Hochschule Offenburg
offenburg.university

Python Implementation of the SUBSCALE Algorithm

Bachelor Thesis

für die Prüfung zum

Bachelor of Science

im Studiengang Angewandte Informatik

Fakultät Elektrotechnik, Medizintechnik und Informatik

an der Hochschule für Technik, Wirtschaft und Medien Offenburg

von

Stanislav Ramin

05. Mai 2021

Online: https://gitlab.com/Stannat/subscale

Prof. Dr. rer. nat. Tobias Lauer, Hochschule Offenburg

M.Sc Jürgen Prinzbach, Hochschule Offenburg

Eidesstattliche Erklärung

Hiermit versichere ich eidesstattlich, dass ich die vorliegende Arbeit mit dem Thema

Python Implementation of the SUBSCALE Algorithm

von mir selbstständig und ohne unerlaubte fremde Hilfe angefertigt worden ist, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen, unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Die Arbeit lag in gleicher oder ähnlicher Fassung noch keiner Prüfungsbehörde vor und wurde bisher nicht veröffentlicht. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

Freiburg im Breisgau, 05.05.2021

[illegible]

Stanislav Ramin

Preface

Hochschule Offenburg - University of Applied Sciences offers students courses for their education and conducts research in its educational disciplines. Prof. Dr. rer. Nat. Tobias Lauer's main field of research is parallel programming, GPU-Computing, and efficient algorithms [2].

He has contributed to the development of the SUBSCALE algorithm [3] by conducting his work and his staff and graduating students' work. I am very thankful for his guidance, support, exciting discussions, and helpful input regarding my implementation problems while working on the thesis during the last months. I would also like to extend my gratitude to Prof. Lauer's assistant, M.Sc. Jürgen Prinzbach for his clear and comprehensible advises and explanations on the theoretical aspects of the algorithm and some specific parts of the implementation.

Abstract

This thesis deals with the implementation of the SUBSCALE algorithm in the Python programming language. First, the current state of research and the needs of the target group are considered. Then, the choice of language is decided based on the findings. On the basis of self-generated requirements, the implementation is carried out.

Finally, the code is evaluated for accuracy, consistency, and execution time, as well as its applicability in practice.

Since the implementation of the current work proved to be unconvincing, an approach is tested in which Python is used only as a front-end.

Kurzfassung

Diese Arbeit beschäftigt sich mit der Implementierung des SUBSCALE-Algorithmus in der Programmiersprache Python. Zunächst werden der aktuelle Stand der Forschung und die Bedürfnisse der Zielgruppe betrachtet. Anschließend wird die Wahl der Sprache auf Basis der Erkenntnisse begründet. Die Implementation wird auf Basis von selbst erstellten Anforderungen verwirklicht.

Abschließend wird der Code auf Perfomanz und Konsistenz und Ausführungsdauer profiliert und die Praxistauglichkeit festgestellt.

Da diese sich als nicht überzeugend erweist, wird ein Ansatz erprobt, bei dem Python lediglich als Front-End verwendet wird.

Table of Contents

1 Introduction

In 2018 Forbes published an article on its blog about the current generated data volume each day. The magazine stated that back then, "There are 2.5 quintillion bytes of data created each day at our current pace, but that pace is only accelerating with the Internet of Things' growth (IoT). Over the last two years, alone 90 percent of the data in the world was generated" [4]. Another prominent information source, the American data storage company Seagate, published a whitepaper [5] in the same year with a forecast (Figure 1) made by the International Data Corporation (IDC) of the annual size of the Global Datasphere (data, that is stored across various storage media) of the following years. Two years later, in 2020, the IDC corrected the forecasted Global Datasphere, predicting that it would already reach 59 zettabytes in the year 2020 because of the contribution of the COVID-19 pandemic [6]. Tang, Ma et al. computed a compound average growth rate of "40% approximately" [7].

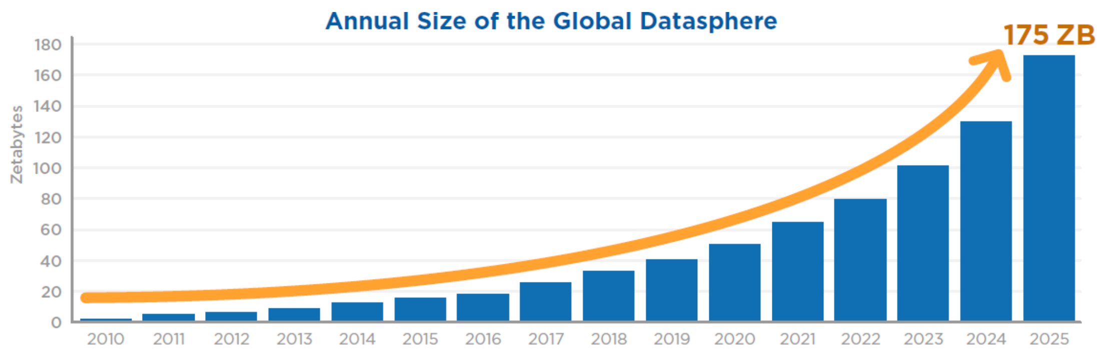


Figure 1 Annual Size of the global Datasphere (2018) [5]

To some extent, parts of this large dump of data can be processed and analyzed. Its domains of origin can reach from media, scientific or industrial to medical domains. The data can be processed by extracting and creating more data in order to gain a better insight and understanding of it. This process is called data mining, and one of its most fundamental tasks is clustering [8, p. 269]. In a paper from Association for Computing Machinery it is stated that "Clustering is the unsupervised classification of patterns [...] into groups" [9, p. 264]. Most of the clustering techniques are based on density heuristics so that groups are formed based on a number of points within a given distance or space. Of course, other measurements are also possible. For example, a gravity-based clustering: Imagine a solar system with planets and their moons that converge with respect to infinite time to clusters of orbs because of the gravitational force.

However, this thesis covers an unsupervised computing technique for metric data points that are grouped based on a distance measurement. The clustering is performed on data sets with an enourmous magnitude of dimensions (or features) which usually makes computation highly ressource consumtive. The SUBSCALE algorithm leverages the runtime performance in comparison to other similar subspace clustering algorithms because of its fewer number of generated subspace clusters and less data base scans while offering a well scaleability with a rising number of dimensions in data sets [10].

1.1 Choice of Implementation Language

Python [11] is the most popular programming language at the moment (with 30% on PYPL-index [12]), especially in the field of data science. For example, as illustrated in Figure 2, in the survey conducted in the year 2020, in the Kaggle data science community, 86.7% of the over 20,000 practitioners have chosen Python as one of their daily used programming languages [1].

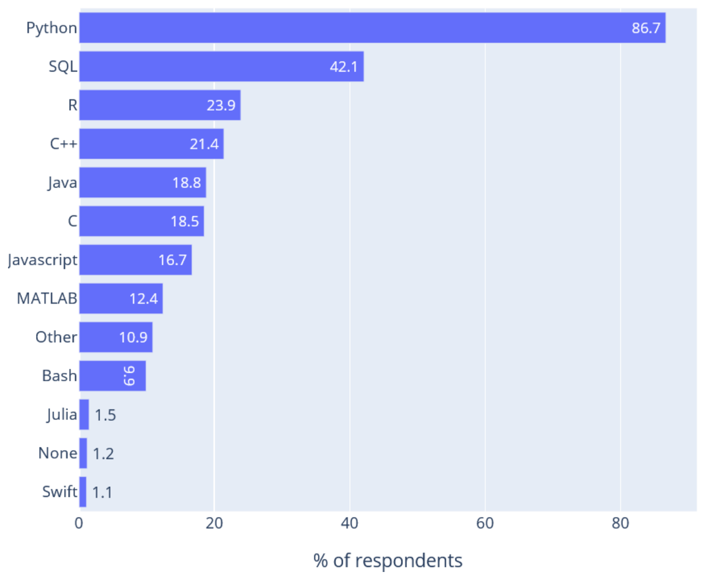


Figure 2: Most popular programming languages in Kaggle data science community community [1]

Based on these facts, it was decided that SUBSCALE needs a reimplementation or a port to Python. Since it is known that Python is not the fastest of languages - In fact, as the Analytics India Magazine states, "Java is pegged to be 25 times faster than Python" [13] - it is insightful to see how the SUBSCALE implementation performs and which parts are mainly responsible for the slowdown. Another measure in the attempt to make the SUBSCALE algorithm popular in the field of machine learning, is that Python is finally used as a front-end for the high performant C++ code with GPU support. The binary is configured and launched from Python. When the application finishes its calculation, the results are agglomerated and displayed with standard pythonic visualization tools.

1.2 Thesis Structure

The thesis is structured coarsely in three parts. The first part is chapter 2, the approach, which analyzes the theoretical and practical state of the art of the SUBSCALE algorithm. First, the motivation for the choice programming language is explained. Then, the theoretical background is put into the picture. This part also covers the program requirements analysis, design, the development tools and possible outcome artifacts. That is, in particular the program and its functional and non-functional requirements.

Parts two and three are the hands-on parts. Chapter 3, the implementation is the second part. This part provides detailed insight into various implementation aspects. Most of them refer to the main project goal, the Python implementation of the SUBSCALE algorithm. Besides the algorithm concerning chapters there is also a chapter about execution speed-up improvements through Python tools like *Just In Time* compiler or specialized Python libraries. The last part, chapter 3.2 describes the second project, a front end to a web interface for displaying and evaluation purposes, that was designed after Python didn’t show any remarkable speed-up improvements.

Chapters 4, 5 and 6 are the last part of the thesis. This chapters cover the results and algorithm artifacts. Chapter 4, covers the analysis of results and describes which algorithmic parts are necessary to log. Chapter 5 discusses the how the implementation was realized, the issues that arose and how the results were validated. Also, both speed-up techniques from chapter 3, the speed-up from Python tools and multiprocessing are critically analyzed. In Chapter 6 the conclusion of for both, the reimplementation in Python and the front-end interface is stated. Eventually some propositions in the outlook for future improvements are suggested.

2 Approach

The implementation requires analyzing the demands for the complete program. Possible solutions are first proposed and then evaluated to find an appropriate solution.

2.1 Theoretical Background

In an unsupervised setting, clustering is a popular information extraction mechanism where there is no prior knowledge of the underlying data. Starting from the early 1970's it was present in different disciplines such as "biology, psychiatry, psychology, archaeology, geology, geography, and marketing" [9, p. 269]. Clustering can give insight into rich data and find patterns that were not evident before. For example, let us consider the commonly used multivariate "Fischer's Iris data set" [14]. The data set is a $n \times d$ matrix that consists of $n = 150$ samples with four attributes (or features) $(d_0 - d_3)$ and the classification (d_4) from each of three species of the Iris flower. Let's assume, that d_4 is not identified in the data set yet. Then, all three species can be identified by setting up suitable parameters for a clustering algorithm regardless of the potential classification labels in d_4. Only the features $(d_0 - d_3)$ are taken as input for the algorithm. Of course, this method is not bulletproof because the clustering parameters have to be as good as possible in order to identify exactly three classes.

On the one hand, with some prior information on the data under inspection, chances can be raised to find the desired number of classes. On the other hand, since the ground truth is unknown, the number of classes in the particular example of the Iris data set is not necessarily three. The decision, what should be found, either the number of possible clusters or the specification for each sample assignment to one of the predefined clusters, is up to the user.

2.2 Motivation and Problem Statement

When coping with high dimensional data, "traditional clustering algorithms like k-means (MacQueen, 1967) and DBSCAN (Ester *et* al., 1996) fail to find meaningful clusters" [3, p. 81]. The reason is the "curse of dimensionality": "data points lose contrast in high-dimensional space" [3, p. 81] because of the density or data/space ratio, as space grows much more than the data within. For example, if a 1 cm^2

square is considered as a data point within a 4 cm² square (as space), then the ratio in this two-dimensional space is ¼. Whereas, in a three-dimensional space, the ratio for a 1 cm³ cube in a 4 cm³ cube will become 1/16. So, the space becomes sparser and similarity measures like the "Euclidean Distance" between two points P_i, P_j become less expressive, since the number of summands increases with respect to $D = \sum d_i$ dimensions (see formula (1) for the distance measure).

This problem can be treated with the "subspace clustering" method by observing the subsets of the data's dimensions. One characteristic of the examined data may become apparent in one smaller subset of dimensions. For example, in figure 3, on the left side, the elements are clustered in a greater subspace. Two elements are put awkwardly in a central cluster and might be misclassified. In contrast, a smaller subset of dimensions may reveal different clusters based primarily only on shapes (Fig. 3 center) or only clustered on color (Fig. 3 right). So, the feature color can help to determine outliers.

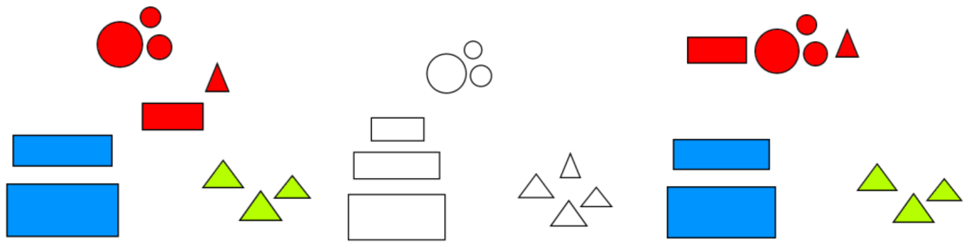


Figure 3, left: elements clustered by shape and color, center: clustered by shape, right: clustered by color

However, this method, called "subspace clustering", comes with a trade-off: For D dimensions, there are "$2^D - 1$ axis-parallel subspaces" [15, p. 5]. Since most subspace clustering algorithms use "enumeration of data points and compute redundant clusters during the clustering process" [3, p. 81], the execution time blows up when high dimensional data is processed. This inefficiency is tackled down by the SUBSCALE algorithm.

2.3 Objective

SUBSCALE is a subspace clustering algorithm that utilizes the *apriori principle* [16], where a cluster C_i of points in a greater subspace S is also present in its subspace S' (in short: $C_i \in S \Rightarrow C_i \in S', when\ S' \subset S$). The other way around, a cluster C_j ($C_i \neq$

C_j) that is not contained in a lower subspace S' is all the more not contained in a higher subspace S. This bottom-up approach reduces the number of computed subspaces significantly. The central principle of the SUBSCALE algorithm is the successive, multileveled clustering.

