universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Exploratory Data Analysis with Google's Tensor Processing Unit (TPU): Enhancing Traditional Data Mining Algorithms with the Use of the TPU on the Example of the k-Means Algorithm"

verfasst von / submitted by

Anna Wolff, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2022 / Vienna, 2022

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 926
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Inform.Univ. Dr. Claudia Plant
Mitbetreut von / Co-Supervisor:	Dipl.-Ing. Dr. techn. Martin Perdacher, BSc

Abstract

In the past few years, there has been an immense increase in the volume of collected data worldwide. Dealing with the continuously growing amount of data requires two strategies. On the one hand, methods and algorithms are needed to extract meaningful information from the amount of data, and on the other hand, there is a need for the computational ability to handle the amount of data. Both challenges are tackled. The research field of data mining is concerned with identifying patterns in data, interpreting the identified patterns and performing qualitative or quantitative predictions or descriptions. One focus of the research in data mining is on scaling algorithms for large and very large data sets. Enterprises like Google are developing powerful hardware to meet the computing needs of modern technologies. In 2016, Google announced the Tensor Processing Unit (TPU), which is specifically designed to meet the computational demands of deep neural networks in Google's data centers. However, other data mining and machine learning techniques can also profit from the computational power of the TPU.
This master thesis aims to enhance the k-Means algorithm with the use of the Google TPU in terms of runtime while preserving the quality of the clustering results. We implemented a k-Means algorithm with matrix-matrix multiplication that is tailored to the requirements of the TPU. We developed two versions that distribute training on the TPU in two different ways. The clustering results of the versions have advantages that complement each other in terms of runtime and accuracy. Therefore, combining both versions is also explored.
The computation time of the k-Means algorithm can be greatly accelerated by using the TPU. However, it should be taken into consideration that loading the data set on the TPU takes additional time and using a TPU compared to a CPU or GPU is more expensive. Therefore, we recommend using the TPU only for large data sets and/or when the k-Means algorithm is repeated multiple times for the same data set.

Kurzfassung

In den letzten Jahren ist Menge der weltweit gesammelten Daten immens gestiegen. Der Umgang mit dieser stetig wachsenden Menge an Daten erfordert zwei Strategien. Einerseits werden Methoden und Algorithmen benötigt, um aussagekräftige Informationen aus der Datenmenge zu extrahieren, und andererseits werden Rechenleistungen benötigt, um die Menge an Daten zu verarbeiten. Beide Herausforderungen werden adressiert. Das Forschungsgebiet Data Mining befasst sich mit der Identifizierung von Mustern in Daten, der Interpretation der identifizierten Muster und der Durchführung qualitativer oder quantitativer Vorhersagen oder Beschreibungen. Ein Schwerpunkt der Forschung im Data Mining ist die Skalierung von Algorithmen für große und sehr große Datensätze. Unternehmen wie Google entwickeln leistungsstarke Hardware, um den Rechenanforderungen moderner Technologien gerecht zu werden. Im Jahr 2016 kündigte Google die Tensor Processing Unit (TPU) an, die speziell für die Rechenanforderungen von tiefen neuronalen Netzwerken (Deep Neural Networks) in den Rechenzentren von Google konzipiert ist. Aber auch andere Data-Mining- und Machine-Learning-Techniken können von der Rechenleistung der TPU profitieren.
Ziel dieser Masterarbeit ist es, den k-Means-Algorithmus durch den Einsatz der Google TPU in Bezug auf die Laufzeit zu verbessern und gleichzeitig die Qualität der Clustering-Ergebnisse zu erhalten. Im Rahmen der Masterthesis wurde ein k-Means-Algorithmus mit Matrix-Matrix-Multiplikation implementiert, der auf die Anforderungen der TPU zugeschnitten ist. Es wurden zwei Versionen entwickelt, die das Training auf der TPU auf zwei verschiedene Arten verteilen. Die Clustering-Ergebnisse der beiden Versionen haben Vorteile, die sich in Bezug auf Laufzeit und Genauigkeit ergänzen. Daher wird auch die Kombination beider Versionen untersucht.
Die Berechnungszeit des k-Means-Algorithmus kann durch die Verwendung der TPU erheblich beschleunigt werden. Es sollte jedoch berücksichtigt werden, dass das Laden des Datensatzes auf die TPU zusätzliche Zeit in Anspruch nimmt und die Verwendung einer TPU im Vergleich zu einer CPU oder GPU teurer ist. Daher wird die Verwendung der TPU nur für große Datensätze epmfohlen und/oder wenn der k-Means-Algorithmus mehrfach für denselben Datensatz wiederholt wird.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

1. Introduction

1.1. Motivation

The amount of data stored in digital form has doubled every nine months on average over the last decades. This is twice the rate Moore's law predicted for the increase of computing power. Moore's law is now replaced by the storage law, as the doubling of the amount of data is called. With regard to the vast amount of data and the fact that data exists in all sorts of varieties, extracting meaningful information is often beyond human capabilities. This leads to the necessity for developing algorithms which can extract meaningful information from the huge amount of data but also for the computational ability to process large amounts of data.

The increase of data is one of the motivations for data mining. Data Mining is also referred to as Knowledge Discovery in Databases (KDD) and is concerned with identifying patterns in data, interpreting the identified patterns and performing qualitative or quantitative predictions or descriptions. For this reason, the algorithms employ a variety of models from the areas of statistics, databases, machine learning, experimental algorithms as well as mathematical approaches. One focus of the research in data mining is on scaling algorithms for large and very large data sets. [1]

In addition to the field of research, the demand for handling these amounts of data has led to the rise of data centers with more than seven million data centers worldwide today. Data centers are facilities that are composed of computing infrastructure, storage systems and network computers. They are used to assemble, process, store and disseminate large amounts of data and are an integral part of companies to support business applications and provide services. Companies rely on the data, applications, and services contained within data centers, which makes them a critical asset for everyday operations.

The massive growth in data requires powerful algorithms as well as the computational ability to continuously evolve and improve. In 2011, Deep Neural Networks (DNNs), used for technologies such as speech or image recognition, achieved a breakthrough. However, this almost led to a crises in Google's data centers: If every Google user used speech recognition for just three minutes a day, that would have doubled the computational demands in Google's data centers. Using conventional Central Processing Units (CPUs), this would have been too cost-intensive.

To tackle this problem, Google started a project and developed the Tensor Processing Unit (TPU) within 15 months. [2, 3] The TPU was announced in 2016 at the Google I/O event [4]. In 2018, it was made available to the public through the Google Cloud Platform (GCP) [5].

TPUs are custom-developed Application-Specific Integrated Circuits (ASICs). They are designed for maximum performance and flexibility and therefore used to accelerate machine learning. Using the TPU, the train time of models can converge in hours instead of weeks as before on other hardware platforms. [6, 7]
TPUs are specialized in handling the computational demands of DNNs. DNNs are predictive models for classification in data mining, which is a supervised learning technique in machine learning. Supervised learning outputs precise recommendations and makes predictions based on an analysis of input data with a given target variable. However, labeled training is often resource intensive to create since data is often unlabeled in reality. Unsupervised learning, in contrast, learns from unlabeled raw data by learning the relationships and patterns within the data. One important and widely used data mining and unsupervised machine learning algorithm is the k-Means algorithm. It is one of the most well-known and popular clustering algorithms. k-Means is powerful because it provides a simple method for partitioning a data set into a user-selected number of clusters and it is well suited for analyzing large data sets.

1.2. Objective and Structure of the Thesis

This master thesis contributes to the research of dealing with large data and explores how to extend the research on the TPU to other methods in data mining on the example of the k-Means algorithm. The aim is to enhance the k-Means algorithm with the use of the Google TPU in terms of runtime while preserving the quality of the clustering results.

For this, the current state of research on the k-Means algorithm and the TPU is presented. Moreover, a k-Means algorithm is implemented with consideration to the requirements of the TPU and validated with regard to runtime and quality of the clustering results. For this purpose, synthetic data sets are generated, which are ideally tailored to k-Means and TPU. Real data sets are used to verify and generalize the results. The results on the TPU are also compared with results on a CPU and a Graphic Processing Unit (GPU) to verify the achievement of the defined goal.

Based on the objectives, the master thesis is structured as follows:

Chapter 1 Introduction In Chapter 1, the motivation for the topic as well as the problem is explained. Furthermore, the objectives are defined and the structure of the work is described.

Chapter 2 Theoretical Background and Related Work In Chapter 2, a brief overview of the topics Exploratory Data Analysis (EDA) and data mining with regards to the k-Means algorithm is given. After that, the k-Means algorithm is described with a focus on well-known extensions and adaptations. Furthermore, an introduction of the TPU is given including a comparison to CPU and GPU as well as an overview of recent research on this topic. Finally, an overview of evaluation metrics is given.

Chapter 3 Proposed Method In Chapter 3, the proposed method to improve the k-Means algorithm in terms of runtime is presented. Furthermore, the goals of and requirements for the method are defined. Lastly, important influencing parameters are discussed and the metrics for evaluation are introduced.

Chapter 4 Implementation In Chapter 4, the implementation of the method is described and the details of the technology and hardware used are stated. Furthermore, the data sets used and the procedure of the experiments are described.

Chapter 5 Evaluation and Discussion In Chapter 5, the results of the experiments are analyzed and discussed with regard to the defined objectives.

Chapter 6 Conclusion In Chapter 6, the results and findings of the experiments are summarized and an outlook for future work is provided.

2. Theoretical Background and Related Work

2.1. Exploratory Data Analysis and Data Mining

Exploratory Data Analysis (EDA) is an essential, fundamental early step in research analysis. It takes place after data is collected and pre-processed. The main goal of EDA is the examination of data to understand the present status of data, i.e., distribution, anomalies and outliers, before making any assumptions. For this, the data is visualized, plotted and manipulated without bias. EDA is used for assessing the quality of data and is applied before models are build. [8]
Many EDA techniques are used in data mining. Data mining is an important step in the KDD process as it transforms data into knowledge. Generally, any kind of data can be used as long as it is meaningful to the field of application. A common categorization of the data mining methods is as follows [9]:

- Characterization and Discrimination,
- Mining of Frequent Patterns,
- Association and Correlation Analysis,
- Classification and Regression,
- Clustering, and
- Outlier Analysis.

The k-Means algorithm, on which is the focus of this thesis, belongs to cluster analysis. Clustering is concerned with discovering groupings or clusters in data. Objects within a cluster should be highly similar to each other and highly dissimilar to objects in other clusters. Unlike in classification, the class label of the data in clustering is unknown. In machine learning, clustering is part of unsupervised learning while classification belongs to supervised learning. With clustering, class labels can be generated. This is necessary since in many cases, class-labeled data is not available. Clustering can be classified into the following categories, even though these may overlap:

- Partitioning methods,
- Hierarchical methods,
- Density-based methods,

- Grid-based methods. [9]

k-Means is a partitioning method. Partitioning methods separate a set of n objects in k clusters ($k \leq n$) where each object belongs exclusively to one cluster. Partitioning is a simple and straight-forward method. The separation in clusters is most of the time distance-based: objects within a cluster are close and objects of different clusters are far away from another. The algorithms usually start with an initial clustering and iteratively improve the partitioning by relocating objects between clusters. A widely used partitioning clustering criterion is the square error criterion. This partitioning strategy is employed by the k-Means method. [9, 10]

2.2. k-Means Algorithm

2.2.1. Origin of the k-Means Algorithm

The k-Means algorithm is one of the most well-known and popular algorithms in data mining. It provides a simple method for partitioning a data set into a user-selected number of clusters and is well suited for analyzing large data sets. The algorithm was voted as one of the top ten algorithms in data mining in 2008 [11].
The term k-Means was introduced by MacQueen [12] in 1967. The aim of his work was to describe a process that, based on a sample, divides an n-dimensional population into k sets. To do this, the population is divided into k groups, each initially consisting of one randomly chosen data point. New points are assigned to the group whose mean is closest to the new point. After adding a point, the mean of the group is recalculated. Therefore, k is always the means of the group and hence the name of the process 'k-Means'. In his work, MacQueen suggests variations of this general approach for different applications. One of them is based on the method of Forgy ([13] as cited in [12]) which obtains partitions with low within-class variance. This improvement varies from the general k-Means process since it extends the initial clustering by a two-step iteration until convergence: (1) compute the means of each partition (2) reassign the points to the (re)calculated means based on the squared distance, thus forming new clusters. MacQueen describes the advantages of the k-Means process as easily programmable and computationally economic and the results as fairly efficient by within-class variance.
Today's most common use of the k-Means algorithm is based on Lloyd's work on pulse-code modulation from 1957 (published in 1982) [14], even though the term k-Means is not used to describe the algorithm. Lloyd's algorithm differs from MacQueen's k-Means process in that it is a batch algorithm: The centroids of the clusters are not calculated after each new assignment of a data point, but applied to all cases at once after the (initial) assignment of the data points.
Another variation of the k-Means algorithm was introduced by Hartigan [15] in 1975. It differs in the (re)assignment phase since it partitions the data set based on locally optimal within-cluster Sum of Squared Error (SSE). It reassigns objects to another cluster only, if the sum of squares (SSE2) is smaller than the present one (SSE1). This means that an

object may be assigned to another subspace even though it is closer to the centroid of another cluster.

2.2.2. Objectives of Cluster Analysis with the k-Means Algorithm

The k-Means algorithm aims to obtain a partitioning of a data set, X, consisting of n objects into k clusters C with a high intracluster and a low intercluster similarity. Each object belongs to exactly one cluster. The k-Means is a centroid-based technique. The data set can be described as a set of d-dimensional vectors $X = \{x_i \mid i = 1, ..., N\}$ where $x_i \in \mathbb{R}^d$ stands for the ith data point. The centroid c_j of a cluster, $C_j, j = 1, ..., k$, which is defined by the mean in the k-Means algorithm, is conceptionally the central point of that cluster and therefore, represents it. For initializing the algorithm, k arbitrary objects in $\mathbb{R}^d$ are selected as the initial centroids.
Based on the sum of squares criterion, the algorithm tries to minimize the sum of squared errors for all objects in the data set and by that iteratively improves the within-cluster variation. The mathematical function of the SSE is depicted in equation 2.1

$$E = \sum_{i=1}^{k} \sum_{x_i \in C_j} ||x_i - c_j||^2 \tag{2.1}$$

where $||x_i - c_j||^2$ is the Euclidean distance. Hence, the k-Means assigns each object to the nearest centroid based on the Euclidean distance. Solving this equation exactly is a NP-hard problem with k^n possible clusterings. The process will always terminate. The k-Means algorithms (e.g. by Lloyd or MacQueen) are heuristics which implement the above mentioned strategy in an efficient way. [9, 10, 11]
The procedure of k-Means is described in Algorithm 1 and visualized in Figure 2.1.

Algorithm 1: Outline of the k-Means Algorithm

```
  Data: Data set $X$ containing $n$ objects
  Result: A set of $k$ clusters
1 Specify the number of clusters $k$
2 Randomly choose $k$ objects from $X$ as initial centroids
3 repeat
4     Assign each data point to its nearest centroid (Assignment Step)
5     Calculate the new centroid (mean) for each cluster (Update Step)
6 until Convergence criterion is met;
```

The time complexity of the k-Means algorithm is $O(nkt)$, where n is the number of objects, k the number of clusters, and t the number of iterations. Usually, $k \ll n$ and $t \ll n$. Therefore, the algorithm is comparably scalable and processing large data sets with k-Means is rather efficient. [9]

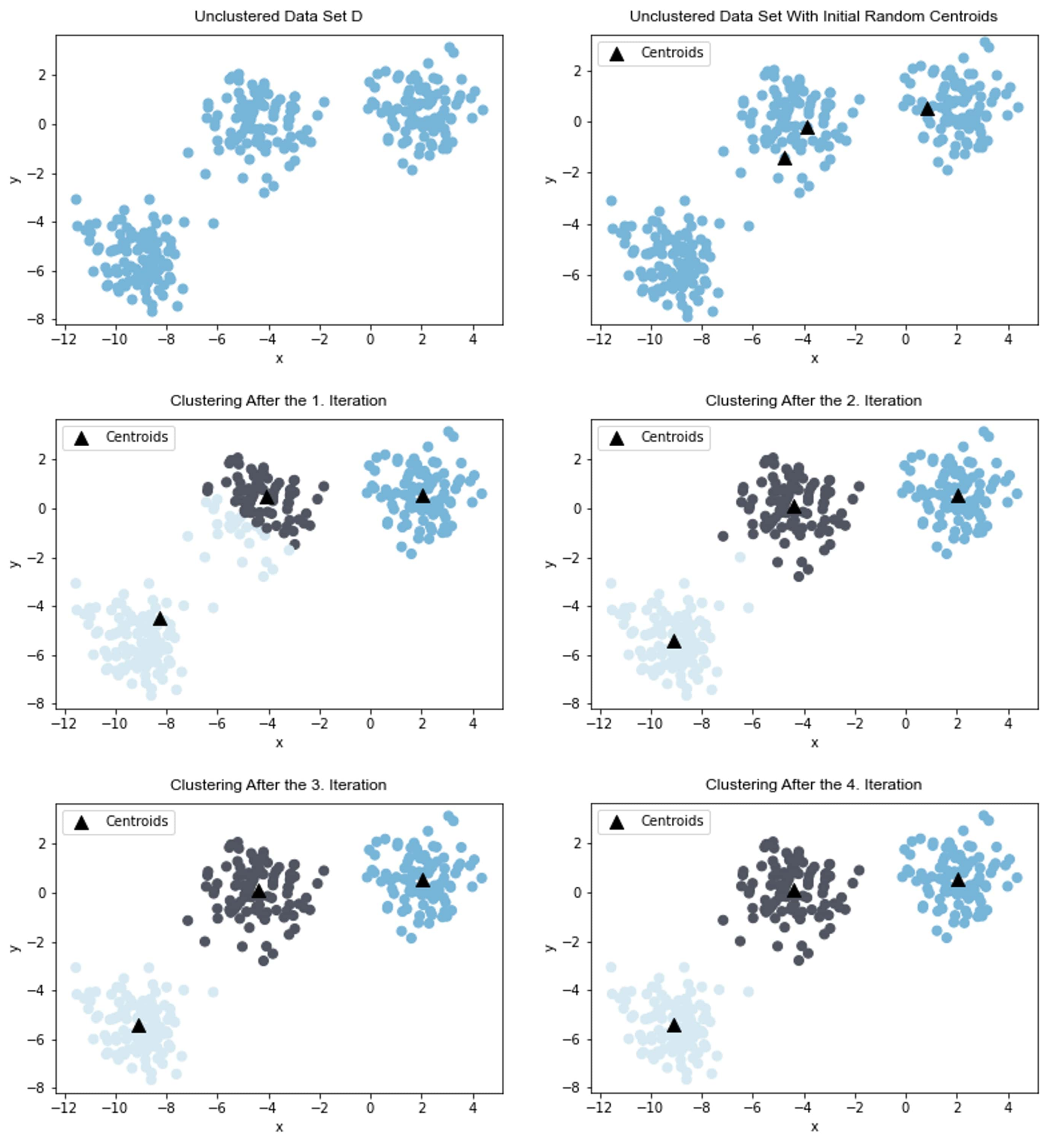


Figure 2.1.: Procedure of k-Means Clustering

2.2.3. Limitations of the k-Means Algorithm

The k-Means algorithm has its limitations. It is appealing because of its speed but not because of its accuracy. One of the limitations is that a global optimum is not guaranteed and the algorithm usually terminates in a local optimum. This is heavily dependent on the initial selection of the centroids (see Figure 2.2). In addition, it is necessary to determine the number of clusters in advance (see Figure 2.3). Another drawback of the k-Means algorithm is its sensitivity to noise and outliers. Even a small amount of such data points heavily influences the position of the centroid, and thus the cluster assignment. Furthermore, the data needs to be convex-shaped, ideally Gaussian-distributed (see Figure 2.4). And last, but not least, the size of clusters needs to be even and the density spread of the data points across the data space needs to be similar (see Figure 2.5).

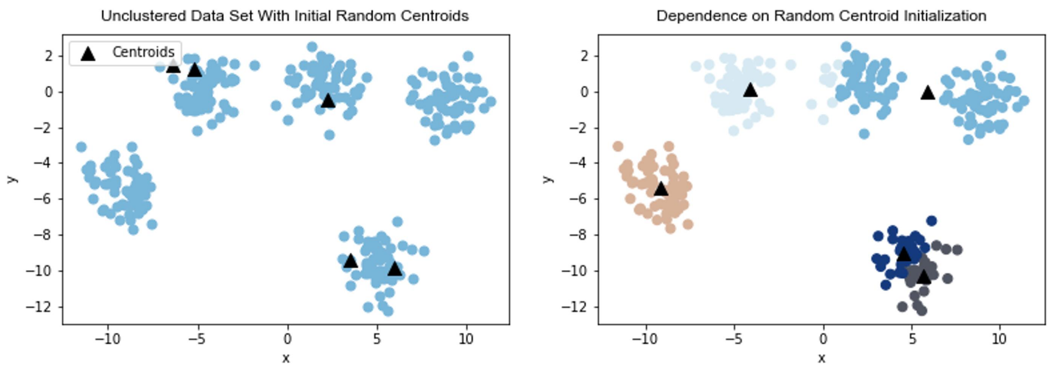


Figure 2.2.: Poor Random Initalization of the Centroids

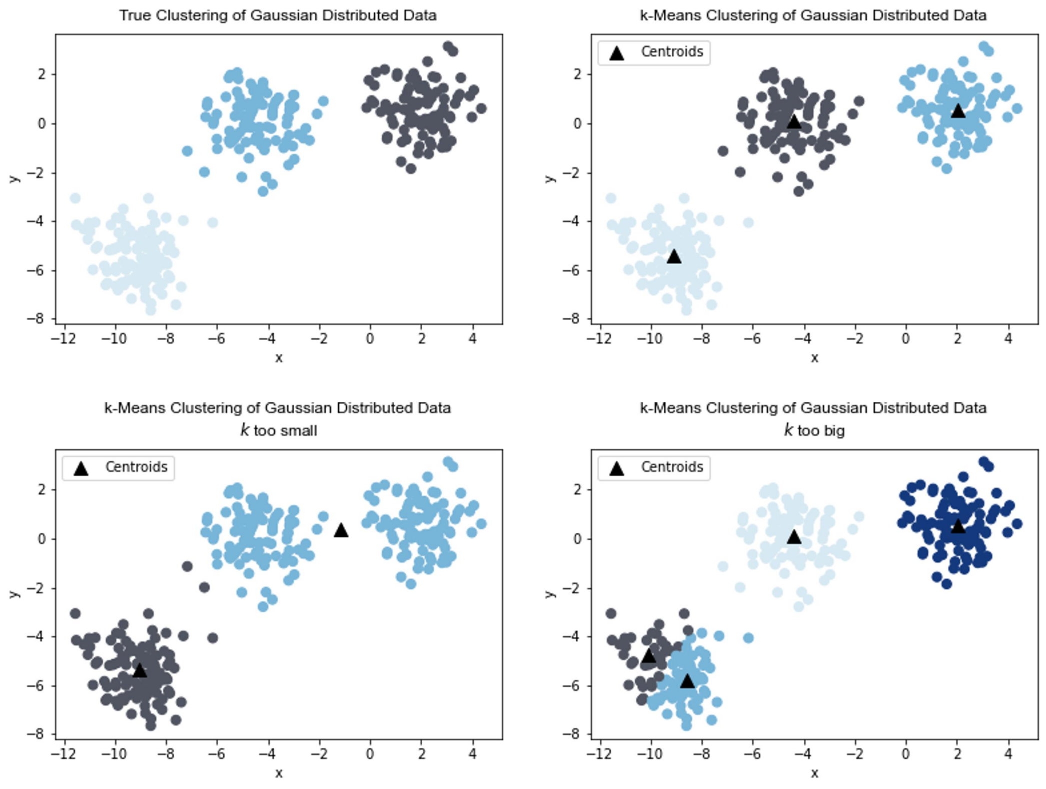


Figure 2.3.: Choosing an Appropriate k

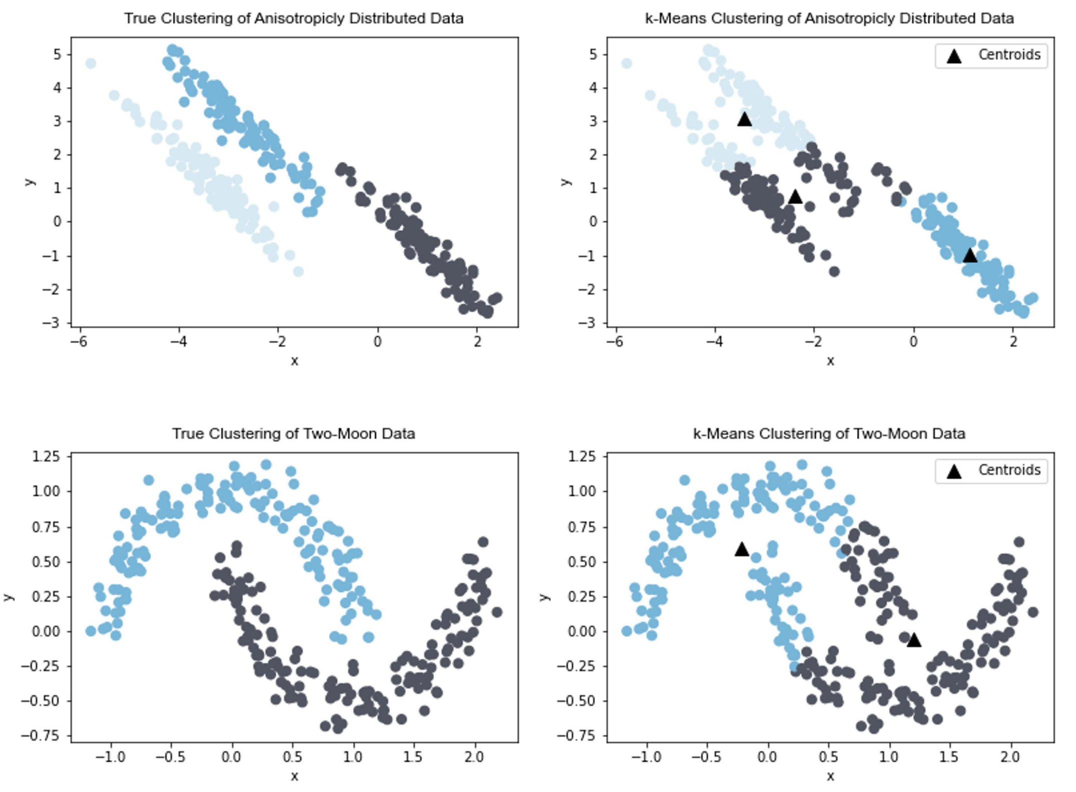


Figure 2.4.: Clustering of Not Gaussian Distributed Data

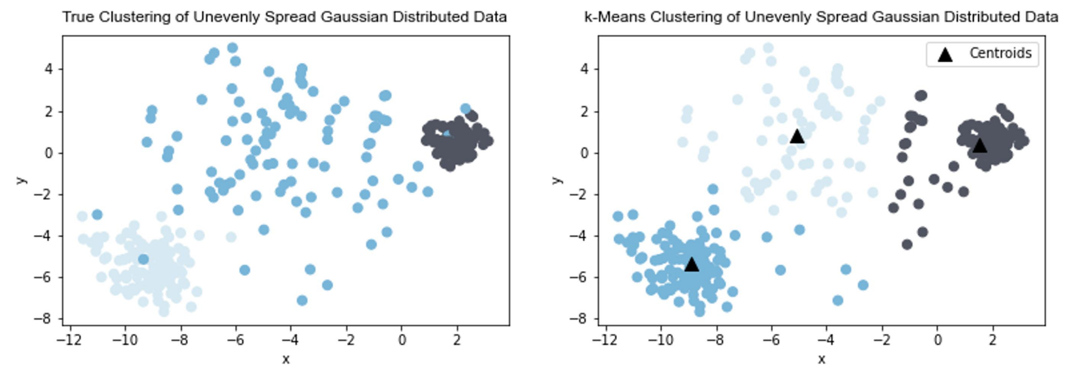


Figure 2.5.: Clustering of Data With Uneven Cluster Density

2.2.4. Adaptations and Extensions of the k-Means Algorithm

k-Means is a simple but efficient algorithm for clustering. For this reason, the algorithm has been and still is subject of many optimization efforts. The k-Median algorithm [10] is a well-known alteration of the k-Means. It uses Manhattan distance instead of Euclidean distance and therefore determines the centroid of a cluster by calculating the Median. The k-Medoids [9] is another alteration. It is a representative object-based technique which picks actual objects of the data set rather than the mean value. It therefore minimizes the absolute-error criterion aiming to reduce the outlier sensitivity of the k-Means.

