random faults or on faults by the Stacking Algorithm .................. 58  
8.6 Barplot of K-means clustering of clustering case 1 .................. 59  
8.7 Sorted values of KMB for 100 restarts of K-Means with random starting points (left) and the KMB during the iterations of 5 restarts of K-Means (right) .......... 61  
8.8 Example of similarity matrix of artifical data and what is should look like after clustering ........................................ 62  
8.9 Example of the similarity matrix sorted by the important vectors of both $S$ and $N$ .................. 63  
8.10 Density of main event types of important vectors for both matrices .................. 65  
8.11 Tree example ...................................................... 68  
8.12 Barplot of how well main event types are clustered for clustering case 1 by SiS on $N$ with several postprocessing methods applied .................. 71  
8.13 Comparison of KMB and the amount of clusters in a clustering for a few clustering methods with no outliers. .................. 73
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Basic analysis of the two datasets.</td>
<td>50</td>
</tr>
<tr>
<td>8.2</td>
<td>Overview of filter results.</td>
<td>54</td>
</tr>
<tr>
<td>8.3</td>
<td>Results of clustering case 1 for Stacking Algorithm and closest points on random faults</td>
<td>57</td>
</tr>
<tr>
<td>8.4</td>
<td>Results of clustering case 2 for Stacking Algorithm and closest points on random faults</td>
<td>58</td>
</tr>
<tr>
<td>8.5</td>
<td>Results for K-Means on clustering case 1</td>
<td>60</td>
</tr>
<tr>
<td>8.6</td>
<td>Results of clustering case 2 clustered by K-Means</td>
<td>61</td>
</tr>
<tr>
<td>8.7</td>
<td>Results for different cutting methods on both matrices</td>
<td>64</td>
</tr>
<tr>
<td>8.8</td>
<td>Results of clustering case 2 without outliers clustered by some SVD based methods.</td>
<td>66</td>
</tr>
<tr>
<td>8.9</td>
<td>Results for different tree pruning methods</td>
<td>67</td>
</tr>
<tr>
<td>8.10</td>
<td>Results for different outlier removal methods</td>
<td>69</td>
</tr>
<tr>
<td>8.11</td>
<td>Results for clustering case 1 clustered by SiS on $N$ with several postprocessing methods applied.</td>
<td>70</td>
</tr>
<tr>
<td>8.12</td>
<td>Results for clustering case 2 clustered by SiS on $N$ with several postprocessing methods applied.</td>
<td>72</td>
</tr>
<tr>
<td>8.13</td>
<td>Comparison of three algorithms for clustering case 1.</td>
<td>72</td>
</tr>
<tr>
<td>8.14</td>
<td>Comparison of three algorithms for clustering case 2.</td>
<td>73</td>
</tr>
<tr>
<td>9.1</td>
<td>Comparison of four different clustering algorithms</td>
<td>75</td>
</tr>
</tbody>
</table>
# List of Algorithms

<table>
<thead>
<tr>
<th>Section</th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Power Method</td>
<td>25</td>
</tr>
<tr>
<td>6.2</td>
<td>K-Means</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>SVD-clustering</td>
<td>36</td>
</tr>
<tr>
<td>6.4</td>
<td>Scanning algorithm</td>
<td>38</td>
</tr>
</tbody>
</table>
# Nomenclature

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Normalized version of vector</td>
</tr>
<tr>
<td>1</td>
<td>Vector of all ones</td>
</tr>
<tr>
<td>$1_X$</td>
<td>Incidence vector of set $X$</td>
</tr>
<tr>
<td>$\partial(C)$</td>
<td>Border of $C$, all edges with vertex in both $C$ and $V/C$</td>
</tr>
<tr>
<td>$\delta(C)$</td>
<td>Sum of weights of $\partial(C)$</td>
</tr>
<tr>
<td>$\theta(C)$</td>
<td>Isoperimetric ratio of cluster $C$</td>
</tr>
<tr>
<td>$v_2$</td>
<td>Second smallest eigenvalue of $N$</td>
</tr>
<tr>
<td>$\sigma_2$</td>
<td>Second largest eigenvalue of $S$</td>
</tr>
<tr>
<td>$\phi(C)$</td>
<td>Conductance of cluster $C$</td>
</tr>
<tr>
<td>$\phi_G$</td>
<td>Conductance of graph $G$</td>
</tr>
<tr>
<td>$\Lambda$</td>
<td>General matrix</td>
</tr>
<tr>
<td>$C$</td>
<td>Cluster, a set of faults</td>
</tr>
<tr>
<td>$C'$</td>
<td>Clustering, a set of clusters</td>
</tr>
<tr>
<td>$c$</td>
<td>Centroid, often corresponding to cluster $C$</td>
</tr>
<tr>
<td>$\hat{c}$</td>
<td>Normalized centroid</td>
</tr>
<tr>
<td>$D$</td>
<td>Degree diagonal matrix</td>
</tr>
<tr>
<td>$d$</td>
<td>Degree vertex, sum of the weights of each row in $S$</td>
</tr>
<tr>
<td>$E$</td>
<td>Edges in a graph</td>
</tr>
<tr>
<td>$E$</td>
<td>The set of all symptoms</td>
</tr>
<tr>
<td>$e$</td>
<td>Symptom</td>
</tr>
<tr>
<td>$F$</td>
<td>Fault, a set of symptoms</td>
</tr>
<tr>
<td>$F'$</td>
<td>Set of all faults</td>
</tr>
<tr>
<td>$f$</td>
<td>Incidence vector corresponding to fault $F$</td>
</tr>
<tr>
<td>$\hat{f}$</td>
<td>Normalized fault, used for cosine similarity</td>
</tr>
<tr>
<td>$G$</td>
<td>Graph</td>
</tr>
<tr>
<td>$I$</td>
<td>Identity matrix</td>
</tr>
<tr>
<td>$h$</td>
<td>Scoring function for scan based cutting algorithms</td>
</tr>
<tr>
<td>KMB</td>
<td>K-Means Benchmark, the sum of the similarity of all faults with their centroid</td>
</tr>
<tr>
<td>$L$</td>
<td>Laplacian matrix</td>
</tr>
<tr>
<td>METS</td>
<td>Main event type score, a benchmark for finding main event types</td>
</tr>
<tr>
<td>$N$</td>
<td>Normalized Laplacian matrix</td>
</tr>
<tr>
<td>$R$</td>
<td>Normalized fault matrix</td>
</tr>
<tr>
<td>$S$</td>
<td>Cosine similarity matrix</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>BoS</td>
<td>Border based splitting method</td>
</tr>
<tr>
<td>CP</td>
<td>Closest point method</td>
</tr>
<tr>
<td>KM</td>
<td>K-Means</td>
</tr>
<tr>
<td>MiCo</td>
<td>Minimum conductance splitting method</td>
</tr>
<tr>
<td>PP</td>
<td>Postprocessing</td>
</tr>
<tr>
<td>QC</td>
<td>Stopping criterion</td>
</tr>
<tr>
<td>SA</td>
<td>Stacking algorithm</td>
</tr>
<tr>
<td>SC</td>
<td>Splitting criterion</td>
</tr>
<tr>
<td>SD</td>
<td>Smart Diagnostics</td>
</tr>
<tr>
<td>SFF</td>
<td>Small fault filter</td>
</tr>
<tr>
<td>SiS</td>
<td>Sign based splitting method</td>
</tr>
<tr>
<td>SVD</td>
<td>Singular value decomposition</td>
</tr>
<tr>
<td>TC</td>
<td>Tree clustering algorithm</td>
</tr>
<tr>
<td>TP</td>
<td>Tree pruning algorithm</td>
</tr>
</tbody>
</table>
## Glossary

<table>
<thead>
<tr>
<th>Word</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASML</td>
<td>The company in association with this project was done</td>
</tr>
<tr>
<td>Centroid</td>
<td>The average fault of a cluster, a representative</td>
</tr>
<tr>
<td>Cluster</td>
<td>Group of faults</td>
</tr>
<tr>
<td>Clustering</td>
<td>A group of clusters, finding a good clustering is the goal of this project</td>
</tr>
<tr>
<td>Fault</td>
<td>A date, time, machine combination with a set of symptoms</td>
</tr>
<tr>
<td>Leaf</td>
<td>The bottom nodes of a hierarchical clustering</td>
</tr>
<tr>
<td>Hierarchical clustering</td>
<td>A clustering created by recursively splitting clusters in multiple smaller clusters, it can be seen as a tree</td>
</tr>
<tr>
<td>Machine</td>
<td>The product that ASML makes</td>
</tr>
<tr>
<td>Main event type</td>
<td>Top level classification known for faults, for example TEST or STARTUP</td>
</tr>
<tr>
<td>Pareto</td>
<td>Histogram of faults and their frequencies, used to identify possible design improvements</td>
</tr>
<tr>
<td>SDT</td>
<td>Event log view/analysis tool that runs on a PC</td>
</tr>
<tr>
<td>Similarity</td>
<td>A measure for comparing two faults</td>
</tr>
<tr>
<td>Smart Diagnostics</td>
<td>New pattern recognition technology implemented inside SDT</td>
</tr>
<tr>
<td>Symptom</td>
<td>An indication something might be wrong in the machine</td>
</tr>
</tbody>
</table>
Introduction

ASML has multiple machines which occasionally stop working properly, a so called breakdown. ASML developed a software tool called System Diagnostics Tool or SDT to analyze these breakdowns to provide easier solution finding. Recently a tool called Smart Diagnostics has been added to SDT. It can recognize certain tasks the machine was doing. A task is a series of items the machine is executing. A task that caused a breakdown is called a fault. The tool then compares and correlates older faults to a current fault. If for any of the highly correlated faults a previously found solution is known, this is shown and the engineer can probably use this solution for the current fault as well. The machine is mostly deterministic, but since it is so complex not everything is known about what happens at every moment. A top level structure of this is known. ASML knows for example when the machine is in STARTUP mode. But ASML is interested if lower level structures can be found as these lower levels can be used to improve their correlation finding.

Clustering is a way to approach this problem. Clustering is a way to create groups of correlated things, in this case groups of faults. Hierarchical clustering is a way to cluster by creating a tree structure, for example by splitting the groups recursively in two. This seems like a way to find these top level structures and also the structures just below it.

There is a second application in which clustering can be used and is already being used at ASML. If we cluster faults and sort the clusters on their size, we get a sorted list that shows which parts of the machine often cause faults. By trying to fix the parts that cause the most faults, newer iteration of the machines can be improved to have less faults or current machines can be upgraded. Or, an expert can find a solution for faults that occur often to have a solution available when it happens again.

The difference with the first type of clustering is that we are now interested in very low level structures. We want to create groups that have a high “internal correlation.” This means we will create a lot of small groups. Thus this problem is different than the first, but both require clustering of faults.

In this thesis we will cover both problems by using the same techniques. First we will explain the current methods used by ASML. Then we will translate this problem to mathematics, covering how to measure correlation between faults and extensively research how we can create (hierarchical) clusterings. Lastly, we will apply these methods to data provided by ASML to see how well the methods perform and which are most useful for ASML.
Current approach

ASML has several machines that break down from time to time. When this happens ASML tries to recover the machine as quickly as possible, as each second a machine is not working, it will cost the machine owner money.

For this SDT with Smart Diagnostics was developed by ASML. This tool can recognize tasks. A task is a time span of a few seconds to several minutes in which the machine is trying to accomplish one specific thing, for example starting some drivers. The tool connects error messages to the current task a machine was doing, detects which task caused the breakdown and tries to find similar tasks. A task that causes a breakdown we will call a fault. The idea behind this is simple: If a machine in the past had a fault and a solution was found, then if a current fault has a task with similar error messages, we can probably apply the same solution to this fault. Smart Diagnostics is the system that currently has these ideas implemented and is being used in the field. This system seems to work well, but there is also room for improvement.

We will discuss the processing steps the data goes through in Smart Diagnostics and will mainly focus on the last layer, which is where clustering is the most useful. We will also cover the Pareto, which can be seen as a spin-off of Smart Diagnostics and provides insight in what type of faults occur most often.

2.1 Smart Diagnostics Pipeline, the first three layers

In Figure 2.1 the stages Smart Diagnostics goes through are shown. We will roughly cover what each layer does. This project mainly focuses on the fourth layer, therefore the output of the third layer and input of the fourth layer needs to be understood.

ASML makes scanners, machines that optically image patterns onto a silicon wafer. The input data of Smart Diagnostics comes directly from the scanner and event log files are created for each scanner. These are then parsed through the L1a Parser to create basic events. These contain time stamped log entries and all kinds of information. Events can range from error messages to warnings or just "normal" events in the machine. Layer 1b links events, but is not very interesting from our perspective.

The second layer makes tasks of the events. The machine is often doing a specific task at a certain moment, for example processing a lot of wafers is one such task. L2 Tasker maps the
events to a task, it also finds out if this task went according to plan. It is also possible that a
task had to abort abruptly or that a task had some problems, but did finish. All this is detected
by this layer and now we can detect which tasks had problems and are the so called faults we
are interested in.

In the third layer we start splitting the tasks. The tasks that had no problems are put in a
database, while faults will continue being processed. In this layer we create symptoms from
the events. Symptoms are indications of something that might be wrong. Often these are just
the events we’ve seen before. But some events get merged into one symptom, while some
events are split out into several symptoms. Also, we are longer not interested in how often
each event occured, but just if they occured. What events are merged or split up is based on
machine knowledge from experts at ASML. Now we can look at the symptoms as the symptoms
a doctor would search for to diagnose a disease. The third layer detects from the tasks that went
okay what symptoms occur often in healthy machines. Since these symptoms happen often in
healthy machines, they will also occur often in unhealthy machines. But they probably don’t
add useful information. Therefore they are filtered out by ASML. The tasks with symptoms
are now called patterns by ASML, but we will call them faults. The reason for a different name
in this report is that one could consider a pattern unique, but it is not. The faults are the final
stage before the diagnozer and can be considered the final form of a preprocessing process. The
final result has to be interpreted, but this can be done in multiple ways: By the Diagnozer or by
the Pareto Analysis, and, as we will see later on, also by our algorithm creating a clusters or a
tree structure that might provide insight in the processes of the machine.

2.2 Smart Diagnostics Diagnozer

The fourth layer in Smart Diagnostics (see Figure 2.1) creates the results usable by the user. The
main idea is that when a fault occurred in the past and a solution was found for this fault, then if
in the present a fault occurs which is similar to the one in the past, it is likely the same solution
can be applied. This philosophy is applied in the Diagnozer. When a fault occurs, SDT compares
this to all previous faults in a certain time span. It uses some calculation on how similar the
faults are and then sorts them based on their similarity. Faults that previously had no solution
can be filtered out and the faults with a solution with a similarity above some threshold are
seen as interesting. Someone in the field should investigate if these solutions are applicable for
the current fault as well.

Figure 2.1: Pipeline for Smart Diagnostics. In this project we will mainly focus on the 4th layer.
The calculation ASML uses for their similarity calculations is simple. If you compare two faults then you take the symptoms they have in common and divide it by the total amount of unique symptoms these two have. From practice it seemed that if this ratio is at least 80%, then the two faults are likely to have the same cause.

In recent developments a new similarity function is used that considers some information that is known of the state of the machine. Detection of lower level states can be useful since it seems that using this information can create more accurate matches of faults. So far this has only been applied to one type of fault and seems to give very good results. However, it is not known how to do this for lower level states and a mere direction in where to search for can already help finding lower level states.

Another recent development is the inclusion of weights to certain symptoms. These weights are added manually to certain symptoms that expert say have a major impact. Adding weights is still in an early stage.

2.3 Pareto Analysis

There is a second use for the data generated by the third layer of the Smart Diagnostics pipeline, which is the Pareto Analysis. The Pareto Analysis tries to find faults which have similar symptoms and sorts them on how often these faults occur. This results in a list which shows how often certain faults occur. The idea behind this is that these are the type of faults that need the most attention. If the cause of these type of faults can be found, one could improve the machine overall or at least have a solution for these often occurring faults ready, which improves overall uptime of the machines. Right now this method uses a system where only faults that have exactly the same symptoms are grouped together, otherwise it leads to performance issues. ASML is actually interested in groups with 80% match, since these are likely to have the same cause. But it is not known how to find them in reasonable time.
Our approach

3.1 Translation to mathematics

The patterns ASML produces (see Figure 2.1) we will call faults. Each fault is a machine-date-time combination and has a set of symptoms. It corresponds to a breakdown or other faulty task. A symptom can be seen as an identifier that something might be wrong. In the case of ASML this translates to error messages and sometimes linked events. In practice we can simply say whether a fault contains a certain symptom or not, preprocessing of this has been done and we can simply use the symptoms as delivered by ASML. Now we want to compare faults with each other. Each fault consists of some symptoms on which the similarity between faults should be related. Eventually we want to find clusters of similar faults.

Denote \( F \) the list of faults. Each fault \( F_i \in F \) consists of some symptoms: \( F_i \subset \mathbb{E} \), where \( \mathbb{E} \) denotes the set of all possible symptoms. Note that \( \mathbb{E} \) is a set, while \( F \) is not, as some faults will occur multiple times (with different time stamps however). Therefore \( F \) is an ordered list. We will abuse set notation for \( F \) a bit. For example if we write \( \sum_{F \in \mathbb{E}} |F| \), we actually take the sum over all faults and not just the unique ones.

We can also look at our faults in a different way. We define some order for our symptoms \( e_j \in \mathbb{E} \) such that we can define an incidence vector \( f_i \) for \( F_i \) as follows

\[
(f_i)_j := \begin{cases} 
1 & e_j \in F_i \\
0 & e_j \notin F_i 
\end{cases}
\]  (3.1)

We would like to define some measurement on how two faults are correlated. ASML uses the Jaccard similarity for this.[1]

**Definition 3.1.** A similarity is a function \( s(a,b) \) such that

1. \( 0 \leq s(a,b) \leq 1 \)
2. \( s(a,b) = 1 \) if and only if \( a = b \)
3. \( s(a,b) = s(b,a) \)
Definition 3.2. The Jaccard similarity $s_J$ between two sets $X, Y$ is defined as

$$s_J(X, Y) := \frac{|X \cap Y|}{|X \cup Y|}$$  \hspace{1cm} (3.2)

To solve dividing by zero we define $s_J(\emptyset, \emptyset) = 1$.

We can define an equivalent for $f_1, f_2$ corresponding to $F_1, F_2 \in \mathcal{F}$:

$$s_J(f_1, f_2) = \frac{f_1 \cdot f_2}{\|f_1 \lor f_2\|_1}.$$  \hspace{1cm} (3.3)

Where $\lor$ denotes a bitwise or-operator. This definition is not very useful however, since the $\lor$-operator is hard to use in practice.

3.2 Clustering

There are multiple ways to define clusters. The most straightforward way is the following.

Definition 3.3. A cluster $C \subseteq \mathcal{F}$ is a list of faults. A clustering $\mathcal{C}$ is a set of clusters. Furthermore, each fault is in at most one cluster $C_i \in \mathcal{C}$. The faults that are not in any cluster are called outliers.

Note that a cluster must be a list of faults since it can contain different faults with the same symptoms (intuitively we expect that faults that have the same symptoms always end up in the same cluster). Also note that some faults do not have to be clustered, the so called outliers. Depending on the problem we can require that there are no outliers, i.e. all faults are in exactly one cluster.

There are some other forms of clustering which are similar to the one defined above, see Figure 3.1. Firstly, we can relax the condition that a fault can be in at most one cluster (Type 2). Secondly, we can relax this even more by saying that each fault corresponds to $k$ clusters each with a certain value $a_i$ such that $\sum a_i = 1$ (Type 3). Thirdly, we can create a tree structure, which gives more than just $k$ clusters. This tree structure can be viewed as follows: One starts with all faults being in one cluster as defined in Definition 3.3. Then we recursively split this cluster in two or more clusters and continue until only one fault is in each cluster or another stopping criterion is met (Type 4). Lastly, one can take this even one step further by including a weight on each of the branches of the tree or even extend this more by creating a dendrogram. A dendrogram is a weighted tree with a parent having a strictly higher weight than its children. The advantage of this is that the weights can be presented visually. If a clustering algorithm creates a tree structure it is also called a hierarchical clustering algorithm. [2]

We will not use the first and second alternative to a clustering definition because they are less intuitive to use. Especially for an end user it makes more sense if a fault is in just one cluster. Also, creating a benchmark when allowing a fault in more than one clusters becomes less intuitive. We will, however, use the tree structure to create a cluster. By taking the leaves of the tree clustering we get back to our original definition of a cluster. We will mostly use the weighted tree structure as this provides benefits in the tree pruning step (see Section 7.3).
Type 1: Separate groups

<table>
<thead>
<tr>
<th>Fault</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault 1</td>
<td>0.2</td>
<td>0.7</td>
<td>0.1</td>
</tr>
<tr>
<td>Fault 2</td>
<td>0.3</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>Fault 3</td>
<td>0.2</td>
<td>0.2</td>
<td>0.6</td>
</tr>
<tr>
<td>Fault 4</td>
<td>0.9</td>
<td>0</td>
<td>0.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Type 2: Overlapping groups

Type 3: Fraction in each group

Type 4: Hierarchical clustering - a tree

Dendrogram

**Figure 3.1:** Different clustering types
However, we will not create a dendrogram, where each weight of the child nodes is strictly lower than the parent node. Because of the huge dataset this becomes hard to visualize. So there is no use for creating a dendrogram, while it puts restrictions on the weights.