The algorithm consists of 5 steps:

1. First, points from each dimension *d* are chosen. From these points P^d, one-dimensional clusters, called *Core Sets* (*CS*) are created. A CS consists of a minimum amount of points (*min_points*) within a given distance in the selected dimension. The *CS*s are created based on a density-similarity-measure (1) (Minkowsky Distance with $p = 1$ and the number of dimensions $k = 1$). This step is repeated for every dimension of the data set.

$$Dist\left(P_i^d, P_j^d\right) = \left(\sum_{d=1}^{k} \left(P_i^d - P_j^d\right)^p\right)^{\frac{1}{p}} \tag{1}$$

Minkowsky Distance $P_i^d \neq P_j^d$.

For SUBSCALE specialized to Manhattan Distance. The parameters p and k are set to 1.

2. Second, every Core Set cluster is so processed that all subsets of min_points size from each Core Set are determined: We call them Dense Units. This part is an essential for the algorithm. Chapter 3 of this paper gives an in-depth insight into the manifold of implementations and tune-up capabilities.
3. A Check is performed to examine if there exists the same Dense Unit in different dimensions. Found occurrences are stored in a table as records for each distinct Dense Unit and its associated dimensions. Storing the Dense Units is closely connected with arising memory capacity problems and is also considered as one of the most important topics.
4. Then a transformation of Dense Units and Subspaces is performed. Points from multiple Dense Units are resolved to a "maximal subspace clusters"

[17, p. 215]. The resulting cluster has the maximum possible subspace size and is associated with its points.

5. Ultimately, the final clustering is performed with a traditional clustering algorithm on each Point-collection of the maximal subspace cluster - candidates.

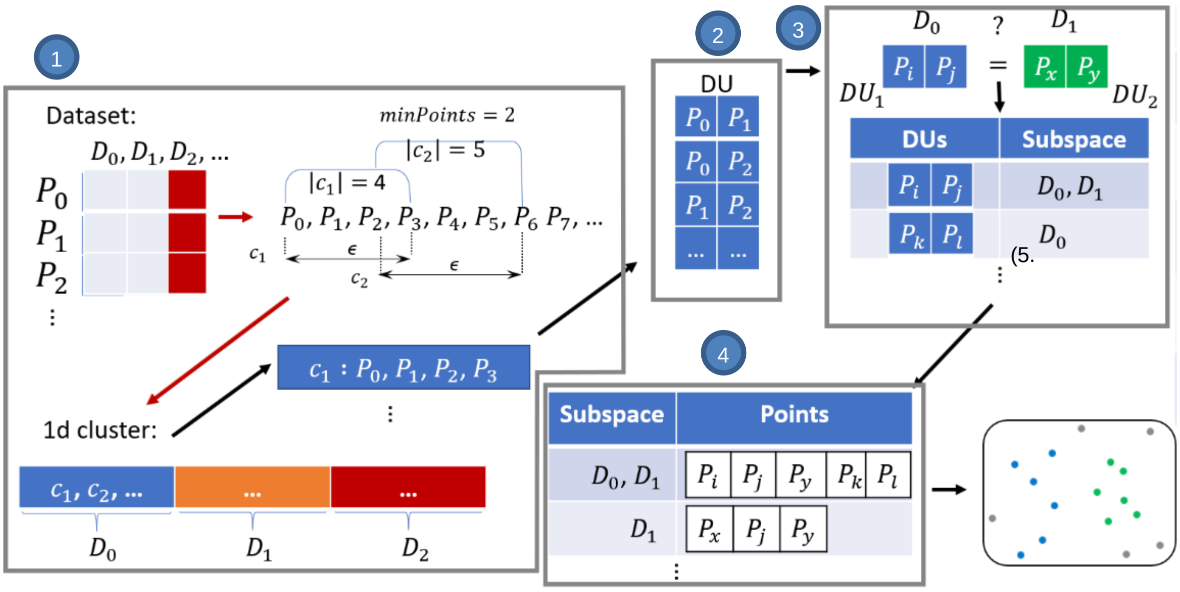


Figure 4: First 4 steps of the SUBSCALE algorithm.

2.4 Terminology

Some notions are used interchangeably within different implementations of SUBSCALE by different authors. Since this thesis is referring to those implementations and is discussing them, a summary of those items is provided:

- Core Sets: Dense Neighbors
- Collision table: Signatures table
- Clusters table: Subspaces table

2.5 Technical Constraints and Chances

The implementation requires analyzing the demands for the complete program. Possible solutions are first proposed and then evaluated to find an appropriate solution.

SUBSCALE underwent implementations in different programming languages. There are implementations in Java and a C++ version with CUDA [18] GPU support. However, in the machine learning community, Python-implementations are most favored. From an engineering point of view, on the one hand, a Python implementation has the advantage that it has a powerful semantic and a clean syntax. Thus, Python code has high readability.

On top of that, it has a rich standard library. On the other hand, it has its performance peaks in different disciplines than C / C++ or Java. Java is fast when it comes to streaming and access operations on objects. C is quick with basic linear algebra calculations and number-crunching tasks. Python, however, is strongly dependent on implementation. Python compiles code usually at runtime and is therefore slow. To speed things up, different approaches A1-A4 for acceleration are possible:

A1: Since the data set is homogeneously typed, packages like NumPy [19] or Pandas could be harnessed for computation on multi-dimensional arrays and matrices.

A2: Just in Time compilers like PyPy [20] or Numba [21] can speed up the program execution but require appropriate, and suitable PyPy supported packages. Numba has CUDA support for GPU computing [22]. However, it requires either to wrap functions in decorators or to annotate them for faster execution.

A3: Another possible speeding up technique is the use of the programming language Cython [23], which claims to be a superset of Python: Unlike in Python, which "uses dynamic typing" [11, p. 249], here types are explicitly annotated in the code. The resulted program is translated from a python semantic-like code to C code and is directly compiled into machine code. No in-between interpreter as is used in normal Python execution is needed from here on.

A4: If all these speed improvements will not lead to success, Python can only be used as a front-end and configuration tool. The C++ program is started as a process from Python code, and status messages of the program standard output are piped to the Jupyter Notebook web interface. The output files generated from the C++ code can be accessed, preprocessed and displayed descriptively with user-friendly functions in Python.

2.6 Requirements Analysis

The following requirements are designated for the implementation:

2.6.1 Specification of the Reimplementation

The Python version must be functionally identical to "SUBSCALE-extended". That means, given input data must produce the same output data as the original program whenever possible. Parts that contain randomness must be recognizable as such.

2.6.2 Platform Independency

Any system with Python installed must run the implementation.

The Python libraries NumPy and Pandas are required to be installed.

The library Scipy is optional, if the DBSCAN algorithm is not required to be executed for final clustering.

2.6.3 Documentation

To make the utilization of the program more user friendly, a user manual for the installation must be provided. A file with examples must show the potential usage.

There should be a change-log file reflecting the decisions that are taken during the program development.

The code-documentation should be meaningful so that it is evident which parts lead to crucial changes and provide a maintainable implementation.

2.6.4 Readability of the Code

The code must be readable. The common norms like documentation of every 2-3 lines of code, for the the typical average size of a function.

Objects should be as shallow-nested as possible. The code should be reduced to simple types whenever possible. In order to make the code maintainable, functions should not cause side effects. Instead, pure functions should be used.

2.6.5 Unit Tests

Deterministic parts of the program must be tested with unit tests. The unit tests must be embedded in a suite to reflect a code change immediately when executing the tests.

2.6.6 Utilizable Results

The program output must be utilizable for further work. Results and significant interim results should be visualized with common graphic packages.

2.7 Possible Algorithmic Solutions

From the 4 steps of the SUBSCALE algorithm (see "Figure 4: First 4 steps of the SUBSCALE algorithm"), there are several properties and thereof resulting trade-offs for the implementation derivable:

2.7.1 Step 1: Creation of Core Sets from Selected Dimension

One central efficiency of the algorithm is its manifold parallelization possibilities. First, since creating a CS in step 1 requires only 1 dimension of the data set, the dimensions can be processed independently. And second, even more efficiency is achieved because each dimension has to be processed only once. However, as a trade-off in later algorithm phases, further processing and cluster-generations are necessary.

2.7.2 Steps 2 and 3: Creation of Dense Units from Core Sets and Agglomeration in a Data Structure

In step 2, there is also a possible parallelization technique. Though this parallelization comes with necessity, the number of possible DUs per CS increases exponentially (2). At some point, hardware constraints are faced: When the computer memory is wholly consumed, the program crashes.

$$\binom{|cs|}{minPoints} \tag{2}$$

The number of all possible DUs in all CSs can be approximated. With this, slices with a defined number of accepted DUs can be calculated. Each slice has a lower and higher boundary. Only if a DU is contained within the currently selected boundary it is subjected to further processing. So, only a fixed number of slices with respect to available computer memory capacity is processed at a time. Another advantage due to a possible assignment of each DU to a dedicated slice is the possibility to save the computed DUs in parallel. This is possible, since different locations of the memory with respect to slices can be accessed for independent writing, without contention between different threads or processes. The trade-off here is that the more slices there are, the longer the computation takes. figure 4 depicts the runtime with raising numbers of split factor (*SP*). The *SP* determines the number of slices.

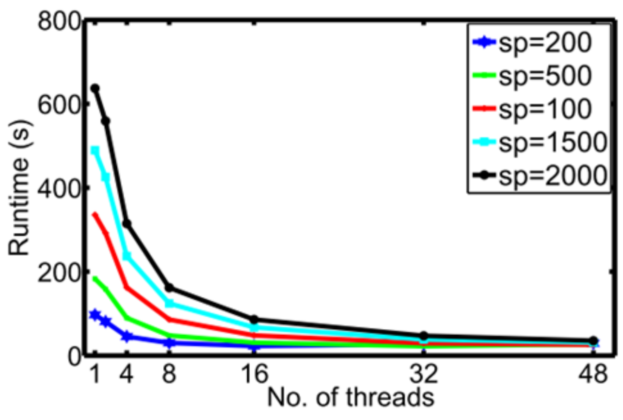


Figure 5: Slices in parallel [23, p. 218], SUBSCALE runtime increases with sp and reduces with increased parallelization

A possible yet not realized (in any implementation) optimization would be the calculation of slices for multiple host systems: All hosts inform the server about their free resources, and the server calculates based on this information the appropriate slice boundaries. Then, each host requests the slice boundaries and the required data. Dependent on program design, the data can be handled as one of these three options:

- the whole data set
- all points in a range of dimensions
- the server-side calculated CSs from a dimension

The calculated DUs per host are saved on the server in one file per host and slice. Then, these files are sent to the server and processed in step 4.

2.7.3 Step 4: Transformation from Collision Table to Subspaces Table

Step 4 is the finishing operation, and this step cannot be parallelized, because, at some point, the data has to be concentrated and unified in one single place.

2.8 Software and Hardware Components

This chapter covers the discussion of possible software and hardware components. The choice of a suitable development environment facilitates rapid prototyping and debugging. Hardware is also an important aspect, since SUBSCALE's performance is highly scalable with hardware.

2.8.1 Development Environment

The pure just standard libraries including Python implementation is executable under both platforms; Windows and Linux. However, speed optimization enhancements may, but do not necessarily, break compatibility to Linux.

For the development, it is beneficial that a suitable integrated development environment (IDE) is used. It must facilitate syntax highlighting, autocompletion, and debugging. An Integrated version control system with access to online repositories is also a must. JetBrains PyCharm 2020 IDE [24] satisfies all these requirements plus it comes with a free license for students of academic institutions. GitLab is used as the online repository of choice because the most similar alternative GitHub is not offering private repositories for free. When the development is finished, the repository will become publicly accessible.

Another IDE that is famous among machine learning applications is Jupyter Lab [25] or Jupyter Notebook. Future users might want to use it for rapid prototyping, and it also synergizes with graphic packages. The IDE can also be used for prototyping the SUBSCALE code.

For the reference implementations, the Java code from the SUBSCALE-extended and SUBSCALE-plus [26] version is examined and debugged with the IDE IntelliJ IDEA 2019. This IDE is also from JetBrains and features similar functionality as PyCharm. The C++ implementation "SUBSCALE-GPU" by Nicolas Kiefer [27] [28] is a Visual Studio solution. Thus, the Visual Studio 2017 IDE is used for necessary proceedings.

2.8.2 Hardware

Except for the SUBSCALE-GPU implementation, the other two existent implementations (JAVA: SUBSCALE-extended and SUBSCALE-plus) do not require a GPU. For this reason, any computer is sufficient for testing and comparing the

Python implementation with the other implementations. However, since there are no familiar benchmarks on consumer hardware, a comparison between the SUBSCALE-GPU multi-core CPU- and GPU-algorithm execution is desirable. Table 1 shows a brief overview of the existing implementations.

Implementation	**Language**	**Single Core CPU**	**Multi-core CPU**	**GPU**
SUBSCALE-plus	Java	+	-	-
SUBSCALE-extended	Java	+	+	-
SUBSCALE-GPU	C++	+	-	+
SUBSCALE-Python	Python	+	-	-

Table 1: Overview of the existing implementations with respect to hardware requirements.

3 Implementation

This chapter describes the developed solution chronologically in Python, extensions for SUBSCALE-GPU, and the Python front end for SUBSCALE-GPU.

In the Python implementation, first, the modules and source files are introduced. Then the underlying data and its organization is described. Next, functions for loading the data into computer memory are introduced, then functions for single-core solutions of the SUBSCALE algorithm, the output functions, and final clustering with the DBSCAN algorithm [29]. Also attempts to realize approaches A1 – A3 for multi-core execution and speed-up are described. Finally, the code of the front-end version is described.

3.1 Python Implementation

Python exists in different versions, and as the official Python website states, it "does not provide backward compatibility" [30]. Backward compatibility is a major concern with respect to Python versions 2.x and 3.x because different language features break the compatibility. E.g., statements like "print" turned into functions, and exceptions are raised and caught differently. The code of the current thesis is guaranteed to run under Python 3.9 version. However, this version's new features are not used to provide backward compatibility to lower 3.x versions.

The Java SUBSCALE-extended version is very modular but has many classes, interfaces, and nested Objects. All these aspects make the program flow hard to follow. Guido van Rossum was convinced, when he developed Python, that "code is read much more often than it is written" [31]. Therefore, it is crucial to make the code most readable and easy to grasp.

3.1.1 Overview of Modules / Project Functional Responsibility

- `IO.py`: The Class for file related I/O operations. Except for one function, all functions are methods that are related to SUBSCALE. They are designated to be called from functions in other files.
- `FourthDegreeFunction.py`: Estimates slice boundaries for DUs. Not used because implementation not finished yet.
- `Subscale.py`: Core functions of SUBSCALE

- `test_DB.py`: Unit tests for DB.py
- `test_SUBSCALE.py`: Unit tests for SUBSCALE.py
- `Timer.py`: Profiling of function execution.
- `main.py`: Connecting code for functions in DB.py and Subscale.py.