The following sections give an overview of the most important adaptations and extensions of the k-Means algorithm. We split the areas of improvement into five different categories: (1) selection of the number of clusters k, (2) initialization, (3) assignment, (4) acceleration of the update step of the algorithm, and (5) environment specific improvements of k-Means.
The selection of the papers is based on their relevance in this category, i.e., publications in important data mining conferences and number of citations. The last category only considers more recent publications since hardware itself underlies continuous development.

Selection of the Number of Clusters

One of the limitations of the k-Means algorithm is that the number of clusters k must be known in advance which is very often not possible. X-Means [16] is an extension of the k-Means algorithm which provides an efficient estimation of the number of clusters given a range within which k reasonably lies. The Unsupervised K-Means (U-K-Means) [17] improves X-Means further. It automatically finds the optimal number of k without the need to specify a range. It adopts the entropy concept introduced by Yang et al. [18] and starts with the number of points n as the initial number of clusters.
The G-Means algorithm [19], like the improvement algorithms presented above, assumes Gaussian data distribution, but follows a different strategy for choosing k. It assumes that one Gaussian cluster should be represented by one cluster center and learns k while clustering by increasing k in a hierarchical manner until this hypothesis is validated.

Initalization

The final clustering result depends on the initialization. A greedy and widely used approach is repeating k-Means with different initial centroids [12] until a reasonably quality of the clustering is reached. The perhaps most famous initialization strategy is k-Means++ [20]. The authors propose a variant of the algorithm that uses a simple, randomized seeding technique. As already known, the initial centers are chosen randomly. The difference to k-Means is that the data points are weighted based on their squared distance to the nearest centroid already selected.
k-Means|| [21] is an optimized version of k-Means++ by parallelizing the initalization of the cluster centers which is better suited for the increasing size of current data sets. Other well known initialization optimizations are bisecting k-Means [22], a hierarchical top-down approach, and the global k-Means clustering algorithm [23].

Assignment Step

The k-Means algorithm uses a hard assignment of the cluster members. However, in real word data sets, clusters can overlap. Prominent weakened k-Means versions are the Expectation-Maximization (EM) algorithm [24], the Fuzzy C-Means (FCM) clustering [25], which is further improved in [26], and the Non-Exhaustive, Overlapping K-Means (NEO-K-Means) [27]. k-Means-- [28] addresses the problem with outliers. The algorithm simultaneously clusters and discovers outliers and even extends to all distance measures that are in the form of Bregman divergences [29].

Update Step

Accelerating the update step of the k-Means algorithm has been proposed by some researchers. Na et al. [30] propose an improved k-Means clustering algorithm that reduces the run-time of k-Means. By storing information about labels and the distance of all data points to the nearest centroid, the calculations of the distance between each data point and all centroids, repeated in each iteration, can be avoided. Elkan [31], Hamerly et al. [19], and Ding et al. [32] avoid unnecessary distance calculations by applying the triangle-inequality and monitoring upper and lower bounds for the distances between data point and centroid. Pelleg and Moore [33] introduce the use of k-d trees to update centroids in bulk instead of object by object. K-d trees can also be applied for initializing the cluster centers more efficiently.

Environment Specific Improvements of k-Means

Today's microprocessors are capable of performing multiple computations such as additions, multiplications and other operations simultaneously. For this reason, more recent advancements of the k-Means also exploit the capabilities of today's microprocessors, such as CPUs and GPUs. Böhm and Plant [34] exploit Single Instruction Multiple Data (SIMD) instruction sets by parallelizing the operations within each core of a CPU. This permits up to 16 double precision floating points operations per clock cycle. Other proposals accelerate the k-Means algorithm using parallel shared-memory of multi-core systems. One approach is splitting the data set into chunks and then clustering each chunk in a separate thread. The data is split into chunks by reading the data line by line until a pre-defined chunk size is reached. The chunks are then put in a queue and distributed to the cores. Independently from another, each core clusters one chunk of data. This allows for multiple clustering in parallel. In the final stage, the results are merged by performing the k-Means on the centroids extracted from the chunks. [35]
Another approach for accelerating k-Means is by distributing the calculations across cores using Open Multi-Processing (OpenMP) [36].
Multi-core k-Means [37] is another extension of k-Means. The algorithm uses two types of parallelism, i.e., SIMD, parallelization within a core as already proposed in [34], and Multiple Instructions Multiple Data (MIMD), parallelization of different code on different CPU cores. This strategy allows for reducing the data transfer between the different

memory components, main memory, cache and register.
Operations cannot only be distributed within a core or across cores of a CPU, but also across multiple nodes [38, 39].

Extensions of the k-Means are not limited to the CPU and exist for the GPU as well. Dhillon and Modha [40] propose an approach based on distributed memory with the Single Program Multiple Data (SPMD) model using Message Passing (MPI). Accelerating k-Means on the GPU via Compute Unified Device Architecture (CUDA) is an approach which executes the distance calculations on the GPU in parallel but sequentially updates the cluster centroids on the CPU. With this optimization strategy targeted at the architecture, Zechner and Granitzer [41] aim to exploit the available computational capabilities of the GPU. A recent approach from 2021, suggests two different clustering extensions of the k-Means on the GPU, one is tailored to low dimensional data and the other one to high dimensional data. The version for low-dimensional data exploits on-chip registers on multi-core systems which reduces the data access latency. Parallelization is achieved by using the OpenMP standard. The high-dimensional data design simulates matrix multiplications by exploiting on-chip shared memory aiming to realize a high compute-to-memory-access ratio.

The k-Means is a well-known, popular algorithm. Its simplicity but yet effectiveness have inspired many adaptations and extensions. Most aim to improve quality at various stages of the algorithm, and only a few exploit the capacities of modern processors. In the following chapter, Google's Tensor Processing Unit is presented, which has been developed specifically for machine learning applications.

2.3. Tensor Processing Units

2.3.1. Introduction of the Tensor Processing Unit

The TPU is a custom ASIC designed by Google. It is specifically built for accelerating machine learning applications and tailored to Google's open-source Artificial Intelligence (AI) program library TensorFlow. [4] TPUs are designed to provide maximum performance and flexibility. The development is targeted at machine learning researchers, engineers, developers and data scientists. Google provides well-optimized, open-source reference models to get started with developing machine learning applications on the TPU using TensorFlow. [42, 7]. The TPU is offered as a service (Cloud TPU) that allows users to access the TPUs through the GCP. The Cloud TPU Application Programming Interface (API) allows for automated and flexible TPU management. [43]

2.3.2. Differentations of the Tensor Processing Unit to CPU and GPU

CPUs, GPUs and TPUs are designed for different purposes and therefore differ in their applications. This section gives a brief overview of the three processing units in order to highlight benefits but also drawbacks of the TPU in comparison to CPU and GPU. However, the comparison does not go into technical details.

CPUs are general purpose processors. They are based on the von Neumann architecture. A great benefit of CPUs is the flexibility they provide. CPUs are suitable for all kinds of software in many different applications, i.e., they can be used for processing in a computer, executing bank transactions, controlling rocket engines, but also for neural networks applications such as image classifications. However, this great flexibility of CPUs is also their weak point. Because CPUs can run all kinds of software, they only know which calculation to execute next when they read the instruction. This leads to the fact that CPUs have to store the calculation results in the registers or in the cache for each calculation. Memory access becomes the drawback of the CPU architecture since it limits the overall throughput and moreover is energy consuming. This downside of the CPU is also known as the von Neumann bottleneck and is especially problematic for huge neural network calculations. [44]

Disregarding technical details, GPUs increase throughput by increasing the number of Algorithmic Logic Units (ALUs) featured, which hold and control the mathematical operations, such as addition and multiplication, compared to a CPU. Additionally, the design of GPU ALUs allows for eight, 16 or even 32 operations at once instead of only one or two. Typically, a modern GPU consists of 2,500 to 5,000 ALUs in a processor allowing for thousands of operations simultaneously. This architecture makes it possible to execute applications with a lot of parallelism such as matrix multiplication. It is the most popular processing architecture for deep learning activities. Even though, the GPU is specifically designed for massive parallelization, e.g., in 3D graphics processing, it is still a general-purpose processor. Furthermore, just as the CPU, it has the issue of the von Neumann bottleneck. [44]

Unlike the CPU and the GPU, the TPU is a custom-specific processor, designed for matrix operations. Its custom design makes it possible to perform massive multiplications and additions at very high speed and low power consumption, as required by large neural networks. The advantage of TPUs over GPUs is that due to their specialization in matrix processing, they already know the next step of the operation. Furthermore, the von Neumann bottleneck can be overcome with the architecture of the TPU. [42]

CPUs are suited for quick prototyping when maximum flexibility is required, for simple models that do not need much time to train or for small models with small effective batch sizes. Meanwhile, GPUs find their use for models with a large number of custom TensorFlow operations or for operations which are not available for the Cloud TPU. GPUs are suitable for models of medium to large size, with large effective batch sizes. The

TPU works well for models that are dominated by matrix computations and that do not have custom TensorFlow operation within the training loop. The TPU converges the runtime of models that took weeks to train to hours and is therefore useful for large and very large models that have very large effective batch sizes. However, the TPU will not perform well for linear algebra programs which need frequent branching or programs that are dominated elementwise by algebra. Furthermore, it is neither suited for high-precision arithmetic workloads such as double-precision arithmetic nor for neural network workloads with custom TensorFlow operations used in the main training loop. And finally, the TPU should not be used for workloads that sparsely access memory. [44]

2.3.3. Tensor Processing Unit Architecture and Versions

TPU v1

The first version of the TPU, the TPU v1, consisted of one chip with one core and could only perform predictions. Google reduced the precision of calculation by only using eight-bit integers instead of 32-bit floating point numbers which allowed to fit more integer multiplier units into a single core and reduced energy consumption: six times less for eight-bit multiplication, and 13 times less for eight-bit additions. Furthermore, Google argues that using eight-bit integers is normally good enough for inference. This first version of the TPU was only used in Google's data centers and never released to the public. [3]

TPU v2

The TPU v2 board consists of four TPU chips. Two cores with 8 Gibibyte (GiB)[1] of High-Bandwidth Memory (HBM) are placed on each chip, which means that a board has a total of 16 GiB of HBM. Each core contains a Matrix Multiply Unit (MXU), Vector Processing Unit (VPU), and scalar unit. [6] Figure 2.6 shows the processor layout of the TPU v2 chip.

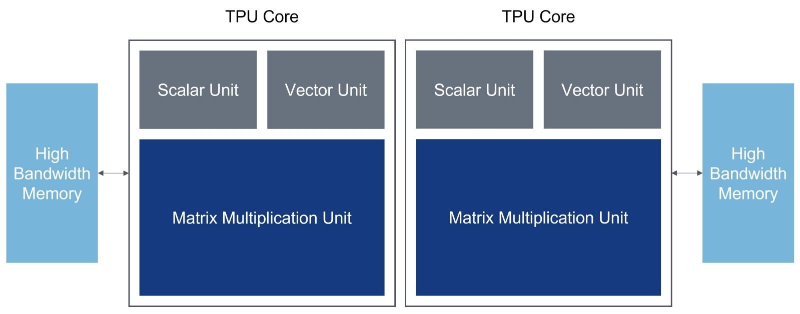


Figure 2.6.: TPU v2 Processor Layout of the Chip (adapted from [6])

[1] 1 GiB $\approx$ 1.073741824 GB = 1024 Mebibytes (MiBs) = 2^{30} Bytes = 1.073.741.824 Bytes

TPU v3

Like the TPU v2, the TPU v3 contains four chips each with two cores equipped. Each core has a memory capacity of 16 GiB, which is double the size compared to the TPU v2 and contains two MXUs. In addition to the MXUs, the core consists of a VPU and a scalar unit. [6] Figure 2.7 depicts the processor layout of the TPU v3 chip. The

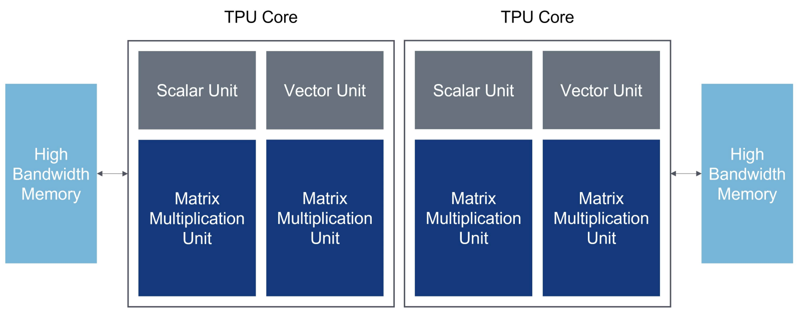

TPU Core
TPU Core
High Bandwidth Memory
Scalar Unit
Vector Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Scalar Unit
Vector Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
High Bandwidth Memory

Figure 2.7.: TPU v3 Processor Layout of the Chip (adapted from [6])

additional memory capacity as well as the increased Floating Point Operations Per Second (FLOPS) per core, allow for an improved performance: Compute-bound models achieve significant better performance on the TPU v3 than on the TPU v2. Memory-bound models achieve performance improvements only if they are not compute-bound at the same time. However, if the data does not fit in the memory of a TPU v2 configuration, an upgrade to the TPU v3 can improve performance as well as reduce the re-computation of interim results. Moreover, TPU v3 allows for larger batch sizes. [6]

Detailed Architecture of TPU v2 and TPU v3

The architecture of the TPU v2 and v3 are quite similar. A single TPU board consists of four chips which contain two TPU cores, respectively. Each TPU core has a Scalar Unit, a VPU, and one or more MXUs. The cores on the TPU board allow for executing user computations independently. The chips can communicate directly with each other via high-bandwidth interconnects. The exact layout of the TPU device depends on the version of the TPU. Furthermore, the amount of HBM, the interconnects between the cores on each board as well as the networking interfaces used for the inter-device communication vary. [6]

Figure 2.8 shows the six major blocks of a TPU core. In Figure 2.9 their placement in the TPU v2 chip is depicted (floorplan of the TPU v2).
When the TPU project was started, the TPU was designed to be a coprocessor connected via Peripheral Component Interconnect Express (PCIe) I/O bus to the CPU which makes it possible to plug it into existing servers. This architectural decision was made in order to reduce development time and to avoid a deployment delay.
The TPU v2 uses the HBM Dynamic Random Access Memory (DRAM). It connects the TPU v2 chip via 32 128-bit buses to four short stacks of DRAM chips. The core

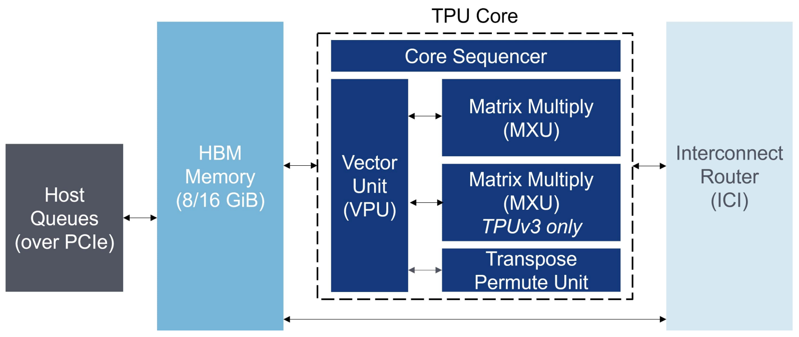


Figure 2.8.: Block Diagram of the TPU Core of TPU v2/v3 (adapted from [45])

sequencer fetches Very Long Instruction Word (VLIW) instructions from the on-chip, software managed instruction memory of the TPU core. It furthermore executes scalar operations and forwards vector instructions to the VPU. In total, eight operations can be launched by the VLIW: two scalar, two vector ALU, vector load and store, as well as a pair of slots which queue data to and from the MXU and transpose units.

The VPU performs vector operations. It executes general computations such as activation or softmax functions. The VPU uses a large on-chip vector memory and 32 two-dimensional vector registers. Each of the registers contains 128 times eight 32-bit elements. From the vector memory, the VPU collects and distributes data via data-level parallelism and instruction-level parallelism. Data is streamed to and from the MXU by decoupling First In – First Outs (FIFOs) from the VPU. [45]

The scalar unit is responsible for maintenance operations such as control flow and calculating memory address [6].

The MXU, comprising of 128 times 128 multiply/accumulators in a systolic array, provides the majority of the compute power. Each MXU is able to perform 16,000 multiply-accumulate operations in each cycle. The MXUs use the brain floating point (bfloat16) number format, which is a custom 16-bit floating point representation[2] providing a higher accuracy of training and model than the half-precision floating point representation of the IEEE standard. [6] The TPU software stack provides automatic, seamless conversion between 32-bit IEEE floating point and bfloat16 on the Cloud TPU. To accelearate the matrix multiplication operations, the format bfloat16 is carefully used. For each multiplication in the multiply-accumulate operation bfloat16 is used, for each accumulation, 32-bit IEEE floating point is used. Even though the conversion is done automatically, in some cases it may be useful to save the values manually in bfloat16 format. [47]
The MXU is the heart of the TPU. It uses systolic executions which saves energy since it reduces read and write access of the unified buffer. In regular intervals, the data arrives

[2]For details refer to [46]

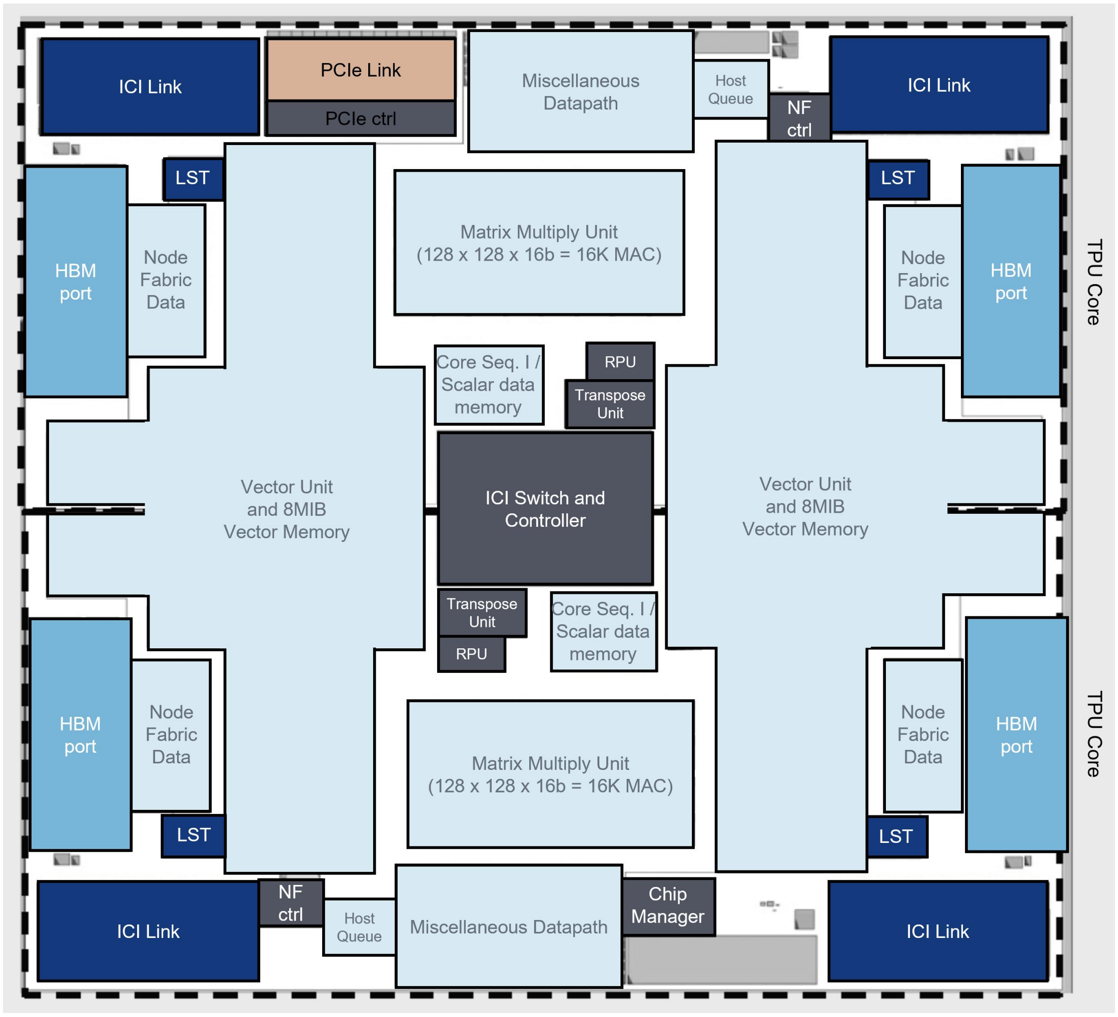


Figure 2.9.: TPU v2 Chip Floor Plan (adapted from [45])

in an array from different directions at the cells where these are combined. As shown in Figure 2.10, the data comes in from the left. The MXU loads the weights from the top, after they were preloaded. This causes the given multiply-accumulate operations to flow through the matrix like a diagonal wave. With each advancing wave, the weights and the data become effective. It seems like the 256 input elements are read at once because control and data are pipelined. Technically, the software does not know about the systolic array but performance-wise it is concerned with the latency of the unit. [3, 48] The required bandwidth for feeding and obtaining results from the MXU is proportional to its perimeter whereas the computation that is provided by the MXU is proportional to its area [45].

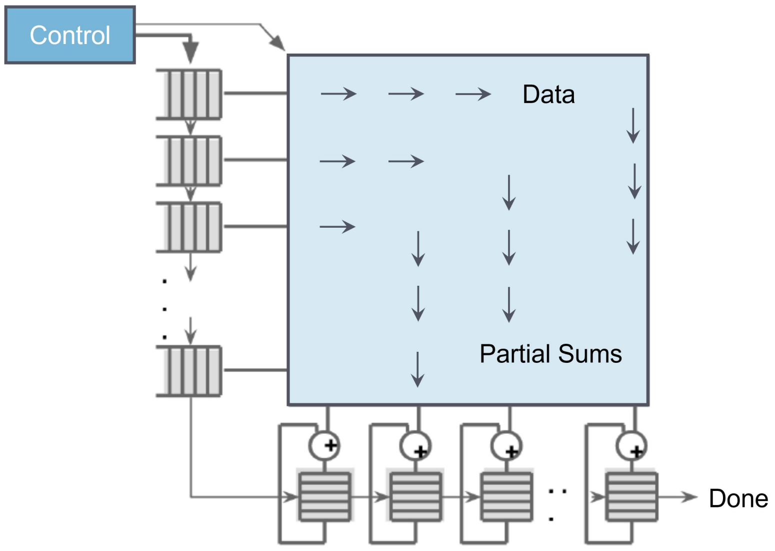


Figure 2.10.: Systolic Data Flow of the Matrix Multiplication Unit (adapted from [3])

TPU v4

The latest version of the TPU, TPU v4, has been announced on May 18, 2021 at the Google I/O event. The TPU v4 chip is twice as fast as the TPU v3 chip. It is able to deliver up to 10^{18} FLOPS computing power. The TPU v4 is already widely deployed in Google's data centers and used for internal machine learning workloads. To the public, it is currently only available upon special request. [49]
Figure 2.11 shows the different versions of the TPU board, v1 - v4.