The goal of a clustering is that the faults in the same cluster are related to each other in some way. For example, we could want a high similarity between all pairs in a cluster, we will speak of a high internal similarity. We could also want to have some relation, often negative, between different clusters. There are several ways to define this and we will give some definition later.

3.3 Project goal

ASML has two problems of which they would like a better understanding. Both of them can be understood better by clustering! The problems by themselves are more complex, but we will only consider the clustering problems for this project.

For the Smart Diagnostics Diagnozer (SDD), they want to find high level structures. They already know what main event type each fault is in, this is a state of the machine they can detect already, for example STARTUP or TEST. They would like to know more about the structure directly below this. If they can detect substates, they can improve their similarity function. This structure can probably be recognized from a clustering with the amount of clusters equal to the expected amount of substates (or subsubstates). It can even be more interesting to use hierarchical clustering to find a tree. We expect that the tree detects the substates and splits these up into subsubstates and so on.

The Pareto already is a clustering, it is created from a list of how often faults with the same symptoms occur and only the top X most occurring faults are listed. This is basically a cluster but with a lot of faults not clustered, i.e. outliers. This is because of performance issues. It makes sense to consider all faults for a clustering. This is a different clustering than the one for the SDD because we are interested in finding clusters with a very high internal similarity. We want the faults in a cluster to at least 80% (Jaccard-) similarity to each other. This way we expect that one cluster represents one cause of a breakdown or faulty task and we expect the same solution can be used for all faults in this cluster.

So in short we want to solve two problems, but they are very related because they are both clusterings of the same dataset. The main difference is how many clusters we want to create for each problem.

3.4 Solution outline

We will try to give a solution to both problems. First we transfer to a different similarity function: the cosine similarity. The reason for this is that we can achieve a significant improvement of the running time for the algorithm. It also provides us with a way to create cluster representatives and gives us a way to benchmark efficiently (Chapter 4). We then explain singular value decomposition and how to use that to calculate Fiedler vectors. This is the main mathematical background we use and on which our main algorithm heavily depends (Chapter 5). We then propose a few clustering algorithms: An equivalent to the current Pareto, the very often used K-Means and an algorithm that splits the data recursively in two, based on Fiedler vectors.
(Chapter 6). Next we create a framework on how to use this splitting based algorithm more effectively that makes it more useful for ASML (Chapter 7). Lastly, we compare all proposed algorithms based on actual data and determine useful settings of the parameters such that it is optimized for ASML (Chapter 8).
Cosine similarity

In this chapter we will show the cosine similarity as an alternative to the Jaccard similarity which is used by ASML. We will compare the two similarities and show in what way they are alike. Then we will introduce centroid which can be seen as cluster representatives. Lastly we will provide a benchmark for clusterings.

4.1 Definitions

The Jaccard similarity is used by ASML because it is intuitive and seems to give good results. One of the nice properties of the Jaccard similarity is that it imposes a metric.

**Definition 4.1.** A metric or distance function $d(a, b)$ on some $a, b, c$ has the following conditions:

1. $d(a, b) \geq 0$
2. $d(a, b) = 0$ implies $a = b$
3. $d(a, b) = d(b, a)$ (symmetry)
4. $d(a, b) + d(b, c) \geq d(a, c)$ (triangle inequality)

One can try to define a metric from a similarity by $d(a, b) = 1 - s(a, b)$. By the properties of a similarity the first 3 conditions of the metric are automatically satisfied. The triangle inequality however has to be checked and is, as we will later see, not valid for all similarities, even simple ones.

**Definition 4.2.** We define the Jaccard metric

$$d_f(X, Y) := 1 - s_f(X, Y),$$

with $d_f(\emptyset, \emptyset) = 0$

**Proposition 4.3.** The Jaccard metric is a metric.
Proof. The first three conditions are trivially met. The fourth, however, is a bit harder. A proof can be found in [1]. \(\square\)

Downside of the Jaccard similarity however is that the vector representation of the sets \(X\) and \(Y\) as in Equation (3.3) is not very useful, it is not possible to use some easy vector summation or multiplication to determine the similarity. One needs a division on a case to case basis, while it is desirable to be able to calculate the similarity for larger data sets at once, for example with matrix multiplication. For this reason we introduce a similarity called the cosine similarity.

**Definition 4.4.** The cosine similarity is defined as

\[
s_{\text{cos}}(X, Y) = \frac{|X \cap Y|}{\sqrt{|X| \cdot |Y|}} \quad (4.2)
\]

We define the initial cosine metric as

\[
\bar{d}_{\text{cos}}(X, Y) = 1 - s_{\text{cos}}(X, Y), \quad (4.3)
\]

with \(\bar{d}_{\text{cos}}(\emptyset, \emptyset) = 0\).

Using the vector representation we can write this in a different way. The vector representation, or bitvector of a set \(X \subset T\) is defined as follows. First we define an order on our set \(T\). We’ll denote the elements of \(T\) as \(t_i, i \in 1, \ldots, |T|\). Now we define our bitvector \(x\) as

\[
x_i := \begin{cases} 1 & t_i \in X \\ 0 & t_i \notin X \end{cases} \quad (4.4)
\]

Note that this is similar to the definition in Equation (3.1). Now we can define our cosine similarity for bitvectors as well. For \(X, Y \subset T\) and corresponding bitvectors \(x, y\).

\[
s_{\text{cos}}(X, Y) = s_{\text{cos}}(x, y) \quad (4.5)
\]

\[
= \frac{x \cdot y}{\sqrt{(x \cdot x)(y \cdot y)}} \quad (4.6)
\]

\[
= \frac{x \cdot y}{\|x\|_2 \|y\|_2} \quad (4.7)
\]

From eq. (4.7) you can see that we can separate \(x/\|x\|\) and \(y/\|y\|\). For any (bit)vector \(x\) we define

\[
\hat{x} = \frac{x}{\|x\|_2} \quad (4.8)
\]

This normalization implies

\[
s_{\text{cos}}(x, y) = \hat{x} \cdot \hat{y} \quad (4.9)
\]

Now we see one of the benefits of the cosine similarity. If we choose the right presentation for a set \(X\), namely \(\hat{x}\), we can calculate similarity with other sets quickly by calculating an inproduct. We will use this later in Section 5.4.
The name of the cosine similarity is derived from the following relation: \( s_{\cos}(X, Y) = \hat{x} \cdot \hat{y} = \cos \theta \), where \( \theta \) is the angle between vectors \( \hat{x} \) and \( \hat{y} \). Since \( \hat{x} \) and \( \hat{y} \) are normalized we can see them as points on a unit ball.

**Proposition 4.5.** The cosine similarity is a similarity, the initial cosine metric is not a metric.

**Proof.** First we proof that it is a similarity, as this will imply conditions 1 to 3 of the metric.

For \( \alpha = |X \setminus Y|, \beta = |Y \setminus X|, \gamma = |X \cap Y| \) we see that \( s_{\cos}(X, Y) = \frac{\gamma}{\sqrt{(\alpha + \gamma)(\beta + \gamma)}} \).

1. We see that \( \sqrt{(\alpha + \gamma)(\beta + \gamma)} \geq \sqrt{\alpha \cdot \beta} = \gamma \). Dividing by zero only occurs if \( \alpha = \beta = \gamma = 0 \), but we defined this case separately.

2. Now \( s_{\cos}(X, Y) \leq 1 \) for \( \alpha > 0 \) or \( \beta > 0 \), and \( s_{\cos}(X, Y) = 1 \) for \( \alpha = \beta = 1 \).

3. Symmetry follows directly from the definition.

Thus it is a similarity. We give a counter example to the initial cosine metric to be a (true) metric.

4. Take \( X = \{1\}, Y = \{1, 2\}, Z = \{2\} \). Now \( \hat{x} = (1, 0), \hat{y} = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}), \hat{z} = (0, 1) \). This means \( \hat{d}_{\cos}(\hat{x}, \hat{y}) = \hat{d}_{\cos}(\hat{y}, \hat{z}) = 1 - \frac{1}{\sqrt{2}} \) and \( \hat{d}_{\cos}(\hat{x}, \hat{z}) = 1 \). Now \( \hat{d}_{\cos}(\hat{x}, \hat{y}) + \hat{d}_{\cos}(\hat{y}, \hat{z}) < \hat{d}_{\cos}(\hat{x}, \hat{z}) \), which violates the triangle inequality. \( \square \)

It is quite unfortunate that the initial cosine metric is not a metric. This means we cannot use it for algorithms that require a metric to work. But quite often only a similarity is required for clustering algorithms and the performance properties of the cosine similarity are useful. It is possible to define a metric out of the cosine similarity. This can be done for example by using the arccos.

**Definition 4.6.** We define the cosine metric as \( d_{\cos}(X, Y) = \cos^{-1}(s_{\cos}(X, Y)) \).

**Lemma 4.7.** The angle \( \theta \) between two vectors \( x, y \) on the unit circle is a metric with \( 0 \leq \theta \leq \pi \)

**Proof.**

1. Follows from definition.

2. \( \theta = 0 \) and \( |x| = |y| \) implies that \( x = y \).

3. This follows from the definition of the angle between two vectors.

4. This follows from simple geometry. \( \square \)

**Theorem 4.8.** The cosine metric is a metric.

**Proof.**

\[
\begin{align*}
    d_{\cos}(X, Y) &= \arccos(s_{\cos}(X, Y)) = \arccos(\hat{x} \cdot \hat{y}) = \arccos(\cos \theta) = \theta. 
\end{align*}
\] (4.10)
COSINE SIMILARITY

Where $\theta$ is the angle between $\hat{x}$ and $\hat{y}$.

Since the angle between two unit vectors is a metric, this will also hold for the cosine metric. The only difference is that we now also have a possible zero-vector, which does not lie on the unit sphere. But from the properties of $s_{\cos}$ we find that $d_{\cos}(\emptyset, \emptyset) = 0$ and $d_{\cos}(X, \emptyset) = 1$, with $X \neq \emptyset$.

We will mostly be using the cosine similarity, but knowing that it is related to and can be converted to a metric might be helpful.

The cosine similarity provides some great computational benefits over the Jaccard similarity, mostly because the cosine similarity can be expressed in matrices. This gives a way to quickly calculate a lot of cosine similarities, but also enables us to use several nice properties that linear algebra provides us with.

**Definition 4.9.** We define the cosine incidence matrix $R$ with dimensions $|F| \times |E|$ as follows

\[
R := \begin{pmatrix}
-\hat{f}_1 \\
-\hat{f}_2 \\
\vdots \\
-\hat{f}_{|F|}
\end{pmatrix}
\] (4.11)

Using the definition of Equation (3.1) and Equation (4.8).

We define the cosine similarity matrix

\[
S := RR^T.
\] (4.12)

$S$ is a $|F| \times |F|$ matrix and we can see from eq. (4.7) that the cosine similarity matrix actually shows the similarity for each fault pair. This is very useful, as we can create $S$ from $R$ or calculate parts of $S$ from $R$. One of the great properties of $R$ is that it is sparse, while $S$ is not. So storing and working with $R$ is in general faster than $S$. We could create a similarity matrix for the Jaccard similarity as well, but this would lack the computational benefits of $S$.

In recent developments ASML is adding weights to their similarity function. An expert can give a weight to certain symptoms which then have a heavier influence on the similarity between two faults. For the Jaccard similarity this is done as follows.

Each symptom $e \in \mathbb{E}$ has a weight $w(e)$. Now the Jaccard similarity for $F_1, F_2 \in \mathbb{F}$ becomes

\[
s_J(F_1, F_2) = \frac{\sum e \in F_1 \cap F_2 w(e)}{\sum e \in F_1 \cup F_2 w(e)}
\] (4.13)

Note that this is the same as our previous definition if all weights are equal.

We can define something similar for the cosine similarity as follows

\[
s_J(F_1, F_2) = \frac{\sum e \in F_1 \cap F_2 w(e)}{\sqrt{\left(\sum e \in F_1 w(e)\right)\left(\sum e \in F_2 w(e)\right)}}
\] (4.14)

We can easily find a similar definition for the cosine similarity by only changing our vectors $f_i$ to include the weights as follows
If we take the inproduct of two normalized vectors \( \hat{f}_1, \hat{f}_2 \) we see that \( \hat{f}_1 \cdot \hat{f}_2 \) will give the same definition as above. So including weights is very natural to include in the cosine similarity.

It might be possible that ASML wants to include parameters as symptoms which can be seen as weights of one symptom, but the weight can vary from fault to fault. There is no clear definition for this if one wants to use the Jaccard similarity. For the cosine similarity it is more or less naturally induced by the new definition of vectors \( f_i \). If this definition is also useful has to be tested in practice.

### 4.2 Cosine compared to Jaccard

We can compare the cosine similarity and Jaccard similarity just by looking at how they behave for different sets. The cosine similarity has several computational advantages. But if the cosine similarity and Jaccard similarity do not behave the same, we cannot justify using the cosine similarity instead of the Jaccard similarity.

As an example we look at the discrete similarity:

\[
s_{\text{disc}}(X, Y) = \begin{cases} 1 & X = Y \\ 0 & X \neq Y \end{cases}
\] (4.16)

The discrete similarity behaves quite different from the Jaccard similarity and from the cosine similarity. When two sets are almost the same, both the Jaccard similarity and cosine similarity will give a high similarity measure (close to 1), while the discrete similarity will give a similarity of 0. So the discrete similarity behaves very differently from the Jaccard similarity and cosine similarity.

To compare the similarity measures, we can generalize the definition with sets we used. Take two sets \( X, Y \). Now we define \( \alpha = \frac{|X \setminus Y|}{|X \cup Y|}, \beta = \frac{|Y \setminus X|}{|X \cup Y|}, \gamma = \frac{|X \cap Y|}{|X \cup Y|} \).

\[
s_f(X, Y) = \frac{|X \cap Y|}{|X \cup Y|} = \gamma
\] (4.17)

\[
s_{\text{cos}}(X, Y) = \frac{|X \cap Y|}{\sqrt{|X| \cdot |Y|}} = \frac{|X \cap Y|}{\sqrt{|X| \cdot |Y| \cdot |X \cup Y|^2}}
\] (4.18)

\[
= \frac{|X \cap Y|}{\sqrt{|X| \cdot |Y|}}
\] (4.19)

\[
= \frac{\gamma}{\sqrt{(|X \cap Y| + |X \setminus Y|) / (|X \cup Y| + (|Y \cap X| + |Y \setminus X|)) / |X \cup Y|}}
\] (4.20)

\[
= \frac{\gamma}{\sqrt{(\alpha + \gamma)(\beta + \gamma)}
\] (4.21)

\[
s_{\text{disc}}(X, Y) = \begin{cases} 1 & \gamma = 1 \\ 0 & \gamma \neq 1 \end{cases}
\] (4.22)
We can also see that $\alpha + \beta + \gamma = 1$. If we now relax the conditions that $\alpha, \beta, \gamma$ are created by sets, we get $\alpha, \beta, \gamma \in \mathbb{R}, 0 \leq \alpha, \beta, \gamma \leq 1$, with $\alpha + \beta + \gamma = 1$.

**Figure 4.1:** Comparison of Jaccard similarity and cosine similarity by fixing the Jaccard similarity $\gamma = 0.8$ and letting $\alpha$ run over the possible values while keeping $\alpha + \beta + \gamma = 1$. This can be seen as keeping the Jaccard similarity fixed on 0.8 and calculating the cosine similarity for different sizes of set $X$. Note that the y-axis has values that lie close to each other.

This allows us to compare similarities more easily, since we can now use three continuous variables instead of two sets. Also, plotting this is a lot easier. For example, ASML often wants the Jaccard similarity to be at least 0.8. In Figure 4.1 we see how the cosine similarity behaves compared to the Jaccard similarity. We fix $\gamma = 0.8$ and using that $\alpha + \beta + \gamma = 1$ we can let $\beta$ depend on $\alpha$. Since $\alpha$ and $\beta$ are symmetric, it does not matter which depends on which. We see from Figure 4.1 that the value of the cosine similarity does not vary much by changing $\alpha$. However, the cosine similarity has a higher value than the Jaccard similarity. So if ASML would switch to the cosine similarity, they need to change their threshold of 0.8.

It appears that if we fix $\gamma$, we always get more or less the same picture as in Figure 4.1, but
it is scaled. It always looks more or less like a parabola, but the maximum and minimum values vary. We can calculate how they behave. If we fix $\beta = 0$ we get $\alpha = 1 - \gamma$ and $s_{\text{cos}}(X, Y) = \sqrt{\gamma}$. If we fix $\alpha = \beta = \frac{1}{2}(1 - \gamma)$ we get $s_{\text{cos}}(X, Y) = \frac{2\gamma}{1 + \gamma}$. We have not yet proven that these values are the minimum and maximum value. For this we determine the derivative to $\alpha$ with $\gamma$ fixed.

$$\frac{\partial}{\partial \alpha} \frac{\gamma}{\sqrt{(\alpha + \gamma)(\beta + \gamma)}} = \frac{\gamma(\alpha - \beta)}{2((\alpha + \gamma)(\beta + \gamma))^{3/2}}$$ (4.23)

Assume $\gamma > 0$, now the denominator is always positive, so the sign is determined by the numerator. The numerator is zero at $\alpha = \beta$ which must be an extreme value, the minimum to be exact. This means that the maximum is attained at the borders. See a comparison in Figure 4.2. The largest difference between the maximum and minimum value can be found by maximizing $\sqrt{\gamma} - \frac{2\gamma}{1 + \gamma}$, which results in finding the roots of a polynomial of degree 4. The approximate difference is 0.1349 at $\gamma = 0.0874$. The relative difference between the maximum and minimum is $\sqrt{\gamma}/(2\gamma/(1 + \gamma)) = (1 + \gamma)/(2 \sqrt{\gamma})$ which tends to infinity if $\gamma$ tends to zero. We are however interested in high values of $\gamma$.

Changing to the cosine similarity from the Jaccard similarity does not seem to be a problem for high values of the similarity. However, for low values of the similarity it can give problems. We are interested in creating high similarities, so this should not be a problem.

![Figure 4.2](image)

**Figure 4.2:** Comparison between the minimum and maximum value of the cosine similarity for different values of $\gamma$, i.e. the Jaccard similarity. The red line is the maximum and the green line the minimum. So all values must lie in the gray area.

For usability at ASML it is useful if the values found by the cosine similarity can be converted
to similar values for the Jaccard similarity. The reason for this is that a user is used to certain values, e.g. 0.8, as a threshold for useful Jaccard similarity. Therefore we would like a function $g$ that roughly converts one in the other like so: $g(s_{\cos}(X, Y)) \approx s_{J}(X, Y)$. From Figure 4.2 we see that we need to consider at least two parameters to create an exact conversion, but we have given an upper and lower bound based on only one, namely $\frac{2s_{J}(X, Y)}{1+s_{J}(X, Y)} \leq s_{\cos}(X, Y) \leq \sqrt{s_{J}(X, Y)}$. For high values of $s_{\cos}$ these bounds are very tight.

### 4.3 Centroids

Take $C \subset \mathbb{F}$. Let $1_C$ denote the incidence vector of $C$ with for each position 1 if that fault is in $C$ and 0 otherwise. We will write $1$ if $C = \mathbb{F}$, we will call this the 1-vector. We can determine an average fault of $C$, this is not possible in set representation, but we can represent it in vector notation.

**Definition 4.10.** We define the average fault or centroid $c$ of $C$:

$$
c := \frac{1}{|C|} \sum_{\hat{f} \in C} \hat{f}
$$

$$
= \frac{1}{|C|} 1_C S 1_C^T
$$

The average fault is simply determined by calculating the average for each symptom over all faults in $C$. Which is intuitive and using the incidence vector we can express this with a matrix multiplication as well.