3.1.2 Data Input and Input Operations in File DB.py

The underlying data is organized as an $n \times |D|$ matrix, where n is the number of rows that represent the point as vectors in the file and $|D|$ is the number of the columns (or features) in a ".csv"-File. For SUBSCALE, each row i corresponds to a point P_i as a vector $\{P_i^1, P_i^2, \dots , P_i^{|D|}\}$ in a $|D|$ – dimensional space. SUBSCALE, processes each dimension $d_j \in D$ independently of other dimensions to create a one-dimensional cluster *Core Set* (*CS*) instead of processing a point in the projection to all dimensions at a time. In contrast, I/O operations on files are performed byte, or line-wise and data is read/written sequentially. Since most files are structured row-wise, it would be wise to transpose rows and columns to utilize this fact. Then dimensions are aligned the same way as they are read.

In the file `IO.py`, the transposition is accomplished by the function `transpose(filepath)`. When the CSV is loaded with the methods of the class ***DB***, that is utilized for caching and keeping file and folder organization related information, with the methods `get_current_row` or `get_next_row`, a possible crash from computer memory overrun is averted, or at least the chance is reduced significantly because only necessary records are kept in the memory at a time. Non-sequential access, e.g., for parallelization to the input file is provided with an optional parameter in the `get_next_row` method. As a side effect, input data is saved as a 2-tuple where the first element is the automatically ascended number in the current row and the second element the float representation of the point value.

In the first stages of the implementation, reading input data was error-prone due to the manual definition of the size of the dataset when the n and $|D|$ parameters were defined manually by the user to avoid reading the whole data set. This resulted frequently in erroneous algorithm calculation and subsequently this option was removed. Currently, the constructor sets both parameters based on the optional parameter `is_transposed`. By default, the latter is set to false since most CSV-files are not transposed. The file, however, is still not read at whole to determine n and $|D|$. Instead, the file lines are counted by processing one line at a time, and their amount is returned as a result.

In case that there is no input data available, a synthetic data set can be created for a smoke test by passing the number of samples and dimensions to the function `create_random_data_set`.

3.1.3 Configuration

First, there were several command line parameters as input to define the amount of clusters and the execution mode. Invalid argument values were recognized and resulted in throwing corresponding exceptions. However, with raising functionality a vast amount of command line parameters resulted in erroneous configuration. Therefore, now the SUBSCALE algorithm requires a configuration file to process the increasing number of options that emerged during implementation.

- `min_points`: The number of minimum points before the cluster is created. It is required that `min_points` ≥ 2.
- `eps`: Distance between Points within a cluster is created.
- `split_fac`: Number of splits, the higher it is, the less computer memory is consumed at once, and the longer is the execution time.
- `is_pivot`: True if the pivoted version of the algorithm has to be used.
- `dbscan_only`: True if only the final clustering with data available in `resultPath` directory shall be used.
- `just_calculate`: True if only the core sets have to be calculated. No Dense Units or Subspaces are kept in memory.
- `use_dbscan`: True if final clustering has to be performed.
- Is_multiprocessing: True if multiprocessing shall be used. Experimental, does not properly work right now.
- `transposed`: True if the input data set is transposed so that dimensions are aligned row-wise.
- `save_slices`: True if the output from slices mode calculation should be saved in distinct files for each slice.

- `dataPath`: The file path to chosen data set.
- `resultPath`: Path to results directory.

3.1.4 Data Output and Output Operations in File DB.py

Before performing the final clustering, the result files are saved to disk. The output directory is created based on the filename of the input data set as the prefix and the suffix “`_results`”.

The entries in the dimensions-clusters are saved to files so that each record has the following structure: Every record is a compound of 2 scopes $[d_i, \ldots\ ] - [id(P_j), \ldots]$. The subspace (or dimensions) and the point-ids scope, where d is a dimension that contains the listed $p's$. The file is named after the agglomerated number of dimensions.

Trivial clusters that contain only 1 dimension are also saved to disk, despite being useless for the final clustering. The reason therefore is, that since 1-dimensional clusters are still generated, the number of those samples in comparison to samples in other subspaces might provide further information about potential clusters existence.

An overview of all saved evaluation, interim and final results are provided in chapter 4, Output of Results / Interim Results.

3.1.5 SUBSCALE Core Algorithm

The central part of the program is the same as in other implementations. All discussed functions are part of the file SUBSCALE.py

3.1.5.1 Labeling Points with Signatures

In order to detect two identical DUs $\mathcal{U}_1^{d_i}, \mathcal{U}_2^{d_j}$ from different dimensions d_i, d_j the contained points in each DU have to be summed up, so that $sum(\mathcal{U}_1^{d_i}) = sum\left(\mathcal{U}_2^{d_j}\right)$. To achieve this, points are labeled with a randomly calculated, high signature number in an interval of two high potentiated numbers ($[10^{13}, 10^{14}]$). Since each point will have a unique number, the sum of multiple points is likely to be unique as well [10, p. 10]. The labeling is performed in the `label_points` function. It is sufficient to process points just from one dimension because other dimensions are projections of the same points. Dependent on whether the data set is transposed or not, different input loading functions are used.

3.1.5.2 Min and Max Signatures

One fundamental feature of SUBSCALE is the use of slice boundaries. These boundaries ensure that only DUs with suited signatures are processed at a time and prevent the system from high memory usage (see Fig. 5). Both signatures, min and max, are used to calculate the slice size and slice boundaries in the configuration code (see slices). To determine both signatures all labeled points (Labeling points with signatures) are checked and the sum of the lowest and highest `min_points` are set to the min and max signature.

3.1.5.3 Slices

There is always a time, when memory capacity becomes an issue. To avoid a high memory usage, it is essential to keep the number of Dense Units low. This is achieved by defining a number of slices so that only Dense Units that belong to a dedicated slice are put in the signatures table. When a slice is finished, the Dense Units are agglomerated to a subspaces table and saved to disk. Then the memory is freed and other Dense Units are computed. Each slice has a boundary of a lower and higher bound. Figure 6 depicts the selection of Dense Units within different slices. For example, in slice $s = 0$ only the Dense Units $\mathcal{U}_1 = [P_0, P_2]$ and $\mathcal{U}_2 = [P_1, P_2]$ are added to the signatures table. Other Dense Units are still computed first but discarded when it is detected that they are not within the current bounds. Only when the slice selection changes to $s = 1$, other Dense Units can be further processed and added to the table.

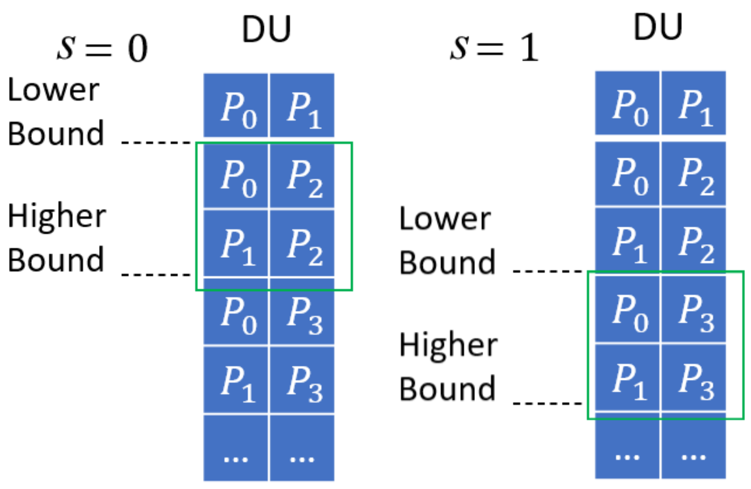


Figure 6: DUs constrained by slices. Bounds belong to associated slice.

To make the slice option effective, the Dense Units from each slice have to be saved to disk so, that the occupied memory is freed. The chapter "Data Output and Output Oprations in File DB.py" specifies the data format in practice. When the next slice is processed, first the saved data from disk is loaded and then agglomerated with Dense Units from the current slice to detect new collisions and to add further dimensions.

There are two implementations from which the more appropriate to the situation has to be chosen. One option is to make the Dense Units from each slice to be stored to files, each containing the same amount of dimensions per Subspaces-dimensions record. After all slices are executed, each file is sifted through, duplicates are removed and the entries are rewritten back into the corresponding file. If the disk space is a constraint this method should be chosen. The drawback in this case is that more time for writing the slice is required, because of many open and write operations that occur when a subspace changes. The other variant is to write each slice as one file with heterogenous subspaces. This variant consumes more disk space but writing to disk is performed faster for writing the slices, because large portions of data are written at once. For each subspace every slice has to be loaded into memory to eliminate duplicates that are present in different slices. Each subspace is processed successively and is saved to disk before for the next subspace all slices are loaded again. This algorithm terminates when no new slices can be found.

- `save_slice` in `IO.py`: Saves whole slice to the slices subdirectory `./results/slices`
- `save_subspaces_to_csv` in `IO.py`: Saves slice to multiple files grouped by subspace size into `results` directory
- `refine_subspaces` in `SUBSCALE.py`: Refines either files from slices or subspaces.

3.1.6 Connecting the Code

In order to assemble different parts of the core algorithm, the connecting code in figure 6 is used. Most of the components are joined in the configuration part, in the function `single_core` of file `main.py`. This also the program entry point. To run SUBSCALE `main.py` has to be launched with a json configuration file as the argument:

```
python.exe main.py --json=config.json
```

The next subchapter covers the detailed algorithm implementation. See Figure 7 for an overview.

3.1.6.1 Algorithm Part: Configuration

At the program start, the options in the configuration file are evaluated and the further program execution is configured. Then file properties like the number points and dimensions and whether the file content is transposed, are kept in the database class. Before further proceeding, a check is performed whether only DBSCAN should be run. The execution causes the program to terminate afterwards.

Further configuration is SUBSCALE specific. Some parameters like `use_evenly_distributed_slices` or `is_multiprocessing`, are set to default values or are not contained in the configuration file because they are experimental and should not be used in practice. The next parameters, point labels and min and max signatures are computed by the SUBSCALE module.

Since the Pascal triangle is used in different execution branches it is also initialized once, early in the program.

Depending on the choice of execution of the main computational part of SUBSCALE, the pivoted computation is performed either by function `start_pivot_execution` in `SUBSCALE.py` or the pivot-less computation by function `start_non_pivot_execution` is run. The pivot execution calculates one Core Set at a time, while for pivot-less computation all Core Sets are calculated beforehand. Each function is designed to perform the calculation with multiprocessing. Therefore, the number of dimensions that are computed at once are set by the variable `dims_at_once`. The counting of those is carried out by function `calculate_dimensions_at_once`, and is dependent of on the alignment of points and dimensions of the input file. At the moment all dimensions are processed by one process because multiprocessing proved itself not be a viable solution in Python.

If slices are used, then the current output files have to be read and refined before saved to disk. If no slices were used before, then the result is written directly to disk.

Finally, DBSCAN can be executed on the resulted cluster candidates.

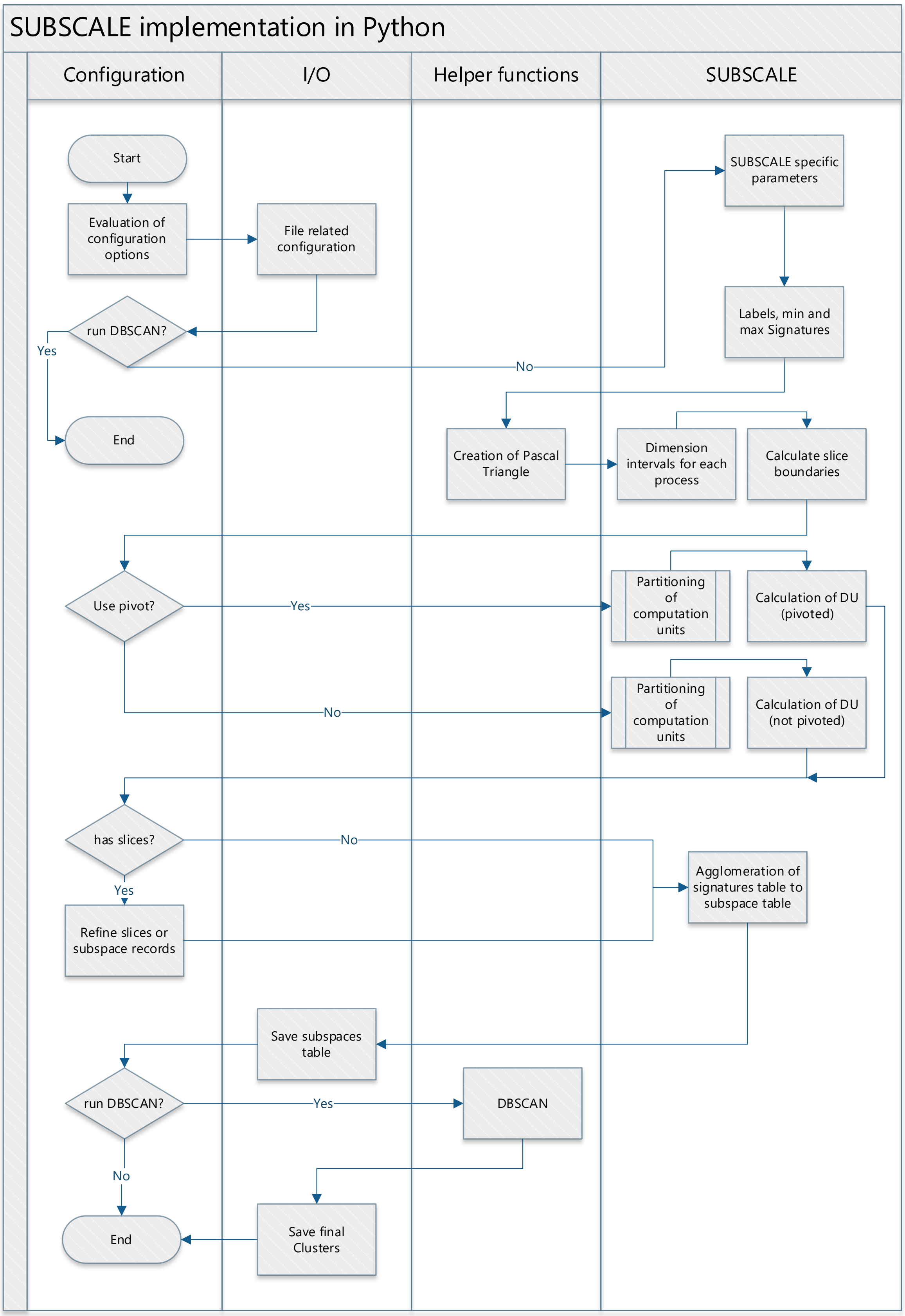

SUBSCALE implementation in Python
Configuration
I/O
Helper functions
SUBSCALE
Start
Evaluation of configuration options
File related configuration
run DBSCAN?
Yes
No
End
SUBSCALE specific parameters
Labels, min and max Signatures
Creation of Pascal Triangle
Dimension intervals for each process
Calculate slice boundaries
Use pivot?
Yes
No
Partitioning of computation units
Calculation of DU (pivoted)
Partitioning of computation units
Calculation of DU (not pivoted)
has slices?
No
Yes
Refine slices or subspace records
Agglomeration of signatures table to subspace table
Save subspaces table
run DBSCAN?
Yes
DBSCAN
No
End
Save final Clusters

Figure 7: Flowchart of the Python implementation of SUBSCALE

3.1.6.2 Computation of Core Sets

A Core Set is a one-dimensional cluster of `min_point` sized set of points within ϵ (epsilon) distance. The function `create_core_sets_per_dimension_v1` calculates all Core Sets for a given dimension for given `min_points` and `eps` (ϵ) parameters.