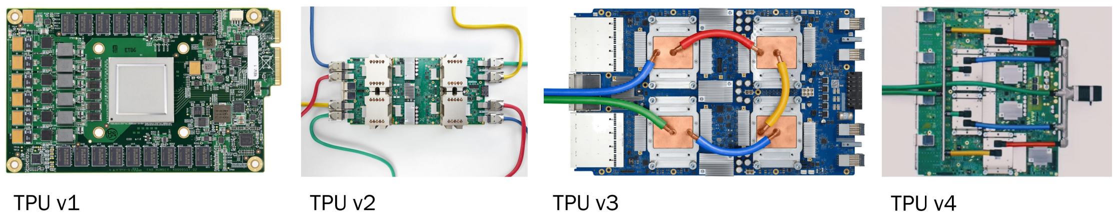


Figure 2.11.: TPU Board Versions v1 - v4 [3, 45, 50]

2.3.4. The Cloud Tensor Processing Unit System

The TPU is accessible to the user as a service, the Cloud TPU, through the GCP. This makes the cloud resources scalable. The Cloud TPU API allows for automated and flexible TPU management. [6]

TPU Configurations

There are three configurations of the TPU available: TPU Pods, as can be seen in Figure 2.12, TPU slices, and single TPU boards which are available to the user via the GCP. TPUs were designed to be extended to a TPU Pod, which is a supercomputer that can

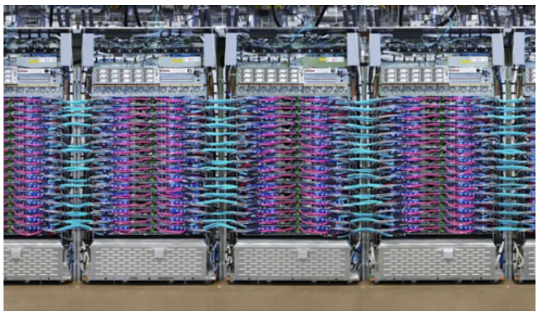


Figure 2.12.: TPU v3 Pod [45]

contain up to 2,048 TPU cores, i.e., up to 1,024 chips (TPU v2 and v3). [6] The TPU v4 Pod contains 4,096 chips [49]. It allows for distributing the processing load across several TPU boards. As a consequence the machine learning workloads have access to a larger group of memory. TPU Pods can be portioned into TPU slices if the entire resources of a TPU Pod are not needed. TPU slices are available for 32, 128, 512, 1,024, or 2,048 cores. A single TPU board is not part of a TPU Pod, but a stand-alone device. [6]

Cloud Tensor Processing Units Virtual Machine Architectures

TPU boards are connected via high-speed network interfaces to a CPU-based host-machine (= TPU host). Since TPUs are only capable of matrix operations, the host machine is used to perform all other operations such as loading and preprocessing data, and finally sending it to the TPU. Within a TPU Pod, each TPU board is connected to a TPU host. Figure 2.13 shows the architecture of TPU and TPU host.

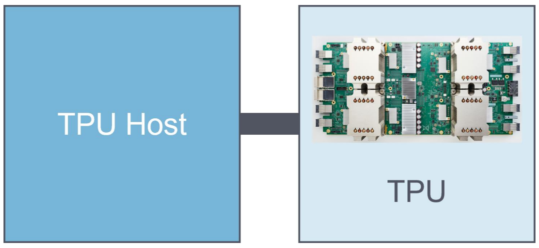


Figure 2.13.: TPU - TPU Host Connection (adapted from [6])

For the interaction between TPU host and TPU, there are two options available: TPU Nodes and TPU Virtual Machines (VMs). TPU Nodes (see Figure 2.14), which is the original setting, need an additional user VM. The communication between user VM and

TPU host works over gRPC[3] which means that the TPU host is only indirectly accessed. Using TPU VMs (see Figure 2.15), the Google Compute Engine VM, which is physically connected to the TPU board, is directly accessed via SSH connection. Getting root access to the VM, the user can run arbitrary code and access debug logs of compiler and runtime, as well as error messages. [6]

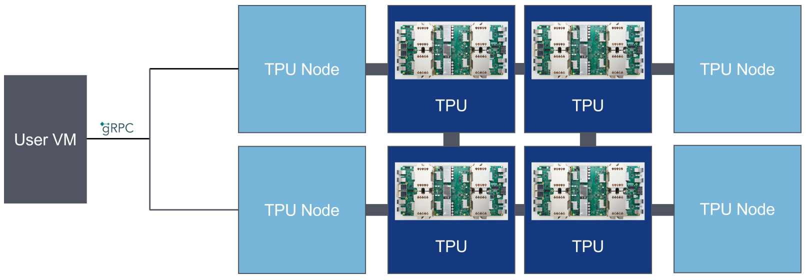


Figure 2.14.: TPU Node Architecture (adapted from [6])

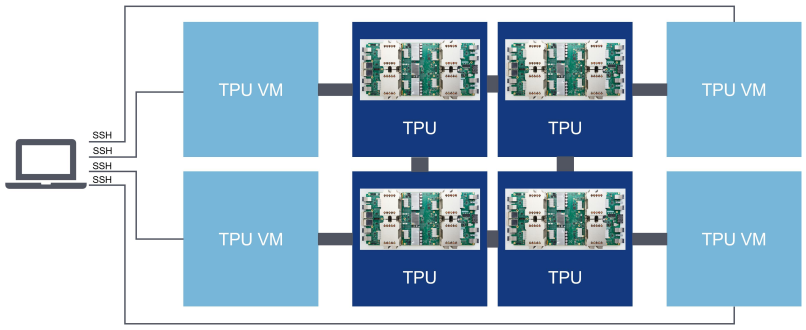


Figure 2.15.: TPU Virtual Machine Architecture (adapted from [6])

Supported Frameworks

The TPU supports several frameworks like TensorFlow, JAX, and PyTorch. The shared library `libtpu` gives these frameworks access to the TPU VM. It includes:

- the Accelerated Linear Algebra (XLA) compiler for compiling TPU programs,
- the TPU runtime for running compiled programs,
- the TPU driver for low-level access to the TPU used by the runtime. [6]

[3]Remote Procedure Call, developed by Google

2.3.5. Performance Evaluations and Applications of the Tensor Processing Unit

When Google announced the TPU, they published a paper as well as an article [3, 51] along with it, which covered together with a general introduction of the TPU, a performance comparison of the TPU (v1) with a CPU (18-core, dual-socket Haswell processor from Intel) and a GPU (NVIDIA K80 GPU) in terms of Roofline, response time, and throughput. The Roofline comparison showed that the TPU works well for DNNs. Five of the six tested applications[4] performed close to the Roofline of the TPU (see Figure 2.16), while they were further below for Roofline of the used CPU (see Figure 2.17) and GPU (see Figure 2.18).

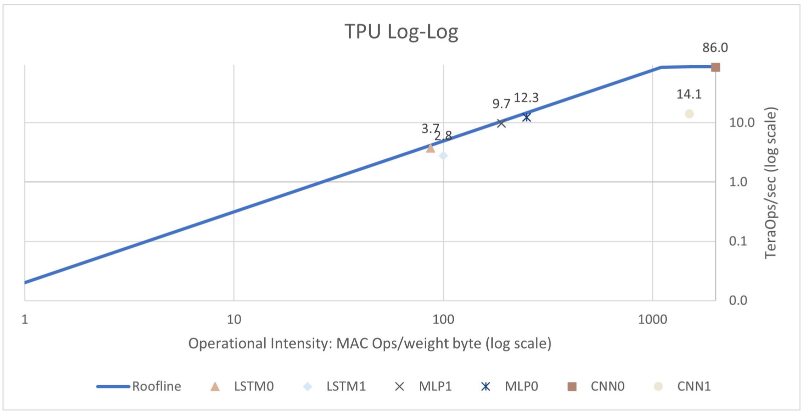


Figure 2.16.: TPU Roofline: The LSTMs and MLPs are under the oblique line of the graph, which means they are constrained by memory bandwidth. The CNNs are bound by the peak computation rate. The ridge point is at 1,350 multiply-accumulate operations per byte. (adapted from [3])

According to Google, response time is the reason for the good performance of the TPU, especially in inference. The 99th-percentile limit for the response time was seven milliseconds, server host time and accelerator time included. The CPU and the GPU only achieved 42 percent and 37 percent, respectively, of the highest throughput that could be achieved for MLP0 when the limit for the response time was loosen up. The TPU was affected by the bounds as well, but achieved 80 percent of its highest MLP0 throughput. The authors explain this due to the minimalism of the domain-specific processor which has no caches, branch predictions, multiprocessing, out-of-order-processing and so on. The throughput can be increased by using larger batch sizes. However, this also increases the response time to such an extent that it exceeds the set limit. For this reason, CPUs and GPUs must use batch-sizes that are smaller and consequently, less-efficient. These

[4]Two Long Short-Term Memory applications, a type of Recurrent Neural Networks (RNNs), (Long Short-Term Memory (LSTM)0, LSTM1), two Multilayer Perceptron applications (Multilayer Perceptron (MLP)0, MLP1), and two Convolutional Neural Networks (Convolutional Neural Network (CNN)0, CNN1)

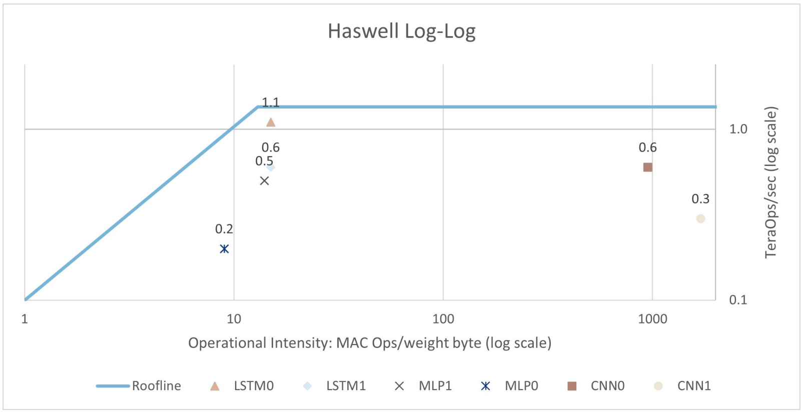


Figure 2.17.: CPU Intel Haswell Roofline: LSTM0 and MLP1 are faster on the CPU than on the GPU, vice-versa the other DNNs are faster on the GPU. The performance of the DNN is limited by response time. The ridge point is at 13 multiply-accumulate operations per byte. (adapted from [3])

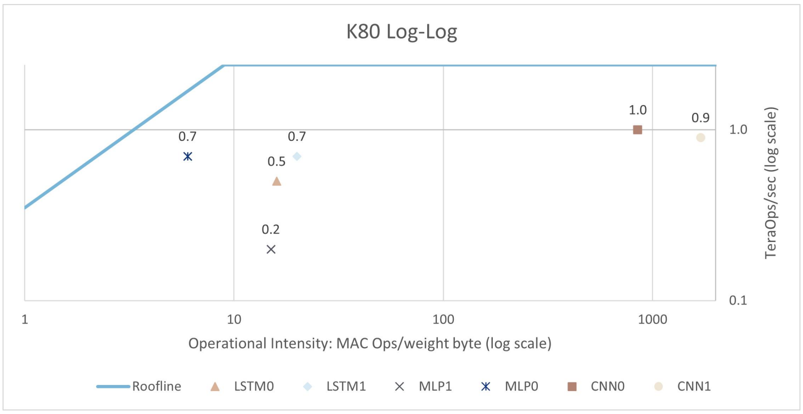


Figure 2.18.: GPU NVIDIA K80 Roofline: DNNs are far away from the Roofline. This is due to response time caps. The ridge point is at nine operations per byte and thus further to the left than in Figure 2.17 and is due to the higher memory bandwidth. (adapted from [3])

comparisons show that although the TPU is connected to a host via an I/O bus and also has limited memory bandwidth, the performance is still significantly better than that of the CPU or GPU used.
In 2020, the hardware and software engineers involved in the TPU project at Google published another paper [45], comparing the performance of the TPU v2, the TPU v3, and the Volta GPU developed by NVIDIA. The results of the comparison are depicted in Figure 2.19. They compared the chip performance based on five programs[5], which were

[5]ResNet50 (Deep Residual Network with 50 layers), Single Shot Detector (SSD), Mask Region-Based Convolutional Neural Network (RCNN), (Google) Neural Machine Translation (NMT), Transformer

submitted by both Google and NVIDIA to MLPerf 0.6 in July 2019 ([52]). The results showed a geometric speedup of these programs of 1.8 for the TPU v3 over the TPU v2 and a speedup of 1.9 for the Volta GPU over the TPU v2 (MLPerf 0.6 Geometric Mean (GM)). Comparing the chip performance for six production applications[6], similar to those of the TPU v1 comparison, the speedup relative to the TPU v2 is 1.8 for the TPU v3, and 0.4 for the Volta GPU (Production GM).

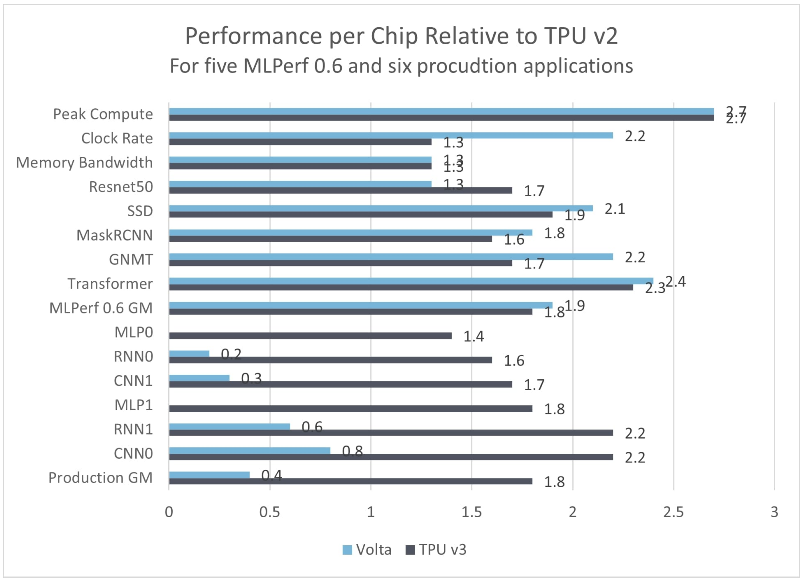


Figure 2.19.: Relative Comparison of TPU v2 to TPU v3 and GPU Volta (adapted from [45])

A concern of the developers was that inference latency could be hurt since the TPU v2 and TPU v3 require large batch sizes to run efficiently but it turned out that DNN models with batch sizes $\geq$ 1,000 can meet the latency targets. Considering that there are billions of Google users daily, the required batch sizes should not be a problem. Comparing the scaling performance on the ResNet50 MLPerf 0.6 of the TPU v3 and Volta, the results are quite similar for the TPU v3 chip at 52 percent accuracy.
Ying et al. [53] analyzed the scaling performance of the TPU v3 Pod with 1,024 chips. They achieved an accuracy of 76.3 percent in 2.2 minutes training ResNet50 on ImageNet. The throughput was more than one million images per second without a drop in accuracy. This shows the strength of the TPU v3 supercomputer in parallel scaling for production applications compared to a single TPU v3 chip or Volta GPU.
The research of Gordienko et al. [54] concludes that even the TPU v2 can achieve higher performance (throughput) compared to a GPU. The authors used algorithmically different DNNs, such as VGG16, ResNet50, Capsule Neural Network (CapsNet), for their analysis. As mentioned before, the performance of the TPU compared to a GPU depends on the batch size. This was also evaluated by Kochura et al. [55]. They investigated the impact

[6]MLP0, MLP1, RNN0, RNN1, CNN0, CNN1

of the largest possible batch size during training and inference on the standard MNIST and Fashion-MNIST data sets. For the training phase, an acceleration of ten times and for the prediction phase, one of up to two times was examined for batch sizes of > 512 images and $> 40{,}000$ images, respectively. Since the TPU v4 Pod was announced, Google bench-marked its performance against other Machine Learning hardware at MLPerf 1.0 [56]. The benchmark results demonstrated the outstanding performance of the TPU v4 supercomputers at scale [57]. It was able to achieve a speedup between 0.49 and up to 1.74 times over the fastest non-Google submission, the NVIDIA A100, in any of the available categories. [49]

The TPUs are deployed in Google's own data centers, for which they were developed and used for their solutions, such as industry solutions, AI, business application platforms, databases, data cloud, smart analytics, and many more [58], and products, e.g. AI and machine learning, data analytics, hybrid and multi-cloud, Internet of Things, and many more [59]. Beyond Google's own application, the TPU v2 and TPU v3 were used for developing a learned performance model. Because contemporary processors are complex, these models are usually difficult to develop. Phothilimthana et al. [60] showed that the developed model for the TPU was able to outperform a massively improved analytical performance model. Furthermore, learning the performance model could be automated and the model can be generalized well to similar programs. Islam et al. [61] used the TPU in an experiment for image recognition to classify flowers.

2.4. Evaluation Metrics

2.4.1. External Metrics

External metrics evaluate the performance of an algorithm by matching the clustering result to a reference result which is considered as the ground truth. An external evaluation criterion measures the degree of similarity of the clustering result obtained to the ground truth, e.g. the number of clusters and the category labels.
One index for comparing the resulting clustering $U = \{U_1, ..., U_r\}$ with r clusters to the ground truth, reference clustering $V = \{V_1, ..., V_s\}$ with s clusters, is expressed in the following contingency Table 2.1 for a two indicator function :

U\V	**Pairs in the same cluster**	**Pairs in different clusters**
Pairs in the same cluster	a	b
Pairs in different clusters	c	d

Table 2.1.: 2x2 Contingency

The parameters a, b, c, and d are defined as follows:

- a: The number of pairs of data points which are in the same cluster in U and V

- b: The number of pairs of data points which are in the same cluster in U but not in V
- c: The number of pairs of data points which are in the same cluster in V but not in U
- d: The number of pairs of data points which are in different clusters in U and V

The total number of pairs of data points is

$$M = a + b + c + d = \frac{n(n-1)}{2} \tag{2.2}$$

where n is the number of data points in the data set.
Indices which are based on this two indicator functions are Rand index, Jaccard index or Folkes and Mallows index. [10]

Since in real world data sets, reference results are often not available, external metrics are usually used for synthetic data and tuning clustering algorithms.

2.4.2. Internal Metrics

Internal metrics evaluate the clustering quality only using the data themselves. They assess the fit between the structure and the data, e.g. the structure of detected clusters and their relations to each other. One approach is to measure the degree to which a partition is justified by the given proximity matrix. Internal evaluation is generally used after the clustering algorithm terminated. For a few clustering algorithms, the metrics are also used in the validation phase. This is the case for k-Means, k-Medoids, and EM among others. [10, 62]

There are two important concepts for evaluating the clustering quality when using internal metrics: compactness and separation.

Compactness

Compactness measures the internal cohesion among objects in a cluster, i.e., how closely are the data points grouped in a cluster. The assumption is that points in a cluster are related to each other by sharing a common feature. Compactness is normally measured by calculating the distance between objects in a cluster. A commonly used approach is through calculating the variance - the average distance to the mean. A cluster has high compactness if the variance is small. [10, 62] Figure 2.20 shows an example for different degrees of compactness: on the left side, the clusters have higher compactness than on the right side.

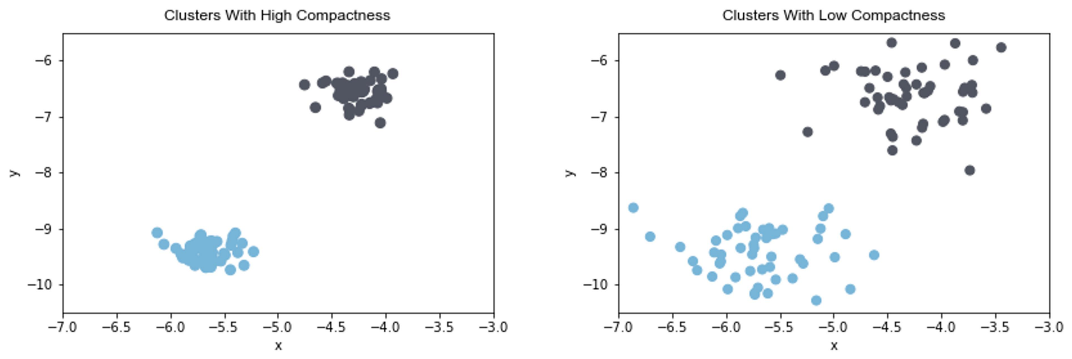


Figure 2.20.: Compactness of Clusters

Separation

Separation measures how isolated a cluster is, i.e., how different the detected cluster (or pattern) is from other clusters (or patterns). As for compactness, distance is a commonly used measure for identifying separation, e.g. by calculating the pairwise distance of centroids or the pairwise minimum distance between data points in different clusters. [10, 62] Figure 2.21 shows an example for different degrees of separation: on the left side, the clusters have better separation than on the right side.

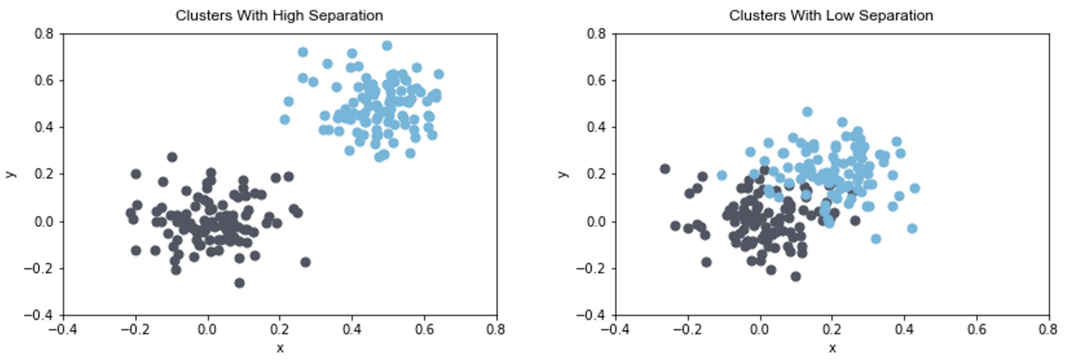


Figure 2.21.: Separation of Clusters

For a cluster to be valid, it must be unusually compact and unusually separated. Clusters are valid in a restricted sense if they are separated but not compact and vice-versa. All indices for the validity of individual clusters measure compactness and separation. It is advisable to define a reference population and a baseline for which "unusual" has a meaning. This meaning of "unusual" should be intuitive and derivable from theory. [10]

2.4.3. Relative Metrics

Relative criteria evaluate which of two structures is better in a certain way, e.g. more stable or better suited for the data. The criterion would e.g. quantitatively measure whether a single-link or a complete-link hierarchy is a better fit for the data. Relative metrics can also be used to find a suitable parameter configuration for an algorithm. The algorithm is repeated with different parameter settings. Using relative metrics, the best parameter setting is determined. [10]

3. Proposed Method

3.1. Goal

The goal is to develop a method to speed up the k-Means algorithm by using the TPU. The method should therefore take advantage of the benefits offered by the TPU. At the same time, the clustering quality should be preserved.

3.2. Requirements

The following three properties should be satisfied: First, the algorithm should be able to start with any initial centroids, so that all available adaptations and extensions that exist for initialization can be applied. Second, the clustering quality should be the same for the implemented k-Means on the TPU as for standard k-Means. And third, the developed method should be executable on the TPU as well as on CPU and GPU.

3.3. Concept

The proposed method builds on a bachelor thesis written at the University of Vienna from 2019 [63], in which the performance of the k-Means algorithm on the TPU (TPU v2) was compared with the performance on a CPU (Intel ® Xeon CPU). As part of the work, three versions of the k-Means were developed. The first version used only one of the eight available cores for the k-Means computations. The second version used all eight cores of the TPU. The input was distributed across, and the centroids were broadcasted to all cores. Each core then performed the assignment step. In-between iterations, the cores communicated their results to the CPU, on which these were combined and the update step was performed. The two steps were repeated, as with standard k-Means, until convergence. In the third version the data was distributed across all cores and each core performed the entire calculation of the k-Means and only reported back the final clustering results which were then combined on the CPU using the mean. The work concluded that the second version provides higher quality because the entire data set is seen before the new centroids are computed. Advantage of the third version is that it is faster since there is less communication between CPU and TPU. The first version is not interesting for further consideration, since it is outperformed by the second and third version.
The method developed in this master thesis uses the idea of implementing different versions, similar to the second and third version of the previous work, as a starting point to build on. The idea is to combine the two versions to achieve both benefits, speed and

accuracy.
As described in Chapter 2.3.4, tasks such as loading and preprocessing of data are performed on the CPU. In the proposed method, in the first step, the data is loaded on the CPU and the centroids are initialized. In addition, the data set is batched into smaller subsets. The actual computation of the k-Means algorithm takes place on the TPU. For this purpose, the batched data set is distributed to the cores and the initial centroids are transmitted to each core. Figure 3.1 and Figure 3.2 show the proposed method on a conceptual level. In version 1 *distribute_ repeat* only the assignment step is performed on the TPU and the assignments, the summed centroids as well as the number of data points per cluster are returned from the TPU to the CPU after each iteration. The sum of the centroids as well as the number of data points per cluster serve as input for the calculation of the new centroids. This calculation is performed on the CPU. The process is carried out until convergence. In version 2 *distribute_ mean*, the batched data set is distributed to the cores and the initial centroids are transmitted to each core just as in version 1 *distribute_ repeat*. In contrast to version 1 *distribute_ repeat*, however, all iterations up to convergence are performed on the respective TPU core and finally the assignments as well as the final centroids per core are communicated back to the CPU. On the CPU, the batches are then concatenated into a complete data set. The final centroids per core are merged to final centroids by calculating the mean.