Now we want to determine the average similarity of the faults in $C$.

$$
\text{avg sim}(C) := \frac{1}{|C|^2} \sum_{F_1, F_2 \in C} s_{\cos}(F_1, F_2)
$$

$$
= \frac{1}{|C|^2} 1_C S 1_C^T
$$

$$
= \frac{1}{|C|^2} 1_C R R^T 1_C^T
$$

$$
= \frac{1}{|C|^2} \|1_C R\|^2
$$

$$
= \|1_C\|^2
$$

So calculating the average similarity is the same as determining the average fault and calculating the inproduct with itself. It makes sense to separate $F$ in several clusters. Determining the average fault of such a cluster could represent the cluster, that is why it is often called a centroid.
One of the downsides of this definition of a centroid is that it is not normalized. So calculating the cosine similarity is not simply the inproduct. Therefore we also use the normalized centroid \( \hat{c} = c/\|c\| \). Now we are also interested in the average similarity of a cluster with its normalized centroid.

\[
\text{avg sim}(C, c) := \frac{1}{|C|} \sum_{F \in C} s_{\cos}(F, c) \\
= \frac{1}{|C|} 1_c R \cdot \hat{c} \\
= \frac{1}{|C|} 1_c R \cdot \left( R^T 1_c^T / (|C|\|c\|) \right) \\
= \frac{1}{|C|^2} \|1_c R\|^2 / \|c\| \\
= \|c\| \\
\] (4.32)

(4.33)

(4.34)

(4.35)

(4.36)

This means that \((\text{avg sim}(C))^2 = \text{avg sim}(C, c)\). So we can explicitly determine one from the other. We will use the average similarity with a centroid more for the benchmark we will see in the next section. We will call \(\|c_i\|\) the internal similarity of cluster \(C_i\).

We can see the centroid and normalized centroid as a cluster representative. Downside of these representatives is that they often have no real fault with a time stamp. Therefore we define a third cluster representative.

**Definition 4.11.** We call \(f\) a medoid of cluster \(C\) if

\[
f = \arg \max_{f' \in C} s_{\cos}(f', c),
\] (4.37)

with \(c\) the centroid of \(C\).

A medoid does not need to be unique, which is a downside as a representative. However, it is a real fault. This means that it is easier to use this cluster representative in practice. Note that it does not matter if we take the normalized centroid or ordinary centroid of the cluster.

The way we defined centroids is a bit biased since in the sum we included the similarities of the faults with themselves in the definition. Since a fault always has similarity 1 with itself, this makes the average similarity biased to 1. We find the following relation

\[
\frac{1}{|C|(|C| - 1)} \sum_{F_1 \neq F_2} s_{\cos}(F_1, F_2) = \frac{1}{|C|(|C| - 1)} \left( \sum_{F_1, F_2} s_{\cos}(F_1, F_2) - |C| \right) \\
= \frac{|C|}{|C| - 1} \frac{1}{|C|^2} \sum_{F_1, F_2} s_{\cos}(F_1, F_2) - \frac{1}{|C| - 1} \\
= \frac{|C|}{|C| - 1} \|c\|^2 - \frac{1}{|C| - 1}
\] (4.38)
If the cluster \( C \) is large, this does not have a major impact since the number of faults is large. If the cluster is small, we already expect a high similarity, so the impact is still relatively small. Our initial definition is more elegant and easier to use, therefore we keep using the original definition.

### 4.4 Benchmark

We want to compare different clusterings to each other. In this section we define a benchmark which gives a score to each clustering. This does not necessarily mean it gives a useful clustering for ASML, but it provides some insight how useful a clustering is without consulting an expert of ASML.

**Definition 4.12.** The K-means problem is the problem of finding vectors \( a_1, \ldots, a_k \) with \( \|a_i\| = 1 \) and maximizing the following function

\[
K_{\text{MB}} = \sum_{i=1}^{|F|} \max_j (s_{\cos}(f_i, a_j)) \tag{4.42}
\]

We can see that this defines a clustering \( C = \{C_1, \ldots, C_k\}, \) where \( C_i = \{f : i = \arg \max_j (s_{\cos}(f, a_j)) \) or in words: each \( a_i \) corresponds to the cluster \( C_i \), for each fault \( f \) we find the maximum similarity with \( a_i \) and put it in the corresponding cluster.

Though we let \( a_j \) be any normalized vector, we can see that they must be normalized centroids as follows

\[
K_{\text{MB}} = \sum_{C_i \in C} \sum_{f \in C_i} a_i \cdot f \tag{4.43}
\]

\[
= \sum_{C_i \in C} a_i \cdot \sum_{f \in C_i} f \tag{4.44}
\]

\[
= \sum_{C_i \in C} a_i \cdot |C_i| \tag{4.45}
\]

This is maximized if \( a_i = \hat{c}_i \) with \( c_i \) the centroid of \( C_i \).

Now we can give an alternate definition for the K-Means problem. Find a clustering \( C \) with cluster \( C_i \in C \) and corresponding centroid \( c_i \) such that the following is maximized

\[
K_{\text{MB}} := \sum_i \left( \sum_{f \in C_i} s_{\cos}(f, c_i) \right) \tag{4.46}
\]

We call \( K_{\text{MB}} \) the K-Means Benchmark and \( K_{\text{MB}}(k) \) the K-Means Benchmark for \( k \) clusters.

**Definition 4.13.** Take \( C \) a clustering with \( k = |C| \). For any \( C_i \in C \), we will write \( 1_i := 1_{C_i} \) for short. We define the matrix representation \( C \) of the clustering \( C \) as

\[
C = \begin{pmatrix}
1_i & \ldots & 1_k
\end{pmatrix} \tag{4.47}
\]
CHAPTER 4

Proposition 4.14. 1. \( \mathbf{1}_i \mathbf{1}_j^T = |C_i| \) and \( \mathbf{1}_i \mathbf{1}_i^T = 0 \) if \( i \neq j \).

2. \( \mathbf{C}^T = \text{diag}(|C_1|, \ldots, |C_k|) \)

3. A binary \( C \) for which condition 2. holds gives a well defined clustering.

Proof. 1. and 2. follow directly from the definition. The third statement can only be a clustering if each row has at most one 1-entry. This must be the case, otherwise \( \mathbf{C}^T \) would not be a diagonal matrix. □

Proposition 4.15. The K-means clustering problem for fixed \( k \) is equivalent to finding \( \mathbf{1}_1, \ldots, \mathbf{1}_k \) such that \( \mathbf{1}_i \cdot \mathbf{1}_j = \delta_{i,j} \) and \( \sum_i \| \mathbf{1}_i \|/\|c_i\|^2 \) is maximized, where \( \delta_{i,j} \) is the Kronecker delta.

Proof. \( \mathbf{1}_i \cdot \mathbf{1}_j = \delta_{i,j} \) is sufficient to define clusters as was shown in Proposition 4.14

\[
\begin{align*}
\text{KMB} & := \sum_i \left( \sum_{f \in C_i} \hat{f} \cdot \hat{c}_i \right) \tag{4.48} \\
& = \sum_i \left( \mathbf{1}_i \mathbf{R} \cdot c_i^T / \|c_i\| \right) \tag{4.49} \\
& = \sum_i \left( \mathbf{1}_i \mathbf{R} \cdot \left( \frac{1}{|C_i|} \mathbf{1}_i \mathbf{R} \right)^T / \|c_i\| \right) \tag{4.50} \\
& = \sum_i |C_i|/\|c_i\| \tag{4.51} \\
& = \sum_i \| \mathbf{1}_i \|^2 /\|c_i\| \tag{4.52} 
\end{align*}
\]

KMB is not perfectly suited for the type of clustering ASML wants to find. They are more interested in finding clusters that have high average similarity and are interested in finding clusters that are large but still have high average similarity. KMB promotes to have a clustering that has clusters with more or less the same size. For example, KMB would rather split up a cluster of size 1000 with average similarity of 0.8 to get two clusters of size 500 with average similarity of 0.85 than to split up a cluster of 100 with average similarity of 0.4 to get two clusters of size 50 with average similarity 0.8. ASML considers the cluster of size 100 rather bad because the similarity is low, while it wants to find big clusters of 0.8 similarity or higher.

We have not been able to find a better benchmark as other benchmarks also have their issues. Quite a few have computational problems because they require pairwise similarities between (normalized) centroids, which became troublesome from an implementation perspective in R.[3][4] Others have too many requirements to use them as benchmark for SA and K-Means.

In literature the K-Means problem is often defined as taking the square of the distance between faults and the centroids. We have defined our problem slightly different because it is hard to calculate a distance for all faults. We can define the K-Means squared distance problem (KMSDP) as follows
We try to minimize KMSDP, which means it is almost the same as KMB, but squares the norm of the centroid. The reason we use a different benchmark is that the reasoning we gave why KMB was not perfect is even worse for this benchmark. The K-Means squared distance problem is NP-complete for \( k \geq 2 \). There is a simple heuristic method called K-Means that tries to solve the K-Means problem, but in general it gets stuck in local optima, see Section 6.2.
Singular value decomposition and Cheeger’s inequality

In this chapter we will lay down the important mathematical theory on which the following chapters are built. First we will introduce singular value decomposition which relates eigenvectors to maximizing and minimizing. Then we will introduce how to compute these eigenvectors numerically. We will introduce the Laplacian and normalized Laplacian of which the latter can provide a tight bound on the conductance measure. Lastly we will show how to compute these vectors faster.

As a convention we will write $f_i$ for the normalized incidence vector of $F_i$, also we will write $f_i \in C$ instead of the $f_i$ that represents $F_i \in C$.

### 5.1 Singular value decomposition

Most of the parts of this subsection are from [5].

**Definition 5.1.** A set of vectors $\{v_1, \ldots, v_n\}$ are called orthonormal if $\forall i, j : v_i \cdot v_j = \delta_{i,j}$, where $\delta_{i,j}$ is the Kronecker delta. This is the same as saying that each vector has norm 1 and all vectors are orthogonal to each other.

The matrix $V = \begin{pmatrix} v_1 & \cdots & v_n \end{pmatrix}$ is called orthonormal.

**Proposition 5.2.** Take $V$ orthonormal matrix of size $m \times n$. We will use $I_k$ to denote the identity matrix of size $k \times k$.

1. $V^TV = I_m$
2. $VV^T = I_n$
3. $V^T = V^{-1}$ if $V$ square

□
**Definition 5.3.** Take \( m \times n \) matrix \( A \). \( \sigma \) is called a singular value of \( A \) with corresponding singular vectors \( u \) and \( v \) if the following two equations hold.

1. \( Av = \sigma u \)
2. \( u^T A = \sigma v^T \)

Note that this definition is related to the normal definition of eigenvalues and eigenvectors. Normally an eigenvector \( v \) has the following relation \( Av = \sigma v \). This is different from our definition of singular vectors. \( u = v \) implies \( v^T A = \sigma v^T \) which is not necessarily true, especially since \( u \) and \( v \) can have different dimensions. They are related in the following way.

**Proposition 5.4.** Let \( v \) denote a right singular vector of \( A \) with \( \sigma \) the corresponding singular value. Then \( v \) is an eigenvector of \( A^T A \) with \( \sigma^2 \) the corresponding eigenvalue.

**Proof.**

\[
A^T A v = A^T \sigma u = \sigma (u^T A)^T = \sigma^2 v \tag{5.1}
\]

The converse is also true. For a proof, see [5]. From eigenvector \( v \) we can find singular vector \( u = A v / \sigma \).

This theorem gives some nice properties. If we want to calculate the eigenvectors of similarity matrix \( S \), we can also calculate the singular vectors of \( R \). Since \( R \) is smaller and more sparse than \( S \), this can be done a lot faster. Actually if we look at the problem at ASML, it is hard to even calculate \( S \), let alone determine its (top few) eigenvectors. But by using Proposition 5.4 we are able to determine the (top few) eigenvectors in a short amount of time. We exploit this in Section 5.4.

**Definition 5.5.** For a \( m \times n \) matrix \( A \) we define the following two norms.

1. The Euclidean norm:

\[
\|A\| := \max_{\|v\| \neq 0} \|Av\| = \max_{\|v\| = 1} \|Av\| \tag{5.2}
\]

2. The Frobenius norm:

\[
\|A\|_F := \sqrt{\sum_{i,j} (A)_{i,j}^2} \tag{5.3}
\]

We will now introduce the distance between a line and a point which we will use to prove some properties. We will see that this is related to minimizing the distance between points and a line that we normally see if we want to minimize the sum of squares and related methods.

We find the distance between a vector \( f \) and the line \( \lambda v \) with \( v \) a vector and \( \lambda \in \mathbb{R} \).

\[
d(f, v) := \| f^T - (f^T v) v^T \| \tag{5.4}
\]

If \( v \) is a unit vector then we can apply the Pythagorean Theorem and we get
\[ ||f||^2 = ||f^T \nu \nu^T|| + d(f, \nu)^2 \] (5.5)

Now we find for any matrix \( A \)

\[
\|A\|^2 = \max_{\|\nu\|=1} \|A\nu\|^2 \\
= \max_{\|\nu\|=1} \sum_i \|f_i^T \nu \nu^T\|^2 \\
= \max_{\|\nu\|=1} \sum_i (||f_i||^2 - d(f_i, \nu)^2) \\
= \|A\|^2_F - \min_{\|\nu\|=1} \sum_i d(f_i, \nu)^2 \\
= \|A\|^2_F - \min_{\|\nu\|=1} \sum_i \|f^T - (f^T \nu)^T\|^2 \\
= \|A\|^2_F - \min_{\|\nu\|=1} \sum_i \|f^T - (f^T \nu)^T\|^2 \] (5.11)

This means that if we’re interested in finding \( \nu \) that maximizes \( \|A\nu\| \), we can also minimize over the squared sum of distances to a line. Which is the same as a least squares method.

**Theorem 5.6.** For a matrix \( A \) we define the following vectors

\[
u_1 = \arg \max_{\|\nu\|=1} \|A\nu\| \\
\nu_2 = \arg \max_{\|\nu\|=1, \nu \cdot \nu_1 = 0} \|A\nu\| \\
\vdots \\
\nu_k = \arg \max_{\|\nu\|=1, \nu \cdot \nu_1 = 0, \ldots, \nu \cdot \nu_{k-1} = 0} \|A\nu\| \] (5.12)

If there are multiple possibilities for an arg max it is broken arbitrarily.

These vectors are the right singular vectors of \( A \). We find singular value \( \sigma_i = \|A\nu_i\|/\|\nu_i\| \) and left singular vector \( u_i = A\nu_i/\sigma_i \). Also, \( \sigma_1 \geq \cdots \geq \sigma_k \), all \( u_i \) are orthogonal to each other and all \( \nu_i \) are orthogonal to each other.

We can write \( A = \sum_{i=1}^n \sigma_i u_i \nu_i^T \)

Now we define the space \( V_k = \text{span}(\nu_1, \ldots, \nu_k) \). Then

\[ V_k = \arg \min_{\dim(V)=k} \sum_i d(A_{(i)}, V)^2, \] (5.13)

where \( A_{(i)} \) denotes the \( i \)th row of \( A \).

\[ A_k := \sum_{i=1}^k \sigma_i u_i \nu_i^T \] (5.14)

\[ = \arg \min_{\text{rank}(D)=k} \|A - D\|^2_F \] (5.15)
A proof can be found in [5]. □

Note that this theorem states that the best least squares approximation of the matrix $A$ by a matrix of rank $k$ is by using the top $k$ singular vectors. What’s even more beautiful is that allowing a rank $l > k$ still uses the same top $k$ eigenvectors found in the rank $k$ approximation.

5.2 SVD approximation algorithm

Determining eigenvectors in practice is often done using numerical analysis. An approximation to the eigenvectors is enough and determining an approximation is quick. An easy numerical method to calculate the top eigenvector is using the power method. [6]

Algorithm 5.1: Power Method

```
function Power Method(Symmetric matrix $A$)
    Take random vector $v$.
    repeat
        $v \leftarrow \frac{Av}{\|Av\|}$
    until Some stopping criterion is met, i.e. $v$ is accurate enough
```

The power method described in Algorithm 5.1 determines the top eigenvector of a symmetric matrix. If $A$ is not symmetric we can use $A^T A$ instead. Note that if $A$ is sparse it is faster to calculate $v \leftarrow \frac{A^T (Av)}{\|A^T (Av)\|}$. We can see that this method indeed converges to the top eigenvector if we consider the singular value decomposition of a matrix. We can write $v$ as a combination of the eigenspaces on which $A$ projects. The eigenspace with the largest eigenvalue will eventually have the most impact if we multiply $v$ multiple times with $A$. If we want to find the second largest eigenvector, we simply use the same method, but use Gram-Schmidt to let $v$ stay in the space orthogonal to the top eigenvector. We can apply this iteratively to find other eigenvectors as well.

There are several other methods to approximate eigenvectors and most of them are faster and more accurate than the power method. We have not focused on finding the best one but used a library in R called “irlba” which applies an “implicitly-restarted Lanczos bidiagonalization”. [7] The reason we used this package is mainly because it can handle sparse matrices well and is able to quickly calculate the top few eigenvectors without spending time on finding the other eigenvectors.

The speed of most of these type of algorithms depend on the matrix you provide. The method applies a few matrix multiplications with a single vector iteratively, similar to the power method. The speed of the algorithm mainly depends on the matrix multiplications with this vector, thus if we have a sparse matrix these operations are faster.

5.3 Laplacian and normalized Laplacian

Most of the parts of this subsection are from [8].

The similarity matrix $S$ (see Equation (4.12)) is an interesting matrix, but there are some other matrices that can be derived from $S$ and which have interesting properties due to graph
theory. These are the Laplacian $L$ and the normalized Laplacian $N$. We will discuss them here and explain why they are interesting.

**Definition 5.7.** A graph $G = (V, E)$ consists of a set of vertices $V$ and a set of edges $E$. An edge is a pair of two vertices. One can view a graph as a collection of points of which some are connected to each other.

A weighted graph $G = (V, E, w)$ is a graph with a weight $w(e) \in \mathbb{R}, \forall e \in E$. A normal graph can be seen as a weighted graph with $w(e) = 1, \forall e \in E$.

**Definition 5.8.** The adjacency matrix $A_G$ of graph $G$ is defined as:

$$A_G(u, v) = \begin{cases} w(e) & \text{if } (u, v) = e \in E \\ 0 & \text{if } (u, v) = e \notin E \end{cases}$$

(5.16)

If we set $V = \mathbb{F}, E = \mathbb{F} \times \mathbb{F}$ and $w(f_1, f_2) = s_{\cos}(f_1, f_2)$ for all $(f_1, f_2) \in \mathbb{F} \times \mathbb{F}$ we have created a graph of our faults with similarity as weight between all faults. If the weight is zero, we can also say that there is no edge between the two faults. We can choose to remove the edges $(f, f)$ since they will always have weight 1. In graph theory one often works with graphs that do not have these loops. This does not matter for the Laplacian or the theory we use for the normalized Laplacian so we include the loops as this will make work easier.

**Definition 5.9.** We define the vector $d$

$$(d)_i = \sum_{j'} A_G(f_i, f')$$

(5.17)

where $(d)_i$ denotes the $i^{th}$ entry in vector $d$.

Thus $d$ is simply the row sum or column sum of $A_G = S$. We can also view it as the sum of weights connected to each vertex.

Now we define

$$D = \text{diag}(d)$$

(5.18)

$$L_G = D - A_G = D - S$$

(5.19)

We call $L_G$ the Laplacian of $G$. We will write $L$ if it is clear what graph we are talking about.

We can use the Laplacian matrix to write the Laplacian quadratic form:

$$x^T L x = \sum_{i,j} w(f_i, f_j)((x)_i - (x)_j)^2$$

(5.20)

Let $w_{i,j} = w(f_i, f_j)$.

To see why this equation hold we write the Laplacian matrix as follows

$$L = \sum_{i \neq j} A_{i,j} = \sum_{i \neq j} \begin{pmatrix} \cdots & \cdots & \cdots & \cdots \\ \cdots & w_{i,j} & -w_{i,j} & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & -w_{i,j} & w_{i,j} & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{pmatrix}$$

(5.21)
This matrix $A_{i,j}$ has only 4 non-zero entries and those are at the $i^{th}$ and $j^{th}$ position in both the columns and rows. Now

$$x^T L x = x^T \left( \sum_{i \neq j} A_{i,j} \right) x$$  \hfill (5.22)

$$= \sum_{i \neq j} (x^T A_{i,j} x)$$  \hfill (5.23)

$$= \sum_{i \neq j} (x_i^2 w_{i,j} - 2(x_i)(x_j) w_{i,j} + (x_j)^2 w_{i,j})$$  \hfill (5.24)

$$= \sum_{i \neq j} w(f_i, f_j)((x)_i - (x)_j)^2$$  \hfill (5.25)

$$= \sum_{i,j} w(f_i, f_j)((x)_i - (x)_j)^2$$  \hfill (5.26)

**Definition 5.10.** The Rayleigh coefficient of vector $x$ with the matrix $A$ is

$$\frac{x^T A x}{x^T x}$$  \hfill (5.27)

**Lemma 5.11.** The maximum value of the Rayleigh coefficient is the largest eigenvalue of symmetric matrix $A$, furthermore this is attained if $x$ is the corresponding eigenvector.