To exploit some optimization, first all of the projected points to dimension d_k are sorted. Then, in **step 2**, two points P_i, P_j are picked pairwise, so that $P_i \neq P_j$. If the Manhattan distance $Dist(\ P_i, P_j\ ) \leq \epsilon$, then P_j is added to the current Core Set CS_i and the next P_j is picked. This step is repeated until either all points P_j have been processed or $Dist(\ P_i, P_j\ ) > \epsilon$. In both cases two requirements have to be satisfied before CS_i and P_i can be added to all Core Sets:

- $size(CS_i) \geq min\_points\ - 1$
- $CS_i \not\subset CS_{i-1}$, where CS_{i-1} is the Core Set from the previous iteration of the algorithm

E.g., in Figure 10 there are two Core Sets cs_1:$\{P_2,\ P_3,\ P_6,\ P_{14},\ P_{10},\ P_5,\ P_{11}\}$ and $cs_2\{P_3,\ P_6,\ P_{14},\ P_{10},\ P_5,\ P_{11}\}$. The first requirement for the new Core Set cs_2 is fulfilled as the size of $cs_2 = 6\ \geq \text{min\_points} = 4$. Yet, the second requirement fails because cs_2 is a subset of the old Core Set cs_1. For the implementation of the second requirement the ID of the last element in CS_{i-1} is kept in a variable (in the current example it is P_{11}) and is compared in the next iteration of this algorithm. The algorithm termination is reached when P_j becomes the last element in d_k because any new CS_i will be a subset of CS_{i-1}. In case that P_j is not the last element, P_i is set to the next point and the algorithm resumes at step 2.

The flowchart in Figure 8 depicts an overview for this algorithm.

One more performance improvement was made by recycling CS_{i-1} for CS_i: Instead of measuring the distance from current P_i to all P_j, all points from CS_{i-1} were used in the new Core Set, except for the first point, so that the next index value j of P_j in step 2 is set to $j = i + size(CS_{i-1}) - 1$, where *i* is the index value of P_i. This concept was further elaborated in function `create_core_sets_per_dimension_v2` to make the index *j* independent of the size of a Core Set: *j* is raised continuously in step 2 and before increasing *i* (for P_i), the first element in CS_i is removed. The latter property is of benefit for adding CS_i to all Core Sets since, by this means, the first point in CS_i is always P_i and the last is P_j.

Since only the current Core Set is required to calculate a Dense Unit, there is no need to keep all Core Sets in memory. Therefore, the computation of Dense Units with pivot element is based on the concept of this function version.

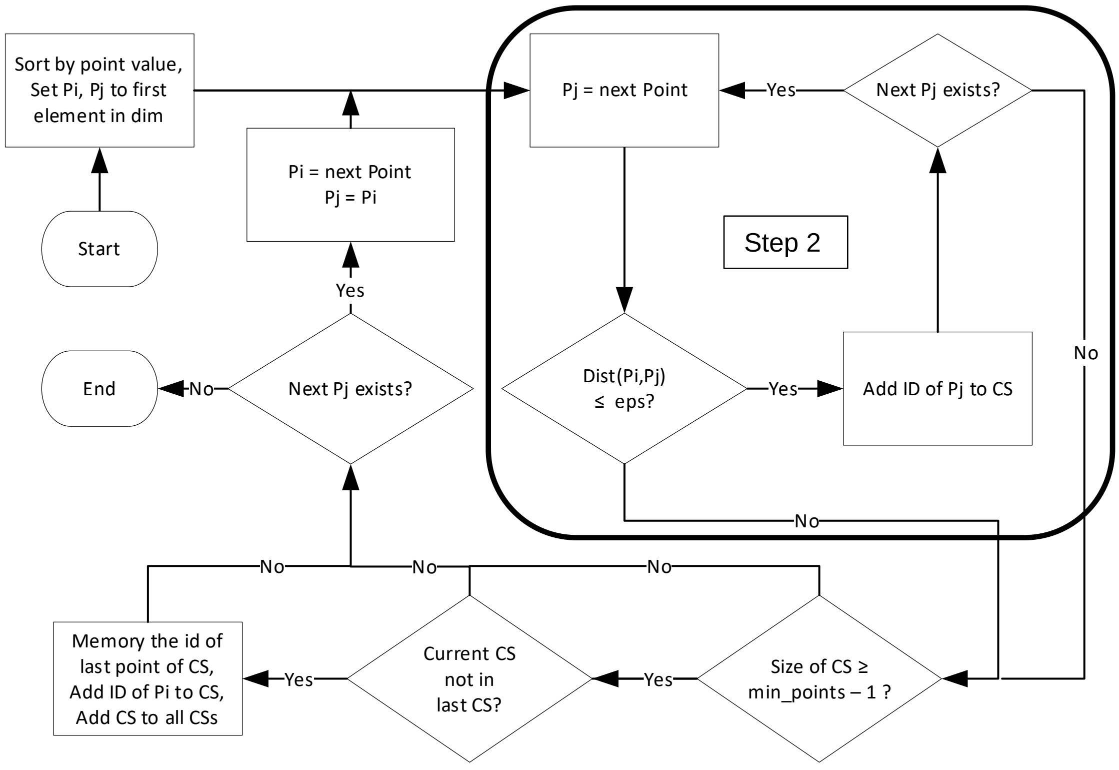

Sort by point value, Set Pi, Pj to first element in dim
Start
Pi = next Point Pj = Pi
Yes
End
No
Next Pj exists?
Pj = next Point
Yes
Next Pj exists?
Step 2
Dist(Pi,Pj) ≤ eps?
Yes
Add ID of Pj to CS
No
No
No
No
No
Memory the id of last point of CS, Add ID of Pi to CS, Add CS to all CSs
Yes
Current CS not in last CS?
Yes
Size of CS ≥ min_points – 1 ?

Figure 8: Computation of all Core Sets per dimension.

3.1.6.3 Computation Mode of Dense Units Without Pivot Element

The pivot-less implementation of the Dense Units creation function requires a prior calculation of CSs. Since the total number of CSs is much less than the total number of Dense Units (2), they are computed all at once. The function `createDU_and_sig_map_no_pivot` creates the Dense Units and puts them into the hashable signatures map. In Python this specific map is a Python built-in type "dictionary" of signatures $\{sig: (\mathcal{U}_i, [d_j, \ldots])\}$, where $\mathcal{U}_i$ is the Dense Units, $sig = sum(\mathcal{U}_i)$ and $[d_j, \ldots]$ is the list of dimensions that contain $\mathcal{U}_i$. sig is the key of the dictionary and the tuple $(\mathcal{U}_i, [d_j, \ldots])$ is the associated value. Due to Python's *Global*

Interpreter Lock (GIL), all built in types, “including critical built-in types such as dict“ [32], are protected against concurrent access. This means, concurrent writing from different threads to different keys (in this specific implementation the signatures) is possible. However, if there is a chance, that the same key is targeted for writing access, then if must be locked by the first accessing thread.

Depending on the number of processes, the function determines the interval of the dimensions of interest within the precalculated CSs. CSs from the selected interval are processed, and the resulted Dense Units are available for further computation. In case that slices are used, only Dense Units that are located within the defined bounds in the current function call are added to the *signatures*-dictionary. The `add_to_dict_same_signatures2` function requires as arguments the current Dense Units $\mathcal{U}_i$, current dimension d_k, and the calculated signature for the Dense Units. If a signature exists, the current dimension d_k is added to the list of dimensions $[d_j, \dots]$. Otherwise, a new entry $sig\colon (\mathcal{U}_i, [d_k])$ is created. See Figure 9 for a schematic explanation.

3.1.6.4 Computation Mode of Dense Units With Pivot Element

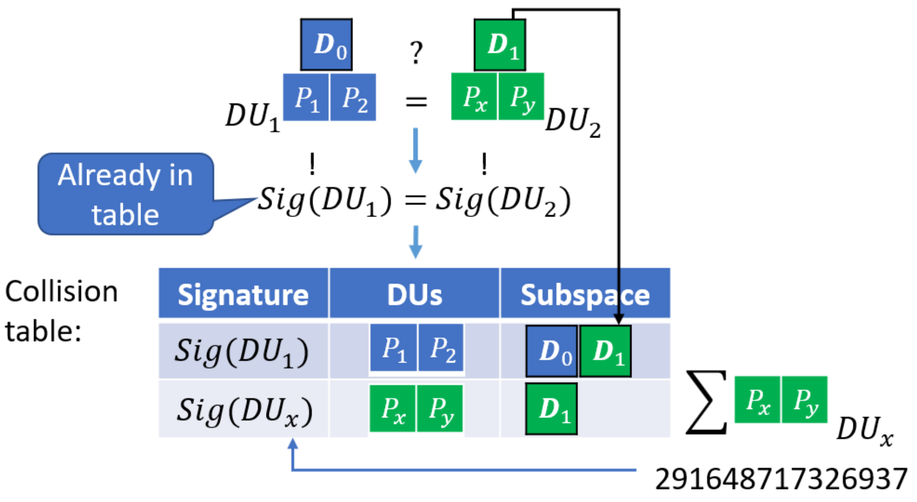
The efficiency Dense Units computation can be improved by utilizing the fact, that some Dense Units are already calculated in a previous Core Set. These Dense Units are portioned by a pivot element that refers to the last point in the previous Core Set.

The function Parameters such as the process number, the dimensions and the slice boundaries are passed to function `pivot_creation` in `SUBSCALE.py` to provide a parallel execution in the better performant version as in the pivot less version.

In contrast to the latter one version, the Core Sets are not calculated beforehand. Instead, only one Core Set is calculated first. Then all Dense Units are calculated from it and any Dense Unit with a signature within current bounds is added to the signatures dictionary. The calculation of Core Sets in the `pivot_creation` function implies that points can be loaded when the function is called. The loading function is dependent on the `transpose` parameter. If transposed is set to true, the whole row is read from the data set file and all points from a dimension are stored in memory. Otherwise, the column wise reading function `loadtxt` from the NumPy library is used to load only the necessary rows in the selected column. Thereby memory is saved because each process or thread loads only the required partition of points from file.

The pivot optimization is only possible if the pivot index, that is, the index of the last element from the last point in previous Core Set is in its current Core Set $\geq \mathrm{min\_points} - 1$. Otherwise, all Dense Units from the current Core Set have to be calculated anew. The calculation of the pivoted elements is based on the fact that if the current Core Set contains at least a `min_points` sized subset of same points from the previous Core Set, then those Dense Units have not to be calculated again. However, the points from the last Core Set are still used for the combination of Dense Units from the current Core Set with points that are disjunct. An illustration for comprehension is displayed in Figure 10 where Core Set cs_3 is partitioned in two subsets cs_{3L} and cs_{3R} by the pivot element $pivot = 5$. The point that the pivot element refers to is the last point P_{11} in the previous Core Set cs_1. In cs_3 however, the index of the fifth element is 4. So, $index(P_{11})\ in\ cs_3 = 4\ \geq min\_points\ - 1 = 3$. For cs_{3R} the normal calculation of Dense Units is mandatory. But in addition, since the points within cs_{3L} are a subset of Core Set cs_2, the Dense Units from these points have not to be calculated again. Yet, these points in cs_{3L} have to be used to calculate Dense Units together with points from cs_{3R}. There are up to "`min_points` – 1" points from cs_{3L} which can be harnessed to form min_points sized Dense Units from the whole Core Set cs_3. In the first step, the algorithm creates 1 element sized partitions from cs_{3R} and combines them with 3 element sized partitions from cs_{3L} (2). In figure 10 the iteration stops already at the second step (3), because cs_{3R} contains only 2 points. In total the calculation results in only 30 Dense Units; 20 from the first iteration (2) and 10 from the second (3) in contrast to 35 from all points in cs_3 (1).

Figure 11 depicts the performance gain from the pivot calculated Dense Units. For small `min_points` values, small sample and dimension size the pivot version has some overhead that makes the calculation less efficient than without pivot element.



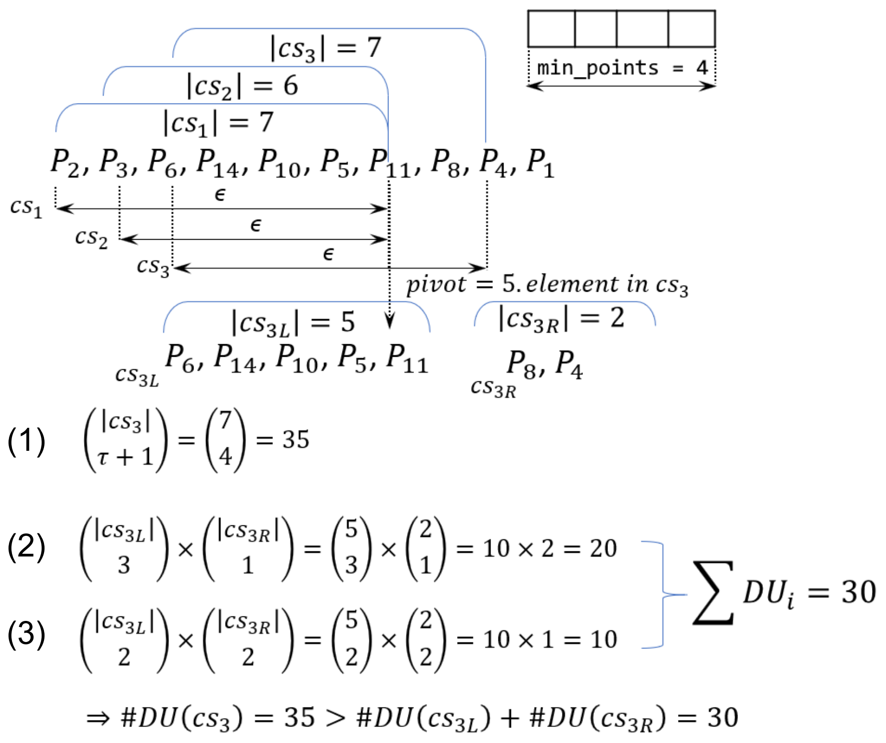
D_0
?
D_1
DU_1 P_1 P_2 = P_x P_y DU_2
Already in table
! !
$Sig(DU_1) = Sig(DU_2)$
Collision table:
Signature | DUs | Subspace
$Sig(DU_1)$ | P_1 P_2 | D_0 D_1
$Sig(DU_x)$ | P_x P_y | D_1
$\sum P_x P_y DU_x$
291648717326937

Figure 9: Adding new entries or dimensions to the signatures table a.k.a. collision table.