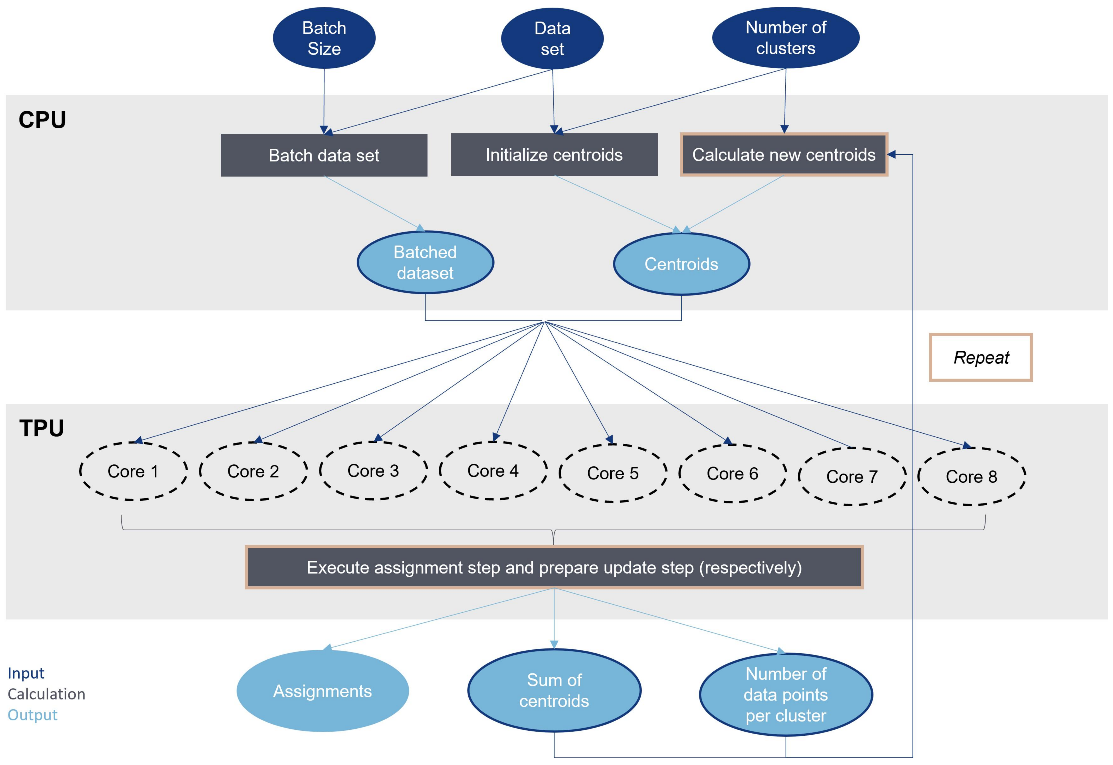


Figure 3.1.: Concept Version 1 *distribute_ repeat*

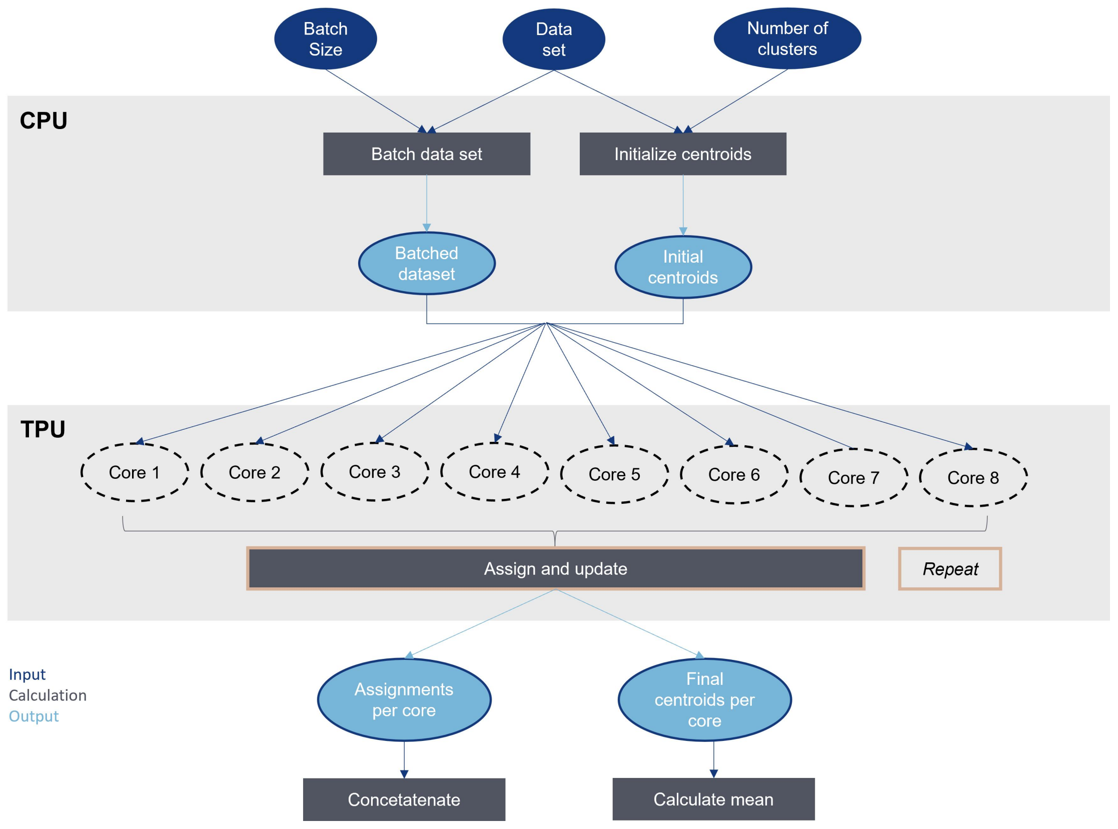


Figure 3.2.: Concept Version 2 *distribute_ mean*

Combining the two versions, there are two conceivable scenarios. First, version 1 *distribute_ repeat* is run for a certain number of iterations and then version 2 *distribute_ mean*. Thus, a high-quality starting position is first created before the algorithm is fully accelerated. Conversely, version 2 *distribute_ mean* can also be executed first and then version 1 *distribute_ repeat*. The algorithm is only accelerated at first and version 1 *distribute_ repeat* is used at the end to increase the clustering quality.

3.4. Parameters and Efficiency

3.4.1. Number of Objects and Number of Features

For efficient memory use, the input data should be structured so that it can be tiled into chunks of 128 by eight. As described in Chapter 2.3.4, the TPU supports several frameworks such as TensorFlow, JAX, and PyTorch which get access to the XLA compiler for compiling programs. The XLA compiler automatically pads tensors, if they do not fulfil the above mentioned criteria. To avoid padding, it is recommended to pick tensor dimensions that are well suited for TPU because padding requires higher on-chip memory storage for a tensor and out-of-memory errors can happen. Furthermore, the TPU cores are under-utilized with padded tensors. Last but not least, padding can significantly increase the execution time. [64]

3.4.2. Batch Size and Number of Features

For maximum runtime performance and to avoid padding, one of the following conditions should be satisfied:

- The batch size is a multiple of 64, which is a batch size of eight per core. The number of feature dimensions is a multiple of 128.

- The batch size is a multiple of 1,024, which is a batch size of 128 per core. The number of feature dimensions is a multiple of eight.

If one of these conditions is fulfilled, the recommendation for the chunk size described in the previous section is fulfilled. To achieve maximum efficiency, a batch size of 1,024 with 128 feature dimensions is recommended. [64]

3.4.3. Formulation of the k-Means Algorithm

As described in Chapter 2.2.2, the k-Means algorithm is composed of the two steps, assignment step and update step. In the assignment step, the data objects are assigned to the nearest centroid, the distance being determined by Euclidean distance. Determining the distance is the most computationally expensive part of clustering with k-Means. Since the systolic array architecture of the MXU (see Chapter 2.3.3), which provides the majority of the compute power, is specifically designed to perform thousands of multiply-accumulate operations, it appears particularly attractive to determine the distance calculations through a matrix multiplication. In both versions, Version 1 *distribute_repeat* and Version 2 *distribute_mean*, as introduced in section 3.3, the assignment step is performed on the TPU. A possible formulation of distance calculation with matrix multiplication can be defined as follows [37]: Let $X \in \mathbb{R}^{n \times d}$ be the matrix of data objects, $M \in \mathbb{R}^{k \times d}$ be the matrix of centroids and $m^T := [\langle c_0, c_0 \rangle, ..., \langle c_{k-1}, c_{k-1} \rangle]$ the vector of scalar products of each centroid with itself. Then the distance matrix $D \in \mathbb{R}^{n \times k}$ is defined as in equation 3.1:

$$D := \frac{1}{2} 1_n m^T - X M^T \tag{3.1}$$

where 1_n is a column vector of n ones. The cluster ID of point x_i is $argmin_j D_{i,j}$ since $D_{i,j} = \frac{1}{2}||x_i - c_j||^2 - \frac{1}{2}\langle x_i, x_i \rangle)$.

3.5. Internal Metrics for Validating the Clustering Quality

3.5.1. Notations

This section presents three well-known internal metrics that are used in the experiments for quality assurance.
First, we introduce the notations used in the formula of these internal metrics:

- X: The input data set
- n: The number of data points in X where x_i is the i-th data point
- C_j: Cluster $C_j, j = 1, ..., k$ with centroid c_j
- k: The number of clusters
- g: The center of the whole data set
- $d(x, y)$: The distance between data points x and y

For convenience, we will introduce an abbreviation for each metric and use it through the rest of this thesis.

3.5.2. Sum of Squared Error

The SSE was already introduced in section 2.2.2 as an error function that the k-Means algorithm tries to minimize. However, it is also suitable as an internal metric that describes the compactness of the clusters. For each cluster, it calculates the squared distance of the points in that cluster to the centroid of the cluster and then sums these values. It can be calculated with formula 3.2.

$$SSE = \sum_{i=1}^{k} \sum_{x_i \in C_j} d(x_i - c_j)^2 \tag{3.2}$$

3.5.3. Calinski-Harabasz Index

The Calinski-Harabasz Index (CH) [65] measures compactness and separation simultaneously. Compactness is reflected by how close the data points within a cluster are gathered around the the cluster centroid; separation is reflected by how much the centroids are spread. CH is formally defined as:

$$CH = \frac{\sum_j d^2(c_j, g)/(k-1)}{\sum_j \sum_{x \in C_j} d^2(x, c_j)/(n-k)} \tag{3.3}$$

A higher value of the CH index means that the clusters are well separated and dense, although there is no "acceptable" threshold.
CH calculates the clustering quality in the form of separation / compactness.

3.5.4. Davies Bouldin Index

Davies Boulding Index (DB) [66] is an old but widely used internal validation metric. The index calculates the average similarity of each cluster with a cluster most similar to it. A lower average similarity score indicates a better cluster separation and consequently a better clustering result. Equation 3.4 shows the formal definition of DB

$$DB = \frac{1}{k} \sum_{t} \max_{j \neq i} \frac{\frac{1}{n_i} \sum_{x \in C_i} + d(x, c_i) \frac{1}{n_j} \sum_{x \in C_j} d(x, c_j)}{d(c_i, c_j)} \tag{3.4}$$

3.6. External Metrics for Validating the Clustering Quality

3.6.1. Normalized Mutual Information Score

The Normalized Mutual Information (NMI) [67] is a measure for characterizing the accuracy of clustering algorithms. The method evaluates how mutual the information of two clusterings is. The method can be used to compare the clustering result of a clustering algorithm with the ground truth or, if this is not available, to compare it with the clustering result of another clustering algorithm. The mutual information is normalized by some generalized mean. The adjusted NMI is adjusted to account chance. NMI is generally higher for two clusterings with a larger number of clusters, regardless of whether more information is actually shared. The NMI is a symmetric measure.

3.6.2. Rand Index

The Rand Index (RI) [68] is a measure for comparing two arbitrary clusterings and evaluating their similarity. The measure is based on the contingency Table 2.1 introduced in Chapter 2.4.1. The metric is defined as follows:

$$RI = \frac{a + d}{M} \tag{3.5}$$

with $M = a + b + c + d$. The Adjusted Rand Index (ARI) [69] is corrected for chance by using the expected similarity. The general form of the index adjusted for chance is defined as follows:

$$ARI = \frac{RI - Expected\, RI}{Maximum\, RI - Expected\, RI} \tag{3.6}$$

The ARI is a symmetric measure.

4. Implementation

4.1. Project Setup

The implementation of the proposed method is written in Python. The project layout looks like this:

```
tpu-k-means/
├── evaluation/
│   └── all_results.csv
├── logs/
│   └── <YYYY-MM-DD>_log.log
├── results_tmp/
│   ├── <YYYY-MM-DD_hh-mm-ss>_<input_filename>_evaluation.txt
│   ├── data_final_labels_<input_filename>_<YYYY-MM-DD_hh-mm-ss>.csv
│   ├── final_centroids_<input_filename>_<YYYY-MM-DD_hh-mm-ss>.csv
│   └── initial_centroids_<input_filename>_<YYYY-MM-DD_hh-mm-ss>.csv
├── arguments.py
├── authenticate.py
├── data.py
├── evaluation.py
├── k-means-authentication.json
├── kmeans.py
├── main.py
├── README.md
└── requirements.txt
```

The Python script is started with the file *main.py*. In this file all important functions and processes are triggered. Important global and constant variables are defined in *main.py*, as well as the procedures for the versions described in Chapter 3.3. This file also queries available devices, in the order TPU, GPU or CPU, connects to the available device and initializes the corresponding strategy.

The functionalities are outsourced to the other Python files. *arguments.py* takes the terminal arguments *data_ dir* and *tpu_ name*. The authentication for Google happens in the file *authenticate.py* which requires a json file with the authentication credentials[1]. The file is provided by Google when creating a service account for the TPU project.

In the file *data.py* the corresponding data set is read in and if available, the true labels are queried. The file also defines the functionality for batching the data set. In addition, the logic for saving and uploading the processed data sets to the cloud is stored. The files

[1]The file contains individual authentication data for the Google Cloud and is therefore not provided.

with the clustering results, initial centroids, and final centroids are stored temporarily in the folder *results_tmp/* and can be uploaded to the Google Cloud Storage bucket.
The class *KMeans* is defined in the file *kmeans.py*. It contains the functions for initializing the centroids as well as the two clustering steps assign and update for version 2 *distribute_mean* or assign and prepare for version 1 *distribute_repeat*. Finally, the clustering results are evaluated in the file *evaluation.py*. Additionally, the set parameters, the quality metrics and the elapsed time are saved in a csv file. The file is stored in the *evaluation/* folder. The folder *logs/* contains the logs written during the execution.

The input data sets as well as the clustering and evaluation results are stored in a Google Cloud Storage bucket. This is recommended by Google as it allows for efficient loading of data and a continuous stream of data if the data is split across multiple files. The structure of the bucket created for the project looks like this:

```
k-means-bucket/
 ├─ evaluation/
 ├─ external/
 ├─ processed/
 └─ raw/
```

The directory *evaluation* contains the evaluation of the clustering results. In the directory *external*, the data sets of third parties, i.e. the real data sets, are stored. The folder *processed* contains the clustering results, i.e. the data sets with the computed labels, the final centroids and the initial centroids. The directory *raw* contains the unprocessed synthetic data sets.

4.2. Used Technology

The TPU supports different Python frameworks such as TensorFlow, JAX and PyTorch. Therefore, the code is is written in Python using TensorFlow (TensorFlow 2.8.0). TensorFlow is selected since the TPU is tailored to Google's library TensorFlow and because it is well suited for large-scale distributed training.
To run the code on TPUs, which are typically Cloud TPU workers, they need need to be initialized at the beginning of the program. This is done using the following APIs as listed in Listing 4.1 with `tpu_name` being the name set for the TPU. [70]

```
tpu_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu=
    tpu_name)
tf.config.experimental_connect_to_cluster(tpu_resolver)
tf.tpu.experimental.initialize_tpu_system(tpu_resolver)
```

Listing 4.1: Initialize TPU

TensorFlow offers several distribution strategies, to run training in parallel. For TPUs, this is `strategy = tf.distribute.TPUStrategy(tpu_resolver)`, for GPUs this is `strategy = tf .distribute.MirroredStrategy(["GPU:0", "GPU:1", "GPU:2"])`, which is the equivalent to the TPU strategy. It takes as many GPUs as arguments as are available. The default strategy is `strategy = tf.distribute.get_strategy()` which is applicable for CPUs as well as GPUs

with one replica in sync. [71]
Listing 4.2 shows the code for the distribution strategies as implemented. This way, the same script can be executed on TPU, GPU or CPU. `tpu_name` is set to None per default and must be set when the code is executed on a TPU.

```
physical_devices_cpu = tf.config.list_physical_devices('CPU')
physical_devices_gpu = tf.config.list_physical_devices('GPU')
try:
    tpu_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu=
    tpu_name)
except ValueError:
    tpu_resolver = None
if tpu_resolver:
    tf.config.experimental_connect_to_cluster(tpu_resolver)
    tf.tpu.experimental.initialize_tpu_system(tpu_resolver)
    strategy = tf.distribute.TPUStrategy(tpu_resolver)
else:
    if len(physical_devices_gpu) > 0:
        strategy = tf.distribute.MirroredStrategy(
            ["GPU:0", "GPU:1", "GPU:2", "GPU:3", "GPU:4", "GPU:5",
            "GPU:6", "GPU:7"])  # 8 replicas
    else:
        strategy = tf.distribute.get_strategy()
```

Listing 4.2: Strategies for Distributed Training

For efficient use of the TPU it is recommended to use the `tf.data.Dataset` API to feed data quickly to the Cloud TPUs. In the implementation, the API `tf.data.Dataset.from_tensor_slices` is used. To improve efficiency, all data files read by the Dataset should be stored in Google Cloud Storage buckets.
For clustering, the input data set is batched. The batch size is determined by two parameters: `BATCH_SIZE_PER_REPLICA`, which is the batch size, each core will see per iteration, and `strategy.num_replicas_in_sync` which is the number of devices available retrieved by the `strategy` API. The argument `drop_remainder` is set to `True` to prevent the smaller batch of being produced. Furthermore, this is recommended when using the XLA compiler which is the case for the TPU. [72] To distribute the training across multiple devices, i.e. the TPU cores or GPUs, it is necessary to create a `tf.distribute.DistributedDataset`. This can be done with the API `strategy.experimental_distribute_dataset`. It takes the batched data set as argument. The objects are Python iterables. [73]
For clustering with k-Means, we implemented a custom training loop using a pythonic for-loop construct which repeats the k-Means clustering for each subset in the `DisributedDatasest`. To replicate and run the computation on all cores, it must be passed into the API `strategy.run`. The API takes two arguments: the function to be executed and the arguments passed to the function. The API returns results from each local replica in the strategy. `tf.distribute.Strategy.reduce` aggregates the results of the replicas to one value. [70, 74] The training loops for version 1 *distribute_ repeat* and version 2 *distribute_ mean* can be found in the Appendix A.1, Listings A.1 and A.2.
The heart of the implementation is the k-Means algorithm. The k-Means is executed when `strategy.run` is called inside the custom training loop. The algorithm is imple-

mented in the class `KMeans`. The class comprises the methods `initialize_centroids`, `assign_fn`, `assign_and_prep_fn`, and `assign_and_update_fn`. The method `assign_fn` is used by both, `assign_and_prep_fn`, which is called when executing version 1 *distribute_ repeat*, and `assign_and_update_fn`, which is called when executing version 2 *distribute_ mean*. All methods use the `@tf.function` decorator for better performance. It converts Python eager code into graph-compatible TensorFlow operations. The assignment step, i.e. the distance calculations (see Listing 4.3), are implemented as matrix multiplication as described in Chapter 3.4.3. The code of the entire class `KMeans` can be found in the Appendix A.2, Listing A.3.

```
@tf.function
def assign_fn(self, points, centroids):
    """assign points to cluster, return assignment"""
    # compute scalar products (m^T) of each centroid with itself
    m = tf.linalg.diag_part(tf.tensordot(centroids, centroids, [[1],
   [1]]))
    # Column vector of n ones (1_n)
    n = tf.ones(tf.shape(points)[0])
    # calculate 0.5 * 1_n * m^T
    d1 = tf.multiply(tf.constant([0.5]), tf.einsum('i,j->ij', n, m))
    # calculate X*M^T
    d2 = tf.matmul(points, centroids, transpose_b=True)
    # calculate distance D:= 1/2 * 1_n * m^T - X * M^T
    distance = tf.subtract(d1, d2)
    # find minimum distance
    assignments = tf.math.argmin(input=distance, axis=1, output_type=tf.
   int32)
    return assignments
```

Listing 4.3: Distance Calculation with Matrix Multiplication

At the end of the script, the labels are assigned to the remainder, that has been dropped while batching. And last but not least, the clustering quality is measured.
The script takes csv files as input files. It is possible to use data sets with and without true labels. Accordingly, the `LABELS_EXIST` variable must be set to True or False at the beginning of the script. By setting the variable `INIT_CENTROIDS_FIXED` to False or True the initialization of the centroids happens randomly or the first k data points are chosen as initial centroids. In addition, suitable values must be found for the following variables: `EPOCHS`, i.e. the number of iterations the first version runs (fixed number of iterations), `MAX_EPOCHS`, i.e. number of iterations the second version runs (not fixed), `TOLERANCE` and `BATCH_SIZE_PER_REPLICA`. The number of clusters is given for each of the used files.

4.3. Used Hardware

TPU Environment: We use a single device Cloud TPU v3 via a VM with 128 GiB RAM memory [7]. TPU v3 is the latest available version. It is is offered in two different regional zones, Iowa (us-central1) and Netherlands (europe-west4). For optimal performance the TPU should be located in the same zone as the Google Cloud Storage bucket.

We choose Iowa because the TPU there is less expensive and the data we use is not sensitive. It costs $8.00 per hour [75]. The cloud storage costs $0.02 per GB per month [76].

GPU Environment: We use a NVIDIA Tesla K80 with 12.68 GB RAM memory via Google Colab (free version). The GPU loads the data from the Google Cloud Storage bucket.

CPU Environment: We use a Intel® Core™ i7-8550U CPU with 1.80GHz and 8,192 MB RAM memory. The CPU loads the data from the Google Cloud Storage bucket.

4.4. Data Sets

4.4.1. Synthetic Data Sets

For a systematic analysis of the proposed method, we generated several synthetic data sets. The data sets vary in the parameters of number of data points n, number of dimensions d and number of clusters k. This is to cover a wide range. The clusters are Gaussian distributed (standard deviation 1.5) and are randomly distributed in the data space. The synthetic data sets are optimized for the TPU to exploit its full potential (see Chapter 3.4.2 for reference). Table 4.1 shows all synthetic data sets used.

No.	***n***	***d***	***k***
1	10,240	8	8
2	10,240	32	16
3	10,240	64	16
4	102,400	64	32
5	102,400	128	64
6	102,400	512	64
7	512,000	128	128
8	512,000	256	64
9	512,000	512	32
10	1,024,000	64	32
11	1,024,000	128	128
12	1,024,000	256	64

Table 4.1.: Synthetic Data Sets

The files are named according to the following scheme: data_<n>_<d>_<k>.csv.

4.4.2. Real Data Sets

We use three publicly available real data sets. The data set *Bank Marketing* [77] is from the UCI Machine Learning Repository [78]. It contains features related with bank client, product and social-economic attributes and is used to predict the success of telemarketing

calls for selling term deposit subscriptions. Therefore, the data set consists of two classes with labels 'yes' or 'no'. Since a few features are text or categorical features, we preprocessed the data set converting these features into numerical features.
The second data set *Internet of Things (IoT) Botnet* [79] is from Kaggle database [80]. It is a collection of nine network attack data sets and is already preprocessed. We used the data of the botnet malware 'Mirai' attack. Since the true labels are available, we derived the number of clusters from there.
The third data set is the *MNIST* [81] data set from Keras database. The data set is split into a train set (60,000 objects) and a test set (10,000) objects. Since we use the data set for clustering, we concatenated the train and test set. The data set contains image data, and for this reason, it is preprocessed with $k = 256$ as the optimal number of clusters.
Table 4.2 gives an overview of the properties[2] of the data sets.

Data Set	n	d	k
Bank Marketing	41,188	48	2
IoT Botnet	764,137	115	2
MNIST	70,000	784	256

Table 4.2.: Real Data Sets

4.5. Experimental Setup

The bachelor thesis was used as inspiration for the proposed method. Since the state of technology is outdated, the project is restarted and set up from scratch again using only the idea of implementing two versions.
The experiment is divided into three parts: In the first part, two important parameters, batch size and convergence tolerance, are optimized or constrained. In the second part, version 1 *distribute_repeat* and 2 *distribute_mean* as described in Chapter 3.3 are compared against each other to verify if the findings of the bachelor thesis still hold for the newer stake of technology. In the third part, both versions are combined on the basis of these findings, and the combination is optimized.
To make all experiments comparable, the first k data objects were always chosen as initial centroids. In order to make the experiments transparent, understandable and replicable, the code is made available in the GitLab repository of the University of Vienna[3]. To run the experiments, *main.py* is adapted. In each section it is mentioned which of the adaptations should be used in order to replicate the experiment. An overview can be found in Table A.4 in Appendix A.4.

[2]Properties after data preprocessing
[3]`https://git01lab.cs.univie.ac.at/wolffa95/tpu-k-means`

4.5.1. Experiment 1: Batch Size

To achieve the highest possible efficiency on the TPU, first the parameters batch size and convergence tolerance ε (allowed deviation between the previous centroids and the new centroids) are optimized by a few experiments.

Five different batch sizes (per replica) are tested for each of the synthetic data sets. For the real data sets, nine different batch sizes are tested; four of them optimized for the data set, and five of them optimized for the TPU. A distinction is made between these two optimizations because drop_remainder is set to True in batching for efficiency reasons. Since real data often does not meet the criterion that the number of objects is divisible by 64 or 1,024, optimized for the data set means that the remainder is kept as small as possible. As a consequence, as many data objects as possible are clustered with k-Means and only as few objects as possible need to be predicted.
Optimized for the TPU means that the batch size is a multiple of 64 or 1,024 and the size of the remainder is neglected.
Table 4.3 shows with which batch sizes the data sets were tested as well as in how many batches this batch size results. The TPU optimum refers to the setting of a global batch size of 1,024, i.e., a batch size per replica of 128. The batch size that results in the shortest runtime is used for further experiments.