**Proof.** We use singular value decomposition to write $A = \sum_i \sigma_i u_i v_i^T = \sum_i \sigma_i u_i v_i^T$. Since $A$ is symmetric we have that $u_i = v_i$. Also we require $\|x\| = 1$. Now the Rayleigh coefficient becomes

$$\frac{x^T A x}{x^T x} = \frac{x^T \sum_i \sigma_i u_i v_i^T x}{x^T x}$$  \hfill (5.28)

$$= \sum_i \sigma_i (u_i^T x)^2$$  \hfill (5.29)

From Theorem 5.6 we now get that this will be maximized if $x = u_1$. \hfill \Box

Minimizing the value of the Rayleigh coefficient is analogue and we find that the smallest eigenvalue and corresponding eigenvector minimize the Rayleigh coefficient. Similarly we can proof that if we require $x$ orthogonal to the eigenvectors corresponding to the $k$ smallest eigenvalues, minimizing the Rayleigh coefficient will be equal to the $k + 1$ smallest eigenvalue with $x$ the corresponding eigenvector.

**Definition 5.12.** For a graph $G = (V, E)$ we define the boundary of cluster $C \subset V$ as

$$\partial(C) := \{(f_1, f_2) \in E| f_1 \in C, f_2 \notin C\}$$  \hfill (5.30)

For an unweighted graph $G = (V, E)$ the isoperimetric ratio of $C \subset V$ is defined as

$$\theta(C) = \frac{|\partial(C)|}{|C|}$$  \hfill (5.31)
The isoperimetric ratio of the graph $G$ is defined as

$$\theta_G = \min_{C: |C| \leq |E|/2} \theta(C)$$  \hfill (5.32)

Let $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ denote the eigenvalues of the Laplacian with $1 = z_1, z_2, \ldots, z_n$ the corresponding eigenvectors. Note that we order the eigenvalues from smallest to largest. The historic reason for this is that often $k$-regular graphs were researched for which the eigenvalues of the Laplacian are the same $\lambda_i$, thus reversed.

**Theorem 5.13.** For any $C \subset V$ in $G = (V, E)$

$$\theta(C) \geq \lambda_2(1 - \rho),$$  \hfill (5.33)

where $\rho = |C|/|V|$.  

**Proof.** Let $L$ be the Laplacian corresponding to $G$. From the section below Lemma 5.11 we get

$$\min_{x \in \mathbb{R}^n : x \cdot z_1 = 0} x^T L x = \lambda_2$$  \hfill (5.34)

So $x^T L x \geq \lambda_2 \|x\|^2$ for any $x$ orthogonal to $1$. We take $x = 1_C - \rho 1$. Which is orthogonal to $1$ since

$$1^T \cdot x = 1^T (1_C - \rho 1) = |C| - |V| \rho = 0$$  \hfill (5.35)

Using this $x$ we find

$$x^T L x = \sum_{i,j} w(f_i, f_j) ((x)_i - (x)_j)^2$$  \hfill (5.36)

$$= \sum_{i,j} w(f_i, f_j) ((1_C - \rho 1)_i - (1_C - \rho 1)_j)^2$$  \hfill (5.37)

$$= \sum_{i,j} w(f_i, f_j) (1_C)_i - (1_C)_j)^2$$  \hfill (5.38)

$$= \sum_{e \in \partial(C)} w(e)$$  \hfill (5.39)

Now

$$x^T x = (1 - \rho 1^T)(1_C - \rho 1) = |C| - 2\rho |C| + \rho^2 |V| = |C| - 2\rho |C| + \rho |C| = (1 - \rho) |C|$$  \hfill (5.40)

We find $\theta(C) \geq \lambda_2(1 - \rho)$. \hfill (5.41)

We define $\delta(C) = \sum_{e \in \partial(C)} w(e)$. We see $\delta(C) = |\partial(C)|$ for the unweighted case. From this theorem we can deduce

$$\frac{|\partial(C)|}{|C|} \geq \lambda_2(1 - \rho)$$  \hfill (5.42)

$$\frac{|\partial(C)|}{|C|} \geq \lambda_2(1 - \frac{|C|}{|V|})$$  \hfill (5.43)

$$\frac{|\partial(C)|}{|C||V - C|} \geq \lambda_2 \frac{|C|}{|V|}$$  \hfill (5.44)
We can view the left-hand side as the following ratio: the amount of edges between $C$ and its complement divided by the total possible edges between these two clusters. Note that the right-hand side is independent of $C$. We can deduce that for the weighted case we should use $\sum_{e \in \partial(C)} w(e)$ instead of $|\partial(C)|$. So now the ratio becomes

$$\frac{\delta(C)}{|C||V - C|} \quad (5.45)$$

This seems as an interesting ratio for the weighted graph. Since in practice we will use a weighted graph with an edge for each vertex pair. In that case this ratio becomes the average weight of all edges between the cluster $C$ and its complement. This seems like a logical ratio to minimize if we want to split $V$ into clusters with high internal similarity. We were not able to find a definition of the isoperimetric ratio for a weighted graph in literature. However we have used the ratio in Equation (5.45) in one of our algorithms as a trial, see Section 6.3.3. We will introduce another ratio which does have a weighted variant and is actually more interesting than the isoperimetric ratio.

**Definition 5.14.** We define

$$a(C) = \sum_{u \in V} \sum_{v \in C} w(u, v) \quad (5.46)$$

Now we define the conductance of a cluster

$$\phi(C) = \frac{\delta(C)}{\min(a(C), a(V - C))} \quad (5.47)$$

and the conductance of the graph $G$ as

$$\phi_G = \min_{C \subset V} \phi(C) \quad (5.48)$$

The conductance can be seen as the ratio of the sum of all weights of the outgoing edges of a cluster, divided by the sum of all edges connected to faults in that cluster. Since we split a cluster in two we would get two different ratio’s, therefore we take the largest ratio. In the unweighted case we can view the conductance as the amount of edges on the border divided by the total amount of edges connected to a cluster. So if the conductance is low, this means that there are little or low weighted edges on the border of a cluster, but there are a lot of edges or high weighted edges inside the cluster. This is something we try to achieve by clustering, thus if we can split a cluster in two with a low conductance we will probably improve our clustering. Not that this ratio looks similar to the isoperimetric ratio, but we can find a tight bound for it as we will show below.

**Definition 5.15.** We define the normalized Laplacian $N$

$$N = D^{-\frac{1}{2}}LD^{-\frac{1}{2}}, \quad (5.49)$$

where $D^{-\frac{1}{2}} = \text{diag}(d^{-\frac{1}{2}})$ and $d^{-\frac{1}{2}}$ is taken by taking the square root element wise, also zero entries in $d$ remain 0.
From this definition follows
\[ N = I - D^{-\frac{1}{2}}A_GD^{-\frac{1}{2}}. \] (5.50)

We define \( v_1 \leq \cdots \leq v_n \) the eigenvalues of \( N \). Now \( v_1 = 0 \) with \( d^{\frac{1}{2}} \) the corresponding eigenvector, since
\[ Nd^{\frac{1}{2}} = D^{-\frac{1}{2}}LD^{-\frac{1}{2}}d^{\frac{1}{2}} = D^{\frac{1}{2}}L1 = 01 = 0d^{\frac{1}{2}}. \] (5.51)

Furthermore, if we substitute \( x = D^{\frac{1}{2}}y \) we find for the Rayleigh coefficient of \( N \):
\[ \frac{x^\top Nx}{x^\top x} = \frac{y^\top Ly}{y^\top Dy}. \] (5.52)

We can prove a similar inequality as for the isoperimetric ratio.

**Theorem 5.16.** For any cluster \( C \)
\[ \phi(C) \geq \frac{v_2}{2} \] (5.53)

For a proof of the unweighted case see [8] for a proof of the weighted version see [9].

The following theorem gives an inequality in the other direction, it also gives the reason why \( v_2 \) and its eigenvector are very interesting and why some of our algorithms in the following sections actually work.

**Theorem 5.17.** (Cheeger’s inequality) Let \( y \) be an eigenvector orthogonal to \( d \). Then there is a value \( \tau \) for which \( C = \{ f_i \in V | (y)_i < \tau \} \) satisfies
\[ \phi(C) \leq \sqrt{2 \frac{y^\top Ly}{y^\top Dy}}, \] (5.54)

where \((y)_i\) denotes the \( i \)th element of vector \( y \).

Again, for a proof of the unweighted case see [8] for a proof of the weighted version see [9].

**Corollary 5.18.** Let \( x \) the eigenvector corresponding to \( v_2 \) and \( y = D^{-\frac{1}{2}}x \). Then there is a value \( \tau \) for which \( C = \{ f_i \in V | (y)_i < \tau \} \) satisfies
\[ \phi_C \leq \sqrt{2v_2} \] (5.55)

**Proof.** Since \( \frac{y^\top Ly}{y^\top Dy} = \frac{x^\top Nx}{x^\top x} \) for \( x = D^{\frac{1}{2}}y \) we see that \( v_2 \) is the value that minimizes Rayleigh coefficient of \( N \) if we require that \( x \) is orthogonal to \( d^{\frac{1}{2}} \). This means that \( y \) is orthogonal to \( D^{\frac{1}{2}}d^{\frac{1}{2}} = d \). Thus the vector \( y = D^{-\frac{1}{2}}x \) with \( x \) the eigenvector corresponding to \( v_2 \) minimizes \( \frac{y^\top Ly}{y^\top Dy} \). Now we apply Theorem 5.17. \( \square \)

This corollary gives us a constructive method to find a cut with a (tight) bound on the conductance! We only need to determine the value of \( \tau \) and we find two sets that have a tight bound on the conductance. Actually we are not even interested in finding \( \tau \). We can just sort \( y \), use the order of this vector and try all cuts based on this order.

Since the second smallest eigenvector of \( N \) is so heavily related to finding this cut it has its own name: the Fiedler vector. After the important work of Fiedler in this field.[10]
5.4 Computing second eigenvectors faster

In the previous section we have shown that determining the eigenvector corresponding to the second smallest eigenvalue of the normalized laplacian $N$ is interesting. In Section 5.2 we have discussed how we can effectively determine the top few eigenvectors of a matrix $A$, especially if $A$ is sparse. We will now show how we can determine the Fiedler vector efficiently, but also how we would determine the top eigenvectors of similarity matrix $S$ as defined in Equation (4.12). The results shown here are our own work, but we have found similar results in [11].

**Theorem 5.19.** Let $v$ denote a right singular vector of the cosine incidence matrix $R^\top$ with $\sigma$ the corresponding singular value. Then $v$ is an eigenvector of cosine similarity matrix $S$ with $\sigma^2$ the corresponding eigenvalue.

**Proof.**

$$Sv = RR^\top v = R\sigma u = \sigma^2 v \quad (5.56)$$

This means that if we determine an approximation of the SVD as in Section 5.2 of the matrix $R$, we also find the eigenvectors of $S$.

We can do something similar for $N$. We define

$$T^\top = R^\top D^{-\frac{1}{2}} \quad (5.57)$$

**Theorem 5.20.** Let $v$ denote a right singular vector of $T^\top$ with $\nu$ the corresponding singular value. Then $v$ is an eigenvector of the normalized Laplacian $N$ with $1 - \nu^2$ the corresponding eigenvalue.

**Proof.**

$$Nu = (I - D^{-\frac{1}{2}}SD^{-\frac{1}{2}})v \quad (5.58)$$
$$= (I - D^{-\frac{1}{2}}RR^\top D^{-\frac{1}{2}})v \quad (5.59)$$
$$= (I - (R^\top D^{-\frac{1}{2}})^\top (R^\top D^{-\frac{1}{2}}))v \quad (5.60)$$
$$= (I - TT^\top) v \quad (5.61)$$

This implies that $1 - \nu^2$ is the eigenvalue with $v$ the corresponding eigenvector. □

**Corollary 5.21.** The (absolute) smallest eigenvalues of $N$ are the (absolute) largest eigenvalues of $R^\top D^{-\frac{1}{2}}$.

**Proof.** From **Proof 12** we see that $I - (R^\top D^{-\frac{1}{2}})^\top (R^\top D^{-\frac{1}{2}})$ reverses the order of the eigenvalues, but this does not necessarily imply that the absolute values of the eigenvalues have this property as well.

$L$ is semi-positive definite since each entry on the diagonal is at least as large as the sum of the entries on the corresponding row.

$$xNx^\top = xD^{-\frac{1}{2}}LD^{-\frac{1}{2}}x^\top \quad (5.62)$$
Substituting \( y = xD^{-\frac{1}{2}} \) shows that \( N \) must also be semi-positive definite. This means that all eigenvalues of \( N \) are at least 0. Also \( (R^TD^{-\frac{1}{2}})^T(R^TD^{-\frac{1}{2}}) \) is positive definite, thus its eigenvalues are at least 0. This means all eigenvalues of \( N \) lie in \([0,1]\), which must also hold for the singular values of \( T \).

The reason why we want to know in what order the singular values of appear is because SVD approximation algorithms often give the eigenvalues and vectors in order of the absolute value of the eigenvalues, starting with the highest eigenvalue. Since we have shown that the order of singular values of \( T \) is reversed for \( T \) we know that the second largest singular value of \( T \) corresponds to the second smallest eigenvalue of \( N \).

If we want to determine \( T \) and use its sparsity benefits we also need to know \( D \). We now show how to compute \( D \) fast, again using sparsity of \( R \).

\[
\begin{align*}
\mathbf{d} & = S \mathbf{1} = R(R^T \mathbf{1}) \\
D & = \text{diag}(d)
\end{align*}
\]

So we can compute \( D \) efficiently using the sparsity of \( R \). Thus we can determine \( T \) efficiently and find the vector \( y = D^{-\frac{1}{2}}x \) with \( x \) the second smallest eigenvector of \( N \). Our SVD-based algorithms in the next chapter depend on finding this vector \( y \).

We will now do a running time analysis of determining the top eigenvectors using an approximation algorithm for singular value decomposition. The running time of the SVD approximation algorithm is based on numerical calculations of the top few eigenvectors, but this is all done by a sequence of matrix multiplications by either \( S \) or \( N \), which would take \( O(s|\mathbb{F}|^2) \) time, where \( s \) is the amount of iterations needed by the numerical SVD algorithm, which is small in general. We have shown that we can determine the eigenvectors of \( S \) and \( N \) using sparsity of either \( R \) or \( T \), which results in a running time of \( O(s|\mathbb{F}|\eta) \), where \( \eta \) is the average number of non-zero entries of a fault \( f_i \in \mathbb{F} \). Note that \( |\mathbb{F}|\eta \) is the total number of non-zero entries in \( R \) and \( T \).

In the case of ASML \(|\mathbb{F}| \approx 250,000 \) is roughly 10 times larger than \(|\mathbb{E}| \approx 20,000 \) and \( \eta \approx 25 \). The amount of memory used to save the complete matrix \( S \) would be \( O(|\mathbb{F}|^2) \approx 62,500,000,000 \) entries. \( R \) uses roughly 8 bytes for each entry, so to store this in \( R \) we would need roughly 0.5 TB. This is just to save this huge matrix, applying operations on it to cluster it become incredibly hard. However, saving the matrix \( R \) or \( T \) only uses roughly 6,250,000 entries, which uses roughly 50 MB. This means we need 10,000 times less space to store the matrix. The same reasoning holds if we want to determine the second largest eigenvector of \( S \) or the second smallest eigenvector of \( N \), the calculation speed increases by a factor 10,000. Note that generating the full matrix \( S \) or \( N \) would take \( O(|\mathbb{F}|^2\eta) \) time to generate, which is not good but would be acceptable. However storing it and efficiently accessing it is a larger issue. All these issues are resolved by using the sparse matrices \( R \) and \( T \).
Clustering algorithms

Clustering is a widely studied subject [12] and several clustering algorithms are available all of which have their own benefits. What separates our problem from most similar problems is twofold. The data we have has no predefined classification which we want to learn, therefore our problem is considered "unsupervised classification", often called clustering. The second difference is that our dataset is big, very high dimensional and binary. Unsupervised clustering is very different from supervised clustering since there is no feedback. One of the problems we encountered is that it is hard to argue why our clustering is good, we need an expert for that. Supervised clustering often uses a training dataset which has been classified already. This way the system can learn. We do not have this type of problem, which makes it considerably different. Our dataset is big, high dimensional and binary, which is something that occurs often in computer science related fields where a lot of data is gathered. For example Netflix or Amazon have these type of datasets, though they are examples of supervised classification since they get feedback from their users. Datasets coming for example from physics related problems often use continuous data and have considerably less dimensions, so are also very different. Lastly, the size of the dataset differs a lot. There are many smaller datasets to be found, for which classifying by hand becomes a possibility to check whether the algorithm works. The size of the dataset has a major influence on what type of algorithms can be used. If it is small enough we can determine matrices $S$, $L$ or $N$ explicitly, but for large datasets this becomes a major issue.

We will look at three types of clustering algorithms: The stacking algorithm is a simple algorithm currently used by ASML. K-Means is an heuristic algorithm that is widely used as it is easy to implement and often gives satisfying results. The last type of algorithm is based on singular value decomposition (SVD) using the second eigenvectors. This is the type of algorithm we have studied most and will be the main topic. However, we want to compare our algorithms with the current algorithm at ASML and since we use K-means for post-processing of the SVD algorithms (see Chapter 7) and K-Means is very widely used [12], we will discuss them here as well.
6.1 Stacking Algorithm

The stacking algorithm (SA) is currently by ASML to create their so called “Pareto Analysis”. For each $f_i$ they determine how many $f_j$ are identical. This can be done using a hash function in linear time. These are the so called “stacks” of identical faults. Next they determine the top $k$ of highest stacks (with the most identical faults). These will be the most important faults and we create $k$ clusters out of them for example by using a closest point method: For each $f_j$ find the stack that has the highest similarity with a fault of this stack and put $f_j$ in the corresponding cluster. Another possibility is to apply K-Means using the stacks as starting centroids. Both of these methods are post processing methods that are described in Section 7.4. ASML never created full clusters as they were mostly interested in cluster representatives which is any fault of a stack of the here found clusters. Since we want to compare several cluster methods we will assume that ASML took the closest points to the stacks, otherwise most faults would be marked outlier.

6.2 K-Means

K-Means is an algorithm that is widely used because it is very easy to understand and implement. It also gives relatively good results.[12] The algorithm starts with $k$ centroids. Then it determines the closest points to these centroids, which is a clustering. It then determines the centroids of the found clustering and repeats this process until a stopping criterion is met or the centroids no longer change, see Algorithm 6.2.

Algorithm 6.2: K-Means

\begin{algorithm}
\begin{algorithmic}[1]
\Function{K-MEANS}{k, centroids \{\hat{c}_1, \ldots, \hat{c}_k\}}
\Repeat
\For{$f \in F$}
\State Put $f$ in cluster $C_i$ with $\arg \max_i s_{\cos}(f, \hat{c}_i)$.\end{algorithmic}
\EndRepeat
\State Determine new normalized centroids $\hat{c}_i$ from the newly formed clusters.
\Until{Centroids no longer change or stopping criterion is met}
\EndFunction
\end{algorithm}


Proof. An iteration of K-Means consists of two steps, we will show both do not decrease KMB. The first step puts every fault in a cluster. Now we will iteratively consider step 2 and 1.

In the second step we determine new centroids for a cluster. As explained in Section 4.4 we know that the normalized vector with highest average similarity with faults in a cluster is the centroid. So taking the new centroid must give a higher KMB.

In the first step we allocate each fault to the cluster with the centroid it has highest similarity to. If we view KMB as $\sum_j \max_i s_{\cos}(c_i, f_j)$ and fix $c_i$, then this can only improve if a fault is in the cluster with the centroid it has highest similarity to. $\square$
The K-means algorithm is quite simple, but requires two things that determine its outcome. First of all, it needs \( k \), the amount of clusters, as an input parameter. So the amount of clusters in \( C \) is fixed. For the K-Means benchmark this is perfect. However in practice we often do not know how many clusters we want to find. Secondly, it needs some starting centroids to initialize. Different starting centroids can heavily influence the outcome, so choosing them is hard.