Two Dense Units from different dimensions are compared.

Dimension D_1 is added to the first entry of the subspace block in the collision table.

The second entry in the table is newly created: The signature $Sig(DU_2)$, the points P_x, P_y from Dense Unit DU_2 and the associated dimension D_1 are added.



min_points = 4
$|cs_3| = 7$
$|cs_2| = 6$
$|cs_1| = 7$
$P_2, P_3, P_6, P_{14}, P_{10}, P_5, P_{11}, P_8, P_4, P_1$
cs_1 ϵ
cs_2 ϵ
cs_3 ϵ
$pivot = 5.element\ in\ cs_3$
$|cs_{3L}| = 5$
$|cs_{3R}| = 2$
cs_{3L} $P_6, P_{14}, P_{10}, P_5, P_{11}$
cs_{3R} P_8, P_4
(1) $\binom{|cs_3|}{\tau+1} = \binom{7}{4} = 35$
(2) $\binom{|cs_{3L}|}{3} \times \binom{|cs_{3R}|}{1} = \binom{5}{3} \times \binom{2}{1} = 10 \times 2 = 20$
(3) $\binom{|cs_{3L}|}{2} \times \binom{|cs_{3R}|}{2} = \binom{5}{2} \times \binom{2}{2} = 10 \times 1 = 10$
$\sum DU_i = 30$
$\Rightarrow \#DU(cs_3) = 35 > \#DU(cs_{3L}) + \#DU(cs_{3R}) = 30$

Figure 10: Pivot optimization for the computation of Dense Units

However, with raising `min_points` values the pivot version is always faster (blueish, lower lines in Figure 11). The measures in Figure 11 are constrained by the system's memory capacity.

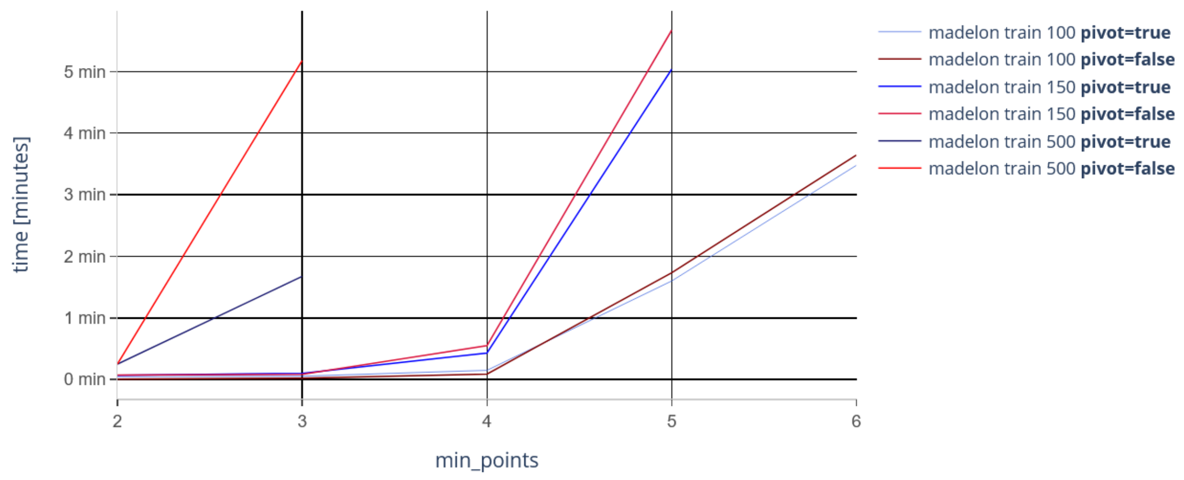


Figure 11: Execution time in different Dense Unit computation modes w.r.t. min_points and size of data set.

3.1.7 Computation Algorithms for Dense Units

Dense Units are computed in several functions, with different purposes. They are united in their purpose, the computation of one distinct subset from all possible $\binom{n}{k}$ combinations without repeats a.k.a. *k*-element sized subsets of an *n*-element set. The calculated subset consists of numbers in a specific order. It is applied as index values on point IDs in a Core Set and thereby mapped to corresponding Dense Units. For example, if a Core Set consists of the following point IDs $[10, 24, 31, 40, 51]$ and $\text{min\_points} = 3$, then based on the ordering function, the first subset would contain an indexing of numbers $[0, 1, 2]$ and the second $[0, 1, 3]$. The resulting Dense Units would be $[10, 24, 31]$ and $[10, 24, 40]$.

There are three sequential computations: One of them is a lexicographic computation and two are colexicographic computations. Also, there is one non-sequential computation which is a colexicographic computation. See Table 2 for an overview.

Function name	sequential	ordering
`create_subsets_lex`	+	lexicographic
`subsequent_colex_index_combination`	+	colexicographic
`subsequent_colex_index_combination_2`	+	colexicographic
`combinadic`	-	colexicographic

Table 2: Functions and their properties for Computation of Dense Units

For pragmatic reasons the function `create_subsets_lex` was calculated first, because the lexicographic ordering is familiar and thus easy to implement and to test but it has a grave downside. Each successor requires a predecessor as input. For parallel execution however, a random and thus a non-sequential access is required.

The non-sequential variant of the colexicographic calculation of Dense Units is performed by the function `combinadic`. A *combinadic* “represents every nonnegative integer *N* uniquely” as a sum of *k* decreasing binomial coefficients [33, p. 360]. This becomes useful when a designated combination N_i of $\binom{n}{k}$ combinations in total has to be calculated without knowledge of all i $(\text{for } 1 \leq i < N < \binom{n}{k})$ predecessors.

For the computation of a Dense Unit the formula (3) for *combinadics* has to be interpreted as following: N_i is the index value of the Nth number of combinations of $\binom{|CS|}{min\_points}$ combinations in total in a colexicographic order. $k = min\_points$, $|CS| > c_{k-0} > c_{k-1} > \cdots > c_{k-j} > c_1$ and $0 \leq j < k$. The c_{k-j}'s represent the point IDs in a Dense Unit.

$$N_i = \binom{c_{k-0}}{k-0} + \binom{c_{k-1}}{k-1} + \cdots + \binom{c_1}{1} \qquad (3)$$

For example, to calculate the 6th combination of $\binom{5}{2} = 10$ combinations, one of the sequential computation functions is required to be executed for 6 times. Each result is used as a new input to get the subsequent combination. With the `combinadic` function the 6th combination / 6th Dense Unit with point IDs [2, 3] can be calculated directly: $N_i = 5 = \binom{3}{2} + \binom{2}{1} = 3 + 2$, where N_i is the index value 5 (or 6th element).

Beginning with the random calculated Dense Unit as *combinadics* any next Dense Unit can be calculated sequentially. This reduces the computation time from $O(k^2 \cdot log_2\, n)$ to $O(1)$ for each subsequent Dense Unit.

Utilizing the principle of the Pascal’s triangle, the computation time for combinadics could be reduced to $O(k \cdot log_2\, n)$: Instead of calculating the binominal coefficient, the result of the number of combinations can be taken from a precalculated look-up table similar to the Pascal's triangle in $O(1)$ asymptotic run time complexity. The number of legs corresponds to the `min_points` parameter of SUBSCALE and the number of entries per leg is defined by the number of points in the processed data set. The values in the Pascal's triangle are calculated successive based on the predecessors. E.g., in order to calculate the value "15" in Figure 12, the predecessor value of the same leg "10" and the respective predecessor value of the previous leg "5" is required. The zeroth leg (on the left side of the triangle, that consists only of values = 1) from the theoretic form of the triangle is not required for the calculation. The values are accessed by getting the index n in the kth ell. For the number of combinations $\binom{6}{2}$ in Figure 12, $n = 6, k = 2$, and the result is 15.

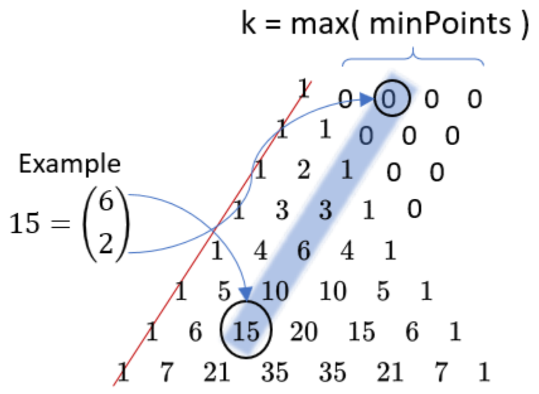


Figure 12: Accessing the element at index 6 in the second leg of the Pascal triangle similiar table with the binomial coefficient. "k" in $\binom{n}{k}$ can become at most of min_points size, n at most the total number of points in the processed data set. There is no utilization of the left leg.

While the look-up for the number of combinations is fast, it has to be checked if the result is wanted. For this purpose, each summand $\binom{c_{k-j}}{k-j}$ of N_i is checked in the function `find_comb_smaller_n_pscal_triangle` such that $N_i = N_{i,j} < \binom{c_{k-j}}{k-j} < N_{i,j+1}$. And for each successive summand $N_{i,j+1} = N_{i,j} - \binom{c_{k-j}}{k-j}$ as long as $j < k$. The search for $\binom{c_{k-j}}{k-j}$ is performed with the *binary search* approach in this function, which results in a runtime of $O(log_2 n)$ with $0 \leq N_i \leq \binom{n}{k}$ for each N_i. In total the runtime is $O(k \cdot log_2 n)$ for *k* summands for one Dense Unit.

A further explanation for the stated run times is given by Nicolas Kiefer in his master thesis "Datenparalleles Subspace clustering mit Grafikprozessoren" [27]. Each Next Dense Unit has to be calculated with the `subsequent_colex_index_combination_2` function in the colexicographic order, to be consistent with other random calculated Dense Units. The original function `subsequent_colex_index_combination` (without the suffix "_2") was taken from [27] where the output of IDs in a Dense Unit was arranged in a descending order [27, p. 15], while for better readability and consistency ascended arranging of IDs is favored.

The synergy of random and sequential access would lead to a significant performance improvement when both functions would be paired with parallel execution. However, on applying parallel execution, the execution time of the Python program increased for each min_point exponentially in comparison to the . Therefore, only the colexicographic, sequential computation is used. Further information on the parallel execution topic can be found in chapter Multiprocessing.

3.1.8 Adding new Entries or Dimensions to the Signatures Table

The data structure for storing new Dense Units is the built-in Python *dictionary*. New entries are added to the *signatures* table as proposed in Figure 9. The code in Listing 1 for adding new entries is straight forward: Each signature in the signatures dictionary is associated with a Python built-in non mutable type *tuple* of a Python built-in type *list* of Dense Units and a *list* with the respective dimension.

```
SubscaleDict.signatures[signature] = (dense_unit_as_indices, [dimension])
…
```

Listing 1: adding Dense Units and dimensions to signatures dictionary

In Listing 2, when a signature that is already contained in the dictionary is found, a check to prevent duplicate dimension entries is performed, the new dimension is added and the dimensions list `dims` is sorted (lines 2 - 4). When non-multicore execution is chosen, this part of algorithm could terminate at line 4.

```
1. dims = SubscaleDict.signatures[signature][1]
2. if dimension not in dims:
3.     dims.append(dimension)
4.     dims.sort()
5.     SubscaleDict.signatures[signature] = (dense_unit_as_indices, dims)
6. …
```

Listing 2: adding new dimension to signatures dictionary

The reference to the dimensions *list* in second position of the *tuple* is preserved and the new dimension is added to it. However, referencing the dimensions list fails, when the signatures dictionary was created as a proxy structure with the multiprocessing manager. The `dims` list is in this case a copy of the list in the second position of the tuple. To make the code both with single process and multi process mode operative, a tuple with the altered `dims` list has to be inserted anew (line 5).

3.1.9 Mapping Dense Units to Subspaces

For the final clustering the output of the SUBSCALE algorithm is required to be in the structure of cluster candidates. Each candidate consists of dimensions and point IDs: $[d_i, d_j, \dots] - [id(P_k),\ id(P_l), \dots]$ where each subscripted index character is unique. This format is used as output for consistency reasons because it is used by other SUBSCALE implementations as well.

The method `add_to_dict_same_subspaces` agglomerates Point IDs from the collisions table to subspaces table.

3.1.10 Final Clustering with DBSCAN

At this point the maximum subspaces are identified, but the task of the SUBSCALE algorithm is to find the maximal subspace clusters [17, p. 215]. To accomplish this task a full dimensional clustering algorithm is used. The algorithm of choice is DBSCAN from the *scikit-learn* machine learning library for Python [34]. It is a density-based clustering algorithm that finds clusters of arbitrary shape.

3.1.10.1 Parameters

DBSCAN requires `eps` and `min_points` as input parameters and a list of k-dimensional samples. Therefore, the point IDs from the cluster candidates are processed by the function `run_DBSCAN` to load the associated points. An optional function parameter `full_dimensional_input` can be toggled to switch the full dimensional data scan on, in order to contain all dimensions of the associated points. Otherwise only the dimensions that are contained in the according subspace are used for clustering.

In case that there are no clusters found, a different metric can be used. The function `minkowsky` provides the Minkowsky distance with the variable `_p` inside the function. To put a reasonable weight to the distance measure `_p` can be adjusted to a suitable value. The function is utilized by NumPy for performant distance measure by processing two passed points as vectors.