Number of Objects / Number of Batches	***TPU Optimized***				
	TPUit Optimum	***20***	***10***	***5***	***1***
10,240 (1-3)	*128*	64	128	256	1,280
102,400 (4-6)	*128*	640	1,280	2,560	12,800
512,000 (7-9)	*128*	3,200	6,400	12,800	64,000
1,024,000 (10-12)	*128*	6,400	12,800	25,600	128,000
41,188 (Bank Marketing)	*128*	256	512	1,024	5,120
764,137 (IoT Botnet)	*128*	4,736	9,536	-	-
70,000 (MNIST)	*128*	384	832	1,728	8,704
	Data Set Optimized				
41,188 (Bank Marketing)	-	257	514	1,029	5,148
764,137 (IoT Botnet)	-	4,775	9,551	19,103	95,517
70,000 (MNIST)	-	437	875	1,750	8,750

Table 4.3.: Tested Batch Sizes

For the data set *IoT Botnet*, only three "TPU-optimized" settings are tested, since the number of objects of the remainder for the other two settings is almost the same number of objects of the entire data set.
The experiment can be replicated by running *main_repeat_batch_loop.py*

4.5.2. Experiment 2: Convergence Tolerance

The convergence tolerance is tested for two values, $\varepsilon = 0.001$ and $\varepsilon = 0.01$, since the goal is to speed up the k-Means algorithm. To ensure that quality is maintained, this experiment is performed with standard k-Means and k-Means with matrix multiplication. The experiment is performed with data set 7 (data_512000_128_128.csv) with a batch size per replica of 12,800. The number of iterations is fixed at 20. The experiment can be replicated by running *main_ repeat.py.*

4.5.3. Experiment 3: Comparison of Versions

In the second part of the experiment, the two versions presented in Chapter 3.3 are each executed in isolation on the TPU using the k-Means Algorithm with matrix multiplication. This step is to check whether the findings from the bachelor thesis still apply to the newer state of technology. The experiment is only applied on the synthetic data sets. Each of the data sets is executed four times with a fixed number of iterations (5, 10, 20, 100) and once with a variable number of iterations of maximum 100 iterations. The experiment can be replicated by running *main_ repeat_ loop.py.*

4.5.4. Experiment 4: Combination of Versions

Assuming Experiment 3 confirms the results, it sounds promising to combine the two versions. This way, it is possible to receive the benefits of both: the quality of version 1 *distribute_ repeat* and the speed of version 2 *distribute_ mean.* Based on the findings, both versions are combined. The combination is tested in both directions: first version 1 *distribute_ repeat* and then version 2 *distribute_ mean* (v1 $\rightarrow$ v2), and vice versa (v2 $\rightarrow$ v1). Both combinations are repeated five times for each of the twelve synthetic data sets. The number of iterations for the first version is fixed to 2, 5, 10, 20, and 50, respectively. For the second version being executed, the number of iterations is variable and set to a maximum of 100. In total, a maximum of 150 iterations is possible. In addition to comparing both combinations to each other, the results of this experiment are also compared to the runtime results of version 1 *distribute_ repeat* and version 2 *distribute_ mean* from Experiment 3, when both versions were executed in isolation. This way, it is possible to see whether combining both versions is beneficial. The experiment is conducted on both, synthetic as well as real data sets. The experiment can be replicated by running *main_ combine_ loop.py.*
When the experiment was conducted, it was extended by two parts. These can be replicated by executing the files *main_ repeat_ loop2.py* and *main_ combine_ loop2.py*

4.5.5. Experiment 5: Performance Comparison to GPU and CPU

In Experiment 5, the performance of the implementation on the TPU is compared to the performance on a GPU and CPU. Based on the results of Experiment 4, the version / combination with the best performance is used for this comparison. The experiment can be replicated by running *main_ repeat_ loop3.py.*

5. Evaluation and Discussion

5.1. Experiment 1: Batch Size

The results of this experiment are saved in the file *all_results_sep.csv* as well as in the log file *2022-04-24_log.log*. To evaluate the experiment, we filtered the csv-file by the number of objects, since the number of batch sizes should correlate to the number of objects in the data set. For each of the two versions, we marked the batch size with the shortest runtime per number of objects n. The runtime is always measured for only the k-Means clustering if not stated differently. The batch size that most often resulted in the best runtime per n is shown in Table 5.1. This batch size is used for all further experiments.

Number of Objects	***Data Sets***	***Best Batch Size Per Replica***
10,240	1-3	256
102,400	4-6	2,560
512,000	7-9	12,800
1,024,000	10-12	25,600
41,188	Bank Marketing	5,120
764,137	IoT Botnet	19,103
70,000	MNIST	1,750

Table 5.1.: Best Batch Sizes

The statement that the setting with a global batch size of 1,024, i.e. a batch size per replica of 128, in combination with 128 dimensions on the TPU is optimal, does not apply to clustering with k-Means. As seen in Figures 5.1, 5.2, and 5.3, the runtime for batch size per replica of 128 is (significantly) worse.

For the real data sets, a distinction was made in the experiment between "data set optimized" and "TPU optimized". As can be seen in Figure 5.4, the results hardly differ. However, it should be considered in the decision that predicting the remainder takes time. The selected batch sizes for the real data sets can also be found in Table 5.1.

Discussion: The experiment showed that large batch sizes are possible when clustering with k-Means on the TPU. The batch sizes are many times larger than recommended by Google in all tested cases. One explanation is that the calculation of the k-Means algorithm is much less complex than that of DNNs, for which the TPU was developed and tested.

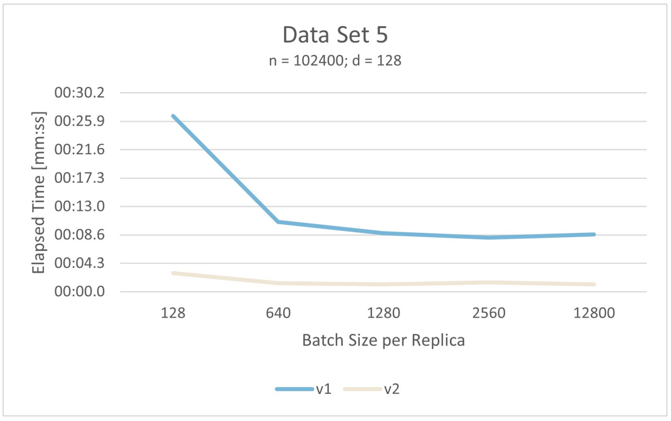


Figure 5.1.: Batch Size Per Replica Runtime Comparison

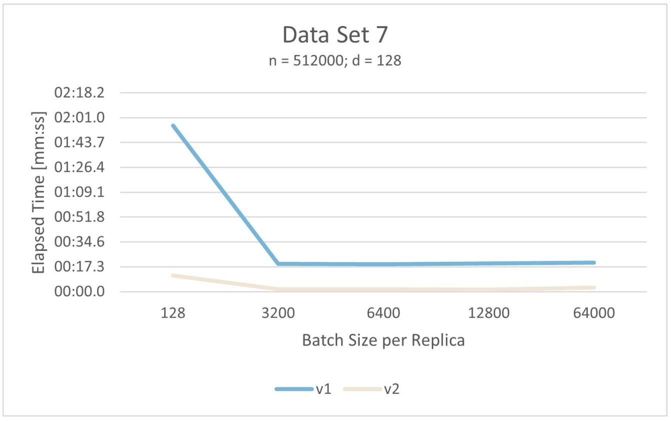


Figure 5.2.: Batch Size Per Replica Runtime Comparison

5.2. Experiment 2: Convergence Tolerance

Experiment 2 shows that reducing the convergence tolerance from $\varepsilon = 0.001$ to $\varepsilon = 0.01$ has no effect on clustering quality as can be seen in Figure 5.5.
As Figure 5.6 shows, the runtime can be improved by lowering the tolerance. For this reason, the convergence tolerance is set to $\varepsilon = 0.01$ for the following experiments.
Furthermore, the metrics are at least equally good for both k-Means, with matrix multiplication and standard k-Means. This holds for version 1 *distribute_ repeat* as well as version 2 *distribute_ mean*. The code for standard k-Means as implemented can be found in Appendix A.3.
The evaluation of this experiment is based on the data collected in the files *all_ results_ sep.csv* and *2022-04-24_ log.log*.

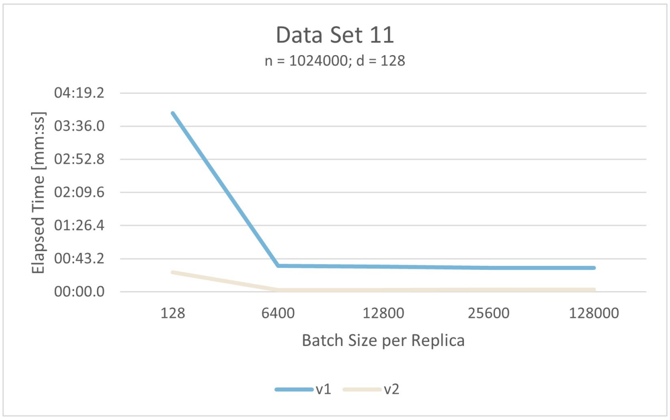


Figure 5.3.: Batch Size Per Replica Runtime Comparison

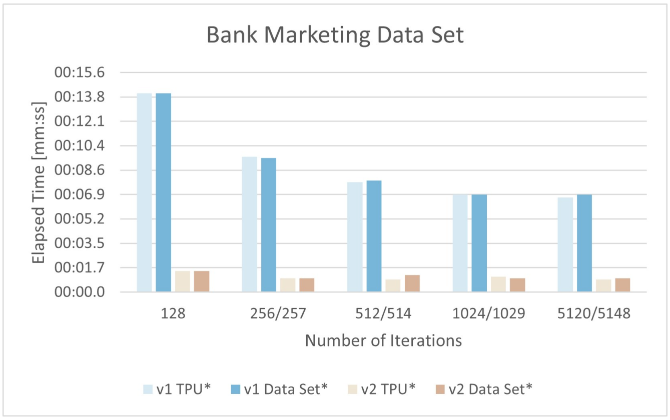
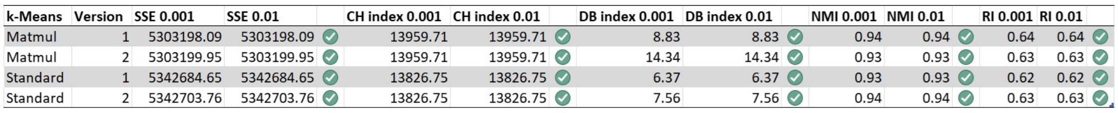
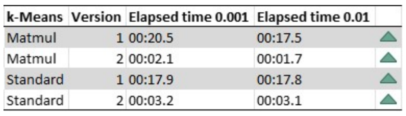


Figure 5.4.: Comparison of TPU-Optimized and Data Set Optimized Batch Size



k-Means	Version	SSE 0.001	SSE 0.01		CH index 0.001	CH index 0.01		DB index 0.001	DB index 0.01		NMI 0.001	NMI 0.01		RI 0.001	RI 0.01	
Matmul	1	5303198.09	5303198.09	✓	13959.71	13959.71	✓	8.83	8.83	✓	0.94	0.94	✓	0.64	0.64	✓
Matmul	2	5303199.95	5303199.95	✓	13959.71	13959.71	✓	14.34	14.34	✓	0.93	0.93	✓	0.63	0.63	✓
Standard	1	5342684.65	5342684.65	✓	13826.75	13826.75	✓	6.37	6.37	✓	0.93	0.93	✓	0.62	0.62	✓
Standard	2	5342703.76	5342703.76	✓	13826.75	13826.75	✓	7.56	7.56	✓	0.94	0.94	✓	0.63	0.63	✓

Figure 5.5.: Quality Comparison of $\varepsilon = 0.001$ and $\varepsilon = 0.01$



k-Means	Version	Elapsed time 0.001	Elapsed time 0.01	
Matmul	1	00:20.5	00:17.5	▲
Matmul	2	00:02.1	00:01.7	▲
Standard	1	00:17.9	00:17.8	▲
Standard	2	00:03.2	00:03.1	▲

Figure 5.6.: Runtime Comparison of $\varepsilon = 0.001$ and $\varepsilon = 0.01$

5.3. Experiment 3: Comparison of Versions

Experiment 3 confirms the findings of the bachelor thesis. For the newer stack of technology it is still true that version 1 *distribute_repeat* results in a higher quality but version 2 *distribute_mean* is faster.
Figures 5.7, 5.8, 5.9, and 5.10 show a runtime comparison of both versions for all data sets when the number of iterations is fixed to 5, 10, 20 or 100. The data sets are summarized by number of objects. Version 1 *distribute_repeat* performs slower than version 2 *distribute_mean*, and the higher the number of iterations, the greater the difference becomes visible. Version 2 *distribute_mean* is at least 50% faster and up to 99% times faster (data sets 10, 11, 12) than version 1 *distribute_repeat*.

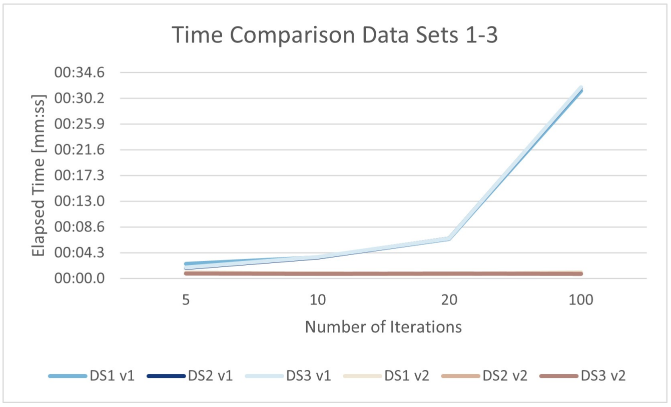


Figure 5.7.: Runtime Comparison of Version 1 *distribute_repeat* and Version 2 *distribute_mean* for Data Sets 1-3

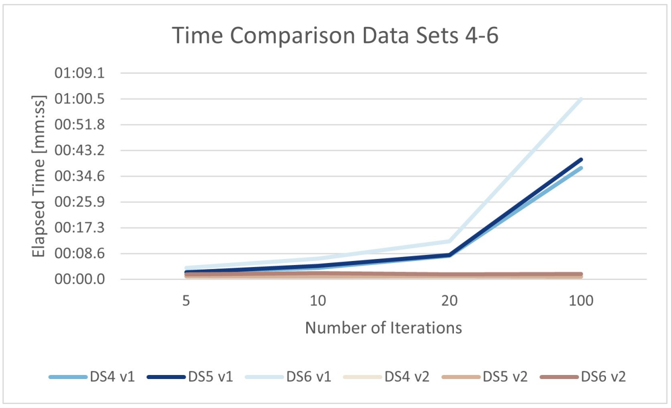


Figure 5.8.: Runtime Comparison of Version 1 *distribute_repeat* and Version 2 *distribute_mean* for Data Sets 4-6

Version 1 *distribute_repeat* achieves (slightly) better quality than version 2 *distribute_mean* in almost all cases (85.83%). A direct comparison is shown for data set 7 in

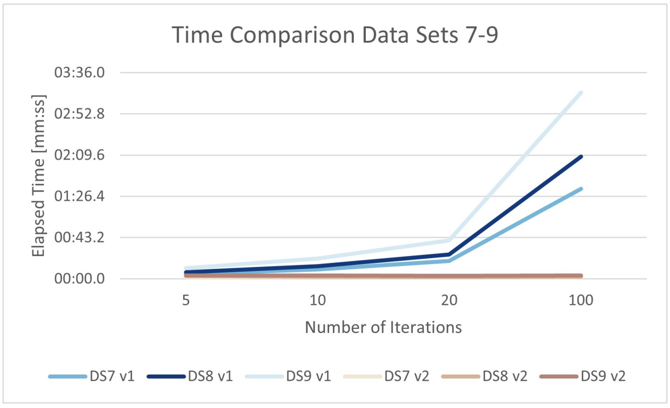


Figure 5.9.: Runtime Comparison of Version 1 *distribute_ repeat* and Version 2 *distribute_ mean* for Data Sets 7-9

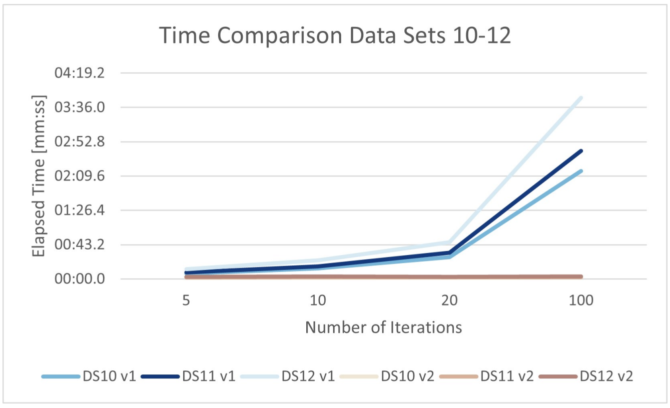


Figure 5.10.: Runtime Comparison of Version 1 *distribute_ repeat* and Version 2 *distribute_ mean* for Data Sets 10-12

Figures 5.11, 5.12, 5.13, 5.14, and 5.15. Referring to the internal metrics, the quality of version 2 *distribute_ mean* is better, referring to the external metrics, the quality of version 1 *distribute_ repeat* is higher. But as can be seen, the range on the y-axis is very small which means that the quality differences are not large. Furthermore, more iterations cannot increase the quality significantly.
Tables 5.2 and 5.3 show the relative improvement of the SSE after 5 and 100 iterations compared to the initial SSE for version 1 *distribute_ repeat* and version 2 *distribute_ mean*, respectively.
For both versions, there is not much of a difference after 5 or 100 iterations. Equally, the difference between v1 and v2 is marginal.
The complete data can be found in Appendix A.5 in Figure A.1. A check mark means that the statements of the bachelor thesis are correct; i.e. version 1 *distribute_ repeat* is more accurate and version 2 *distribute_ mean* is faster.

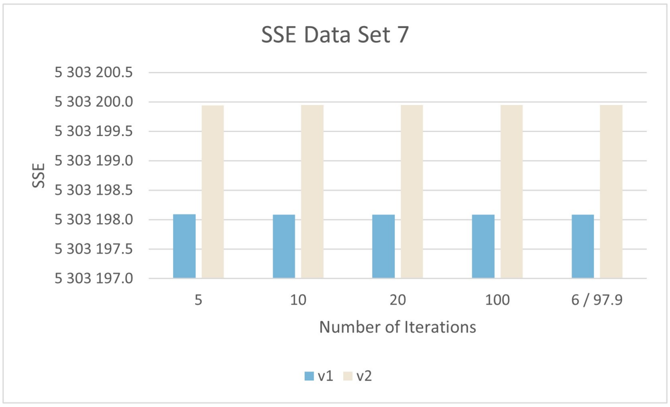


Figure 5.11.: Quality Comparison of Version 1 *distribute_repeat* and Version 2 *distribute_mean* Based on Sum of Squared Error

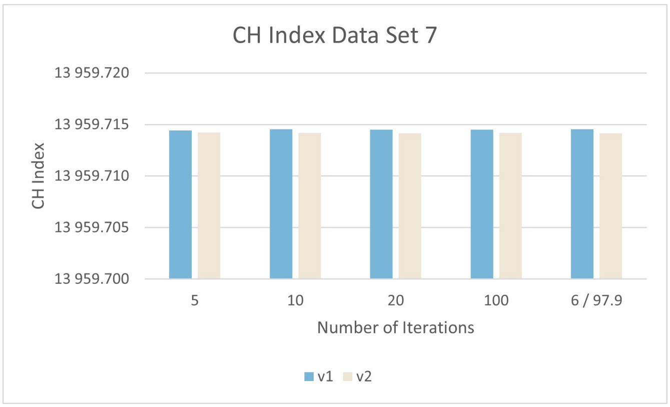


Figure 5.12.: Quality Comparison of Version 1 *distribute_repeat* and Version 2 *distribute_mean* Based on Calinski-Harabasz Index

Furthermore, we compared the number of iterations needed by version 1 *distribute_repeat* and version 2 *distribute_mean* until convergence with a convergence tolerance $\varepsilon = 0.01$. The number of iterations for version 2 *distribute_mean* is an average value based on all distributed data sets per replica. As can be seen in Figure 5.16, version 2 *distribute_mean* sometimes requires significantly more iterations than version 1 *distribute_repeat* (data sets 7-9, 11-12) with the exception of data set 5 and 6, where version 1 *distribute_repeat* needs more iterations. Nevertheless, version 2 *distribute_mean* is significantly faster than version 1 *distribute_repeat* as can be seen in Figure 5.17.
The clustering quality of the internal metrics are very similar for most cases. A few exceptions occur, where the quality of the clustering results of version 2 is clearly lower than of version 1 *distribute_repeat*. The external metrics, i.e. the correctness of the label assignments, is close to the same for both versions. Table A.2 and Table A.3 in Appendix

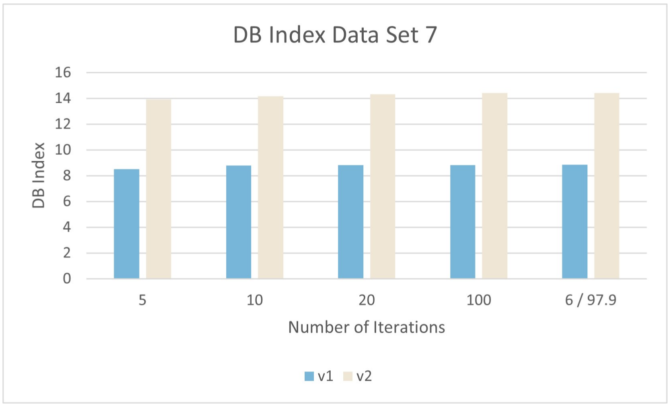


Figure 5.13.: Quality Comparison of Version 1 *distribute_repeat* and Version 2 *distribute_mean* Based on Davies Boulding Index

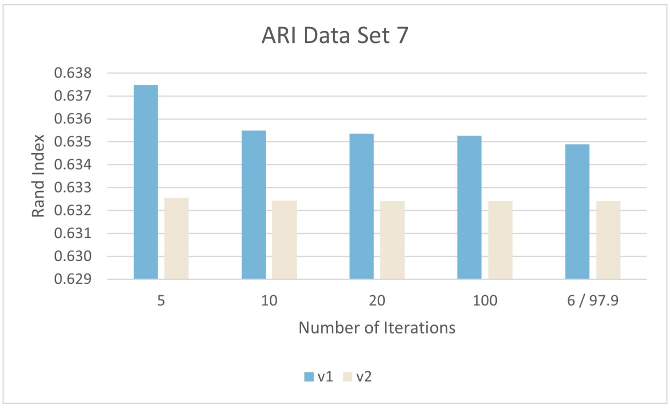


Figure 5.14.: Quality Comparison of Version 1 *distribute_repeat* and Version 2 *distribute_mean* Based on Adjusted Rand Index

A.5 display a detailed quality comparison.
The evaluation of this experiment is based on the data collected in the files *all_results_sep.csv* and *2022-04-24_log.log.*

Discussion: Version 1 *distribute_repeat* results in a higher clustering quality because, unlike version 2 *distribute_mean*, the new centroids are calculated based on the entire data set, whereas in version 2 *distribute_mean* the new centroids are calculated based on a small fraction of the entire data set. This can also be the reason why version 2 *distribute_mean* needs more iterations than version 1 *distribute_repeat.*
The large increase in the clustering quality right at the beginning and the subsequent slight improvement can be explained by the fact that the centroids shift by the greatest distance in the beginning and then only position themselves within the cluster. This is also true if the initial centroids are chosen poorly.

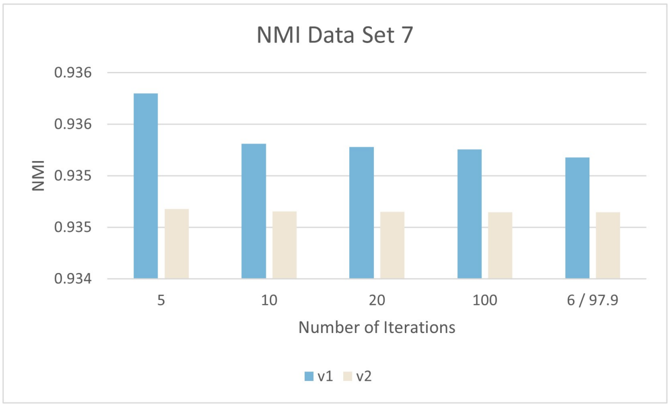


Figure 5.15.: Quality Comparison of Version 1 *distribute_repeat* and Version 2 *distribute_mean* Based on Normalized Mutual Information

Data Set	***Initial SSE***	***SSE after 5 Iterations***	***Rel. Imp.***	***SSE after 100 Iterations***	***Rel. Imp.***
1	11,306.46	579.43	95%	579.41	95%
2	316,460.86	29641.47	91%	29641.46	91%
3	562,967.09	34947.23	94%	34947.23	94%
4	2,503,102.55	306,057.01	88%	306,057.01	88%
5	19,761,435.11	2,889,620.30	85%	2,889,620.30	85%
6	82,070,990.02	10,027,015.22	88%	10,027,015.20	88%
7	46,302,607.44	5,303,198.09	89%	5,303,198.09	89%
8	42,652,781.73	5,282,051.37	88%	5,282,051.37	88%
9	21,079,112.58	3,428,496.28	84%	3,428,496.28	84%
10	2,558,262.59	224,798.01	91%	224,798.01	91%
11	45,724,417.04	5,074,685.35	89%	5,074,685.35	89%
12	40,792,315.20	4,975,061.66	88%	4,975,061.66	88%

Table 5.2.: SSE Development Version 1 *distribute_repeat*

The calculation of the k-Means with version 2 *distribute_mean* is faster than with version 1 *distribute_repeat*, because the communication between CPU and TPU between the iterations is omitted. In fact, the calculation speed is so fast that a required high number of iterations is hardly significant.

5.4. Experiment 4: Combination of Versions

Experiment 4 was extended by two additional parts due to the results, which is why the evaluation is divided into part 1, part 2 and part 3.