To choose starting centroids, one intuitive way is picking \( k \) faults at random as starting centroids. If all picked faults are non-identical, this will result in \( k \) clusters after the first iteration. During iterations of the algorithm it is possible that a cluster becomes empty if centroids get close together and some centroids do not find any closest points. Finding empty clusters does not break the K-Means problem, but the score can probably be improved if \( k \) non-empty clusters are found. K-means can be run multiple times with different starting centroids. We can select the best clustering based on any benchmark, in our case this will be KMB.

Downside of K-means is that it is a greedy algorithm and often gets stuck in local optima. Also, in the case of outliers, faults that have little similarity with others are still clustered and actually influence the clustering procedure.

Another downside mentioned before is that the starting centroid heavily influence the outcome. Also, if using identical faults as starting centroids we get empty clusters. One way to solve this is determining the unique faults and picking random faults out of those. But this might select a fault that has very little similarity with other faults. This fault might not find many other faults it has a good similarity with and because of the greedy nature of K-Means this probably results in some very small clusters. A better way to solve this is by taking any fault and check if an identical fault has been picked before. This way we use the distribution of faults, which results in starting centroids that are more likely to be close to several other faults.

The running time of K-Means is \( O(|F| \eta k \rho) \) where \( \eta \) is the average amount of symptoms a fault has and \( \rho \) is the average amount of iterations needed by K-Means. The analysis is relatively simple: Each iteration the slowest operation K-Means does is the matrix multiplication of \( R \) and the centroids which takes \( O(|F| \eta) \) time. The amount of iterations needed by K-Means vary for different starting centroids and \( k \). From experiments with real data we see that it often takes between 5 and 20 iterations. So it is relatively small. We often will do multiple iterations of K-Means which has a running time of \( O(|F| \eta k \rho \pi) \), where \( \pi \) is the amount of restarts.

### 6.3 SVD-based Clustering algorithms

Now we will introduce SVD-based algorithms, based on the theory of singular value decomposition and other theory of the previous chapter. First we will start with a rough idea of why this algorithm should work.

Singular value decomposition is trying to approximate a matrix. So for example if the SVD approximates our similarity matrix \( S \), the first eigenvector will roughly be responsible for approximating the average of the matrix. The second eigenvector will start showing the differences and because of the “sum of squares” nature of SVD, the second eigenvector will show the largest differences. Using this very rough analysis it makes sense to use SVD’s to split a large cluster in two or more smaller clusters. As shown in the previous section determining
the second smallest eigenvector of $N$ instead of $S$ actually gives us a tight bound. We will call the second largest eigenvector of $S$ the important vector of $S$ and we will call $D^{1/2} \phi_2$ with $\phi_2$ the second smallest eigenvector of $N$ the important vector of $N$.

We use this idea to cluster. We recursively split our clusters in two to create a clustering. In each iteration we first determine the important vector $v$ of either matrix $S$ or $N$ and we split the cluster $C$ in two by $C_1 = \{ f_i \in C : (v)_i \leq t \}$ and its complement. Determining the optimal cut would take $O(2^{|F|})$ time, thus quickly becomes infeasible as $|F|$ grows in size. Using just a single vector and a cutoff value $t$ makes the cutting problem 1-dimensional and solvable in near-linear time.

There are two dependencies we consider on which the complete algorithm will depend. The general algorithm will be the same. Firstly, we can choose different matrices $A$ to apply SVD on and secondly we can use different splitting criteria. One naive choice of a matrix $A$ is the similarity matrix $S$, but a smarter choices is the normalized Laplacian $N$ as shown in Section 5.3. The second dependency is the splitting criterion SC. We want to split a cluster in two or more clusters based on the important vector. We can divide the splitting criteria in two categories. The first one is solely based on the values found in the important vector. The second category scans through the order of the sorted important vector and determines an optimal cut based on the sorting of that eigenvector and some scoring function.

Since all clustering algorithms are based on the same splitting mechanism, they all have the same global structure, which is shown in Algorithm 6.3. The algorithm is recursive starting with $C = F$. The algorithm also needs a stopping criterion QC.

Algorithm 6.3: SVD-clustering

```plaintext
1 function SVD-clustering(C,A,SC,QC)
2  Find top eigenvectors $v_i$ using SVD of the part of $A$ corresponding to $C$
3  Determine new clustering $C$ of $C$ using SC by splitting $C$ in two
4  if QC is met then
5      return $C$
6  else
7      Determine for each $C_i \in C$: $C_i' \leftarrow$ SVD-clustering($C_i$,A,SC,QC)
8      return $\bigcup_i C_i'$

```

The stopping criterion QC can be based on several things. Some natural choices are the size clusters or the average similarity. This means that worst case the algorithm will continue until only clusters of size 1 will remain.

The worst case running time of this algorithm is $O(|F|)$ iterations, but we expect to cut the size of the clusterings roughly in half by each step. So the expected iterations is $O(\log |F|)$. The total running time then becomes $O(\log |F| \cdot (SVD + SC + QC))$ where we require that our splitting algorithm splits our cluster nicely and SVD, SC and QC are the running times of the SVD-algorithm, splitting criterion and stopping criterion respectively. Formally we have to require that it will split the cluster $C$ in sets of size at least $\gamma |C|$ where $\gamma$ is some (independent) constant larger than zero. We will mostly ignore this condition since we can alter our SC such that it will always return a clustering with the required sizes by using some threshold.
practice we will not even do this as we will check during the iterations of the algorithm if it progresses fast enough. The running times of SVD, SC and QC can depend linearly on the size of $C'$ of their current recursion step without increasing the running time of the complete algorithm. It cannot, however, depend on $|F|$ in each recursion step. We will show that the total running time is $O(|F| \log(|F|) \eta (\log |F| + \varsigma))$, where $\varsigma$ is the amount of iterations needed for the SVD approximation algorithm to converge.

### 6.3.1 Matrices

The running time of SVD is based on numerical calculations of the top few eigenvectors, but this is all done by a sequence of matrix multiplications by either $S$ or $N$, which take $O(\varsigma |C| \eta)$ time, where $\eta$ is the average number of non-zero entries of $f_i$ and $\varsigma$ is the amount of iterations needed by the numerical SVD algorithm, which is small in general. So the total running time becomes $O(|C| (\varsigma \eta + SC))$. As shown in Section 5.3 there is a bound if we use $N$, so we expect the best results from this matrix.

### 6.3.2 Value based splitting

We want to split a cluster $C$ in two by $C_1 = \{ f_i \in C : (u)_i \leq t \}$ and its complement, where $u$ is the important vector.

**Sign SVD (SiS)**

Since the important vector is showing differences between faults it makes sense to split just based on the sign, so we take $t = 0$. This is the first thing one might try and it appears to give some decent results already. It was also suggested by [13].

**Border SVD (BoS)**

This method is more an historic method, it was the first SVD-based method we came up with. At first it was not being applied recursively. It was simply based on the top few eigenvectors $x_i$ of $S$ and their value, not just the second. Plotting these values with their corresponding showed that some clustering already appeared (basically a principal component analysis). It appeared that a logic cut based on the main events was $t = \pm (u)_j > 1 / \sqrt{|C|}$. We assume $\|u\| = 1$. Basically if the value was “above average” it was clustered. Note that this splits our cluster in three subclusters instead of two, though often one cluster appears to be empty. Assuming that the eigenvectors are sorted in order of importance, one can iterate over the top eigenvectors and if a fault is not clustered, i.e. landed in the middle section, we try to cluster it by the next eigenvector. This appeared to work quite well for the main events, but didn’t work well if one would want to go deeper and create more clusters.

### 6.3.3 Scan based splitting

Scan based splitting is a more logical choice if one looks at Theorem 5.17. The value of $t$ is undetermined, so we try to determine it. The SVD approximation algorithm already takes
$O(|C|\eta_2)$ running time, so the value based splitting rule uses less time than the SVD algorithm does. It makes sense to use more time to analyze the produced eigenvectors.

We do this by scanning through faults ordered by the important vector. Using this order we determine an optimal splitting point. We use some scoring function $h$ that we want to optimize with $h(C_l, C_r) \rightarrow \alpha \in \mathbb{R}$, where $C_l$ and $C_r$ are sets of faults ($C_l$ represents the left side of the splitting point and $C_r$ represents the right side). $h$ can use more data than is in the important vector, since we only use the order of the important vector. If we use $N$, the most logical choice would be using the conductance as this is what Cheeger’s inequality gives a bound for. We generalize it a bit, since conductance is not what we will benchmark on. Also, what we do have to consider is that the running time of the scoring function needs to be fast, otherwise the scan will be slower than the SVD algorithm (which is not necessarily a problem, but if we are not careful this quickly becomes $O(|C||E|)$, while we can do it in roughly the same time as the SVD approximation algorithm as we will show below).

The scanning algorithm will roughly be as follows: We start with an empty set $C_l$ and $C_r = C$. We sort the important vector Using the order of the important vector we iteratively move one fault $f$ from $C_r$ to $C_l$. For each iteration we determine $||c_l||, ||c_r||$ and $c_l \cdot c_r$. We do not want to recompute these values in each iteration as that will take $O(|E|)$ time. Since $f$ has $n$ non-zero entries on average we can determine the values of these three quantities in $O(n)$ time for each iteration. Based on these quantities we want to determine an optimal cut. We let the optimal cut depend on a scoring function $h$. Let the scoring function $h$ be of the following form

$$h = h(c_l \cdot c_r, ||c_l||, ||c_r||, |C_l|, |C_r|).$$

We will later show some ways to define the scoring function in a meaningful way. A pseudocode implementation of this algorithm is shown in Algorithm 6.4.

**Algorithm 6.4:** Scanning algorithm. This algorithm uses the order of important vector $v$ to scan through all faults. It starts with an empty left cluster and a full right cluster. Each iteration the next fault in $v$ is moved from the right cluster to the left. For each cluster we keep track of the 5 parameters used for scoring function $h$.

```python
function SCANNING ALGORITHM(important vector $v$, $h$)
    bestScore \leftarrow \infty
    $f_l \leftarrow (0, \ldots, 0), f_r \leftarrow R1$
    $\alpha \leftarrow 0 \equiv ||f_l||^2, \beta \leftarrow ||f_r||^2, \gamma \leftarrow 0, n \leftarrow length \ v$
    for $i \in order(v)$ do
        $y \leftarrow y + f_i \cdot f_r^\top - f_i \cdot f_r^\top f_i - ||f_i||^2 = (f_i + f_l) \cdot (f_r - f_i)^\top$
        $\alpha \leftarrow \alpha + 2 f_i \cdot f_r^\top + ||f_i||^2 = ||f_i + f_l||^2$
        $\beta \leftarrow \beta - 2 f_i \cdot f_r^\top + ||f_i||^2 = ||f_r - f_i||^2$
        $f_i \leftarrow f_i + f_l, f_r \leftarrow f_r - f_i$
        newScore \leftarrow $h(y/(i(n-i)), \alpha/\beta, (n-i)^2, i, n-i)$
        if newScore > bestScore then
            bestScore \leftarrow newScore, bestI \leftarrow i
    return bestI
```
The values $\alpha, \beta, \gamma$ in Algorithm 6.4 are simply to calculate the input values of $h$, but it only depends on the new $f_i$ in each iteration. The only operations that are performed in each step are summation of or taking the inner product with the sparse vector $f_i$. We cannot use $f_l$ or $f_r$ directly, since these are in general not sparse. Each step in the for-loop takes on average $O(\eta)$ time, so the loop of the scanning algorithm takes $O(|C| \eta)$ time, which is slightly faster than the SVD algorithm. We also need to sort the eigenvector $v$, which takes $O(|C| \log |C|)$ time, so the running time of one recursion step is $O(|C| (\eta + \log |C|))$.

The total running time of the SVD-based algorithm can be deduced using the Master’s theorem [14], this becomes

$$O(|F| \log(|F|) \eta(\log |F| + \zeta)) \tag{6.2}$$

We now propose some meaningful functions for $h$.

**Minimal Conductance (MiCo)**

This method is based on Cheeger’s inequality and is the most logical choice if we use the important vector of $N$. We simply use the conductance as defined before and try to find the minimum conductance cut.

$$h_{\text{MiCo}} := \frac{(c_l \cdot c_r |C_l||C_r|)}{(\delta(C_l) + \min(\|c_l\|^2 |C_l|^2, \|c_r\|^2 |C_r|^2))} \tag{6.3}$$

$$= \frac{\delta(C_l)}{\min(a(C_l), a(C_r))} \tag{6.4}$$

$$= \phi(C_l) \tag{6.5}$$

**Minimal Isoperimetric Ratio (MIR)**

We have seen that the Laplacian $L$ and eigenvector $\lambda_2$ have a relation with the isoperimetric ratio. We can also cut based on the isoperimetric ratio, though this ratio does not have such a nice bound as we have seen for the conductance.

$$h_{\text{MIR}} := \frac{(c_l \cdot c_r |C_l||C_r|)}{(|C_l||C_r|)} \tag{6.7}$$

$$= \delta(C_r) / (|C_l||C_r|) \tag{6.8}$$

$$= \theta(C_r) \tag{6.9}$$

**Scan K-Means Optimized (SKO)**

Since we use a benchmark based on K-Means, it makes sense to use a scoring function that tries to find the cut that gives the best K-Means score. This does mean we apply a somewhat greedy algorithm based on the K-Means benchmark, but it might give better results for our specific benchmark.

$$h_{\text{SKO}} := -\|c_l||C_l| - \|c_r||C_r| \tag{6.10}$$
The reason why we want to minimize this function is the following. First we recall that 
\[ \text{KMB}(\mathcal{C}) = \sum_{C_i \in \mathcal{C}} ||c_i||C_i| \] with \( c_i \) the average fault of the cluster \( C_i \). If we have a clustering \( \mathcal{C} \) and we want to create one extra cluster, we split one of the clusters \( C_i \in \mathcal{C} \) in two smaller clusters \( C_l \) and \( C_r \), the improvement of KMB would be 
\[ |C_l|||f_l||^2 + |C_r|||f_r||^2 - |C_i|||f_i||^2. \] If we want to maximize this function, but fix \( C_i \), we see that it is the same as minimizing \( h_{SKO} \).

As long as \( -h_{SKO} > |C_i|||c_i|| \) it always increases the score of KMB.
Clustering framework

In Chapter 6 we’ve shown several clustering algorithms, but these can be extended to match the needs of ASML better. Especially the SVD-based clustering algorithms can use extra pre- and postprocessing to improve their performance. In Figure 7.1 we see the overview of the complete processing steps the data goes through for the different algorithms. Our focus is mainly on the SVD-based clustering algorithms, but Pareto and K-means are shown on the left to give a complete picture. We call this the clustering framework.

Our data first goes through some preprocessing to filter out some of the faults. Then we have to choose which algorithm to use, it can be either K-means, the original stacking algorithm (SA) or SVD clustering. Our main focus is on SVD clustering. SVD clustering is split in three parts. First a decision tree is built, this is the most important part of the algorithm and is based on Section 6.3. Then we prune the tree to make it better fit the type of clustering we want to find. Finally we apply some post processing which is mostly to make the final clustering better fit our benchmark. Note that SA also uses some postprocessing. This is because the clustering it finds only includes a small amount of faults, we apply a closest points algorithm on SA to cluster more faults.

Now we will focus on the SVD clustering. In Figure 7.1 the SVD clustering processing steps are shown. In each processing step several choices has to be made. We will explain all choices in slightly more detail now, then we will discuss them in full detail. For a complete overview of all possible choices see Figure 7.3.

The first step processing step is preprocessing. This is a simple step that alters our data to make it better suited for clustering. Then our data is being fed to Tree Clustering algorithm (TC), which can be any algorithm that produces a decision tree as a solution, in our case this is always an SVD-based algorithm. We have several clustering algorithms available, which were covered in Section 6.3. All of these algorithms can run based on two different matrices: the similarity matrix $S$ or the normalized Laplacian $N$. The splitting can be based on the different splitting criteria proposed in the same section. For the stopping criterion QC we use that each cluster must have an internal match of 1, i.e. all faults are identical. The reason we do this is because we decide how big we want the tree to be in the next step: Tree Pruning (TP). The tree created in TC is too big in general, which is on purpose. We create a smaller tree using the tree from TC, but we can completely redo the decision making of the tree. This is because we know
Figure 7.1: Overview of the different processing steps the clustering algorithms can go through. After preprocessing the user has to decide what clustering algorithm has to be used. We can view the faults removed in preprocessing as separate clusters and add them to the results as indicated by the dotted line.
more of the final structure after creating the full tree. Also, if we want to apply the K-Means benchmark we want to have a fixed amount of clusters. Which is easy to create in the TP step by pruning until the right amount of leaves are left. Furthermore we can remove outliers in this step. We expect to be able to detect them from our tree structure. Finally we can decide to apply some post-processing. Post processing removes the tree structure and only uses the leaves as clusters. To be more precise, it only uses the centroids of the leaves. In this step we can also detect outliers. Our benchmark is based on K-means. So if we want to give good results based on that benchmark, it is smart to optimize for that, this can be done more easily in the Postprocessing step, therefore our postprocessing are closely related to the K-Means algorithm. The Stacking algorithm needs some post processing since it will not give results for the whole data set, it will only find the stacks. We can use a closest points algorithm to find clusters, but we can also use other algorithms of the Postprocessing step.

We will now discuss all steps in full detail.

### 7.1 Preprocessing

Preprocessing is the step we apply to make the data better suited for clustering by filtering out symptoms and/or faults. We have two algorithms of which we can choose whether to apply both, one or none of them. The first algorithm is the SDT pattern filter, the second algorithm is the Small fault filter.

The SDT pattern filter is the filter ASML uses in Smart Diagnostics. This is the exact same filter as the filter applied in layer 3 of Figure 2.1 that filters out symptoms that occur often in healthy machines. The reason we incorporate it as a filter in our framework is because we suspect that the filter Smart Diagnostics normally applies does not have a major influence on our results. Also, the filter removes some of the symptoms based on the different main event types corresponding to each fault. These are top level labels that ASML knows for each fault. We expect that our algorithms can detect these main events automatically, but the SDT pattern filter colors the data. So if we want to see if our algorithm can detect these main event types, it makes sense to turn this filter off.

The second filter is the Small Fault Filter (SFF). ASML is often interested in finding clusters that have an internal Jaccard similarity of 80% or more. If a fault has very few faults, i.e. 3 or less, then it can never have at least 80% Jaccard similarity with any non-identical fault. The SFF filters out any fault with less than $\alpha$ symptoms, where $\alpha$ is a parameter we have to set. $\alpha = 4$ should be used if we want to ensure that faults can have 80% Jaccard similarity with non-identical faults. If we want to create a clustering we can add these faults again after postprocessing. This filter makes most sense if we want to create a clustering with many clusters and should probably be turned off if we want to create a clustering with just a few clusters since the internal similarity of a few clusters will be relatively low.

### 7.2 Tree Clustering (TC)

This algorithm is based on Section 6.3. We recursively split our data in two based on the important vector of a matrix. Instead of only storing the final clusters as was suggested by
Algorithm 6.3 we now keep track of each time we split a cluster in two. This way we create a tree. We can choose between two types of matrices, also we can choose between several splitting criteria. The stopping criterion QC we do not use, we continue until all clusters consist of identical faults.

7.2.1 Matrix selection

This was covered in Section 6.3.1. In short we can choose between the original similarity matrix \( S \), but we expect better results from the normalized Laplacian \( N \).

7.2.2 Split algorithm

This was extensively covered in Section 6.3.2 and Section 6.3.3. In short we can choose between two types of splitting criteria. Both are based on the important vector of the chosen matrix, but we can let it depend on the values in that eigenvector or sort the eigenvector and scan through all faults using that order determining an optimal splitting point based on a scoring function.

7.3 Tree Pruning (TP)

The Tree Pruning step takes the tree structure of Tree Clustering and makes a smaller tree out of it. For this TP can use more information than was available in TC since TC is recursive and does not know what the bottom leaves will be like. In this step we can make a decision on what we select as our clusters. For our benchmark KMB we want a fixed amount of clusters, we can select which branches of the tree we want to prune such that we get the desired amount of leaves. We can also use different criteria to create a tree which might give results that are more useful in practice. Lastly, we can detect outliers in this step.

7.3.1 Leave selection

In this step we take the tree from TC and start pruning it until it is has certain properties we want. For each node in the tree we calculate some values of interest, these include the size of a cluster and internal similarity. We can do pruning in two different ways: Recursively based on the calculated values of interest, or iteratively such that we get a fixed amount of leaves which is required for our benchmark KMB.