3.1.10.2 Output

The output of DBSCAN is formatted as S, P_i, c -records such that $S^k = (d_1, \ldots, d_k)$ is the subspace containing k dimensions of the associated, adjacent points. K is also the filename for all k-dimensional subspaces. $P_i = [P_i^1, P_i^2, \ldots, P_i^k]$ is a point that is represented as a *k*-dimensional vector if the parameter `full_dimensional_output` is set to false (otherwise P_i has the full dimensional length), and c is the number of the cluster that P_i is assigned to. Points that are identified as noise are marked with $c = -1$. There are multiple records for each Subspace S with different P_i's. For example, a record of file `2.csv` can have the following records:

- $[2, 3, 7] - [0.123,\ 0.841, \ldots], 0$
- $[2, 3, 7] - [0.694,\ 0.191, \ldots], 0$
- $[2, 3, 7] - [0.3446,\ 0.1267, \ldots], 1$
- $[2, 3, 7] - [0.3446,\ 0.1267, \ldots], 1$
- $[4, 3] - [0.3446,\ 0.1267, \ldots], -1$
- $[4, 3] - [0.961,\ 0.971, \ldots], -1$

In this example the subspace $[2, 3, 7]$ contains clusters 0 and 1, each with two points per cluster. Whereas, the subspace $[4, 3]$ has no clusters at all. Both points are outliers w.r.t. `min_points` and `eps` parameters.

3.1.10.3 Statistics

Before the function terminates a count of clusters per subspace is performed and for each cluster the size is noted. The output of this counting is in the form of $S =$

$(C_i: n_i, \dots)$ per record, such that S is the subspace, i is the number of the cluster and n_i is the number of points in the cluster. For example, the entry $[1, 4, 5, 13] = \{0: 3, 1: 3, -1: 4\}$ specifies that in the subspace $[1, 4, 5, 13]$ there are 2 clusters found: The first cluster with 3 points and the second one with 2 points. The cluster "-1" denotes that 4 points are outliers that cannot be clustered at all. All records are written into the file `count_clusters.txt` in the DBSCAN directory.

3.1.11 Execution Speedup and Optimization

In practice, when Python is used in computation intensive tasks, it is usually mainly utilized to connect not speed relevant parts of Code with computationally intensive parts. The latter decide the total execution time of the program. For this purpose, libraries like NumPy, *Just In Time* compilers (*JIT*), or alternative compilations are used to speed up the program.

Since SUBSCALE processes a large amount of data, execution time is an issue. In order to counter the slow execution time, different approaches are taken.

3.1.11.1 NumPy

The NumPy library is one of the most supported libraries in other Python compilers and *JIT*. In the implementation it is used to wrap Python lists into NumyPy arrays as input for DBSCAN. By doing that, NumPy determines automatically the inherent type of the elements in the array for the cost of having them of uniform type. The fixed data type has the advantage of consuming less memory and make execution faster, when NumPy or other optimizations are used.

3.1.11.2 PyPy

Due to Pypy's ability to generate machine code that is run directly on hardware, instead of interpretable bytecode that is run by a virtual machine (as it is done with standard CPython VM), performance improvements are achieved: "The basic approach of a tracing JIT is to only generate machine code for the hot code paths of commonly executed loops and to interpret the rest of the program" [35]. According to PyPy's official "speed center" website the average geometric speedup in benchmarks is 0.23 or PyPy is 4.4 times faster than CPython [36]. However, PyPy requires its own implementations of libraries [37] so that the choice of libraries, manual integration due to error prone automated installation and compatibility issues have to be taken into consideration. For example, a precompiled NumPy library version had to be installed manually for PyPy because the automated building of the *wheel* package with the “pip installs packages” (*PIP*) package manager failed. Further problems emerged when *sklearn* library for the contained DBSCAN algorithm had to be installed. The compatibility failed, so that eventually the final clustering could not be tested with PyPy. Another issue is that there exists only a 32-bit precompiled Numpy library for PyPy. Therefore, to preserve compatibility, the 32-bit PyPy version had to be put to use. This resulted first in aborted program execution once the memory consumption exceeded 4 GB RAM. Since, NumPy is only used for I/O operations, a workaround for reading the whole data set without resorting to this library was established. The trade-off for sacrificing NumPy functionality in this implementation is that the whole data set is read at once and kept in memory for the whole program runtime.

3.1.11.3 Numba

Numba is another JIT that works well on loops and NumPy arrays. Since it features GPU support it is also attempted to run SUBSCALE. Numba has two compilation modes that affect performance. The best performant *nopython* mode is activated by annotating a function with the `@njit` decorator, which causes the function to be entirely compiled into machine code. However, compilations fail when non supported structures or functions come across. For example, the Python built-in function `sum` is not supported for lists. To discern non compatible code, Numba can be run in *object* mode utilizing the `@jit` decorator. In this case Numba will automatically identify functions and loops that can be run without the Python interpreter. Since lists are heavily used in this SUBSCALE implementation as the container of choice for collections, the *object* mode was harnessed for the most functions. Functions that contain only operations on lists and no other called functions are decorated with *nonpython* mode. The `SubscaleDict` class in file `SUBSCALE.py` is not annotated at all, because it would require comprehensive typing of functions, variables and a change of the structure of the signatures and subspaces dictionaries. However, the execution time did not decrease. On the contrary, with longer computation it increased. The suspected cause for this is probably that not all variable types are recognized, when multidimensional or nested structures are used. Also, functions that use the efficient *nonpython* mode (the mode of choice for maximum performance improvement) cannot operate on Python built-in collections of *lists, tuples, sets* structures. In such a case the collection is copied and its contents are converted to any of the corresponding reflected PyPy structures like a *reflected list*. This behavior nullifies the profit or even generates more overhead than benefit. The remedy comes with using NumPy *arrays* instead of *lists*. However, this would require heavy dependencies to NumPy and the code has to be rewritten at scopes that feature lists.

3.1.11.4 Multiprocessing

Multiprocessing is a problem field of Python, because in its beginnings this was not meant to be available. The background for this reason is the integrated garbage collection und compatibility to C programs. To exploit multiprocessing the *multiprocessing* Python package was harnessed. This package effectively side-steps the Python mutex *Global Interpreter Lock* (*GIL*), a security mechanism to provide compatibility to C and Fortran Code, by using subprocesses instead of threads, but forces utilization of specific data structures. E.g., for shared access and in order to obtain a proxy of the regular Python dictionary, the latter object has to be created with the process manager first. However, speed tests for the implementation resulted in a high overhead administrative management effort. In Figure 13 the execution time is depicted. It increases to a high magnitude when Dense Units are written into the signatures and subspaces dictionaries.

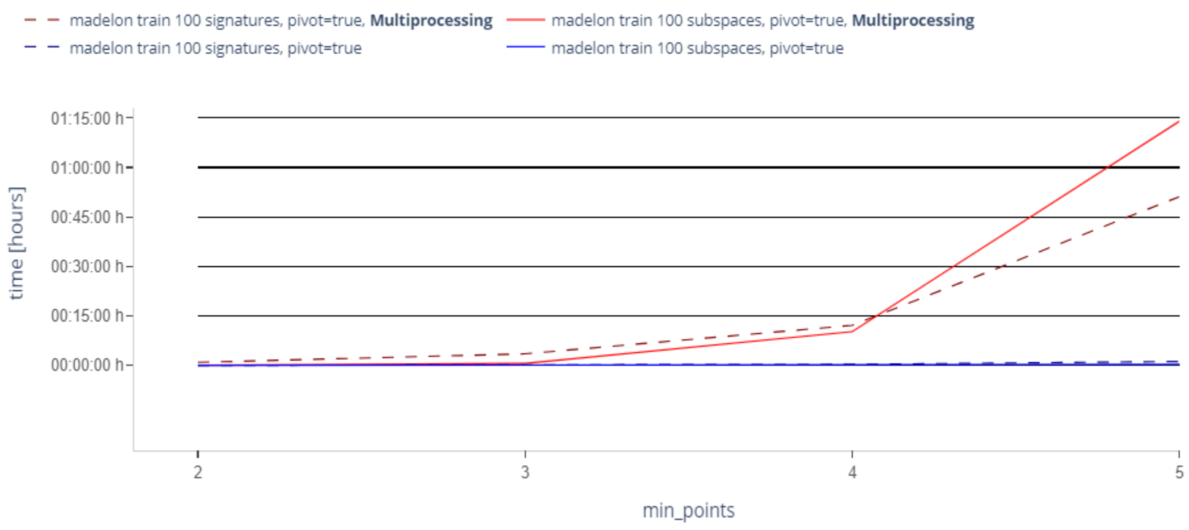


Figure 13: Execution time for inserting elements into the signatures and subspaces dictionaries with and without multiprocessing in Python, eps=0.03

In particular, with raising min_points, the amount of subspaces entries in the dictionary increases more than then amount of subspace entries in according dictionary (see Figure 14).

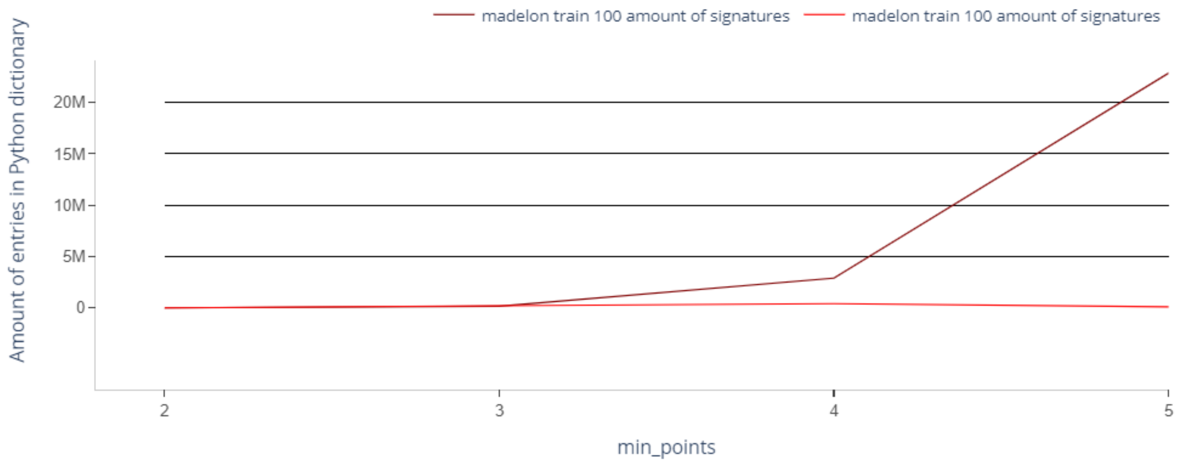


Figure 14: Amount of signature entries vs. subspaces entries in dictionaries, eps=0.03

However, the access time remains linear and independent of multiprocessing, when the dictionaries are accessed for reading in case that the contained data is written to disk (see Figure 15).

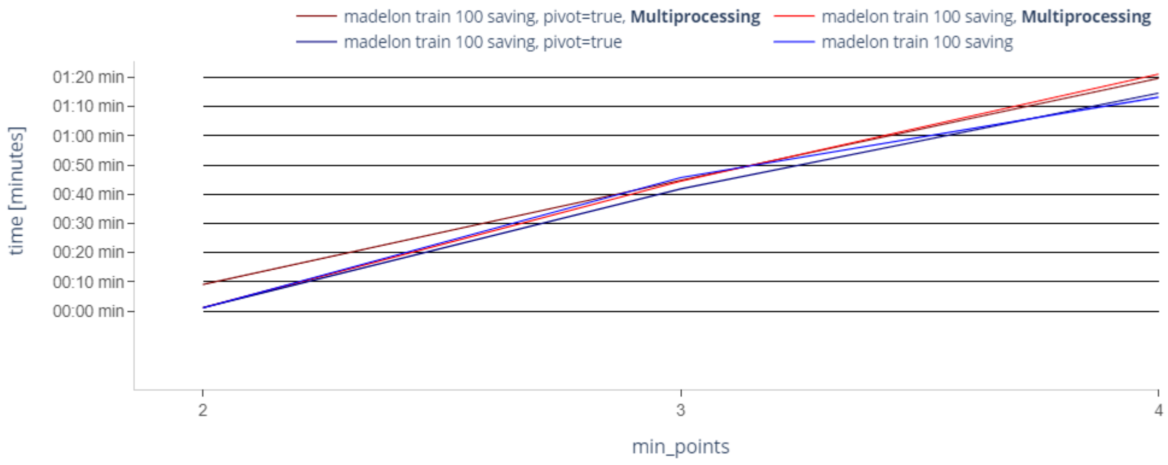


Figure 15: Duration of saving the dictionary in different modes and data set size, eps=0.03

3.1.12 Unit Tests

Input and output data processing is one of the first precise software requirements inferred from the program specification. The algorithm can only be profiled and evaluated if there is data available that can be harnessed. On top of that, some errors and bugs might only emerge when real-world data is used thereon. For example, some edge cases might be omitted because they were not identified during the conception phase of the test cases.

For those reasons the unit tests for the IO.py file are implemented first and before the implementation of the database code. All other tests relate to the SUBSCALE algorithm. They are created during further implementation.

In order to achieve a better structured and validated project, integration tests are automated with the *pytest*-library [38] and embedded in the PyCharm IDE.

3.2 Python as Front End

As table 3 shows, despite utilizing different speed up technologies and techniques no significant increase in execution time was achieved. No technology could overtake the Java nor C++ implementation.

Interpreter/JIT / Component	Python	PyPy	Numba	Java	*C++
Dense Units creation	00:35.795	00:07.110	04:42.959	-	00:15.878
Signatures table	00:46.857	00:53.004	03:15.918	-	00:01.582
Subspaces table	00:56.793	00:13.292	00:19.363	-	00:00.693
Saving subspaces	00:18.424	00:16.797	00:11.147	-	00:00.179
total time	02:37.495	02:48.339	06:42.096	00:30.235	00:18.396

Table 3: Execution time in minutes of 4_madelon_train_100 data set, min_points=5, eps=0.03 with different technologies.

The total time is the sum for all program operations, not limited to the listed ones in this table.

***C++: Execution with 100 slices. Other benchmarks used only 1 slice.**

Yet, in practice, Python is often used to connect fast executable compiled machine code. Since, there exists already a faster implementation in C++ [28] it seems natural to use that instead of a Python implementation. For this purpose, Python is used as a frontend with *Jupyter Lab* that facilitates the algorithm configuration, visualizes and utilizes the results.