Data Set	***Initial SSE***	***SSE after 5 Iterations***	***Rel. Imp.***	***SSE after 100 Iterations***	***Rel. Imp.***
1	11,306.46	682.66	94%	682.70	94%
2	316,460.86	30,346.58	90%	30,346.58	90%
3	562,967.09	46,506.13	92%	46,506.13	92%
4	2,503,102.55	326,028.89	87%	325,939.80	87%
5	19,761,435.11	2,923,537.61	85%	2,923,541.66	85%
6	82,070,990.02	10,148,912.70	88%	10,148,912.70	88%
7	46,302,607.44	5,303,199.94	89%	5,303,199.95	89%
8	42,652,781.73	,5282,051.97	88%	5,282,051.99	88%
9	21,079,112.58	3,428,496.61	84%	3,428,496.64	84%
10	2,558,262.59	224,985.25	91%	224,988.94	91%
11	45,724,417.04	5,074,685.87	89%	5,074,685.88	89%
12	40,792,315.20	4,97,5062.04	88%	4,975,062.05	88%

Table 5.3.: SSE Development Version 2 *distribute_ mean*

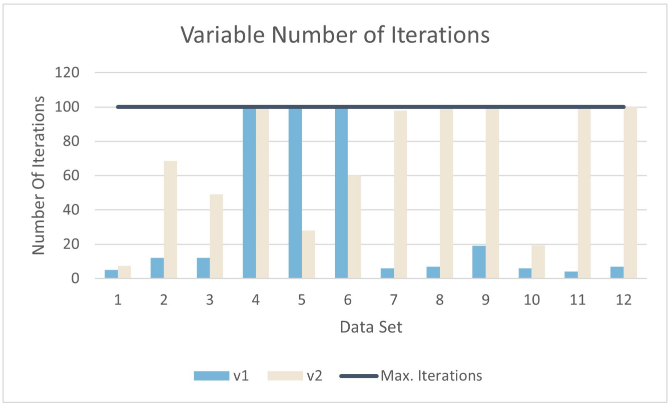


Figure 5.16.: Comparison of Number of Iterations Needed by Version 1 *distribute_ repeat* and Version 2 *distribute_ mean*

5.4.1. Part 1: Comparison of the Combinations and Versions

In Experiment 4 we have combined version 1 *distribute_ repeat* and version 2 *distribute_ mean* in both directions; v1 → v2 and v2 → 1. As before, the measures runtime, number of variable iterations and quality were evaluated. First runtime as well as the required number of iterations were evaluated. The results for data sets 5, 7 and 9 can be seen in Figures 5.18, 5.19 and 5.20, respectively. These three data sets seemed to be the most representative. The results for the other data sets can be found in Appendix A.6.1 for reference. The figures show the fixed number of iterations tested on the x-axis and the total runtime as well as the number of iterations needed by the second version in the combination on the y-axis. Furthermore, the runtime of the combinations was compared with those of version 1 *distribute_ repeat* and version 2 *distribute_ mean* from Experiment

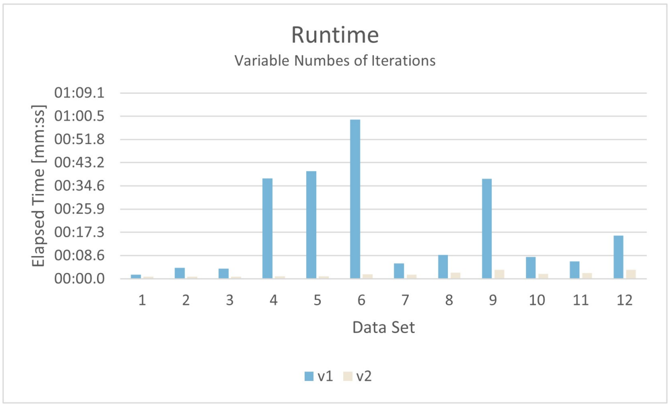


Figure 5.17.: Runtime Comparison of Version 1 *distribute_repeat* and Version 2 *distribute_mean* (Variable Numbers of Iterations)

3, since the goal is to speed up the calculation of the k-Means using the TPU. Combining the two versions is just one approach to test. The runtime of version 1 *distribute_repeat* and version 2 *distribute_mean* refers to the measurement with a variable number of iterations and is shown as a constant.
As can be seen in the figures, the results of the options are not very consistent. The performance of the option v1 → v2 deteriorates the more iterations of version 1 *distribute_repeat* are performed. This can be observed for all twelve data sets. The performance of option v2 → v1 varies a lot, as can be seen especially in figure 5.19 and 5.20.

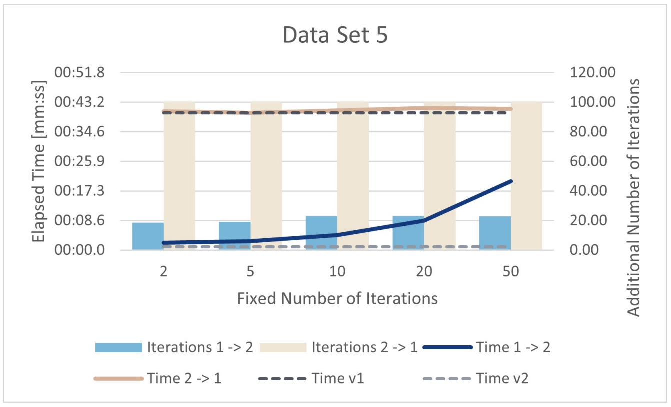


Figure 5.18.: Runtime Comparison of the Combinations for Data Set 5

It is also noticeable - and this applies to both combinations - that regardless of how many iterations the first version has performed, the second one requires about the same number of iterations. This is also, with a few exceptions, very well seen in the figures of the other data sets in Appendix A.6.1. The runtime performance of the combination v2 → v1, when not fluctuating, is comparable to that of version 1 *distribute_repeat*. The runtime

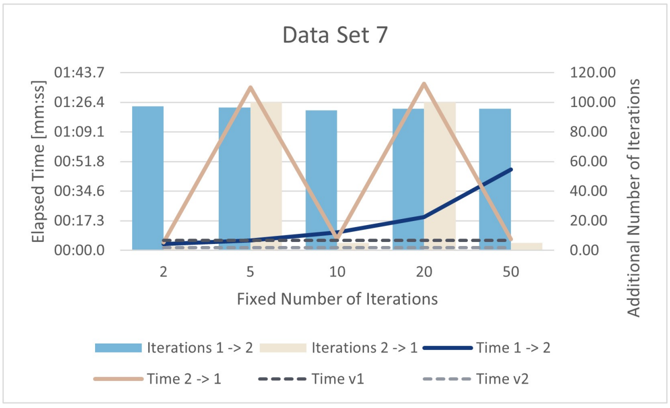


Figure 5.19.: Runtime Comparison of the Combinations for Data Set 7

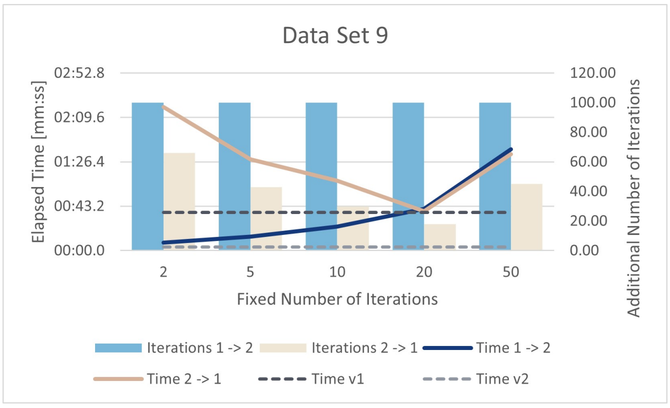


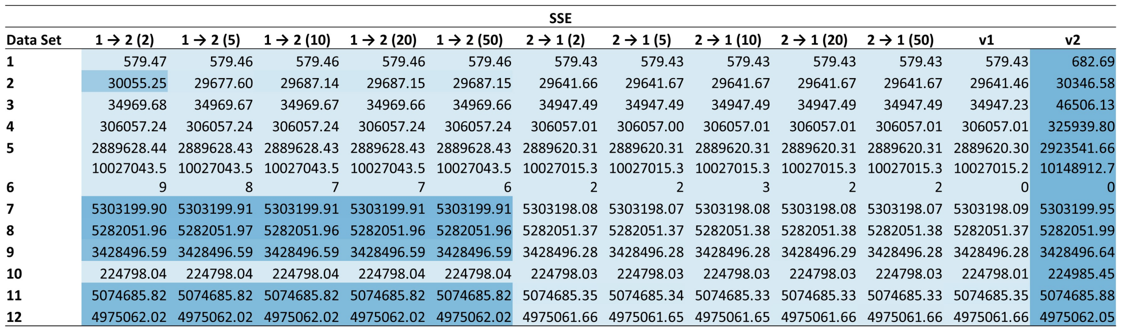
Figure 5.20.: Runtime Comparison of the Combinations for Data Set 9

performance of the combination v1 → v2 is similar to that of version 2 *distribute_mean*, although this is only true if the proportion of iterations of version 1 *distribute_repeat* in the combination is low.

The quality of version 1 *distribute_repeat* or also the combination v2 → v1 is better than that of version 2 *distribute_mean* or the combination v1 → v2. Figure 5.21 shows the quality comparison based on the SSE as a heatmap. The heatmaps for the other metrics are attached in Appendix A.6.2 (Figures A.11, A.12, A.13, A.14). The scale of the heatmap goes from lowest to highest value, with the highest quality marked in light blue and the lowest quality in medium blue. However, looking closely at the figures it can be seen that the values are usually very close to each other. For this reason, the focus in the next part is on runtime, neglecting quality.

To further narrow down the results, this experiment is extended as follows. However, only the combination v1 → v2 and version 2 *distribute_mean* are considered.

The evaluation of the first part of this experiment is based on the data collected in the files *all_results_comb.csv*, *all_results_sep.csv* and *2022-04-30_log.log*.



	SSE											
Data Set	1 → 2 (2)	1 → 2 (5)	1 → 2 (10)	1 → 2 (20)	1 → 2 (50)	2 → 1 (2)	2 → 1 (5)	2 → 1 (10)	2 → 1 (20)	2 → 1 (50)	v1	v2
1	579.47	579.46	579.46	579.46	579.46	579.43	579.43	579.43	579.43	579.43	579.43	682.69
2	30055.25	29677.60	29687.14	29687.15	29687.15	29641.66	29641.67	29641.67	29641.67	29641.67	29641.46	30346.58
3	34969.68	34969.67	34969.67	34969.66	34969.66	34947.49	34947.49	34947.49	34947.49	34947.49	34947.23	46506.13
4	306057.24	306057.24	306057.24	306057.24	306057.24	306057.01	306057.00	306057.01	306057.01	306057.01	306057.01	325939.80
5	2889628.44	2889628.43	2889628.43	2889628.43	2889628.43	2889620.31	2889620.31	2889620.31	2889620.31	2889620.31	2889620.30	2923541.66
6	10027043.59	10027043.58	10027043.57	10027043.57	10027043.56	10027015.32	10027015.32	10027015.33	10027015.32	10027015.32	10027015.20	10148912.70
7	5303199.90	5303199.91	5303199.91	5303199.91	5303199.91	5303198.08	5303198.07	5303198.08	5303198.08	5303198.07	5303198.09	5303199.95
8	5282051.96	5282051.97	5282051.96	5282051.96	5282051.96	5282051.37	5282051.37	5282051.38	5282051.38	5282051.38	5282051.37	5282051.99
9	3428496.59	3428496.59	3428496.59	3428496.59	3428496.59	3428496.28	3428496.28	3428496.28	3428496.29	3428496.28	3428496.28	3428496.64
10	224798.04	224798.04	224798.04	224798.04	224798.04	224798.03	224798.03	224798.03	224798.03	224798.03	224798.01	224985.45
11	5074685.82	5074685.82	5074685.82	5074685.82	5074685.82	5074685.35	5074685.34	5074685.33	5074685.33	5074685.33	5074685.35	5074685.88
12	4975062.02	4975062.02	4975062.02	4975062.02	4975062.02	4975061.66	4975061.65	4975061.65	4975061.66	4975061.66	4975061.66	4975062.05

Figure 5.21.: Quality Heatmap Based on SSE

Discussion The experiment showed that no additional benefits can be obtained by combining the versions. However, one should keep a few points in mind when considering the results. First, the runtime comparison is not entirely fair, since up to 150 iterations are possible with the combination, but only 100 iterations were possible in Experiment 3. Also, many parameters have an impact that cannot be tested in detail within the scope of the master's thesis. These include number of objects, number of dimensions, number of clusters as well as initial centroids. In the experiments so far, the first k data objects have always been used as initial centroids without making any statement about whether they are good or not. But it is known that initial centroids have a great impact on the performance of the k-Means algorithm. For these reasons, a second part is added to Experiment 4.

5.4.2. Part 2: Verification With Random Initial Centroids

In the second part of the experiment only the combination v1 $\rightarrow$ v2 as well as version 2 *distribute_mean* are considered. The initial centroids are now chosen randomly, with both options receiving the same random initial centroids so that the results are comparable. In addition to that, the final centroids are stored. Both options are run ten times. Version 2 *distribute_mean* runs through a maximum of 100 iterations. Combination v1 $\rightarrow$ v2 is also repeated ten times, with version 1 *distribute_repeat* making two, five, and ten iterations, respectively, and version 2 *distribute_mean* making a maximum of 98, 95, and 90 iterations, respectively, so that a total of 100 iterations is possible.

Even with different randomly chosen initial centroids, version 2 *distribute_mean* remains much faster than the combination v1 $\rightarrow$ v2. The quality of the combination is usually better, but when looking closely at the numbers, the difference often only becomes apparent in the decimal places. The document in Appendix A.6.3 shows a detailed list of the results. The results for each data set, each of the initial centroids (the filename reference is given) and each of the versions, i.e., combination v1 $\rightarrow$ v2 with 2 iterations fixed and up to 98 iterations variable, with 5 iterations fixed and up to 95

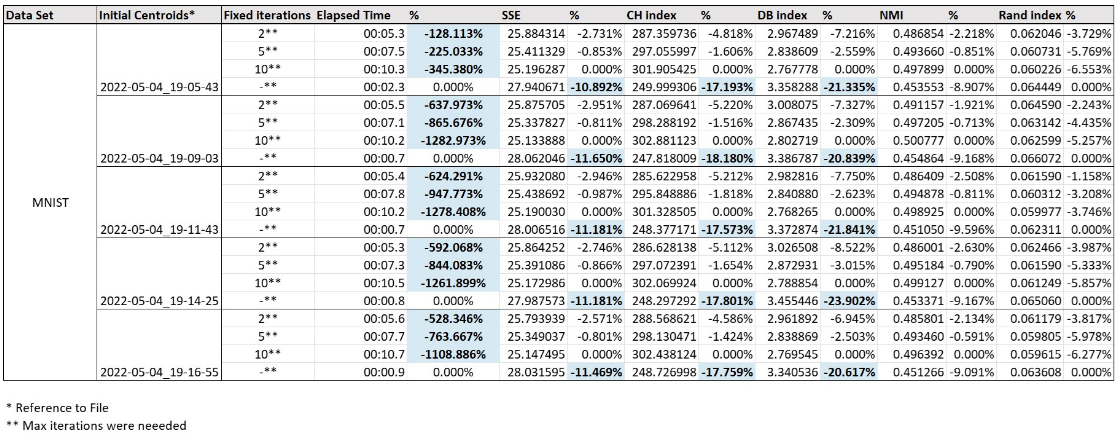
iterations variable, with 10 iterations fixed and up to 90 iterations variable, and version 2 *distribute_ mean* with up to 100 iterations variable (shown as "-") are listed here. For each of the initial centroids of each data set, the best value is marked in gray.
The evaluation of the second part of this experiment is based on the data collected in the files *all_ results_ comb.csv*, *all_ results_ sep.csv* and *2022-05-04_ log.log*.

Discussion Randomizing the initial centroids does not change the runtime behaviour of the combinations / versions. This is not surprising since both options, though randomly selected, received the same initial centroids and performed basically the same calculations.

5.4.3. Part 3: Verification With Real Data Sets

In the third part of the experiment, the second part of Experiment 4 is repeated with real data sets, to see if these findings only apply for "perfect" Gaussian distributed data or also for data in different shapes.

Part 3 of Experiment 4 confirms the findings of part 2 for real data sets. As in part 2, version 2 *distribute_ mean* is a lot faster than the combination v1 → v2 but the quality is better when the data set is clustered with the combination v1 → v2. For the runtime, this is true for each of the data sets and each of the initial centroids without exception. For the evaluation metrics, this is true with the exception of the ARI for the MNIST data set. The full report on the results is attached in Appendix A.6.4. The best result per initial centroid per data set is highlighted in gray.
The impact of the quality differences in the clustering results for the real data sets is more difficult to recognize at first glance. For this reason, we calculated the relative deviation to the benchmark result, i.e., the best result per initial centroids per data set. Figure 5.22 shows this for the MNIST data set. The calculations for all data sets can be found figure A.15 in Appendix A.6.4.



Data Set	Initial Centroids*	Fixed iterations	Elapsed Time	%	SSE	%	CH index	%	DB index	%	NMI	%	Rand index	%
MNIST		2**	00:05.3	**-128.113%**	25.884314	-2.731%	287.359736	-4.818%	2.967489	-7.216%	0.486854	-2.218%	0.062046	-3.729%
		5**	00:07.5	**-225.033%**	25.411329	-0.853%	297.055997	-1.606%	2.838609	-2.559%	0.493660	-0.851%	0.060731	-5.769%
		10**	00:10.3	**-345.380%**	25.196287	0.000%	301.905425	0.000%	2.767778	0.000%	0.497899	0.000%	0.060226	-6.553%
	2022-05-04_19-05-43	-**	00:02.3	0.000%	27.940671	**-10.892%**	249.999306	**-17.193%**	3.358288	**-21.335%**	0.453553	-8.907%	0.064449	0.000%
		2**	00:05.5	**-637.973%**	25.875705	-2.951%	287.069641	-5.220%	3.008075	-7.327%	0.491157	-1.921%	0.064590	-2.243%
		5**	00:07.1	**-865.676%**	25.337827	-0.811%	298.288192	-1.516%	2.867435	-2.309%	0.497205	-0.713%	0.063142	-4.435%
		10**	00:10.2	**-1282.973%**	25.133888	0.000%	302.881123	0.000%	2.802719	0.000%	0.500777	0.000%	0.062599	-5.257%
	2022-05-04_19-09-03	-**	00:00.7	0.000%	28.062046	**-11.650%**	247.818009	**-18.180%**	3.386787	**-20.839%**	0.454864	-9.168%	0.066072	0.000%
		2**	00:05.4	**-624.291%**	25.932080	-2.946%	285.622958	-5.212%	2.982816	-7.750%	0.486409	-2.508%	0.061590	-1.158%
		5**	00:07.8	**-947.773%**	25.438692	-0.987%	295.848886	-1.818%	2.840880	-2.623%	0.494878	-0.811%	0.060312	-3.208%
		10**	00:10.2	**-1278.408%**	25.190030	0.000%	301.328505	0.000%	2.768265	0.000%	0.498925	0.000%	0.059977	-3.746%
	2022-05-04_19-11-43	-**	00:00.7	0.000%	28.006516	**-11.181%**	248.377171	**-17.573%**	3.372874	**-21.841%**	0.451050	-9.596%	0.062311	0.000%
		2**	00:05.3	**-592.068%**	25.864252	-2.746%	286.628138	-5.112%	3.026508	-8.522%	0.486001	-2.630%	0.062466	-3.987%
		5**	00:07.3	**-844.083%**	25.391086	-0.866%	297.072391	-1.654%	2.872931	-3.015%	0.495184	-0.790%	0.061590	-5.333%
		10**	00:10.5	**-1261.899%**	25.172986	0.000%	302.069924	0.000%	2.788854	0.000%	0.499127	0.000%	0.061249	-5.857%
	2022-05-04_19-14-25	-**	00:00.8	0.000%	27.987573	**-11.181%**	248.297292	**-17.801%**	3.455446	**-23.902%**	0.453371	-9.167%	0.065060	0.000%
		2**	00:05.6	**-528.346%**	25.793939	-2.571%	288.568621	-4.586%	2.961892	-6.945%	0.485801	-2.134%	0.061179	-3.817%
		5**	00:07.7	**-763.667%**	25.349037	-0.801%	298.130471	-1.424%	2.838869	-2.503%	0.493460	-0.591%	0.059805	-5.978%
		10**	00:10.7	**-1108.886%**	25.147495	0.000%	302.438124	0.000%	2.769545	0.000%	0.496392	0.000%	0.059615	-6.277%
	2022-05-04_19-16-55	-**	00:00.9	0.000%	28.031595	**-11.469%**	248.726998	**-17.759%**	3.340536	**-20.617%**	0.451266	-9.091%	0.063608	0.000%

* Reference to File
** Max iterations were neeeded

Figure 5.22.: Detailed Benchmark Comparison for the MNIST Data Set

The deviations for the runtime as well as the evaluation metrics SSE, CH, DB, NMI and ARI are shown. The runtime is highlighted if it is more than 100% slower, half as fast as the benchmark. The values of the metrics are highlighted if they deviate more than 10% from the benchmark value. The runtime for the combination deviates by more than 100% in all cases. In the best case this is -128% and in the worst case it is -1,282.97%. In contrast, the metrics of version 2 *distribute_mean* deviate from the benchmark of the combination v1 $\rightarrow$ v2 by more than 10% only for the internal metrics. The largest deviation is -23.9%.
The evaluation of the second part of this experiment is based on the data collected in the files *all_results_comb.csv*, *all_results_sep.csv* and *2022-05-04_log.log*.

Discussion It can be stated that the combination v1 $\rightarrow$ v2 performs better in terms of quality, but significantly worse in terms of runtime. The goal of the master thesis is to enhance the k-Means with the use of the TPU with respect to the runtime without major losses in quality. As Experiments 3 and 4 show, version 2 *distribute_mean distribute_mean* gives the best results on the TPU. Therefore, this is used for the comparison with CPU and GPU in Experiment 5.

5.5. Experiment 5: Performance Comparison to CPU and GPU

We analyzed and compared the performance of TPU, GPU and CPU in terms runtime and quality. On the TPU as well as on the GPU, there are eight replicas in sync, i.e. the training is distributed across eight cores. The CPU has one replica in sync, which means that the training is not distributed. The runtime for both, the pure calculation and in total, i.e. including reading, batching and distributing the data set as well as initializing the centroids, is measured. The results for runtime performance are depicted for the data sets 1, 9 and *Bank Marketing* in the Figures 5.23, 5.24, 5.25, respectively. The results for all data sets can be found in Appendix A.7, Figures A.16 and A.17. The TPU is shown in light blue in all figures, the GPU in gray and the CPU in beige.
The shortest runtime for the calculation of data set 1 (10,240 objects, 8 dimensions, 8 clusters; 1.46 MB) is achieved on the CPU. The total runtime is shortest on the GPU by 0.4 seconds compared to the CPU. For data set 9 (512,000 objects, 512 dimensions, 32 clusters; 4.4 GB) this looks different. Here, the TPU delivers by far the best calculation runtime (302% faster than the GPU and 2,215% faster than the CPU).
Looking at the total runtime, however, the GPU performs better. The difference is 3% compared to the TPU and 302% compared to the CPU.
The runtime performance for the data set *Bank Marketing* (41,188 data objects, 48 dimensions, 2 clusters; 4.57 MB) is best with the TPU.
Comparing the quality, the best results are often achieved on the TPU. Generally, the differences in quality between the three processors are small. The results of this comparison can be found in Appendix A.7 Figure A.17.
The evaluation of Experiment 5 is based on the data collected in the files *all_results_sep.csv*

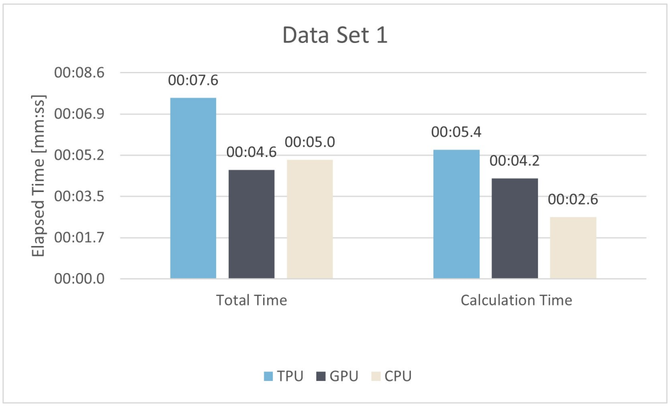


Figure 5.23.: Runtime Comparison of TPU, GPU, and CPU for Data Set 1

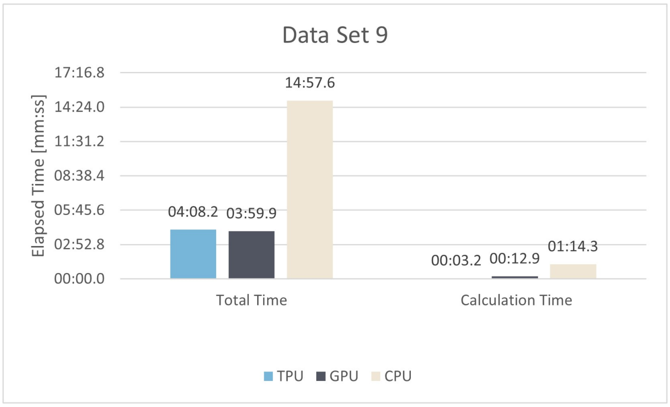


Figure 5.24.: Runtime Comparison of TPU, GPU, and CPU for Data Set 9

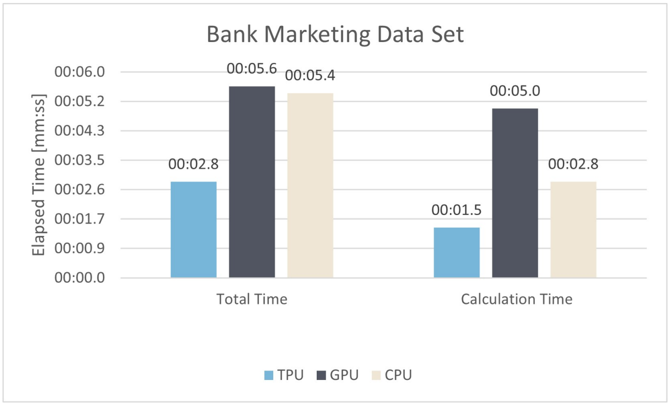


Figure 5.25.: Runtime Comparison of TPU, GPU, and CPU for *Bank Marketing* Data Set

and *2022-05-05_log.log*.

Discussion: The runtime required for clustering with k-Means is on the TPU a lot faster than on GPU or CPU. Nonetheless, the total runtime takes longer compared to the other processors. It can be deduced from this that a lot of time is needed for reading in the data and, in the case of the TPU, also for transferring the data from the CPU to the TPU. Moreover, it should be taken into consideration that for the GPU Google colab was used which is part of the Google infrastructure and hosted in the Cloud. This could be one of the reasons why the overall runtime performance on the GPU was often better.
k-Means converges to a local optimum instead of a global optimum. For this reason, in practice, the algorithm is often repeated multiple times with different initial clusters. In the experiment however, k-Means was only executed once per data set. But running k-Means multiple times for one data set, reduces the proportion of the runtime required to load the data set in relation to the calculation time. This makes the TPU competitive overall.
Additionally, there are some potential improvements that can shorten execution time spent with loading the data set. These include, for example, using the TFRecord data format instead of csv files. This could improve the overall runtime on all three processors, since the data format is specific to Tensorflow. Another measure that could improve runtime is to split large data sets into smaller ones and read them in chunks. Using version 2 *distribute_mean*, this is possible since the data set does not need to be synchronized between assignment steps. It only needs to be ensured, that there is a continuous income stream of data.
Last but not least, the final experiment proved that the clustering quality of the k-Means algorithm performed on the TPU is equally good compared to GPU and CPU.