Pruning based on the calculated values of interest is very straightforward. Starting from the top node we recursively go down the tree until some value of interest passes a threshold. For example, it appeared that requiring that each node has at least a certain internal similarity has useful results. If we require that each node has at least 80% internal similarity we get clusters that seem interesting for ASML, but are less good for KMB. We can also select nodes to optimize for K-means. We select nodes in a greedy manner such that the child nodes improve KMB the most. Lastly, we can select nodes on the scoring function we have seen in scan based algorithms. For example, we can select child nodes that have the smallest conductance.

We can also prune such that we get a fixed amount of leaves. This is a requirement for our benchmark KMB and can be done as follows. We pick one value of interest, for example internal
similarity, and we try to determine a threshold for this value such that the amount of leaves is equal to a fixed \( k \). We do this by taking two values for our threshold: \( c_0 \) small and \( c_1 \) large. Now we take the average \( c_{\text{avg}} \) of these two values as a threshold and calculate how many leaves this gives. If the amount of leaves is smaller than \( k \) we set \( c_0 = c_{\text{avg}} \) otherwise \( c_1 = c_{\text{avg}} \). This way we quickly find the optimal cut. This method might give clusterings with \( k \) slightly off unless all the childs have a higher value of interest than its parent.

The reason we use this method is because there were some implementation issues in R. The implementation of the tree was not object oriented which made it harder to prune to a fixed amount of clusters. This is how we would advise to implement it in a better object oriented implementation environment: We create a list of “selected clusters” which at the start only contain the root of the tree. Also we create a “child list” with the children of the previous list. The children can be sorted on a pruning criterion. Now we iteratively take the best child, add it to the “selected clusters”, remove it from the “child list” and add the children of this cluster to the “child list.” We update the order of the “child list.” This can be done efficiently with a heap for example, creating a running time below that of TC.

Actually, it is possible to combine TC and TP, using the algorithm mentioned above. Instead of creating every branch of the tree during TC, we could just select the ones we are interested in. The tree pruning scoring functions we have used only use information of the childs or the node itself. This means that we would need to alter the previously mentioned algorithm to include a “child of child list.” This only increases the running time by a constant compared to the analysis of Algorithm 6.4. Using this approach we can reduce the running time of our algorithm to \( O(|\mathcal{F}| \log(k) \eta(\log |\mathcal{F}| + \zeta)) \) and if we use SiS we reduce this even further to \( O(|\mathcal{F}| \log(k) \eta \zeta) \).

### 7.3.2 Remove outliers

We can remove outliers in the TP step. In the previous section we have shown how to select leaves for a clustering. We can also use this in a slightly different way and detect outliers with it. We do this as follows: First we apply a leave selection algorithm using some value of interest and some threshold. Then we mark the faults of some of the clusters as outliers. For this we use some criterion based on one of the values of interest. For example, we can first do a leave selection based on the internal similarity of clusters. We require that each cluster has at least 80% internal similarity. Now we mark all the clusters that consist of only 1 fault as outliers. The idea is that these faults did not have a high similarity with most other faults. So it makes sense to mark them as outlier. Downside of this method is that it is possible that some of these faults should not be outliers. They were split into the wrong cluster somewhere higher up the tree. But in general we expect that most of the faults found this way have low similarity with all other faults and are correctly marked outliers. After removing outliers we probably want to prune our tree further to create a tree with the type of leaves we desire.

We can summarize the processing steps in TP as in Figure 7.2. First, leave selection for finding outliers takes place. This leaf selection should create a lot of leaves, we say it uses a “soft criterion.” Then we mark some of the clusters as outliers. Lastly we apply the leaf selection algorithm again, but now we use a “hard criterion” since we want to create only a few clusters, possibly equal to \( k \).
**Figure 7.2:** Overview of the Tree Pruning step. First leaves are selected using a “soft criterion”, of those the often small leaves are marked as outliers. Finally another leave selection takes place using a “hard criterion” to create the clusters.

### 7.4 Post Processing (PP)

Since we use a benchmark based on K-Means it makes sense to do some post processing to optimize for that benchmark. We propose a few possible postprocessing algorithms. These methods are only based on the centroids. Often this means that we lose the hierarchical clustering we have found. We can also detect outliers in this step.

The simplest post processing algorithm is Closest Points (CP), where we take the centroid of each cluster and for each fault we determine to which of the centroids it is closest. The idea behind this is that during the splitting in tree building we might have made some cuts that were not very accurate. Some of the faults will be in the wrong cluster. By allowing faults to switch to different clusters with centroids closer the fault, we expect the benchmark score to improve. We can view this method as moving faults to different clusters, but most of the original clusters remain intact. So we can still use the hierarchical structure we have created before. One can wonder if this hierarchical structure still makes sense. But since the centroids still clearly belong to a leaf of the tree, we think it does.

An extension to this idea is to actually apply K-Means. Normally one needs to take starting centroids for K-Means, this can be done at random, but we can also use the centroids that were found by our tree. Taking the starting centroids at random will give varying results based on how well the starting distribution is. If we want a lot of different clusters the chance that the initial distribution is bad increases. Thus if we take the centroids based on our tree we expect to have a better starting distribution and therefore better results. Downside is that the hierarchical structure gets completely lost by this postprocessing method, since K-Means will be moving
There is a second way to use starting centroids based on the tree. Instead of using the
centroids we can take one random fault of the found clusters. This way we still use a better
distribution than completely random starting centroids, so we expect better results than K-
Means with random starting points. The main benefit of this method is the possibility to restart
the algorithm and pick the best solution. This way it might give better results than K-means on
the centroids. But restarting it increases the running time as well.

One downside of all these post processing methods is that they are based on K-Means and
therefore have a slower running time than both tree building and tree pruning. This also is
true for CP since it is basically one iteration of K-Means. These postprocessing methods will
take $O(|F|\eta k)$ time, which is slower than creating the tree for large $k$. Here we ignore the
amount of iterarions needed by K-Means and SVD approximation algorithms. Using restarts
with randomized starting points multiplies our running time with the amount of restarts, thus
increasing the running time by another factor.

For ASML this still takes reasonable amounts of time, but another practical problem is
the size needed to store the centroids in R. Centroids are no longer sparse in the library of R
we used, thus they use $O(|E|)$ storage space each, which requires a total of $O(k|E|)$ storage
space. Because of this some problems in our software arise for large $k$ since we try to store the
centroids in our working memory. It is possible to circumvent these problems but this requires
a lot of work. Considering the time scope of this project, we did not do this.

A second downside of most of these postprocessing methods is that they lose the structure
of the tree. The tree gave arguments why a certain clustering was formed. By applying any
of the post processing algorithms we ignore this structure as the faults can move to different
clusters which are not based on splitting.

### 7.4.1 Remove outliers

We can also remove outliers in the postprocessing step. We do this as follows: We apply any
of the above algorithms which can all be seen as K-Means with one or more iterations. In the
final iteration we sort the faults on how similar they are to their closest centroid and remove
the faults that have the least similarity. We can remove as many faults as we want using this
method.

For the Stacking Algorithm and K-Means we use this method to remove outliers. If we
want to compare different algorithms on performance we need to remove the same amount of
outliers in all of them. We can use this method to remove outliers in all cases thus removing
the same amount of outliers for all algorithms. This is great for benchmarking.

### 7.5 Summary

In Figure 7.3 you can see a complete overview of the clustering framework. All possible choices
of algorithms covered in this section are shown on the right. We did not include the different
parameter choices for each algorithm as that would make the picture even more complex.
Figure 7.3: Overview of the complete pipeline the clustering algorithms go through. On the right the different possibilities of algorithm implementations are shown. From each collection on the right, one should be picked to create a complete clustering algorithm. An exception on this is the outlier removal.
Results

There are a lot of results we can show, but we have made a selection of the results we think are most interesting. It is hard to show the eventual clusters that we have found in a way that it is easy to analyze and can be used for all clustering algorithms. We have created several ways to visualize the results that we’ve found, but they often do not show everything. We will try to argue what they do show and why they are useful. For the SVD based algorithms we have created a file that can be used in the tool yEd[15], there is a lot of information in that file and we think it is one of the best ways to analyze the hierarchical results.

We have used two datasets in the results below, a dataset of ASML. It contains the faults on NXT machines in the archive. This will be our default dataset and if we do not specify which dataset we are using, it will be this one. The second dataset consists of 300,000 articles from the New York Times, which is preprocessed to a bag-of-words format. This means we only know of each word how often it occurs in each article. Words like “is” and “the” have been removed.[16] The reason why we use this dataset as well is because some of the data of ASML is confidential. So in some cases we want to show some functionality of our algorithms but would provide sensitive information. In that case we will use the New York Times dataset. Also we want to compare the ASML dataset to a more conventional dataset. This way we can see the differences between what these type of algorithms are normally applied to. Lastly, without an expert the data output by our algorithms is hard to analyze. We cannot analyze how good our clusters are in practice. The NY Times dataset provides insight in how well the algorithms work while being understandable by non-experts as well. The algorithms are mainly designed to work well for the ASML dataset, thus we expect that they will not work as well for the NY Times dataset. From the analysis in Section 8.1 we will already see that the two datasets differ. However, it was one of the few datasets we found that came at least a little bit close to our default dataset. For the NY Times dataset the terminology “article” and “word” are more appropriate than “fault” and “symptom” respectively. We will sometimes use them as well, but for consistency we will mostly use “fault” and “symptom”.


8.1 Basic analysis

We can do some basic analysis of the dataset that provides a bit of insight such as the the size of the dataset and sparsity. We discuss those here.

**Table 8.1:** Basic analysis of the two datasets.

<table>
<thead>
<tr>
<th></th>
<th>ASML dataset</th>
<th>NY Times dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr. of faults/articles</td>
<td>242,132</td>
<td>299,752</td>
</tr>
<tr>
<td>Nr. of symptoms/words</td>
<td>19,608</td>
<td>102,660</td>
</tr>
<tr>
<td>Sparsity</td>
<td>0.00101</td>
<td>0.00226</td>
</tr>
<tr>
<td>Data size (amount of non-zero entries)</td>
<td>4,803,436</td>
<td>69,679,427</td>
</tr>
<tr>
<td>Mean unique symptoms/words per fault/article</td>
<td>19.84</td>
<td>232.46</td>
</tr>
<tr>
<td>Mean faults/articles per symptom/word</td>
<td>244.97</td>
<td>678.74</td>
</tr>
<tr>
<td>Average similarity to other faults</td>
<td>0.0877</td>
<td>0.0557</td>
</tr>
<tr>
<td>Average similarity to centroid</td>
<td>0.2953</td>
<td>0.2359</td>
</tr>
</tbody>
</table>

In Table 8.1 we see a few numbers that show some characterization of the data. The Nr. of faults/articles shows how many items we have to cluster and is therefore an indication of how complex our problem will be. Both datasets have the same order of magnitude for this parameter. The Nr. of symptoms/words shows how many different symptoms and words there are in total. This has some relation with the complexity of the problem, but less so than the Nr. of faults/articles. This is because our algorithm are mainly dependent on the total data size and the amount of clusters we want. The amount of clusters we want is (partially) dependent on the Nr. of faults/articles, which is why it is interesting also. Note that the order of magnitude of the Nr. of symptoms/words of both datasets are arguably the same or different. This has little influence on the speed of the algorithm, but it can have some influence on the similarity between fault/article pairs, which we will discuss later.

The sparsity of the dataset is defined as the amount of non-zero entries in the fault-symptom matrix \( R \) divided by the total amount of entries in this matrix. It shows how much the data reduction is if we use a sparse matrix representation instead of a full matrix representation. Both datasets have the same order of magnitude for this parameter. We can see that using the sparse matrix representation can give a data reduction of up to 1000 times, which is a lot. This is an indication that our algorithms can make use of this. The Data size shows how many non-zero entries are actually in the matrix, this indicates how much storage is needed, either in memory or on a hard drive. We have used R for our implementation and the library “Matrix” which provides sparse matrices for R. This sparse matrix implementation seems to use 12 bytes for each entry in the sparse matrix. Thus storing our matrix will use roughly 58 MB for the ASML dataset and 800 MB for the NY Times dataset. So here the dataset seem to differ in order of magnitude. Since R stores this data in working memory the NY Times dataset is quite large, especially if we want to apply operations on this dataset with the same order of magnitude that also needs to be stored in working memory. This also indicates how large the dataset is if we do not store it as a sparse matrix. Actually, several algorithms in literature we have
encountered[11] seem to use the similarity matrix $S$ as input data, but even storing this matrix becomes cumbersome as we would need over 60GB to store this matrix if we are able to store each entry in just 1 byte. In the case of R we would need 8 bytes for each entry, thus increasing the needed size to around 500 GB. Using a way to store it in either working memory or a hard drive becomes troublesome, of which the latter will slow down the algorithm because of slower access time of a hard drive. Also, ASML considers increasing the time span in which their faults are gathered, which would multiply the number of faults by 4. Storing this as a sparse matrix would cost roughly 231 MB as it would increase only linearly. While the matrix $S$ would grow quadratically in size, thus increasing to around 8TB. This shows that the scalability of methods using $S$ as input data is bad and not viable for our datasets. But since our algorithms depend mostly on the data size and the number of clusters $k$, we are able to use these algorithms even on a personal computer.

Figure 8.1: Distribution of both symptoms per fault and faults per symptom for both datasets.

The Mean symptoms/words per fault/article is related to the data size. We are actually interested in what the distribution of symptoms per fault looks like. It will provide insight in how the similarity between faults will be formed. For example, if there are very few symptoms in a fault, it will always get a low similarity with any other non-identical fault, a fault with just 2 symptoms can get at most a similarity with a non-identical fault of $\frac{2}{n}$. In Figure 8.1 we see the actual distribution of the symptoms per fault. In the same figure we show the faults/articles per symptom/word. This gives a different perspective. Some of the symptoms occur a lot of
times, while others rarely occur. One of the things we cannot see from these distributions is how the similarity of faults will be. We expect that there are more correlations between faults than we can see from this distribution. For example, in the NY Times dataset we expect that words related to sports will often occur in the same article, this is however completely invisible in the distribution shown in Figure 8.1. From this figure we can see that the datasets indeed differ. The amount of symptoms per fault is a lot lower in the ASML dataset. It makes sense that an article has a certain amount of unique words in it, but for the ASML dataset the amount of unique symptoms in a fault is relatively low. There are several faults with a very low amount of symptoms per fault which as discussed before can cause problems with finding non-identical faults that have high similarity. Also the faults per symptom differ for the two datasets. There are a lot of symptoms in the ASML dataset that only occur once. They will only make similarities between faults lower. Occurring a few times, but more than once, can be interesting as it might be more likely that these faults have a high similarity if they have this rarely occurring symptom in common. The NY Times dataset again differs here. Apparently most words occur multiple times. The tail of the distribution is also interesting. In the ASML dataset it shows that there are only a few symptoms that occur a lot. The NY Times dataset has several symptoms that occur very often.

For the ASML dataset we also have information on what main event type each fault belongs to. This is the sole classification we know of the data. In Figure 8.2 we show the symptoms per fault for the four main event types that occur most often. One of the things this shows is that the similarity between main event types can differ just because the amount of symptoms per fault differs. But that this by no means is a way to distinguish the main event types from each other. It also shows if a main event type has several or a few symptoms per fault. For example, LOT is the hardest to analyze for ASML. This is because it produces a lot of symptoms. The spike at 2 symptoms for LOT is because several of the faults in LOT are caused by a user aborting the operation which often causes just 2 symptoms to appear. Also TEST seems to be the main event type that has the most faults with small amounts of symptoms. This makes sense as a TEST will often be specialized at testing just one thing in the machine and is run under restricted conditions, which only produces a few symptoms.

Finally we can take a look at the average similarity of all faults. We can see that the average similarity to other faults is small for both datasets. This makes sense, if we take two random faults out of the dataset, we expect them to have very few symptoms in common. It is likely that a main portion of the average similarity is gained from the identical faults that occur several times. We also give the average to one centroid. This value can be seen as the KMB for having one big cluster. This is the value upon which the clustering algorithm must improve.

### 8.2 Preprocessing

During the preprocessing phase of our clustering framework we filter out some things. Since we can do this for each of our three algorithms: K-Means, Stacking and SVD based algorithms, we can analyze the results for all these algorithms at once. We do not add it at the end of the processing steps to make analysis of the results easier. We will now compare the different types of data we get after preprocessing.
We can analyze the data for four cases. The first one is unprocessed data that didn’t even go through layer 3 of the Smart Diagnostics processing steps (see Section 2.1). Secondly, we can use the data after the third layer. Thirdly, we can look at the data where we applied the full preprocessing step of our algorithm (see Section 7.1), essentially using the data after layer 3 of SD and then removing the faults with very few symptoms. We will remove the faults with 3 or less symptoms. The fourth case is a bit strange: we take the data without using layer 3 of SD, but we still remove the faults with very few symptoms.

For each of these we can argue why they might be interesting. Originally ASML used the data after layer 3 of SD, but what layer 3 of SD does is filtering out the symptoms that also occur in healthy machines. So we expect these symptoms to be noise. But if this was truly noise, our algorithm should be able to handle it relatively well because the SVD techniques are generally good at ignoring noise. So it is interesting to see if the filtering done by ASML is necessary. Especially since the machine is mostly deterministic, we expect that there is a causal relation between all symptoms, also between the ones that occur often in healthy machines. Downside of including these symptoms is that the average similarity will probably go down. The second filter we apply removes the faults with very few symptoms. The reason for this is twofold: If a fault has less than four symptoms, than the highest possible similarity with a non-identical fault is 0.75, which is quite low. The second reason is that these type of faults disturb our Tree Clustering step. Making the algorithm slower and less accurate. The reason why we would not
want to apply this filter is because it removes some data of our algorithm, of which we then have to decide what to do with: Remove it completely or add it later on. All four combinations of these filter settings can be interesting, thus we will analyze them now.

**Table 8.2: Overview of filter results**

<table>
<thead>
<tr>
<th>Filter Combination</th>
<th>Off</th>
<th>On</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD layer 3 filter Off</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD layer 3 filter On</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Small fault filter Off</td>
<td>24.24</td>
<td>25.37</td>
</tr>
<tr>
<td>Small fault filter On</td>
<td>19.84</td>
<td>21.84</td>
</tr>
<tr>
<td>Symptoms per fault Off</td>
<td>299.50</td>
<td>298.09</td>
</tr>
<tr>
<td>Symptoms per fault On</td>
<td>244.97</td>
<td>242.21</td>
</tr>
<tr>
<td>Faults per symptom Off</td>
<td>0.0872</td>
<td>0.0897</td>
</tr>
<tr>
<td>Faults per symptom On</td>
<td>0.0877</td>
<td>0.0945</td>
</tr>
<tr>
<td>Average similarity to other faults</td>
<td>0.0872</td>
<td>0.0897</td>
</tr>
</tbody>
</table>

In Table 8.2 we see a comparison of the filters. We can see that the filters do not majorly change the data, but in all cases it does have some influence. We can see that the SD layer 3 filter has a major influence on the symptoms per fault and the faults per symptom. Both decrease roughly 20%. The small fault filter has less influence. It increases the symptoms per fault and slightly decreases the faults per symptom. The average similarity seems to increase only by a significant amount if we turn the small fault filter on. It is remarkable that turning the SD layer 3 filter on as well increases the average similarity a lot more.

We can see that the two filters give very different results. We try to give a reason why these results happen. The SD layer 3 filter removes some of the symptoms. This is reflected in the decrease of symptoms per fault by turning this filter on. Since this filter removes different symptoms for the different main event types, some symptoms get removed for some faults but not all of them. This explains the decrease of faults per symptom. The Small fault filter makes the symptoms per fault increase mostly, which makes sense as the faults with few symptoms are removed. This affects the faults per symptom barely. This filter does affect the average similarity. But that was the actual reason to introduce this filter. Small faults are hard to match with other faults, thus decrease similarity. By removing them we see that the average similarity increases by a lot, thus the idea behind this filter seems just. It is remarkable that using both filters has such a massive effect on the average similarity. We think this is mostly because of the following. The SD layer 3 filter removes some of the symptoms, so there are more faults with very few symptoms. Thus the small fault filter removes more faults in this case, thus increasing the average similarity more.

We will use different settings of the filters for different type of clusterings we want to find. We will consider two cases we want to cluster and introduce them in the next section. We will also explain why we use certain filter settings for them.