3.2.1 SUBSCALE-GPU Extensions

As table 3 hints, Nicolas Kiefer's C++/GPU SUBSCALE-GPU implementation [28] features the fastest execution time. To verify this statement, it was necessary to enhance the data output of the implementation. Originally, the data was outputted to a single file for cluster candidates (a.k.a. subspaces – point id records) and a single file for clusters, classified by DBSCAN. Each file record has the form $[S] - [id(P_1), id(P_2), \dots]$, where S is the subspace of multiple dimensions and $id(P_1)$, is the ID of the point. For example, $[2,3] - [7,11,53]$ denotes that in subspace $[2,3]$ the 7th, 11th and 53rd points are clustered together. To achieve an insightful comparison in validation and execution time, the output of the SUBSCALE-GPU implementation was provided in the same form as in the SUBSCALE-extended and Python implementation. The structure of the records stays the same but subspaces are agglomerated by their size. E.g., the output of maximal clusters from DBSCAN for the subspace $[2,3]$ is added to file `2.csv` in the `clusters` subdirectory in the `results` directory of [28]. The same applies for the file and record structure of candidate clusters.

3.2.1.1 Configuration File

Both output format extensions, for clusters and candidates can be switched independently and inclusive with other output options in the configuration file of [28]. Three options are added to it.

- `saveCandidatesMultiFiles`

  Saves output from DBSCAN to one single file.
- `useDBSCAN_MultiFiles`

  Finds maximum clusters. Requires: `saveCandidatesMultiFiles` to be set to `true`.
- `saveClustersMultiFiles`

  Saves output from DBSCAN in files grouped by the amount of dimensions.

3.2.1.2 Output Methods

The corresponding output methods are

- `CsvDataHandler::writeSubspacesToFile`

  Creates cluster candidates

- `ISubscale::calculateAllSlices2`
  Logs the number of Dense Units and subspaces per slice
- `Clustering::calculateClusters2`
  Outputs the slices as files after final clustering

To save programing time and keep the added code congruent to the original code, a simple implementation with only two lines of code change were done: While in method `calculateClusters` all maximum (final clusters) were put into the C++ standard library structure `std::vector<Cluster> clusters`, in the second implementation the cluster size was mapped into the two dimensional, built-in map structure `std::map<size_t, vector<Cluster>> clusters_dimensions2`. This enabled a convenient write access by calling `clusters_dimensions2[dimensions.size()].push_back(cluster).`

The subfolder directories in `results` have neither to be created nor deleted in advance. Instead, they are created automatically when the “multi files” options in the config file are switched on, before the output data is processed into files.

3.2.1.3 Logs

Additional counting was logged for the calculation of the number of records for different processes. The logs have found its application in the front end to illustrate the performance of different algorithm steps.

For each candidate file that is written with the output method `writeSubspacesToFile` into the `candidates` subfolder, this method also records the number of entries for the candidate file into the file `SubspaceClusterFilesSizes.txt`.

In method `ISubscale::calculateAllSlices2` in addition to the functionality of the original method, the number of Dense Units per slice is written out to the file `entriesPerSlice.txt` while the number of subsets per slice is written out to `condensedEntriesPerSlice.txt` of the `results` directory.

3.2.2 Visualization Tools

According to the “2020 Kaggle Machine Learning & Data Science Survey” top visualization tools are Matplotlib, Seaborn and Ploty and ggplot [39]. The first three tools are used for data visualization in the front-end for this thesis.

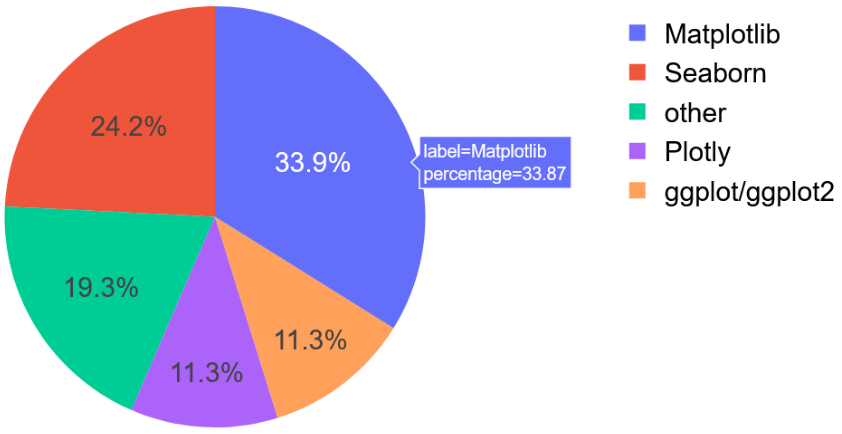


Figure 16: Visualization tools used on daily basis by data scientists of the Kaggle community. The label is displayed with Plotly in Jupyter Notebook on mouse hover over the blue piece of the pie.

Matplotlib is the most popular since it is one of the oldest visualization tools. *Seaborn* enhances *Matplotlib* with different visual styles. *Plotly* is not based on *Matplotlib*, since its primary domain of application is not Python. So, no enhancements of *Matplotlib* visualization tools are possible. Instead, Plotly enables interactive visualization and data analysis tools. The latter feature is beneficial when the visualizations are not only utilized in print media but also for interactive use. For example, when the mouse hovers over a pie chart slice in Figure 16, additional information in a label is emerging next to it. This convenient information technique is utilized to display various information that is logged during the execution of the C++/GPU implementation of SUBSCALE. Figure 16 shows how the execution time of each SUBSCALE operation of the current slice by hovering the mouse over the bar segment can be examined. The algorithmic steps in each bar segment are explained in [27, p. 39]. In the front end both Plotly and Seaborn were used. Seaborn is put to use when Plotly reaches its customization limits, since Matplotlib facilitates more sophisticated tweaking of the visualizations.

3.2.3 Code Structure of the Front End

Since the group of interest is the field of interactive scientific work, the front end is provided for use in a web interface Jupyter Notebook and its derivative Jupyter Lab.

Further in text both interfaces will be referred together as “Jupyter”. Both interfaces feature independent executable cells, that can contain formatted text and code. With such layouting it delimitates itself from common programming by shifting the focus to a hybrid of programming and document-oriented workflow.

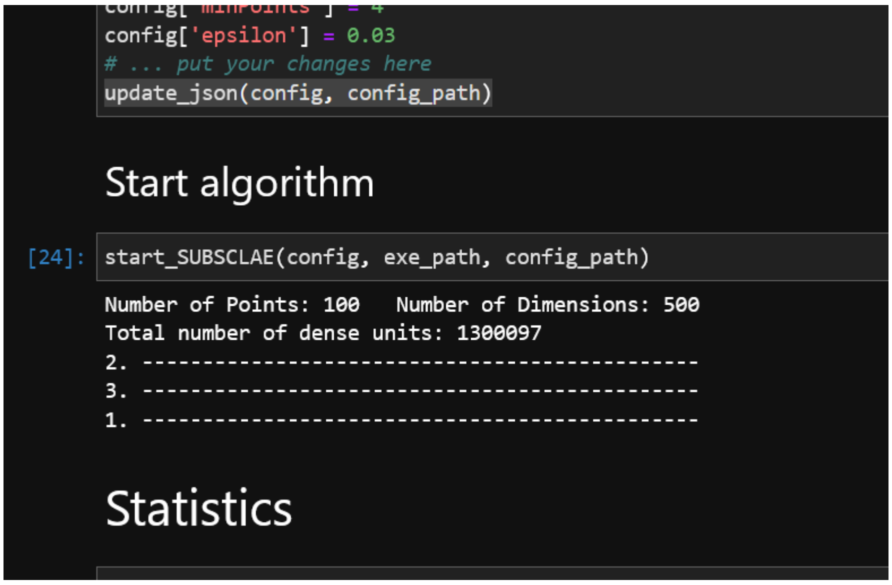
The web interface has to be launched through command line within the folder that contains the “`.ipynb`“ file with the command `jupyter notebook` or `jupyter lab`.

Once Jupyter is open, the working directory, therein contained configuration file and the SUBSCALE executable have to be defined first, because the “`.ipynb`“ file is located in a separate directory. The first parts of the code have to be run sequential to initialize all necessary parameters. Namely it is the functions `change_dir` and `load_json`. The first function changes the working directory and the second is required to get all C++ implementation related parameters, the result and data path. If SUBSCALE is to be executed, then it is advised to print the configuration file first and then adapt the configuration to the user’s needs. Otherwise, any of the functions in the cells below can be executed. Once the configuration loaded it is kept in a Python dictionary. All changed values are written back at once when the configuration file is updated with the function `update_json`.

```
config['dataPath'] = "./../data/4_madelon_train_100.csv"
config['minPoints'] = 4
…
update_json(config, config_path)
```

Listing 3: adding Dense Units and dimensions to signatures dictionary

When the SUBSCLAE C++program is executed, the progress is printed to the console output. However, this interactive behavior is diminished by Jupyter static output possibilities. Only when the program execution is finished, the output can be printed to the cell. A screenshot is in Figure 17 provided therefor. Eventually, this problem was solved in a way, that no output is printed to Jupyter. Instead, a console window is launched and the user can observe the state of the slices progression.



```
config['minPoints'] = 4
config['epsilon'] = 0.03
# ... put your changes here
update_json(config, config_path)
```

Start algorithm

```
[24]: start_SUBSCLAE(config, exe_path, config_path)

Number of Points: 100   Number of Dimensions: 500
Total number of dense units: 1300097
2. ---------------------------------------------
3. ---------------------------------------------
1. ---------------------------------------------
```

Statistics

Figure 17: Execution of SUBSCALE in Jupyter lab.

The output is displayed only after the termination of the of C++ program.

All further functions provide basic data visualization in the form of bar charts. The data, which consists of log files that are generated through the C++ execution for the functions, must be available in the directory that is defined in the configuration dictionary. An example is depicted in Figure 18. The available bar plots model the following information:

- Number of Dense Units per slice.
- Number of subspaces per slice.
- Time of loading and joining the slices.
- Time of all operations concerning processing all steps from the creation of Dense Unis to writing subspace clusters to files.
- Number of Dense Units per file.

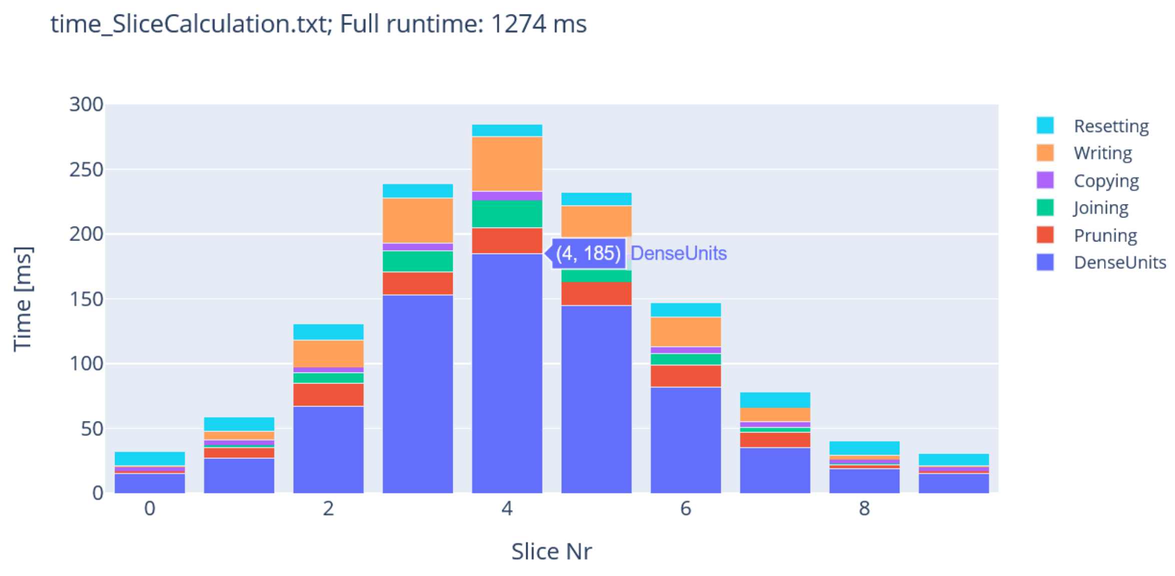


Figure 18: Visualization of execution time coverage

Parameters: 4_madelon_train_100.csv min_points=4, eps=0.02, splitting factor 10.

4 Analysis of Results

It is likely that SUBSCALE will be used on an appropriate platform for processing high dimensional data, e.g., Amazon Web Services or Microsoft Azure. Yet, consumer systems or semi-professional high end systems might also be used as a platform, since SUBSCALE aims to be scalable in both directions, for high-end and low-end hardware. For evaluation purposes of this thesis, other existing implementations and the Python implementation have been executed on different consumer systems.

4.1 Profiling Logs

All implementations create log files from profiling for each processed data set. Those files contain the number of elements and time for the components listed in table 4:

Implementation / Component	Python	SUBSCALE-GPU	SUBSCALE-Extended	SUBSCALE-Plus
Cores Sets	+	+	+	+
Dense Units	+	+	+	+
Signatures-Cluster	+	+	+	-
Subspace-Cluster	+	+	+	-
Slices	+	+	+	+

Table 4: Overview of the components that are logged in different SUBSCALE implementations.

4.2 The Output of Results / Interim Results

The interim results are handled differently by the algorithms (see Table 5). Some of the data saving steps are omitted for pragmatic reasons. For example, the number of CSs is very much less than the number of DUs (2). Thus, less space is required for keeping the CSs in memory in comparison to DUs. The same goes for the amount of signatures as there are always much less subspaces (Figure 19).

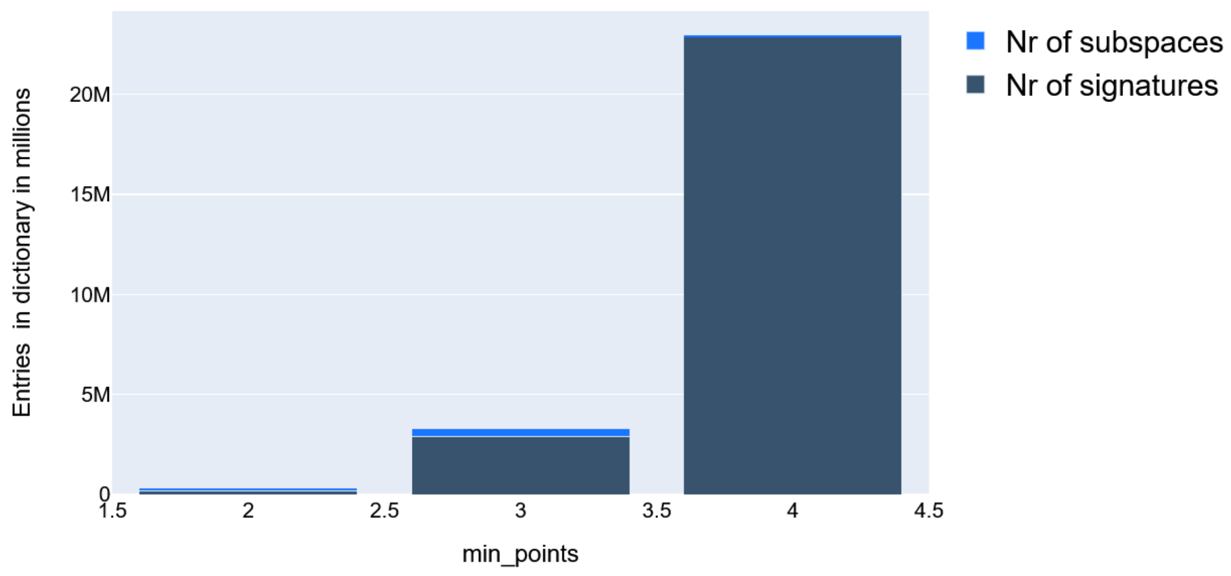


Figure 19: Number of signatures vs. Number of subspaces.