6. Conclusion

6.1. Summary

The massive increase in data stored in digital form leads to two necessities: On the one hand, algorithms are needed that can extract meaningful information from these volumes of data, and on the other hand, the computational ability to process these volumes of data. The former is the subject of data mining, the goal of which is to interpret large data sets. The latter challenge has also been recognized by Google, which developed the TPU to stem the user demands in their data centers. In Google's data centers, the TPU is mainly used for executing DNNs. The master thesis explored to extend this research to other Data Mining algorithms. The k-Means algorithm is one of the most popular algorithms in data mining. It is scalable and well suited for analyzing large data sets. Furthermore, it can be formulated as matrix calculation which makes it possible to exploit the potentials of the TPU. The aim of the master thesis is to enhance the k-Means algorithm with the use of the TPU in terms of runtime while preserving the quality of the clustering results. We implemented two versions, version 1 *distribute_repeat* and version 2 *distribute_mean*, for which the concept is based on a bachelor thesis written at the University of Vienna. Since both versions have complementary advantages we furthermore, combined these two versions to the two options v1 $\rightarrow$ v2 and v2 $\rightarrow$ v1 and experimented with different fixed number of iterations, the first version is being executed.
The comparison of all four options showed that version 2 *distribute_mean* is the fastest with comparable clustering results to the other three options. The slower runtime performance of version 1 *distribute_repeat* can be explained by the communication overhead between TPU and CPU in each iteration and that the update step is entirely performed on the CPU.
This obviously takes so much time to do that version 2 *distribute_mean* is faster even if it takes significantly more iterations to converge. This observation can also be applied to the combinations. The larger the share of version 1 *distribute_repeat* in the total number of iterations is, the slower the runtime performance. Combination v1 $\rightarrow$ v2 came close to the runtime performance of version 2 *distribute_mean*, but only if version 1 *distribute_repeat* has a share of 2 or maximum 5 iterations. The clustering results of version 2 *distribute_mean* are of lower quality than for other options but the deviation is small with a maximum of 24%. The runtime was up to 1,283% faster in the experiments compared to the other options.

Furthermore, it can be stated that the clustering quality based on Concept 2 with matrix multiplication is as good as that of the standard k-Means.

Comparing the clustering results of version 2 *distribute_ mean* executed on the TPU to the results on the CPU or GPU, the quality is at least equally.
In the experiments, it was also found that the runtime for calculating the k-Means on the TPU is significantly faster than on the CPU or GPU. However, reading in the data and loading it onto the TPU takes more time. However, if one considers that k-Means often has to be repeated several times with different initial centroids in practice, this is less significant. In the experiment, k-Means was only executed once, so this effect could not be recorded. In addition, optimization approaches for reading in the data were proposed. The results described above apply to both the TPU optimized synthetic data sets and real data sets.
In the experiments, we additionally investigated possible batch sizes. For k-Means, much larger batch sizes can be chosen than recommended by Google, although it can be assumed that this recommendation refers to DNNs. The recommended optimal batch size by Google is 128 per replica, but we were able to successfully test the k-Means with a batch size of 128,000 per replica, which corresponds to a global batch size of 1,024,000 on a single TPU v2 or v3 device. It is reasonable to assume that larger batch sizes are possible for k-Means because the computations performed are far less complex than those of DNNs. We were also able to show with our experiments that the implementation can be run with any initial centroids and thus the optimizations presented at the beginning of the paper can be applied. And last but not least, the same code is executable on TPU, GPU and CPU and thus can be used flexibly.

In summary, k-Means can be enhanced with the TPU in terms of runtime without major deviations in the quality of the clustering results. However, it has been found that combining the two versions is not beneficial, since the increase in quality does not justify the longer runtime. The results that can be achieved with version 2 *distribute_ mean* are, however, sufficient.
Using a TPU is much more expensive than using a GPU or CPU. For this reason, costs and performance need to be put in relation. The use of the TPU is beneficial for large and very large data sets in terms of number of objects, number of dimensions and number of clusters and/or if k-Means is repeated several times for a data set.

6.2. Future Work

In future work the influence of the data set properties number of objects n and number of dimensions d as well as the number of clusters k can be investigated in more detail so that better statements can be made about the performance of the k-Means on the TPU and to be able to better assess under which conditions the use of the TPU is beneficial. This also includes making a meaningful comparison with CPU and GPU, i.e. with more powerful processors. On this basis, a good cost-performance comparison could be made and better statements made as to when the use of the TPU is really worthwhile.
Future work could also include the investigation of common k-Means extensions on the TPU. It is conceivable that this could lead to further benefits from the use of the TPU.

Furthermore, it could be interesting to extend the research and explore the use of the TPU for other clustering algorithms as well, or for other methods in the area of Unsupervised Learning. The only prerequisite for this is that the algorithms can at least be expressed as matrix operations, but ideally as matrix multiplications.

Bibliography

[1] K. R. Chowdhary. Data Mining. In K.R. Chowdhary, editor, *Fundamentals of Artificial Intelligence*, pages 507–555. Springer India, New Delhi, 2020.

[2] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. Motivation for and Evaluation of the First Tensor Processing Unit. *IEEE Micro*, 38(3):10–19, May 2018.

[3] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, June 2017. Association for Computing Machinery. Comment: 17 pages, 11 figures, 8 tables. To appear at the 44th International Symposium on Computer Architecture (ISCA), Toronto, Canada, June 24-28, 2017.

[4] Norm Jouppi. Google supercharges machine learning tasks with TPU custom chip. `https://cloud.google.com/blog/products/ai-machine-learning/google-supercharges-machine-learning-tasks-with-custom-chip/`, May 2016. Accessed 17 Nov 2021.

[5] Release Notes | Cloud TPU. `https://cloud.google.com/tpu/docs/release-notes`. Accessed 23 Oct 2021.

[6] System Architecture | Cloud TPU. `https://cloud.google.com/tpu/docs/system-architecture-tpu-vm`. Accessed 19 Nov 2021.

[7] Cloud Tensor Processing Units (TPUs) | Google Cloud. https://cloud.google.com/tpu/docs/tpus. Accessed 23 Oct 2021.

[8] Chong Ho Yu. Exploratory data analysis in the context of data mining and resampling. *International Journal of Psychological Research*, 3, June 2010.

[9] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, Boston, third edition edition, 2012.

[10] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data.* Prentice-Hall, Inc., USA, 1988.

[11] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, January 2008.

[12] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *In 5-Th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[13] Edward Forgy. Cluster analysis of multivariate data: Efficiency vs. interpretability of classification. *Biometrics. Journal of the International Biometric Society*, 21(3):768–769, 1965.

[14] S. P. Lloyd. Least squares quantization in PCM. *IEEE Trans. Inf. Theory*, 1982.

[15] John A. Hartigan. *Clustering Algorithms.* John Wiley & Sons, Inc., USA, ninety-ninth edition, 1975.

[16] Dan Pelleg, Andrew W Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *Icml*, volume 1, pages 727–734, 2000.

[17] K. P. Sinaga and M. Yang. Unsupervised K-Means Clustering Algorithm. *IEEE Access*, 8:80716–80727, 2020.

[18] Miin-Shen Yang, Chien-Yo Lai, and Chih-Ying Lin. A robust EM clustering algorithm for Gaussian mixture models. *Pattern Recognition*, 45(11):3950–3961, November 2012.

[19] Greg Hamerly and Charles Elkan. Learning the k in k-means. *Advances in neural information processing systems*, 16:281–288, 2003.

[20] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical Report 2006-13, Stanford InfoLab, June 2006.

[21] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable K-Means++. *arXiv:1203.6402 [cs]*, March 2012.

[22] Michael Steinbach, George Karypis, and Vipin Kumar. A Comparison of Document Clustering Techniques. Report, University of Minnesota, May 2000.

[23] Aristidis Likas, Nikos Vlassis, and Jakob J. Verbeek. The global k-means clustering algorithm. *Biometrics*, 36(2):451–461, February 2003.

[24] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data Via the EM Algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[25] J. C. Dunn. A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters. *Journal of Cybernetics*, 3(3):32–57, January 1973.

[26] James C Bezdek, Robert Ehrlich, and William Full. FCM: The fuzzy c-means clustering algorithm. *Computers & geosciences*, 10(2-3):191–203, 1984.

[27] Joyce Jiyoung Whang, Inderjit S. Dhillon, and David F. Gleich. Non-exhaustive, Overlapping k-means. In *Proceedings of the 2015 SIAM International Conference on Data Mining (SDM)*, Proceedings, pages 936–944. Society for Industrial and Applied Mathematics, June 2015. doi:10.1137/1.9781611974010.105.

[28] Sanjay Chawla and Aristides Gionis. K-means–: A unified approach to clustering and outlier detection. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 189–197. SIAM, 2013.

[29] Arindam Banerjee, Srujana Merugu, Inderjit S. Dhillon, and Joydeep Ghosh. Clustering with bregman divergences. *Journal of Machine Learning Research*, 6(58):1705–1749, 2005.

[30] S. Na, L. Xumin, and G. Yong. Research on k-means Clustering Algorithm: An Improved k-means Clustering Algorithm. In *2010 Third International Symposium on Intelligent Information Technology and Security Informatics*, pages 63–67, April 2010.

[31] Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 147–153, 2003.

[32] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 579–587, Lille, France, July 2015. PMLR.

[33] Dan Pelleg and Andrew Moore. Accelerating exact k-means algorithms with geometric reasoning. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 277–281, 1999.

[34] Christian Böhm and Claudia Plant. Mining Massive Vector Data on Single Instruction Multiple Data Microarchitectures. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, pages 597–606, November 2015.

[35] Ali Hadian and Saeed Shahrivari. High performance parallel k-means clustering for disk-resident datasets on multi-core CPUs. *The Journal of Supercomputing*, 69(2):845–863, August 2014.

[36] Wojciech Kwedlo and Michał Łubowicz. Accelerated K-Means Algorithms for Low-Dimensional Data on Parallel Shared-Memory Systems. *IEEE Access*, 9:74286–74301, 2021.

[37] Christian Böhm, Martin Perdacher, and Claudia Plant. Multi-core K-means. In *Proceedings of the 2017 SIAM International Conference on Data Mining (SDM)*, Proceedings, pages 273–281. Society for Industrial and Applied Mathematics, June 2017.

[38] David Pettinger and Giuseppe Di Fatta. Scalability of efficient parallel k-means. In *2009 5th IEEE International Conference on E-Science Workshops*, pages 96–101, 2009.

[39] Maria Florina Balcan, Steven Ehrlich, and Yingyu Liang. Distributed k-Means and k-Median Clustering on General Topologies. 2020.

[40] Inderjit S. Dhillon and Dharmendra S. Modha. A Data-Clustering Algorithm on Distributed Memory Multiprocessors. In Mohammed J. Zaki and Ching-Tien Ho, editors, *Large-Scale Parallel Data Mining*, Lecture Notes in Computer Science, pages 245–260, Berlin, Heidelberg, 2000. Springer.

[41] Mario Zechner and Michael Granitzer. Accelerating K-Means on the Graphics Processor via CUDA. In *2009 First International Conference on Intensive Applications and Services*, pages 7–15, April 2009.

[42] Cloud TPU beginner's guide | Google Cloud. `https://cloud.google.com/tpu/docs/beginners-guide`. Accessed 23 Oct 2021.

[43] System Architecture | Cloud TPU | Google Cloud. `https://cloud.google.com/tpu/docs/system-architecture-tpu-vm`. Accessed 23 Oct 2021.

[44] Introduction to Cloud TPU. `https://cloud.google.com/tpu/docs/intro-to-tpu`. Accessed 19 Nov 2021.

[45] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Communications of the ACM*, 63(7):67–78, June 2020.

[46] Bfloat16 and TensorFlow models | Cloud TPU. `https://cloud.google.com/tpu/docs/bfloat16`. Accessed 03 Nov 2021.

[47] BFloat16: The secret to high performance on Cloud TPUs. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus/. Accessed 30 Mar 2022.

[48] Kaz Sato, Cliff Young, and David Patterson. An in-depth look at Google's first Tensor Processing Unit (TPU). `https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu/`. Accessed 20 Nov 2021.

[49] Tao Wang and Aarush Selvan. Google wins MLPerf benchmarks with TPU v4. `https://cloud.google.com/blog/products/ai-machine-learning/google-wins-mlperf-benchmarks-with-tpu-v4/`.

[50] Sundar Pichai. Google I/O 2021: Hilfreicher sein in Momenten, in denen es darauf ankommt. `https://blog.google/intl/de-de/unternehmen/inside-google/google-io-2021-hilfreicher-sein-momenten-denen-es-darauf-ankommt/`, May 2021.

[51] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Communications of The Acm*, 61(9):50–59, August 2018.

[52] MLPerf™ Training v0.6 results. `https://mlcommons.org/en/training-normal-06/`. Accessed 23 Nov 2021.

[53] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image classification at supercomputer scale. *arXiv preprint arXiv:1811.06992*, 2018.

[54] Yuri Gordienko, Yuriy Kochura, Vlad Taran, Nikita Gordienko, Alexandr Rokovyi, Oleg Alienin, and Sergii Stirenko. Scaling Analysis of Specialized Tensor Processing Architectures for Deep Learning Models. In Witold Pedrycz and Shyi-Ming Chen, editors, *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence, pages 65–99. Springer International Publishing, Cham, 2020.

[55] Yuriy Kochura, Yuri Gordienko, Vlad Taran, Nikita Gordienko, Alexandr Rokovyi, Oleg Alienin, and Sergii Stirenko. Batch Size Influence on Performance of Graphic and Tensor Processing Units During Training and Inference Phases. In Zhengbing Hu, Sergey Petoukhov, Ivan Dychka, and Matthew He, editors, *Advances in Computer Science for Engineering and Education II*, Advances in Intelligent Systems and Computing, pages 658–668, Cham, 2020. Springer International Publishing.

[56] MLPerf™ Training v1.0 Results. `https://mlcommons.org/en/news/mlperf-training-v10/`. Accessed 23 Nov 2021.

[57] V1.0 Results. `https://mlcommons.org/en/training-normal-10/`. Accessed 23 Nov 2021.

[58] Cloud Solutions | Google Cloud. `https://cloud.google.com/solutions`. Accessed 23 Nov 2021.

[59] Products and Services | Google Cloud. `https://cloud.google.com/products`. Accessed 23 Nov 2021.

[60] Mangpo Phothilimthana, Mike Burrows, Sam Kaufman, and Yanqi Zhou. A Learned Performance Model for the Tensor Processing Unit. Technical report, 2020.

[61] Towhidul Islam, Nurul Absar, Abzetdin Z. Adamov, and Mayeen Uddin Khandaker. A machine learning driven android based mobile application for flower identification. In Mufti Mahmud, M. Shamim Kaiser, Nikola Kasabov, Khan Iftekharuddin, and Ning Zhong, editors, *Applied Intelligence and Informatics*, pages 163–175, Cham, 2021. Springer International Publishing.

[62] Marwan Hassani and Thomas Seidl. Using internal evaluation measures to validate the quality of diverse stream clustering algorithms. *Vietnam Journal of Computer Science*, 4(3):171–183, August 2017.

[63] Thomas Spendlhofer. *Evaluating the Usage of Tensor Processing Units (TPUs) for Unsupervised Learning on the Example of the k-Means Algorithm.* Bachelor thesis, University of Vienna, Vienna, 2019. Accessible in the archive of the University of Vienna.

[64] Cloud TPU performance guide. https://cloud.google.com/tpu/docs/performance-guide. Accessed 13 Feb 2022.

[65] Tadeusz Caliński and Joachim Harabasz. A dendrite method for cluster analysis. *Communications in Statistics-theory and Methods*, 3:1–27, 1974.

[66] David L. Davies and Donald W. Bouldin. A cluster separation measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227, 1979.

[67] Byron E. Dom. An information-theoretic external cluster-validity measure. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, UAI'02, pages 137–145, San Francisco, CA, USA, August 2002. Morgan Kaufmann Publishers Inc.

[68] William M. Rand. Objective Criteria for the Evaluation of Clustering Methods. *Journal of the American Statistical Association*, 66(336):846–850, December 1971. doi: 10.1080/01621459.1971.10482356.

[69] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of Classification*, 2(1):193–218, December 1985.

[70] Use TPUs | TensorFlow Core. https://www.tensorflow.org/guide/tpu. Accessed 27 Mar 2022.

[71] Distributed training with TensorFlow | TensorFlow Core. https://www.tensorflow.org/guide/distributed_training. Accessed 18 Dec 2021.

[72] Tf.data.Dataset | TensorFlow Core v2.8.0. https://www.tensorflow.org/api_docs/python/tf/data/Dataset. Accessed 27 Mar 2022.

[73] Tf.distribute.DistributedDataset | TensorFlow Core v2.8.0. https://www.tensorflow.org/api_docs/python/tf/distribute/DistributedDataset. Accessed 27 Mar 2022.

[74] Custom training with tf.distribute.Strategy | TensorFlow Core. https://www.tensorflow.org/tutorials/distribute/custom_training. Accessed 28 Mar 2022.

[75] Pricing | Cloud TPU. https://cloud.google.com/tpu/pricing. Accessed 28 Mar 2022.

[76] Pricing | Cloud Storage. https://cloud.google.com/storage/pricing. Accessed 28 Mar 2022.

[77] Sérgio Moro, Paulo Cortez, and Paulo Rita. A data-driven approach to predict the success of bank telemarketing. *Decision Support Systems*, 62:22–31, 2014.

[78] UCI Machine Learning Repository: Bank Marketing Data Set. https://archive.ics.uci.edu/ml/datasets/Bank+Marketing. Accessed 22 Apr 2022.

[79] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection. *arXiv:1802.09089 [cs]*, May 2018.

[80] Kitsune Network Attack Dataset. https://www.kaggle.com/ymirsky/network-attack-dataset-kitsune. Accessed 23 Apr 2022.

[81] MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges. http://yann.lecun.com/exdb/mnist/. Accessed 24 Apr 2022.

Acronyms

AI Artificial Intelligence. 13, 25

ALU Algorithmic Logic Unit. 14, 17

API Application Programming Interface. 13, 19, 36, 37

ARI Adjusted Rand Index. 34, 55, 56, 98

ASIC Application-Specific Integrated Circuit. 2, 13

bfloat16 brain floating point. 17

CapsNet Capsule Neural Network. 24

CH Calinski-Harabasz Index. 33, 56, 97

CNN Convolutional Neural Network. 22, 24

CPU Central Processing Unit. 1–3, 12–14, 16, 20, 22, 23, 29, 30, 35–37, 39, 42, 50, 56–60, 104

CUDA Compute Unified Device Architecture. 13

DB Davies Boulding Index. 34, 56, 97

DNN Deep Neural Network. 1, 2, 22–24, 43, 59, 60

DRAM Dynamic Random Access Memory. 16

EDA Exploratory Data Analysis. 3, 5

EM Expectation-Maximization. 12, 26

FCM Fuzzy C-Means. 12

FIFO First In – First Out. 17

FLOPS Floating Point Operations Per Second. 16, 19

GCP Google Cloud Platform. 2, 13, 19, 20

GiB	Gibibyte. 15, 16, 38
GM	Geometric Mean. 24
GPU	Graphic Processing Unit. 2, 3, 12–14, 22–24, 29, 35–37, 39, 42, 56–58, 60, 104
gRPC	Remote Procedure Call, developed by Google. 21
HBM	High-Bandwidth Memory. 15, 16
IoT	Internet of Things. 40, 41, 43
KDD	Knowledge Discovery in Databases. 1, 5
LSTM	Long Short-Term Memory. 22, 23
MiB	Mebibyte. 15
MIMD	Multiple Instructions Multiple Data. 12
MLP	Multilayer Perceptron. 22–24
MPI	Message Passing. 13
MXU	Matrix Multiply Unit. 15–18, 32
NEO-K-Means	Non-Exhaustive, Overlapping K-Means. 12
NMI	Normalized Mutual Information. 34, 56, 98
NMT	Neural Machine Translation. 23
OpenMP	Open Multi-Processing. 12, 13
PCIe	Peripheral Component Interconnect Express. 16
RCNN	Region-Based Convolutional Neural Network. 23
ResNet	Deep Residual Network. 23, 24
RI	Rand Index. 34
RNN	Recurrent Neural Network. 22, 24
SIMD	Single Instruction Multiple Data. 12
SPMD	Single Program Multiple Data. 13

SSD Single Shot Detector. 23

SSE Sum of Squared Error. 6, 7, 33, 47, 50, 51, 54, 56, 97

TPU Tensor Processing Unit. 2, 3, 13–25, 29–32, 35–39, 41–43, 45, 50, 52, 56–61, 104

U-K-Means Unsupervised K-Means. 11

VLIW Very Long Instruction Word. 17

VM Virtual Machine. 20, 21, 38

VPU Vector Processing Unit. 15–17

XLA Accelerated Linear Algebra. 21, 31, 37

A. Appendix

A.1. Custom Training Loops

A.1.1. Custom Training Loop v1 *distribute_repeat*

```
stop_criterion = False
assert EPOCHS > 0, "EPOCHS must be at least 1"
for ep in range(EPOCHS):
    per_batch_centroids_sum = 0
    per_batch_count = 0
    count = 0
    # iterate over the ‘tf.distribute.DistributedDataset‘
    for x in dataset:
        per_replica_centroids_sum, per_replica_count, per_replica_output
    = strategy.run(clf.assign_and_prep_fn, args=(x, centroids))

        per_batch_centroids_sum += strategy.reduce(tf.distribute.ReduceOp
    .SUM, per_replica_centroids_sum, axis=None)
        per_batch_count += strategy.reduce(tf.distribute.ReduceOp.SUM,
    per_replica_count, axis=None)

        # concatenate assignments only in last iteration
        if ep + 1 == EPOCHS or stop_criterion:
            if strategy.num_replicas_in_sync > 1:
                per_replica_values = tf.convert_to_tensor(
    per_replica_output.values)
                a, b, c = per_replica_values.shape
                per_batch_output = tf.reshape(per_replica_values, [a * b,
     c])
            else:
                per_batch_output = tf.convert_to_tensor(
    per_replica_output)

            if count == 0:
                output = per_batch_output
            else:
                output = tf.concat([output, per_batch_output], axis=0)

        count += 1

    prev_centroids = centroids
    centroids = per_batch_centroids_sum / per_batch_count

    if not epochs_fixed:
        # stopping condition = no change (small tolerance)
```

```python
        relative_change = tf.sqrt(
            tf.square(tf.divide(tf.subtract(centroids, prev_centroids), prev_centroids)))
        change_bool = tf.math.less_equal(relative_change, TOLERANCE)
        num_true = tf.reduce_sum(tf.cast(change_bool, tf.int32))
        if num_true == n_clusters * n_features:
            if stop_criterion:
                break
            else:
                stop_criterion = True
```

Listing A.1: Custom Training Loop v1 *distribute_ repeat*

A.1.2. Custom Training Loop v2 *distribute_ mean*

```python
per_batch_iterations_sum = 0
per_batch_centroids = 0
count = 0

for x in dataset:
    per_replica_centroids, per_replica_output, per_replica_iterations = strategy.run(clf.assign_and_update_fn, args=(x, centroids, epochs_fixed))
    per_batch_centroids += strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_centroids, axis=None)
    per_batch_iterations_sum += strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_iterations, axis=None)

    if strategy.num_replicas_in_sync > 1:
        per_replica_values = tf.convert_to_tensor(per_replica_output.values)
        a, b, c = per_replica_values.shape
        per_batch_output = tf.reshape(per_replica_values, [a * b, c])
    else:
        per_batch_output = tf.convert_to_tensor(per_replica_output)

    if count == 0:
        output = per_batch_output
    else:
        output = tf.concat([output, per_batch_output], axis=0)
        count += 1
denominator = n_objects // BATCH_SIZE_PER_REPLICA
centroids_final = per_batch_centroids / denominator
num_iterations_avg = per_batch_iterations_sum / denominator
```

Listing A.2: Custom Training Loop v2 *distribute_ mean*

A.2. Class *KMeans*

```
class KMeans:
    """K-means computation
        Attributes
        ----------
            n_clusters                -- the number of clusters (k) in the
                                         dataset
            n_features                -- number of dimensions for a single
                                         data point
            tolerance                 -- the dataset
            max_iter                  -- maximum iterations

        Methods
        -------
            initialize_centroids      -- return initial centroids
            assign_fn                 -- return cluster labels for each
                                         data point
            assign_and_prep_fn        -- return new centroids, number of
                                         points per cluster, data points
                                         with labels
            assign_and_update_fn      -- return final centroids, data
                                         points with labels, number of
                                         iterations needed
       """
    def __init__(self, n_clusters, n_features, tolerance, max_iter):
        self.n_clusters = n_clusters
        self.n_features = n_features
        self.tolerance = tolerance
        self.max_iter = max_iter

    @tf.function
    def initialize_centroids(self, data, centroids_fixed=False) -> tf.slice:
        """ initialize random centroids """
        if centroids_fixed:
            init_centroids = tf.slice(data[:, :-1], [0, 0], [self.n_clusters, -1])
        else:
            shuffled_data = tf.random.shuffle(data)
            init_centroids = tf.slice(shuffled_data[:, :-1], [0, 0], [self.n_clusters, -1])
        logging.info('Centroids initialized; fixed = {}'.format(centroids_fixed))
        return init_centroids

    @tf.function
    def assign_fn(self, points, centroids):
        """assign points to cluster, return assignment"""
        # compute scalar products (m^T) of each centroid with itself
        m = tf.linalg.diag_part(tf.tensordot(centroids, centroids, [[1], [1]]))
        # Column vector of n ones (1_n)
        n = tf.ones(tf.shape(points)[0])
        # calculate 0.5 * 1_n * m^T
        d1 = tf.multiply(tf.constant([0.5]), tf.einsum('i,j->ij', n, m))
        # calculate X*M^T
        d2 = tf.matmul(points, centroids, transpose_b=True)
        # calculate distance D:= 1/2 * 1_n * m^T - X * M^T
        distance = tf.subtract(d1, d2)
        # find minimum distance
        assignments = tf.math.argmin(input=distance, axis=1, output_type=tf.int32)

        return assignments

    @tf.function
    def assign_and_prep_fn(self, data, centroids):
        points = data[:, :-1]
        assignments = self.assign_fn(points, centroids)

        centroids = tf.math.unsorted_segment_sum(points, assignments, self.n_clusters)
        counts = tf.math.unsorted_segment_sum(tf.ones_like(points, dtype=tf.float32), assignments, self.n_clusters)

        assignments_expanded = tf.expand_dims(assignments, 1)
        output = tf.concat([tf.cast(points, tf.float32), tf.cast(assignments_expanded, tf.float32)], 1)

        return centroids, counts, output

    @tf.function
    def assign_and_update_fn(self, data: tf.float32, centroids: tf.float32, epochs_fixed) -> tuple:
        points = data[:, :-1]
        assert self.max_iter > 0, "max_iter must be at least 1"
        for it in tf.range(self.max_iter):  # must be tf-loop because of stopping condition
            assignments = self.assign_fn(points, centroids)

            centroids_prev = centroids
            centroids = tf.math.unsorted_segment_mean(points, tf.cast(assignments, tf.int32), self.n_clusters)

            if not epochs_fixed:
                # stopping condition = no change (small tolerance)
                change = tf.square(tf.divide(tf.subtract(centroids, centroids_prev), centroids_prev))
                cx = tf.maximum(change, 1e-9)
                rel_change = tf.sqrt(cx)

                bool_change = tf.math.less_equal(rel_change, self.tolerance)
                num_true = tf.reduce_sum(tf.cast(bool_change, tf.int32))
                # if num_true == self._n_clusters * self._n_features:
```

```
                if tf.equal(num_true, tf.multiply(self.n_clusters, self.n_features)):
                    break
        assignments_expanded = tf.expand_dims(assignments, 1)
        output = tf.concat([tf.cast(points, tf.float32), tf.cast(assignments_expanded, tf.float32)], 1)
        return centroids, output, it + 1

    @tf.function
    def predict_fn(self, data, centroids):
        points = data[:, :-1]
        assignments = self.assign_fn(points, centroids)
        assignments_expanded = tf.expand_dims(assignments, 1)
        output = tf.concat([tf.cast(points, tf.float32), tf.cast(assignments_expanded, tf.float32)], 1)
        return output
```