### 8.3 Clustering algorithm analysis

We will discuss the several clustering algorithms here. We will use the same settings for all of them and we will use similar visualizations. We take two cases we want to analyze. We will call these clustering cases. We will discuss the two clustering cases now and give an explanation why we chose them.
Since there are 5 main event types that occur often, it makes sense to try and find these main events as a clustering. This is the only classification data we have, so it is the only way to check whether our algorithm works well for ASML without using an expert. For the first clustering case we set \( k = 5 \) and hope to find the five main event types in five different clusters. We can visualize this as shown in Figure 8.3. We have two quality measures for this clustering. The first one is the K-Means benchmark (KMB). The second one is similar but makes use of the main event types. For each cluster we determine the main event type that occurs the most and use the percentage of faults that match this as how well a cluster scores. The main event type scoring function (METS) is the following, where \( \text{ME}(C) \) returns the main event type of each fault in that cluster:

\[
\text{METS}(C) = \frac{1}{|C|} \sum_{C \in C} |\{ f \in C | \text{ME}(f) = \arg \max(\text{ME}(C)) \} |
\]

We can visualize how well the main event types get clustered as shown in Figure 8.3. This visualization is only possible for the main event types.

We also want to go deeper and find several smaller clusters. ASML uses a minimum similarity of 0.8 in Smart Diagnostics, so it makes sense to find clusters that have this similarity. So we would like to require that all clusters have at least 0.8 similarity, but this is not possible to guarantee in all of our algorithms. Using an SVD based algorithm, where this is possible, we discovered that 8,000 to 10,000 clusters should be enough to achieve this, but we do need to allow outliers. For the second clustering case we take \( k = 1,000 \) and we allow a maximum of 20% of the faults to be outliers. The reason we take less clusters is because the processing time is high and some major memory issues occur in R if we want to compare multiple algorithms for larger \( k \). \( k = 1,000 \) should give enough information on which algorithm works best but reduces processing time by roughly 8-10 times. The percentage of outliers is based on both information of ASML and our own results. There are approximately 20% of faults that have less than 0.8 similarity with any other fault. To compare results of our algorithms on this level we actually need an expert. We can compute the KMB, ignoring the outliers, but this only gives a slight indication of how well the algorithm has performed.

For each clustering case we include the processing time as well to show how fast an algorithm performs. We only include the time the algorithm needs and exclude the time that is needed for loading the dataset, preprocessing the dataset or analyzing the results. All tests were done on an HP elitebook 8470p running R 3.0.2 x64. This means R runs on one core of 2.6G GHz with at most 8 GB of RAM. During the tests other programs have been running, so the measured speed of an algorithm may vary from the actual speed, though not by much.

We have to determine filter settings for our two clustering cases. We specify them here and we will use them for all following algorithms. For clustering case 1 we use no filters. In case 1 we try to retrieve the fault’s main event types, so it makes sense not to use the SDT pattern filter. The SDT pattern filter colors the data because it already knows what main event type a fault is in, which is what we try to retrieve without knowing the main event type. The reason why we would use the Small fault filter is to remove faults that cannot get high similarities with non-identical faults. But since we only create very few clusters in case 1, we can better turn the filter off. For clustering case 2 we the reverse is true. In this case we want to find several
Figure 8.3: Example of barplot visualization. There are 5 clusters formed in this case. The width of the bars shows how many faults are in that cluster. The colors show how many of the faults have a certain main event type. Thus the area of one color corresponds to the amount of faults with that specific main event type and are in that specific cluster. Sometimes faults are not clustered, then they are put in the “none” cluster on the right, which is empty right now.

clusters and are less interested in the main event types. Also, we are interested in clusters with high internal similarity. So we turn both filters on for case 2.

8.3.1 Stacking Algorithm

The Stacking Algorithm (SA) is the algorithm ASML currently uses to create their Pareto. We expect that it will be outperformed by the other algorithms. A downside of the stacking algorithm is that it does not consider how closely related clusters are. It simply takes the top $k$ of most occurring identical faults. Especially for low $k$ we expect that this algorithm is bad. This makes sense since it was designed for large $k$, so a comparison for clustering case 1 is not completely fair, but we include it nonetheless.

In Figure 8.4 we see how well SA performs for clustering case 1. We see that LOT, TEST and STARTUP are separated, but two of the clusters found are very small and mainly consist of STARTUP. This is most likely caused by the way SA works. It takes the 5 largest clusters of identical faults and then applies CP. But if any of these 5 largest clusters are very similar, we create two separate clusters for faults that a human would probably put together. The standard deviation of the similarity of faults with its normalized centroid is quite high. This also indicates
that the clustering can be improved by taking better starting centroids. METS agrees with the fact that three main event types are mostly clustered separately. Running time is low, of this hashing takes most time.

Table 8.3: Results of clustering case 1 clustered by Stacking algorithm. Also including a version of closest point postprocessing algorithm on random starting points.

<table>
<thead>
<tr>
<th></th>
<th>Stacking Algorithm</th>
<th>CP with random starting points</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMB</td>
<td>0.4242</td>
<td>0.4523</td>
</tr>
<tr>
<td>METS</td>
<td>0.8484</td>
<td>0.8487</td>
</tr>
<tr>
<td>Running time (mins)</td>
<td>0.77</td>
<td>8.02 (100 runs)</td>
</tr>
</tbody>
</table>

SA seems to be very sensitive for fluctuations in the data. This is because it only considers the top identical faults. It is likely the top identical faults are similar to each other, so they are bad starting points. To show this, we take 5 random faults from the whole dataset and use these as starting points. Again we apply a closest points postprocessing method. We expect that these random starting points on average are worse than the ones of SA. But we expect that there will be some starting points that are better. We do this 100 times, with 5 random starting points each time. The results are shown in Figure 8.5.

The results in Figure 8.5 confirm our expectations: SA performs better on average, but restarting the algorithm a few times quickly gives a better solution. This most likely happens because of the problem we noticed before: top identical faults are probably quite similar to each other, thus being bad starting points. This is indeed the case: The three topmost identical faults are all STARTUP and have pairwise similarities of 0.4364, 0.6299 and 0.3849 and similarity 0 with the 4th and 5th topmost identical fault. We visualize the best result of CP with random
**Figure 8.5:** 100 times 5 random starting points with a closest points postprocessing method applied result in a very basic clustering. The KMB of these are sorted and plotted. The red line shows the KMB of SA. This was done for clustering case 1.

Starting points in Figure 8.4 as well. So in conclusion SA is not a good choice for clustering case 1.

Now we take a look at clustering case 2. For clustering case 2 we can exclude some outliers, but SA does not detect these by default. We use the following heuristic to solve this: For each fault we determine how close it is to its centroid. The 20% faults that have the lowest similarity with their centroid will be marked as outliers. The results are shown in Table 8.4

**Table 8.4:** Results for clustering case 2 clustered by Stacking algorithm. Also including a version of closest point postprocessing algorithm on random starting points.

<table>
<thead>
<tr>
<th></th>
<th>Stacking Algorithm</th>
<th>CP with random starting points</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMB</td>
<td>0.8770</td>
<td>0.8696</td>
</tr>
<tr>
<td>Nr. of non-empty clusters</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Running time (mins)</td>
<td>1.23</td>
<td>60.80(100 runs)</td>
</tr>
</tbody>
</table>

Again we compared this to random starting points instead of the ones found by taking the top identical faults. Now we encounter a problem for taking 1,000 random faults. We could
simply take 1,000 random faults, but then we would most likely use identical faults multiple times. So it makes sense to first find all unique faults and then pick some at random. There is a third option, using a hash we can just pick a fault at random, but we check if we already took it and repeat this process. The last option will most likely give the best results. These results are shown in Table 8.4. Important to note is that SA performs better than CP with random starting points. This clustering case is what SA was originally used for. So it seems to work well.

So for small $k$ random starting points create enough variation that they outperform SA, but for large $k$ SA is a huge improvement over CP with random starting points.

### 8.3.2 K-Means

K-Means can be seen as an extension to CP postprocessing, it determines the closest points to some centroids multiple times and then calculates the new centroids. K-Means is often applied in practical situations because it is relatively fast and seems to give good results while the algorithm is simple.

![Barplot of K-means clustering of clustering case 1 with random starting centroids and 100 restarts.](image)

**Figure 8.6:** Barplot of K-means clustering of clustering case 1 with random starting centroids and 100 restarts.

The results of K-Means for case 1 are shown in Figure 8.6. On the eye it seems that the main event types are clustered a lot better than we have seen for SA. STARTUP is still split, but only in two clusters. TEST has one big cluster, but several TEST faults are clustered in other clusters. This is most likely because some tests are actually focused on testing certain aspects of other main event types. The reason why LOT is split up is because there are two types of
Table 8.5: Results for K-Means with random starting centroids and 100 restarts on clustering case 1.

<table>
<thead>
<tr>
<th></th>
<th>K-Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMB</td>
<td>0.4619</td>
</tr>
<tr>
<td>METS</td>
<td>0.9003</td>
</tr>
<tr>
<td>Running time (mins)</td>
<td>77.57 (100 runs)</td>
</tr>
</tbody>
</table>

faults corresponding to LOT. Either a few wafers of the lot can be rejected or the complete lot is aborted. They seem to produce different types of errors, which is why they are split up in our clustering. It is nice that this can be detected since this looks like one of the structures directly below the main event types, which is what we were searching for. Since a relatively small amount of faults have SDDRIVERS as main event type it makes sense that a clustering rather splits up the two types of LOT than SDDRIVERS. The same holds for MXACTION, which is even smaller than SDDRIVERS.

It is remarkable that CP postprocessing seems to detect the main event types better as shown in Figure 8.4, while the KMB is better for K-Means clustering, see Table 8.5. This seems to confirm that KMB is not very suitable for ASML’s purposes, but the differences are small and the algorithms seem quite sensitive to small changes in the data. So we cannot say with certainty that CP postprocessing on random starting points is actually better. We tested this sensitivity by taking half of the faults at random. We expect to get the same results as before since we create only a few large clusters which should not be affected significantly by taking half of the data. However, if the data is truly sensitive to small changes, taking half of the data could be enough to create very different clusters again. This is indeed what happens. Using K-Means we now find clusters very similar to the ones found in Figure 8.4. So K-Means is very sensitive to change if we look at METS.

We investigated if K-Means is also sensitive for the KMB. Therefore we kept track of the KMB scores in the 100 restarts. We sorted them and plotted them in the left picture of Figure 8.7. We compare them to CP with random starting points as well. We can see that KMB of K-Means is not very sensitive. Roughly 50 out of 100 runs of K-Means get a high KMB score, so apparently the local maxima in which K-Means gets stuck are relatively close to each other. The graph also shows that restarting indeed is useful, but only slightly. A few restarts should be enough to get a local maximum that is almost as good as we can get with a lot of restarts. But applying a lot of restarts does increase the KMB score slightly, so applying several can be interesting if one has time for it. In the picture on the right of Figure 8.7 we see how the KMB increases each iterations for 5 runs of K-Means. We see that each restart converges to roughly the same value, but also that the starting value differ a lot. These points directly relate to CP on random starting centroids. We can conclude that KMB for CP with random starting points is quite sensitive to changes. However, KMB of the K-Means algorithm on random starting centroids seems relatively stable for clustering case 1.
Figure 8.7: Sorted values of KMB for 100 restarts of K-Means with random starting points (left) and the KMB during the iterations of 5 restarts of K-Means (right).

Table 8.6: Results for K-Means with random starting centroids and 100 restarts on clustering case 2.

<table>
<thead>
<tr>
<th>K-Means</th>
<th>K-MB</th>
<th>Nr. of clusters</th>
<th>Running time (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.8858</td>
<td>922</td>
<td>1243 (100 runs)</td>
</tr>
</tbody>
</table>

8.4 SVD based algorithms

We will now analyze the SVD based algorithms. If we want to analyze all possible combinations of algorithms proposed in the framework in Chapter 7 we would need to run a lot of tests and analyzing all of them would be a lot of work. Therefore we go through the processing steps of the framework one by one and analyze the choices for the algorithms separately. In each step we choose the best one and continue using that one in the next steps. This does not necessarily mean we find the optimal framework, but it is likely we find a good one.

We will use the same clustering cases as in Section 8.3 to test performance.

8.4.1 Matrix

We first will compare the performance of similarity matrix $S$ and the normalized Laplacian $N$. We expect that $N$ gives better results since it has a nice mathematical bound. We cannot create a full clustering yet since we need a splitting criterion for that. Therefore we will analyze this in a different manner.
Figure 8.8: On the left we show the similarity matrix \( S \) for a sample of 1000 random faults of an artificially created dataset. We start with four groups of faults that have 100% internal similarity, but none with the other faults. In the left picture the faults are in the same order on both axes, but randomly permuted random. The color of each fault pair shows its similarity. The goal of finding a good clustering is creating clusters with high internal similarity. This means that if we order the faults such that faults in the same cluster are next to each other we get the picture on the right. If we sort our faults using one of the important vectors we expect to get a picture similar to the right. This is indeed what happens, we find our artificially created similarity matrix back.

In Figure 8.8 we see roughly the goal of what we try to accomplish. The left picture shows the similarity matrix \( S \) for 1000 random faults (on both axes) of an artificially created dataset. When we try to find a good clustering we actually try to sort the axes such that the picture on the right is created. Of course we cannot find the exact same picture in practice since that would be the optimal case and in real life there will always be some relations between clusters and not all faults in a cluster have 100% similarity. Either way, it shows something similar to what we try to achieve by using matrix \( S \) and \( N \) in the SVD-based algorithms: We try to sort the faults such that we can make a nice cut. So if we sort the faults on a vector, we hope to see something as in the picture on the right.

We apply this idea on some actual data and get Figure 8.9. We used the order of the second largest eigenvector of \( S \) and the \( D^2 \phi_2 \) where \( \phi_2 \) is the second smallest eigenvector of \( N \) to sort the faults, i.e. the important vectors. We then use this order to sort the faults and show the similarity with the other faults again. We get something similar to the picture on the right of Figure 8.8. This shows that the ideas for clustering based on the order of an important vector works for both matrices.

We now wonder which one is better. For this we show the main event types on the axes and see that they are sorted as well. We note that the matrix \( S \) is better in separating the
Figure 8.9: An example of the similarity matrix of real data sorted by important vectors. In the picture on the left we again show the similarity matrix of 1000 faults, but now of actual data. We sort them based on the important vectors of $S$ and $N$. We also include the main event type for each fault on the axes. For the top level clustering we expect to be able to find the main event types back.
main event types. Somehow LOT and TEST get mixed up for matrix $N$ (It looks like there are more TEST faults in the picture on the left, this is due to sampling of the picture and should be ignored). However, STARTUP is separated very well if we use matrix $N$. If we look at the minimal conductance cut for sorted faults by $N$ we see that the cut is roughly at the border between STARTUP and the other main event types. So the cut implied by Cheeger’s inequality indeed gives a very good cut in practice.

Based on these two pictures we cannot say one is better than the other. We will test both using some splitting criterion and then continue with the best one.

### 8.4.2 Splitting criterion

We will now compare the splitting criteria. We have 5 different splitting criteria and we would like to apply them to both our important vectors and for both our clustering cases. To make the results suitable for our clustering cases we will use a technique of tree pruning to make the amount of clusters equal to the amount requested by the clustering case. For this we will recursively take the children that improve KMB the most in a greedy manner. Also, to be able to test several cases quickly we took only 30,000 faults at random and use the same sample for all tests. We expect that this sample is large enough to give accurate results. Furthermore, we stop the recursion as soon as clusters have at least 55% similarity, since we are only interested in finding 5 clusters this recursion should be deep enough.

We will not analyze BoS here since it does not split the clusters in two, which makes it less applicable to other steps in the framework. Also, some initial testing shows that the other algorithms are much better than BoS.

After some testing we quickly discovered that MIR has some issues. It often creates very small clusters. This means that the KMB will get a horrible result. MIR was mostly a test if the isoperimetric ratio can be used as a cut using the Fiedler vector, the two do not necessarily have a good relation with each other. This is confirmed in practice and we will not test any further on MIR.

We show the results in Table 8.7. We can see that the KMB are all very close. MiCo for $N$ gives the best results and then both SiS methods, but we cannot conclude that any one is better than the others. METS gives more indications that MiCo for $N$ and SiS give the best results, but we need more analysis to make conclusions.

The running time of scan based cutting methods seems a lot higher than the value based

---

### Table 8.7: Results for different cutting methods on both matrices.

<table>
<thead>
<tr>
<th></th>
<th>SiS</th>
<th>MiCo</th>
<th>SKO</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KMB</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>0.4516</td>
<td>0.4498</td>
<td>0.4485</td>
</tr>
<tr>
<td>N</td>
<td>0.4515</td>
<td>0.4522</td>
<td>0.4498</td>
</tr>
<tr>
<td><strong>METS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>0.8917</td>
<td>0.8561</td>
<td>0.8661</td>
</tr>
<tr>
<td>N</td>
<td>0.8920</td>
<td>0.8779</td>
<td>0.8447</td>
</tr>
<tr>
<td><strong>Running time (mins)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>0.2617</td>
<td>6.12</td>
<td>6.18</td>
</tr>
<tr>
<td>N</td>
<td>0.2422</td>
<td>6.93</td>
<td>6.14</td>
</tr>
</tbody>
</table>
Figure 8.10: The density of the different main event types of the important vectors. Each point on a curved line corresponds how many faults of a certain main event type exist with that value in the important vector. The curved black line is the density of all faults. The vertical lines show where the different algorithms would make their cut in this case. Only 10,000 faults were used as a sample. Also, the build in density function of R was used, resulting in the curves being more smooth than they truly are.

methods. This is caused by our implementation in R. The sparse matrix package “Matrix” we have used in R is not good in recursively scanning through a matrix. We have implemented it such that a lookup in the matrix is done in each iteration which makes it a lot slower than it has to be. We think this can be circumvented using some smart preprocessing of the matrix R, but we were not able to do this during this project. This means that in theory the running time of scan based algorithms is only slightly higher than the running time of value based algorithms. But our current implementation seems to result in a quadratic running time instead of near-linear.

Since Table 8.7 does not give very convincing results we analyze the cutting a bit further. We take a more in depth look at the first iteration of the cutting algorithm. We take the important vectors of both S or N and look at the different main event types and how well the important vectors split up the faults for different Event types. We take the value in the important vector for each fault and plot the density for each main event type. The results are shown in Figure 8.10.

The density plots show that both important vectors split up the main event types quite nicely. However, the important vector of N gives better results. It looks like it should be possible to cut this vector such that the STARTUP events are separated of the other main event types quite accurately. Since STARTUP is the most occurring main event type, this is perfect for cutting. The important vector based on S seems to split the main event types quite well, but they still overlap on several places. It looks like the best cut based on this vector is by splitting TEST of
the rest.

Now we look at the cuts that our algorithms make which are represented by the vertical lines in Figure 8.10. It appears they are all quite close to each other, so the algorithms do not differ by much. However, the cuts based on the important vector of \( S \) seem to cut main event types quite badly. The optimal cut position is very different from the cut our algorithms find. This is a lot better for the important vector of \( N \), where the cuts are very close to the optimal cut. It is remarkable that SiS seems the closest to the optimal cut, this probably means that there is more information in the important vector than just the order for cutting. Somehow the important vectors are balanced around zero. We do not have an explanation for this. This does explain why SiS performs well in Table 8.7. Also MiCo seems to have a very nice cut which correlates with the nice score it got in Table 8.7. SKO, however, is still close to the optimal cut, but seems to cut SDDRIVERS. This is most likely due to the greedy nature of SKO. It favors a high KMB improvement over splitting faults that are not very similar.

Using this analysis we think we should base our algorithms on \( N \) and not on \( S \). Those cuts are likely to be more useful for ASML. The cutting algorithm to use is a close call between SiS and MiCo as both seem to give good cuts. The advantage of MiCo is that it has nice theory behind it that seems to work well in practice. The main advantage of SiS is that in the current implementation in R it runs faster than MiCo and is slightly faster in general.

We now compare these two splitting criteria for clustering case 2. We expect that they will give the best results. We now use all faults again since we are creating a lot of clusters. Using a sample of 30,000 would mean that a cluster contains 30 faults on average, which is small. The results are shown in Table 8.8. We will look at MiCo on \( N \) and SiS on both matrices.

**Table 8.8:** Results of clustering case 2 without outliers clustered by some SVD based methods.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>( S )</th>
<th>( N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cutting method</td>
<td>SiS</td>
<td>SiS</td>
</tr>
<tr>
<td>KMB</td>
<td>0.8124</td>
<td>0.8147</td>
</tr>
<tr>
<td>Nr. of clusters</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Running time (mins)</td>
<td>13.36</td>
<td>14.36</td>
</tr>
</tbody>
</table>

It is remarkable that SiS has a much higher KMB. Somehow SiS gets this remarkable score while needing only a fraction of the running time of MiCo. We suspect that MiCo tries to create very distinct clusters, but does not necessarily look at how large these clusters are. While SiS somehow balances itself out around 0. We included both SiS for matrix \( S \) and \( N \). They get a similar KMB score, but if we base the algorithm on \( N \) we find a slightly better score.