Parameters: 4_madelon_train_100.csv eps=0.03

Implementation / Component	Python	SUBSCALE-GPU	SUBSCALE-Extended	SUBSCALE-Plus
Cores Sets	-	-	-	-
Dense Units	-	-	-	-
Signatures-Cluster	-	-	-	-
Subspace-Cluster	+	+	+	+
Slices	+	+	+	+

Table 5: Overview of interim results in different SUBSCALE implementations.

4.3 Time Measuring

The file `Timer.py` provides all necessary functions to enable the logging of necessary characteristic values for the Python implementation. Before execution, the `filepath` of the timer file has to be set by the function `init`. Each timer logging is initialized by the function `start` with the timer's number and is finished with the call of the function `time_display`. 2 of 4 parameters are mandatory: The reference number of the timer and the user-defined message to log. The log is the collection of the entries from all `time_display` calls of every timer. On termination of the algorithm, the log is saved to `filepath` by calling the `out` function.

5 Discussion

The Python implementation of the SUBSCALE algorithm was developed in order to expand its popularity to a wider public, namely the field of data science. During the implementation process, a variety of difficulties and issues arose.

5.1 Correctness of Implementation

The implementation went as planned until the execution time benchmarking. The first components were implemented quickly and within the first month the subspace clusters could already be generated. After one and a half months all basic functionalities of the Java implementation were reached. This was possible due to a variety of code templates: There were Java and pseudocode implementations available. The progress also had one major speed up because, one component, a hash structure for the tables of Dense Units and subspaces had not to be implemented because of the available, built-in Python dictionary structure. Yet, to understand some aspects better, some time had to be spent on the theory that was obtained from the thesis supervisors, from papers and the doctor thesis of A. Kaur [15]. Following the investigation of the provided Java application the first functions were taken confident, propped with testcases. With the raising complexity however, the advantage diminished, because functions became extensive and were less testable. E.g., errors emerged only within higher amount of data – small, synthetic data sets were insufficient in order to find a difference between the subspaces data of the Python implementation and the analogous Java data. Different approaches have been undertaken to counter these issues. Interim results, like the amount of Core Sets were analyzed, which turned out to be equivalent to the Java paragon. The error appeared only within a specific data set and has been found after two weeks of intensive search. The cause was a relational operator within the pivot calculation combined with the manual definition of the number of points and dimensions in the file instead of an automated determination. Both errors masked each other and the output data differed. Such cumbersome implementation of the file property was utilized in order to save memory and to evade loading the whole data set.

5.2 Execution Speedup

After the basic functionality was established, it was found that Python was much slower than expected. Table 3 shows that the Java implementation is more than 5 times faster than the Python one. In order to speed up the execution, different speedup enhancements were tested. The integration proved itself to be sophisticated and error prone. A lot of time was spent on workarounds and yet it was not enough to achieve desired results. No speed up could be observed. Instead, the execution time increased because those technologies bring some overhead before a noticeable ROI is achieved.

5.2.1 JIT and Compilers

Yet so far, it became apparent, that the most convenient integration was Numba. Code parts, that were not able to be efficiently compiled in *nonpython* mode, were executed in *object* mode and some parts that failed to run even in this mode, could be left out from annotations at whole. Numba is also rated as the best of the three techniques, because it stood out due to least speed decrease.

PyPy led to a lot of trial-and-error configuration because of incompatibility and compile errors during *wheel* libraries installation. Eventually, the code had to be stripped off type hints of collections and classes to not conflict with PyPy own type inferring technique. Also, any execution of NumPy code was bypassed because no compatible PyPy 64-bit library could be installed. The utilization of the 64-bit PyPy version is crucial because otherwise the program crashes the moment 4 GB RAM are consumed.

Tests with Cython utilization of the `main.py` file went well first. But, once the `SUBSCALE.py` file had to be compiled, linking errors regarding Visual Studio libraries emerged. This issue could not be solved and the evaluation of the Cython speed-up was aborted.

5.2.2 Multiprocessing

SUBSCALE is aimed to be massively parallel executed, so it is only natural, that multi and manycore support should also be featured with the Python implementation. However, the Python language was not designed to be used in multiprocessing tasks. Even though with Python 3 some of the constraints were lifted and the GIL can be bypassed there are still performance issues with parallel execution. Test in Figure 11 showed that the execution time was raised to high magnitude when Python's multiprocessing library was harnessed for write access on the multiprocessing dictionary data structure.

6 Conclusion

The aim of this thesis was first to reimplement the SUBSCALE algorithm in Python in order to expand its popularity in the field of data science and machine learning. The choice fell for Python, because it is mainly used on daily basis by the group of interest [1] and as it facilitates to write a readable code, it is comprehensible and thus is often recommended as the primary programming language. But, since most of the Python libraries are not interpreted but precompiled to machine code, Python can also be used as a front end and used only in order to launch the performant C++ implemented code. In this thesis it was demonstrated, that as well both the combination of the C++ backend for solving the task and the Python front end, for visualization of the results can be utilized. Beyond that, the Python front end can be used to display prior calculated results. Therefore, the Python-only approach should not be pursued any more. Instead, further development should focus on the C++ implementation.

6.1 Outlook

In the C++ implementation Nicolas Kiefer showed, that SUBSCALE GPU implementation is approximately 70 times faster in parallel computation and mapping of Dense Units. However, since the hard drive proves itself as a bottleneck, the total speedup is as he states approximately 3 times. Yet, in a typical consumer system the data from the hard drive is loaded first to the computer memory and afterwards to the GPU memory. So, the main issue might not be the slow hard drive but the transmission through computer memory. Another point of criticism is that the hardware for the benchmarks was not chosen fairly. Even though both devices have the same amount of RAM and a comparable frequency, the GPU under test was a high-end device of approximately $1300 - 1550\ \%$ much higher monetary value than the CPU. Therefore, an interesting performance comparison would be between a GPU and CPU of same monetary value. Up to now, only sequential execution is implemented. So, a fair comparison could be achieved through a multithreading or multiprocessing implementation. If, however, the write access to disk space is really the bottleneck, then another approach is possible. Neither the computation of cluster candidates, nor Dense Units has to be performed on the same system. In the end of chapter 2.7.2 a possible approach was outlined for a client-server model, where the computationally demanding parts are outsourced to clients.

List of Tables

Table of Figures

7 References

[1] *2020 Kaggle Data Science & Machine Learning Survey.* [Online]. Available: https://www.kaggle.com/paultimothymooney/2020-kaggle-data-science-machine-learning-survey (accessed: Mar. 21 2021).

[2] T. Lauer, *Curriculum vitae Prof. Dr. Tobias Lauer.* [Online]. Available: https://cms5.hs-offenburg.de/fileadmin/Sonstige_Unterseiten/Altos/images/persProfile/ALTOS_Curriculum-_vitae_Lauer.pdf

[3] A. Datta, A. Kaur, T. Lauer, and S. Chabbouh, “Exploiting multi–core and many–core parallelism for subspace clustering,” *International Journal of Applied Mathematics and Computer Science*, vol. 29, no. 1, pp. 81–91, 2019, doi: 10.2478/amcs-2019-0006.

[4] B. Marr, “How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read,” *Forbes*, 21 May., 2018. https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=5a2a7d3c60ba (accessed: Mar. 21 2021).

[5] David Reinsel, John Gantz, and John Rydning, “The Digitization of the World from Edge to Core,” [Online]. Available: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

[6] IDC: The premier global market intelligence company, *IDC's Global DataSphere Forecast Shows Continued Steady Growth in the Creation and Consumption of Data.* [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS46286020 (accessed: Mar. 21 2021).

[7] J. Tang, T. Ma, and Q. Luo, “Trends Prediction of Big Data: A Case Study based on Fusion Data,” *Procedia Computer Science*, vol. 174, pp. 181–190, 2020, doi: 10.1016/j.procs.2020.06.073.

[8] L. Rokach, “A survey of Clustering Algorithms,” in *Data Mining and Knowledge Discovery Handbook*, O. Maimon and L. Rokach, Eds., Boston, MA: Springer US, 2010, pp. 269–298.

[9] A. K. Jain, M. N. Murty, and P. J. Flynn, “Data clustering: a review,” *ACM Comput. Surv.*, vol. 31, no. 3, pp. 264–323, 1999, doi: 10.1145/331499.331504.

[10] Amardeep Kaur and Amitava Datta, “A novel algorithm for fast and scalable subspace clustering of high-dimensional data,” (in En;en), *Journal of Big Data*, vol. 2, no. 1, pp. 1–24, 2015, doi: 10.1186/s40537-015-0027-y.

[11] A. Kumar and S.P. Panda, “A Survey: How Python Pitches in IT-World,” in *Proceedings of the International Conference on Machine Learning, Big Data, Cloud and Parallel Computing: Trends, prespectives and prospects : COMITCON-2019 : 14th-16th February, 2019*, Faridabad, India, 2019?, pp. 248–251.

[12] *PYPL PopularitY of Programming Language index.* [Online]. Available: https://pypl.github.io/PYPL.html (accessed: Mar. 21 2021).

[13] R. Bhatia, “Why Do Data Scientists Prefer Python Over Java?,” *Analytics India Magazine*, 18 May., 2018. https://analyticsindiamag.com/why-do-data-scientists-prefer-python-over-java/ (accessed: Mar. 21 2021).

[14] *UCI Machine Learning Repository: Iris Data Set.* [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Iris (accessed: Mar. 22 2021).

[15] A. Kaur, “THESIS_DOCTOR_OF_PHILOSOPHY_KAUR_Amardeep_2016-freigeschaltet,” The University of Western Australia, Crawley, WA 6009, Australia, 2016.

[16] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo, “Fast discovery of association rules,” *Advances in knowledge discovery and data mining*, vol. 12, no. 1, pp. 307–328, 1996.

[17] Marite Kirikova *et al.,* “New Trends in Databases and Information Systems,”

[18] NVIDIA Developer, *CUDA Zone.* [Online]. Available: https://developer.nvidia.com/cuda-zone (accessed: Mar. 30 2021).

[19] S. van der Walt, S. C. Colbert, and G. Varoquaux, “The NumPy Array: A Structure for Efficient Numerical Computation,” *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 22–30, 2011, doi: 10.1109/MCSE.2011.37.

[20] T. P. Team, *PyPy - Features.* [Online]. Available: https://www.pypy.org/features.html (accessed: Mar. 26 2021).

[21] *Numba: A High Performance Python Compiler.* [Online]. Available: https://numba.pydata.org/ (accessed: Mar. 27 2021).

[22] *Numba for CUDA GPUs — Numba 0.52.0.dev0+274.g626b40e-py3.7-linux-x86_64.egg documentation.* [Online]. Available: https://numba.pydata.org/numba-doc/dev/cuda/index.html (accessed: Mar. 27 2021).

[23] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, “Cython: The Best of Both Worlds,” *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 31–39, 2011, doi: 10.1109/MCSE.2010.118.

[24] JetBrains, *Features - PyCharm.* [Online]. Available: https://www.jetbrains.com/pycharm/features/ (accessed: Mar. 28 2021).

[25] *JupyterLab Documentation — JupyterLab 3.1.0a3 documentation.* [Online]. Available: https://jupyterlab.readthedocs.io/en/latest/ (accessed: Mar. 28 2021).

[26] A. Kaur, *amkaur/subscaleplus.* [Online]. Available: https://github.com/amkaur/subscaleplus (accessed: Mar. 28 2021).

[27] N. Kiefer, “Datenparalleles Subspace Clustering mit Grafikprozessoren,” Hochschule Offenburg, 2020.

[28] N. Kiefer, *KieferN/SUBSCALE_GPU.* [Online]. Available: https://github.com/KieferN/SUBSCALE_GPU (accessed: Mar. 28 2021).

[29] *sklearn.cluster.DBSCAN — scikit-learn 0.24.1 documentation.* [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html (accessed: Mar. 30 2021).

[30] Python.org, *PEP 606 -- Python Compatibility Version.* [Online]. Available: https://www.python.org/dev/peps/pep-0606/ (accessed: Mar. 31 2021).

[31] Guido van Rossum, *PEP 8 -- Style Guide for Python Code.* [Online]. Available: https://www.python.org/dev/peps/pep-0008/ (accessed: Apr. 1 2021).

[32] *Glossary — Python 3.9.4 documentation.* [Online]. Available: https://docs.python.org/3/glossary.html (accessed: Apr. 8 2021).

[33] Donald E. Knuth, *The Art of Computer Programming: Volume 4A Combinatorial Algorithms Part 1*. Boston: Addison-Wesley, 2011.

[34] *scikit-learn: machine learning in Python — scikit-learn 0.24.1 documentation.* [Online]. Available: https://scikit-learn.org/stable/ (accessed: Apr. 11 2021).

[35] I. Rogers, *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. New York, NY: ACM, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1565824

[36] *PyPy Speed.* [Online]. Available: https://speed.pypy.org/ (accessed: Apr. 17 2021.909Z).

[37] *PyPy - packages.* [Online]. Available: http://packages.pypy.org/ (accessed: Apr. 16 2021.794Z).

[38] *pytest: helps you write better programs — pytest documentation.* [Online]. Available: https://docs.pytest.org/en/stable/ (accessed: Apr. 4 2021).

[39] italomarcelo, “2020 Kaggle ML and DS Survey - Brazil,” *Kaggle*, 16 Feb., 2021. https://www.kaggle.com/italomarcelo/2020-kaggle-ml-and-ds-survey-brazil (accessed: Apr. 22 2021.551Z).