Note that we cannot compare Table 8.8 with previous results of clustering case 2 since we did not remove outliers yet.

Summarizing we can say that SiS on \( N \) seems to give the best results for case 2 and MiCo on \( N \) and SiS on either matrix give the best results for case 1. If we want to make one decision that holds for any type of clustering, we should take SiS on \( N \). We do think the evidence gathered so far is not enough to make this decision, especially since we already applied on version of pruning. Therefore we will continue looking at other cases as well for the next section.
8.4.3 Tree pruning

After creating a tree we can start pruning it to the size we want. This is what we already did in the previous section to make the SVD based algorithms suitable for the two clustering cases. We chose to use KMB optimized pruning in the previous section. We will now show that this is indeed the best choice for KMB.

Table 8.9: Results for different tree pruning methods. We prune on MiCo on N and SiS on both matrices. If we prune based on the score, we can only include a scan based algorithm thus we omit SiS.

<table>
<thead>
<tr>
<th>Pruning method</th>
<th>KMB optimized</th>
<th>Internal similarity</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
<td>S</td>
<td>N</td>
<td>S</td>
</tr>
<tr>
<td>Cutting method</td>
<td>SiS</td>
<td>SiS</td>
<td>MiCo</td>
</tr>
<tr>
<td>KMB</td>
<td>0.8124</td>
<td>0.7902</td>
<td>0.6609</td>
</tr>
<tr>
<td>Nr. of clusters</td>
<td>1000</td>
<td>1000</td>
<td>990</td>
</tr>
<tr>
<td>Smallest internal similarity</td>
<td>0.1272</td>
<td>0.1392</td>
<td>0.3617</td>
</tr>
</tbody>
</table>

The results in Table 8.9 show the results of the three algorithms we want to continue analyzing for pruning as discussed in the previous section. For each we look at the different pruning algorithms which all go down the tree in a greedy manner. Which children we include depends on what pruning criterion we use. Since pruning based on scoring is only possible for scan based cutting we can apply this only to MiCo. We introduce a new interesting value: The smallest internal similarity. This is the similarity of the cluster with the lowest similarity. Since one of our pruning algorithms is based on this it makes sense to analyze the differences of this. Also, ASML is interested in clusters with high internal similarity. As explained in Section 7.3 we can have clusterings that do not have the exact amount of clusters we required. Therefore we include a field for the amount of clusters as well. We did not include running time since pruning is relatively fast.

Pruning for clustering case 1 was not very interesting since it only has 5 clusters, therefore we excluded it.

Clustering case 2 is more interesting. The first thing we can notice is that pruning based on scoring is a stupid idea. With too much clusters it finds the lowest KMB. It appears this method goes down one side of the tree while ignoring the rest, resulting in a horrible score. The second thing we can notice is that pruning optimized for KMB is a good idea, it gives a better benchmark in all three cases. If we look at the cluster with smallest internal similarity we see that pruning based on internal similarity does work well. Especially SiS based on N works good. This probably means that the reason why SiS for N gives a good KMB is because it splits the clusters very well, which also causes it to perform well if we use a different pruning method.

So pruning based on KMB is the type of pruning we want to use for our benchmark. Also, we have more reason to believe that SiS based on N is the best choice. In Figure 8.11 the tree for this version of the algorithm is shown. We can see that it cuts the main event types relatively well, though not all are cut perfectly. Furthermore, we can recognize that TEST has several small clusters which have little similarity with each other, they split off one by one.
**Figure 8.11:** The tree created by SiS on N pruned optimized for KMB to 1000 leaves. This visualization was created by yEd and is called the circular layout. It starts in the white node and recursively splits a cluster in two. The size of each node shows the size of the cluster. The color shows what main event type occurs most in a cluster.
We would like to note that pruning based on internal similarity can be used for finding useful clusters. ASML tries to find large clusters with at least 80% Jaccard similarity in their current pareto. So pruning based on internal similarity up to a minimum internal similarity gives clusters that are close to the Pareto Analysis. We also tried this on the NY Times dataset and it appeared that using this form of pruning gave clusters that seem to contain one recognizable category. For example it seems that sports, politics and even cooking can be recognized, while this works less well if we prune based on KMB. We have created an interactive example which can be found here: [http://www.student.tue.nl/w/t.p.a.slenders/](http://www.student.tue.nl/w/t.p.a.slenders/). We created this example using tf-idf for the word frequencies.[17] Also, we have used SiS on S and pruning based on internal similarity. We chose to use the matrix S because it seems to give better results. We do not know why this is the case, but it only happened for the NY times dataset.

### 8.4.4 Outlier removal

We have two types of outlier removal methods. One is based on the small leaves of the tree, the other based on points that have small similarity with its centroid. We will now compare the two. We will only compare them for clustering case 2 since that is the only clustering case that removes outliers.

For outlier removal based on small leaves we will only look at SiS based on N since it is the most promising case as shown in the previous section, especially since this form of outlier removal has a direct relation with pruning. First we will do soft pruning as follows: We use pruning based on internal similarity of at least 0.9 (which is roughly 80% Jaccard similarity, the threshold often used by ASML). After pruning, all clusters with only 1 fault are marked outliers. Then we prune our tree to 1000 leaves with pruning criterion optimized for KMB.

For outlier removal based on small similarity with centroids we have to apply a postprocessing method. This means that to be able to compare both outlier removal methods we should apply postprocessing to both. We will only apply a closest point method, since this destroys the least of the tree structure. A second problem is that we want to have the same amount of outliers marked in both methods. Downside of marking outliers based on small leaves is that it becomes hard to mark 20% of the faults which is what we require for clustering case 2. Therefore we will mark the rest of the outliers by a small centroid similarity method. This way we get results for clustering case 2.

<table>
<thead>
<tr>
<th>Outlier method</th>
<th>Small leaves</th>
<th>Small centroid similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postprocessing</td>
<td>none</td>
<td>Closest points</td>
</tr>
<tr>
<td>Outliers removed</td>
<td>14.9%</td>
<td>20%</td>
</tr>
<tr>
<td>KMB</td>
<td>0.8563</td>
<td>0.8688</td>
</tr>
<tr>
<td>Nr. of clusters</td>
<td>1000</td>
<td>963</td>
</tr>
</tbody>
</table>

In Table 8.10 we see the results of the two outlier removal methods. We compare two methods, but since they are not completely comparable we need to apply postprocessing to both and remove the same amount of outliers. Therefore the most interesting cases are the two on the right.
two steps of the small leaves method to see how it develops. It seems that the iterations of the small leaves method work well since each iteration improves the KMB. However, removing the faults with small similarity to its centroid has better results. Also, the latter has less clusters. So it achieves a significantly higher KMB while having less clusters. We can conclude that the small leaves outlier removal method should not be used, we should find them using small centroid similarity.

8.4.5 Postprocessing

Postprocessing is the last step of all processing steps. We can even decide to skip it. All postprocessing methods we discuss are more or less based on K-Means. The first one is a closest points method on the centroids of the leaves. Which can be seen as one iteration of K-Means, but the tree structure is better preserved. The second method is K-Means on the centroids of the leaves. The third method is K-Means on random faults of the clusters. This last method can be restarted, thus giving a chance of finding better clusterings.

Table 8.11: Results for clustering case 1 clustered by SiS on N with several postprocessing methods applied.

<table>
<thead>
<tr>
<th>Postprocessing</th>
<th>none</th>
<th>CP</th>
<th>KM on centroids</th>
<th>KM on random faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMB</td>
<td>0.4624</td>
<td>0.4633</td>
<td>0.4633</td>
<td>0.4693</td>
</tr>
<tr>
<td>METS</td>
<td>0.8600</td>
<td>0.8472</td>
<td>0.8472</td>
<td>0.9013</td>
</tr>
<tr>
<td>Running time (mins)</td>
<td>-</td>
<td>0.06</td>
<td>0.13</td>
<td>54.35 (100 runs)</td>
</tr>
</tbody>
</table>

We analyze the results for clustering case 1 first, the results are shown in Table 8.11 and a visualization of the main event types is shown in Figure 8.12. It appears that the K-Means on centroids stops after its second iteration because it does not improve anymore. This means that K-Means on centroids gives the same results as the closest points. It needs the second iteration to detect this. This indicates that our SVD-based algorithm works quite well, its centroids are well picked since K-Means is not able to improve the centroids. However, it is possible to improve the KMB by using random faults of the clusters. This indicates that K-Means works well, but that one of the weaknesses of K-Means, picking of starting centroids, is very important and can be done well by an SVD-based algorithm. If we look at the results of K-Means in Table 8.5 we can see that using good starting centroids can have major influence. Except that KMB improves in Figure 8.12, we can also see that the running time is reduced. This is because we need less iterations if we start with better centroids. We can determine the chance that an ordinary K-Means algorithm would pick faults with the same distribution as K-Means on the clusters produced by SiS on N. This chance is roughly 0.019. This means that roughly 2 of 100 restarts of ordinary K-Means would have faults of the same clusters as SiS on N gives. This explains why we get such an improvement with 100 runs on good starting centroids. Lastly, if we look at the barplot of the main event types clustered by SiS on N with K-Means postprocessing with random faults we can see that it clusters the main event types very well. The 4 dominant main event types are separated and LOT, which can be split relatively easy in two, is indeed splitted to create the fifth cluster. Furthermore, TEST seems spread around the other main event types.
**Figure 8.12:** Barplot of how well main event types are clustered for clustering case 1 by SiS on $N$ with several postprocessing methods applied. CP stands for closest points and KM stands for K-Means. The results for K-Means on centroids appears to be the same as closest points on centroids.

As well. Since TEST can be one of several test types it makes sense that tests, focussed on for example LOT, will be clustered together with LOT. The only downside of this method is the running time, which is significantly higher than the others. Actually, the total algorithm for SiS on $N$ with CP including loading the input data of the hard drive is roughly a minute, of which loading the data of the hard drive now becomes a major part.

In conclusion for clustering case 1, if we want to keep our tree structure or have very low running time, SiS on $N$ with CP or no postprocessing give good results. However, better results are achieved by using K-Means with random faults as postprocessing method at the cost of higher running time.

In Table 8.12 we can see the results of different postprocessing methods for clustering case 2. Now some different things happen than for clustering case 1. We see that CP is no longer the same as K-Means on centroids. Also, K-Means on centroids outperforms K-Means on random faults, it gets a higher KMB with less clusters in less time.
Table 8.12: Results for clustering case 2 clustered by SiS on N with several postprocessing methods applied. (*) No outliers were removed in this case. (**) The amount of clusters was increased by using more leaves. This was done manually so the running time is an approximation.

<table>
<thead>
<tr>
<th>Postprocessing</th>
<th>none</th>
<th>CP</th>
<th>KM on centroids</th>
<th>KM on random faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMB</td>
<td>0.8147</td>
<td>0.8864</td>
<td>0.8925</td>
<td>0.8974</td>
</tr>
<tr>
<td>Nr. of clusters</td>
<td>1000</td>
<td>812</td>
<td>1000**</td>
<td>886</td>
</tr>
<tr>
<td>Running time (mins)</td>
<td>-</td>
<td>2.18</td>
<td>25**</td>
<td>10.23</td>
</tr>
</tbody>
</table>

We now note one advantage of K-Means on the leaves of a tree: We can adapt the amount of clusters generated by the complete algorithm easily. We can simply use more leaves and apply K-Means on the centroids again. Using more leaves is fast, so the running time increases only due to a few more restarts of K-Means. We tried this manually by increasing the number of leaves and running K-Means a few times until we passed 1000 clusters. This gives even better results: the best result we have found for clustering case 2. It is harder to use this idea for K-Means on random faults and ordinary K-Means since they are not deterministic. They produce different amounts of clusters each time they are restarted, which makes it hard to find one with 1000 clusters and a high KMB. But since K-Means on centroids already gave better results for less clusters we expect that it will give better results either way.

8.5 Comparison

We have gathered the results of the three algorithms here. We will use the best results found in the previous sections. We will split the SVD-based algorithm in one that has a hierarchical clustering by only applying CP, and one without the hierarchical clustering.

Table 8.13: Comparison of three algorithms for clustering case 1.

<table>
<thead>
<tr>
<th></th>
<th>SA</th>
<th>K-Means</th>
<th>SVD-based, hierar</th>
<th>SVD-based, no hierar</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMB</td>
<td>0.4242</td>
<td>0.4619</td>
<td>0.4633</td>
<td>0.4693</td>
</tr>
<tr>
<td>METS</td>
<td>0.8484</td>
<td>0.9003</td>
<td>0.8472</td>
<td>0.9013</td>
</tr>
<tr>
<td>Running time (mins)</td>
<td>0.77</td>
<td>77.57 (100 runs)</td>
<td>1.0</td>
<td>55 (100 runs)</td>
</tr>
</tbody>
</table>

In Table 8.13 we see the results for clustering case 1. We can see that the Stacking algorithm currently used by ASML, though for different purposes, gets heavily outperformed by the other two algorithms. The SVD-based algorithm without hierarchical clustering is the better one, although only slightly.

Now we look at the results for clustering case 2 in Table 8.14. Now SVD-based algorithms definitely outperform the other algorithms. The SVD-based algorithms perform more flexibility in how many clusters should be created. Even if we do not want to apply to much postprocessing to preserve the tree structure we see that SVD-based algorithms perform better than the Stacking algorithm and ordinary K-Means.

Now we analyze the different KMB scores can be achieved for different amount of clusters.
Table 8.14: Comparison of three algorithms for clustering case 2.

<table>
<thead>
<tr>
<th></th>
<th>SA</th>
<th>K-Means</th>
<th>SVD-based, hierar</th>
<th>SVD-based, no hierar</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMB</td>
<td>0.8770</td>
<td>0.8858</td>
<td>0.8967</td>
<td>0.8974</td>
</tr>
<tr>
<td>Nr. of clusters</td>
<td>1000</td>
<td>922</td>
<td>1000</td>
<td>999</td>
</tr>
<tr>
<td>Running time (mins)</td>
<td>1.23</td>
<td>1243</td>
<td>≈25</td>
<td>≈40</td>
</tr>
</tbody>
</table>

This is interesting to see how many clusters are needed to create such a score, or reversively if we want to know what kind of score can roughly be achieved by a certain amount of clusters. The results are in Figure 8.13.

**Figure 8.13:** Comparison of KMB and the amount of clusters in a clustering for a few clustering methods with no outliers.

The results show that for small amounts of clusters all algorithms except the Stacking Algorithm give more or less the same KMB. However, for larger amounts of clusters the results start to diverge a little. Eventually all algorithms converge to 1. The dent in the curve of the Stacking Algorithm shows the weakness of it for large amount of clusters. We have also shown that the Stacking Algorithm is bad at finding the main event types, so we can conclude in general that the Stacking Algorithm performs a lot worse than the others.

Lastly we investigated if we were overfitting our data. Especially for clustering case 2 we expect that overfitting can become an issue since it has several small clusters. To test for this we ran SiS on $N$ with half of the faults picked at random. We use K-Means on the centroids as
postprocessing. We are interested in the the centroids and save them. Now we use the other half of the faults and apply a closest point method. We did not mark outliers.

It appears that the learning group of faults gets a KMB of 0.8381. The test group gets 0.8371. Since this is lower we can say that indeed some overfitting occurs. But the difference is relatively small, so we do not think that overfitting is a problem.
Conclusion

We have seen three types of clustering algorithms. The Stacking algorithm that is currently used by ASML in their so-called “Pareto”. Secondly, K-Means, which is often used in other applications because it is easy to implement and generally gives good results. Lastly, SVD-based algorithms, which can give the best results, but require mathematical background to understand. SVD-based algorithms can give an hierarchical clustering, which can provide extra insight. We give an overview of the pro’s and con’s of each algorithm. For the SVD-based algorithm we include two, one which gives a hierarchical clustering and one which does not, but gives better results.

Table 9.1: Comparison of four different clustering algorithms. SA is Stacking Algorithm. We include two SVD-based algorithms, one with hierarchical structure and one without.

<table>
<thead>
<tr>
<th></th>
<th>SA</th>
<th>K-Means</th>
<th>SVD, hierar</th>
<th>SVD, no hierar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection high level structures</td>
<td>Poor</td>
<td>Good</td>
<td>Acceptable</td>
<td>Best</td>
</tr>
<tr>
<td>Detection low level structures</td>
<td>Acceptable</td>
<td>Good</td>
<td>Good</td>
<td>Best</td>
</tr>
<tr>
<td>Includes hierarchical structure</td>
<td>No</td>
<td>Yes</td>
<td>Good</td>
<td>No</td>
</tr>
<tr>
<td>Running time</td>
<td>Best</td>
<td>Acceptable</td>
<td>Good</td>
<td>Acceptable</td>
</tr>
<tr>
<td>Implementation complexity</td>
<td>Minimal</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Uses cosine similarity</td>
<td>Possible</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The results are in Table 9.1. Detection of high level structures shows how well the algorithms will be able to detect high level structures which can be useful for improving Smart Diagnostics. Detection of low level structures shows how well the algorithm will perform for finding an equivalent to the Pareto Analysis. One algorithm includes an hierarchical structure which can provide extra insight. The running time shows how fast each algorithm runs. The implementation complexity shows how much work it would be if one tries to implement such an algorithm. Lastly, most algorithms need the cosine similarity to work, which would require extra work for ASML, but provide the benefit of cluster representatives.

We see that each clustering algorithm has its pro’s and con’s. The best results are achieved by the SVD-based algorithm without hierarchical clustering. But some good results are also
achieved by the same algorithm that includes an hierarchical clustering. However, K-Means
gives good results while the implementation complexity stays low. The Stacking Algorithm is
a bit easier to implement but the main benefit is that no convergence to a different similarity
measure is needed. The detection of it for high level structures is bad, but it was not designed
for this. The detection of low level structures is acceptable, meaning that it works reasonably
well but can easily be improved.

For ASML we think the SVD-based algorithms will be too much work to implement. The
prototype created during this project can be used as a black box, but a full implementation
is most likely not worth the effort. Implementing K-Means might be interesting. It is easy to
understand and gives relatively good results. This requires the cosine similarity to be used. But
considering the limitations of the currently used Jaccard similarity, this might be worth the
effort.

\section{Future work}

There are several things we wanted to investigate but did not find time for, we give a short
overview of them here.

\subsection{K-Means improvements}

There are two interesting aspects of K-Means we think are interesting to investigate. Firstly the
starting centroids of K-Means, secondly a hierarchical version of K-Means.

Especially the starting centroids of K-Means have a huge impact on how the algorithm
performs. Restarting is one solution to this, but it is worth investigating if some easy other
solutions exist, different than finding them using the complicated SVD-based methods.

We encountered some ideas for a hierarchical version of K-Means. This version applies
K-Means on only two centroids and recursively applies K-Means again on the clusters. This
means that the running time of K-Means is reduced to the same time as SVD-based algorithms
using a value based method. This means you can get the benefits of a hierarchical method,
but still use the seemingly powerful K-Means. However, restarting and finding good starting
centroids might become an issue again. Also, one of the downsides of hierarchical algorithms is
that a cluster might be split wrong and this cannot be repaired during the recursion. We expect
that this becomes even more true for this hierarchical version of K-Means. This algorithm will
run faster than ordinary K-Means, replacing $k$ with $\log k$. This is roughly the same speed as SiS
using the optimization described in \autoref{section:7.3.1}.

\subsection{SVD-based algorithm improvements}

We have three items we think are worth investigating further for the SVD-based algorithms:
splitting based on the (ordinary) Laplacian, an interesting postprocessing method by merging
clusters, and why SiS performs so well.

We have seen that we can let our algorithm depend on the similarity matrix and the
normalized Laplacian. But we could also use the (ordinary) Laplacian as suggested by \cite{13}. This
is slightly less easy to implement since we cannot just use a sparse matrix like $R$ or $T$. The
library we used only uses one matrix as input, while a few smart matrix multiplications would
provide a way to determine the eigenvectors of the Laplacian fast. This can be circumvented by
implementing an SVD approximation algorithm ourselves, which was out of the scope of this
project.

ASML is interested in finding faults that are different from a top level perspective but are
still very similar. It can be interesting to create an hierarchical clustering and then look at the
clusters that are similar to each other. They are likely to be clusters that should be merged, but
they are interesting for ASML since they might provide extra insightful information.

Lastly, we are curious why SiS gives the best results. We expected that the scan-based algo-
rithms perform better. They use more information and especially MiCo has a nice mathematical
bound. Somehow SiS performs better, which indicates the sign somehow is related to finding
good splitting points. Investigating this further might provide insight in this phenomena and
give ways to improve the algorithm even more.
Bibliography