Listing A.3: Class KMeans

A.3. Standard k-Means

```python
class KMeans:
    def __init__(self, n_clusters, n_features, tolerance, max_iter):
        self.n_clusters = n_clusters
        self.n_features = n_features
        self.tolerance = tolerance
        self.max_iter = max_iter

    @tf.function
    def initialize_centroids_fn(self, data, centroids_fixed=False) -> tf.slice:
        """ initialize random centroids """
        if centroids_fixed:
            init_centroids = tf.slice(data[:, :-1], [0, 0], [self.n_clusters, -1])
        else:
            shuffled_data = tf.random.shuffle(data)
            init_centroids = tf.slice(shuffled_data[:, :-1], [0, 0], [self.n_clusters, -1])
        logging.info('Centroids initialized; fixed = {}'.format(centroids_fixed))
        return init_centroids

    @tf.function
    def assign_fn(self, points, centroids):
        # calculate distance to each centroids (result = array of cluster_num arrays)
        centroids_expanded = tf.expand_dims(centroids, 1)
        d = tf.reduce_sum(tf.square(tf.subtract(points, centroids_expanded)), 2)
        distance = tf.sqrt(tf.maximum(d, 1e-9))
        assignments = tf.math.argmin(input=distance, axis=0)
        return assignments

    @tf.function
    def assign_and_prep_fn(self, data, centroids):
        points = data[:, :-1]
        assignments = self.assign_fn(points, centroids)

        centroids = tf.math.unsorted_segment_sum(points, tf.cast(assignments, tf.int32), self.n_clusters)
        counts = tf.math.unsorted_segment_sum(tf.ones_like(points, dtype=tf.float32), assignments, self.n_clusters)

        assignments_expanded = tf.expand_dims(assignments, 1)
        output = tf.concat([tf.cast(points, tf.float32), tf.cast(assignments_expanded, tf.float32)], 1)
        return centroids, counts, output

    @tf.function
    def assign_and_update_fn(self, data: tf.float32, centroids: tf.float32, epochs_fixed) -> tuple:
        points = data[:, :-1]
        assert self.max_iter > 0, "max_iter must be at least 1"
        for it in tf.range(self.max_iter):

            assignments = self.assign_fn(points, centroids)

            centroids_prev = centroids
            centroids = tf.math.unsorted_segment_mean(points, tf.cast(assignments, tf.int32), self.n_clusters)
            logging.info('Centroids updated')

            if not epochs_fixed:
                change = tf.square(tf.divide(tf.subtract(centroids, centroids_prev), centroids_prev))
                cx = tf.maximum(change, 1e-9)
                rel_change = tf.sqrt(cx)

                bool_change = tf.math.less_equal(rel_change, self.tolerance)
                num_true = tf.reduce_sum(tf.cast(bool_change, tf.int32))
                if tf.equal(num_true, tf.multiply(self.n_clusters, self.n_features)):
                    break
        assignments_expanded = tf.expand_dims(assignments, 1)
        output = tf.concat([tf.cast(points, tf.float32), tf.cast(assignments_expanded, tf.float32)], 1)
        return centroids, output, it + 1

    @tf.function
    def predict_fn(self, data, centroids):
        points = data[:, :-1]
        assignments = self.assign_fn(points, centroids)
        assignments_expanded = tf.expand_dims(assignments, 1)
        output = tf.concat([tf.cast(points, tf.float32), tf.cast(assignments_expanded, tf.float32)], 1)
        return output
```

Listing A.4: Standard k-Means

A.4. Adpations of the *main.py* Script

Table A.4 lists all adaptations of the *main.py* Script that are used for the experiments.

Python Script	**Description**
main.py	Contains the optimal version
main_combine.py	Combines v1 → v2 or v2 → v1, depending on which version is set as first.
main_combine_loop.py	Combines v1 → v2 or v2 → v1, depending on which version is set as first, and loops over all synthetic or real data sets. It must be specified whether the synthetic or real data sets should be used. It is important to set the correct directory of the files.
main_combine_loop2.py	main_combine_loop2.py adapted for Experiment 4 part 2 and 3
main_repeat.py	Executes version 1, resets the centroids to the initial centroids, and repeats the execution with version 2.
main_repeat_loop.py	Executes version 1, resets the centroids to the initial centroids, and repeats the execution with version 2, and loops over all synthetic or real data sets. It furthermore, loops over the iterations array and switches between max iterations fixed or not fixed.
main_repeat_loop2.py	main_repeat_loop.py adapted for Experiment 4 part 2 and 3
main_repeat_loop3.py	main_repeat_loop.py adapted for Experiment 5
main_repeat_batch_loop.py	main_repeat_loop.py adapted for Experiment 1

Table A.1.: Python Scripts for the Experiments

A.5. Additional Data Experiment 3

Additional data for Experiment 3 include Figure A.1, Table A.2 and Table A.3.

A.6. Additional Data Eperiment 4

A.6.1. Part 1: Runtime Comparison

Additional data for Experiment 4 include Figures A.2, A.3, A.4, A.5, A.6, A.7, A.8, A.9, and A.10 as part of the runtime comparison.

A.6.2. Part 1: Quality Comparison

Additional data for Experiment 4 include Figures A.11, A.12, A.13, and A.14 as part of the quality comparison.

A.6.3. Part 2: Using Random Initial Centroids

The results of the second part of the experiment are summarized in the included document 1.

Comparison v1 -> v2 vs. version 2	Min. von Elapsed time	Min. von SSE	Max. von CH	Min. von DB	Max. von NMI	Max. von RI
1	**00:01.0**	**670.753411**	**12901.444909**	**1.346372**	**0.936263**	**0.835041**
2022-05-04_08-37-43	**00:02.0**	**2 054.087881**	**3228.593942**	**2.225034**	**0.846085**	**0.606418**
2	00:06.0	2 054.095767	3228.574657	2.250556	0.846085	0.605058
5	00:02.0	2 054.090721	3228.586529	2.233243	0.845954	0.604704
10	00:04.0	2 054.087881	3228.593942	2.225034	0.845970	0.604747
-	00:02.0	2 065.723432	3203.427739	2.404847	0.840596	0.606418
2022-05-04_09-00-37	**00:01.0**	**2 667.998269**	**2507.714557**	**2.655368**	**0.797421**	**0.532506**
2	00:02.0	2 667.998269	2507.714557	2.655368	0.797421	0.532506
5	00:02.0	2 668.011881	2507.693458	2.705751	0.797419	0.532499
10	00:04.0	2 668.012460	2507.692601	2.708536	0.797406	0.532465
-	00:01.0	3 108.320917	2010.333955	2.710281	0.751393	0.497446
2022-05-04_09-20-38	**00:01.0**	**1 135.566700**	**7022.255962**	**2.371902**	**0.874884**	**0.706561**
2	00:02.0	1 135.589763	7022.084824	2.440531	0.870459	0.695019
5	00:02.0	1 135.572950	7022.208906	2.388058	0.869947	0.693623
10	00:04.0	1 135.566700	7022.255962	2.371902	0.869614	0.692706
-	00:01.0	1 136.973965	7021.551088	2.649958	0.874884	0.706561
2022-05-04_09-39-14	**00:01.0**	**670.753411**	**12901.444909**	**1.346372**	**0.936263**	**0.835041**
2	00:02.0	670.757105	12901.365221	1.356758	0.936263	0.835041
5	00:02.0	670.753411	12901.444909	1.346372	0.936129	0.834696
10	00:04.0	670.757626	12901.352198	1.362896	0.936119	0.834670
-	00:01.0	746.384647	11448.053231	1.432045	0.914965	0.814181
2022-05-04_09-55-59	**00:01.0**	**1 495.500016**	**4980.347479**	**1.625007**	**0.877883**	**0.718004**
2	00:02.0	1 495.510153	4980.305473	1.650821	0.877472	0.717264
5	00:03.0	1 495.506820	4980.318782	1.639682	0.877355	0.717055
10	00:04.0	1 495.500016	4980.347479	1.625007	0.877362	0.717067
-	00:01.0	1 495.537464	4980.193595	1.726418	0.877883	0.718004
2022-05-04_10-12-39	**00:01.0**	**3 330.287765**	**1431.884912**	**2.786382**	**0.676360**	**0.383577**
2	00:02.0	3 330.330102	1431.848947	2.900287	0.673327	0.378044
5	00:03.0	3 330.313113	1431.863371	2.855036	0.673030	0.377480
10	00:04.0	3 330.287765	1431.884912	2.786382	0.672984	0.377391
-	00:01.0	3 346.692480	1431.775740	3.066695	0.676360	0.383577
2022-05-04_10-28-09	**00:01.0**	**1 245.878237**	**7316.993797**	**1.651507**	**0.888775**	**0.729184**
2	00:02.0	1 245.880331	7316.979740	1.677964	0.888773	0.729179
5	00:03.0	1 245.878237	7316.993797	1.668650	0.888775	0.729184
10	00:04.0	1 245.878237	7316.993797	1.668650	0.888775	0.729184
-	00:01.0	1 353.915679	6424.305121	1.651507	0.883294	0.726398
2022-05-04_10-45-34	**00:01.0**	**1 253.557546**	**6252.307714**	**1.746219**	**0.918142**	**0.821316**
2	00:02.0	1 253.557546	6252.307714	1.746219	0.918112	0.821235
5	00:03.0	1 253.558129	6252.305051	1.748291	0.918085	0.821164
10	00:04.0	1 253.558129	6252.305051	1.748291	0.918085	0.821164
-	00:01.0	1 253.566848	6252.250545	1.781867	0.918142	0.821316
2022-05-04_11-00-32	**00:01.0**	**2 196.853528**	**2923.699262**	**2.416261**	**0.751513**	**0.411513**
2	00:02.0	2 196.877546	2923.651641	2.485141	0.751513	0.405640
5	00:02.0	2 196.853528	2923.699262	2.416261	0.751276	0.405009
10	00:04.0	2 196.857784	2923.690453	2.444552	0.751234	0.404930
-	00:01.0	2 215.661064	2901.408396	2.729213	0.749165	0.411513
2022-05-04_11-16-11	**00:01.0**	**2 370.583566**	**2602.302121**	**1.848841**	**0.800682**	**0.553394**
2	00:02.0	2 370.612162	2602.252373	1.940779	0.797623	0.550050
5	00:02.0	2 370.599563	2602.273731	1.884074	0.797314	0.549674
10	00:04.0	2 370.583566	2602.302121	1.848841	0.797368	0.549745
-	00:01.0	2 396.301391	2602.105617	2.158462	0.800682	0.553394
2	**00:01.0**	**10 211.322187**	**9384.739580**	**1.621636**	**0.955481**	**0.858327**
2022-05-04_08-37-43	**00:01.0**	**34 645.720982**	**2285.327288**	**3.178137**	**0.873789**	**0.534151**
2	00:02.0	34 651.687998	2285.326628	3.184435	0.873789	0.531296
5	00:03.0	34 651.586539	2285.327235	3.178137	0.873552	0.530723
10	00:04.0	34 645.720982	2285.327288	3.183387	0.873390	0.530401
-	00:01.0	35 801.873536	2202.650879	3.334753	0.871648	0.534151
2022-05-04_09-00-37	**00:01.0**	**14 205.330793**	**6554.451386**	**2.107353**	**0.952860**	**0.849106**
2	00:02.0	14 214.536929	6554.446865	2.107353	0.952860	0.848631
5	00:03.0	14 205.331079	6554.451316	2.113392	0.952325	0.846976
10	00:04.0	14 205.330793	6554.451386	2.113178	0.952319	0.846955
-	00:01.0	14 867.397891	6263.321764	2.120321	0.948742	0.849106
2022-05-04_09-20-38	**00:01.0**	**41 228.213139**	**1811.610230**	**3.920738**	**0.854221**	**0.506951**
2	00:02.0	41 241.757999	1811.609017	4.095633	0.847706	0.492433
5	00:03.0	41 228.213221	1811.610196	4.023640	0.847459	0.491722
10	00:04.0	41 228.213139	1811.610230	4.025102	0.847354	0.491508
-	00:01.0	42 085.807168	1811.603713	3.920738	0.854221	0.506951
2022-05-04_09-39-14	**00:01.0**	**26 577.264675**	**3186.044879**	**3.235666**	**0.900024**	**0.689539**

2	00:02.0	26 577.272682	3186.043805	3.262069	0.899281	0.687239
5	00:02.0	26 577.264675	3186.044879	3.235666	0.899300	0.687268
10	00:04.0	26 577.270171	3186.044224	3.254231	0.899310	0.687292
-	00:01.0	26 586.505508	3186.041336	3.306567	0.900024	0.689539
2022-05-04_09-55-59	**00:01.0**	**14 531.003190**	**6392.683228**	**1.621636**	**0.955481**	**0.858327**
2	00:02.0	14 531.012347	6392.677953	1.654713	0.955481	0.858327
5	00:03.0	14 531.003190	6392.683228	1.621636	0.955379	0.858108
10	00:04.0	14 539.255199	6392.682616	1.624489	0.955397	0.858147
-	00:01.0	16 937.775060	5456.995047	1.784149	0.943611	0.844734
2022-05-04_10-12-39	**00:01.0**	**10 211.322187**	**9384.739580**	**2.004475**	**0.953789**	**0.851369**
2	00:02.0	10 211.365903	9384.699862	2.068631	0.953789	0.851369
5	00:03.0	10 211.322199	9384.739580	2.004475	0.952379	0.847146
10	00:05.0	10 211.322187	9384.739345	2.007998	0.952339	0.847019
-	00:01.0	13 068.914611	7464.936904	2.398649	0.938981	0.834922
2022-05-04_10-28-09	**00:01.0**	**26 007.471108**	**3272.269361**	**2.919374**	**0.913868**	**0.665584**
2	00:02.0	26 007.471108	3272.267033	2.980367	0.910780	0.656911
5	00:03.0	26 007.649370	3272.267783	2.967881	0.910701	0.656672
10	00:04.0	26 007.763866	3272.269361	2.919374	0.910682	0.656618
-	00:01.0	26 354.287848	3272.258637	2.994696	0.913868	0.665584
2022-05-04_10-45-34	**00:01.0**	**29 223.800241**	**2836.470947**	**3.353110**	**0.901154**	**0.691382**
2	00:02.0	29 223.860239	2836.469388	3.434047	0.899684	0.688206
5	00:03.0	29 228.712077	2836.470923	3.355261	0.899580	0.687954
10	00:04.0	29 223.800241	2836.470947	3.353110	0.899448	0.687649
-	00:01.0	29 286.748368	2836.465709	3.425148	0.901154	0.691382
2022-05-04_11-00-32	**00:01.0**	**35 519.633387**	**2380.570772**	**3.454202**	**0.874625**	**0.617923**
2	00:02.0	35 519.633387	2380.569360	3.512228	0.874625	0.617509
5	00:03.0	35 596.354438	2380.570772	3.454202	0.874384	0.616968
10	00:05.0	35 595.438311	2380.570294	3.486982	0.874352	0.616858
-	00:01.0	35 638.831167	2221.648877	3.618125	0.872705	0.617923
2022-05-04_11-16-11	**00:01.0**	**56 497.376792**	**1140.317720**	**3.910056**	**0.797083**	**0.445414**
2	00:02.0	56 584.124591	1140.317132	3.934891	0.797083	0.433782
5	00:03.0	56 501.629188	1140.317720	3.912601	0.796655	0.433145
10	00:04.0	56 497.376792	1140.317640	3.910056	0.796673	0.433218
-	00:01.0	58 136.274359	1109.293291	4.108474	0.793783	0.445414
3	**00:01.0**	**34 573.149269**	**5531.723054**	**2.645216**	**0.952240**	**0.846705**
2022-05-04_08-37-43	**00:01.0**	**34 573.149269**	**5531.723054**	**2.706848**	**0.952200**	**0.846581**
2	00:02.0	34 573.153538	5531.722249	2.729315	0.952196	0.846568
5	00:03.0	34 573.150092	5531.722964	2.710654	0.952197	0.846570
10	00:04.0	34 573.149269	5531.723054	2.706848	0.952200	0.846581
-	00:01.0	40 993.916398	4613.602780	2.792303	0.937358	0.829585
2022-05-04_09-00-37	**00:01.0**	**86 686.892472**	**1927.497266**	**3.489726**	**0.878638**	**0.630382**
2	00:02.0	86 686.892472	1927.496627	3.537746	0.878638	0.630382
5	00:03.0	86 782.049405	1927.497123	3.489726	0.878555	0.630083
10	00:04.0	86 720.369245	1927.497266	3.507597	0.878368	0.629671
-	00:01.0	99 069.714471	1622.595489	5.060531	0.859227	0.604314
2022-05-04_09-20-38	**00:01.0**	**109 661.490999**	**1277.288666**	**6.211214**	**0.832020**	**0.508598**
2	00:02.0	109 661.513099	1277.288339	6.275143	0.828635	0.499271
5	00:03.0	109 661.495631	1277.288567	6.222025	0.828680	0.499425
10	00:04.0	109 661.490999	1277.288666	6.211214	0.828486	0.498810
-	00:01.0	109 902.367532	1277.286325	6.494032	0.832020	0.508598
2022-05-04_09-39-14	**00:01.0**	**71 941.413593**	**2469.198999**	**3.218409**	**0.906958**	**0.704575**
2	00:02.0	71 941.428599	2469.198089	3.296002	0.906958	0.704575
5	00:03.0	71 941.413593	2469.198745	3.242219	0.906816	0.704258
10	00:04.0	71 955.526893	2469.198999	3.218409	0.906849	0.704333
-	00:01.0	87 216.812721	1952.661664	3.672842	0.872940	0.659208
2022-05-04_09-55-59	**00:01.0**	**88 500.016088**	**1749.712361**	**4.470173**	**0.865980**	**0.603205**
2	00:02.0	88 559.388856	1749.711890	4.492235	0.865980	0.581431
5	00:03.0	88 500.016088	1749.712361	4.470173	0.865511	0.580385
10	00:04.0	88 546.793507	1749.712317	4.481732	0.865525	0.580401
-	00:01.0	95 708.320591	1600.815923	4.492428	0.862037	0.603205
2022-05-04_10-12-39	**00:01.0**	**74 585.522129**	**2355.778958**	**4.019196**	**0.910545**	**0.734520**
2	00:02.0	74 585.535383	2355.778400	4.084544	0.910545	0.734520
5	00:03.0	74 608.880489	2355.778921	4.027328	0.910463	0.734254
10	00:04.0	74 585.522129	2355.778958	4.019196	0.910461	0.734248
-	00:01.0	75 630.744569	2186.705411	4.069523	0.907252	0.732630
2022-05-04_10-28-09	**00:01.0**	**91 568.080864**	**1664.348465**	**5.098081**	**0.862879**	**0.571478**
2	00:02.0	91 568.099926	1664.348001	5.157808	0.862879	0.571089
5	00:03.0	91 568.080871	1664.348461	5.106541	0.862723	0.570566
10	00:04.0	91 568.080864	1664.348465	5.098081	0.862754	0.570658

-	00:01.0	101 080.535840	1522.514156	5.210931	0.856534	0.571478
2022-05-04_10-45-34	**00:01.0**	**67 621.315580**	**2495.132652**	**3.319564**	**0.902688**	**0.699132**
2	00:02.0	67 621.334891	2495.131795	3.396798	0.902251	0.698261
5	00:03.0	67 621.321901	2495.132420	3.350709	0.902286	0.698336
10	00:04.0	67 621.315580	2495.132652	3.319564	0.902285	0.698335
-	00:01.0	68 357.909107	2495.125685	4.224682	0.902688	0.699132
2022-05-04_11-00-32	**00:01.0**	**36 021.333842**	**5281.974105**	**2.645216**	**0.952240**	**0.846705**
2	00:02.0	36 021.343879	5281.972415	2.694364	0.952234	0.846687
5	00:03.0	36 021.336328	5281.973611	2.659951	0.952239	0.846703
10	00:04.0	36 021.333842	5281.974105	2.645216	0.952240	0.846705
-	00:01.0	37 451.808607	5054.855857	2.797664	0.948777	0.844905
2022-05-04_11-16-11	**00:01.0**	**72 371.449840**	**2287.215756**	**4.079615**	**0.910997**	**0.740305**
2	00:02.0	72 371.929761	2287.215529	4.107905	0.906145	0.728369
5	00:03.0	72 371.724723	2287.215672	4.095621	0.906020	0.728115
10	00:04.0	72 371.449840	2287.215756	4.079615	0.906055	0.728212
-	00:01.0	72 731.717583	2287.211393	4.096720	0.910997	0.740305
4	**00:01.0**	**159 802.938193**	**24922.775636**	**3.461554**	**0.954134**	**0.810830**
2022-05-04_08-37-43	**00:01.0**	**326 156.329310**	**10470.697562**	**7.271524**	**0.907667**	**0.642741**
2	00:02.0	326 156.332950	10470.697312	7.315030	0.907647	0.642676
5	00:03.0	326 156.331211	10470.697442	7.320942	0.907660	0.642712
10	00:05.0	326 156.329310	10470.697562	7.271524	0.907667	0.642741
-	00:01.0	346 920.748190	9728.558033	8.980401	0.900031	0.628546
2022-05-04_09-00-37	**00:01.0**	**211 415.971428**	**17424.335895**	**3.821869**	**0.944719**	**0.783689**
2	00:02.0	211 415.974064	17424.335652	3.836762	0.944336	0.782925
5	00:03.0	211 415.971428	17424.335895	3.821869	0.944348	0.782946
10	00:05.0	211 415.972158	17424.335877	3.850465	0.944348	0.782946
-	00:01.0	211 416.024453	17424.330533	4.782028	0.944719	0.783689
2022-05-04_09-20-38	**00:01.0**	**159 802.938193**	**24922.775636**	**3.461554**	**0.954134**	**0.810830**
2	00:02.0	159 802.945724	24922.774459	3.618257	0.954124	0.809372
5	00:03.0	159 802.938193	24922.775522	3.461554	0.954134	0.809401
10	00:05.0	159 802.938322	24922.775636	3.464646	0.954132	0.809392
-	00:01.0	167 046.679883	24326.051922	4.336114	0.953712	0.810830
2022-05-04_09-39-14	**00:01.0**	**324 045.623560**	**10220.346023**	**7.720566**	**0.904982**	**0.635729**
2	00:02.0	324 045.623600	10220.346023	7.720566	0.904641	0.634394
5	00:04.0	324 045.623560	10220.345910	7.752955	0.904634	0.634392
10	00:05.0	324 045.623662	10220.345975	7.748616	0.904627	0.634373
-	00:01.0	325 669.359235	10220.341810	10.619167	0.904982	0.635729
2022-05-04_09-55-59	**00:01.0**	**213 086.270544**	**17261.878302**	**4.951564**	**0.949864**	**0.797269**
2	00:02.0	213 086.270544	17261.878302	4.951564	0.949864	0.795718
5	00:04.0	213 086.272600	17261.878098	4.987335	0.949860	0.795702
10	00:05.0	213 086.272498	17261.877990	4.990176	0.949862	0.795710
-	00:01.0	214 750.453452	17105.273655	6.545633	0.948838	0.797269
2022-05-04_10-12-39	**00:01.0**	**249 929.988671**	**14704.885051**	**6.234330**	**0.927242**	**0.741364**
2	00:02.0	249 929.991027	14704.884850	6.318372	0.927242	0.741251
5	00:04.0	249 929.989149	14704.885048	6.234330	0.927236	0.741232
10	00:05.0	249 929.988671	14704.885051	6.245515	0.927237	0.741237
-	00:01.0	265 775.105939	14497.111933	7.762724	0.926280	0.741364
2022-05-04_10-28-09	**00:01.0**	**223 722.758543**	**16284.183834**	**4.744204**	**0.937674**	**0.775941**
2	00:02.0	223 722.758543	16284.183834	4.744204	0.937118	0.774768
5	00:03.0	223 722.762736	16284.183584	4.831559	0.937089	0.774713
10	00:05.0	223 722.761975	16284.183592	4.807547	0.937087	0.774705
-	00:01.0	223 722.808682	16284.179467	5.610474	0.937674	0.775941
2022-05-04_10-45-34	**00:01.0**	**225 804.771219**	**16103.602798**	**6.629067**	**0.933313**	**0.758307**
2	00:02.0	225 804.771950	16103.602798	6.629067	0.932967	0.756957
5	00:04.0	225 804.771940	16103.602613	6.649635	0.932967	0.756957
10	00:05.0	225 804.771219	16103.602656	6.641034	0.932967	0.756960
-	00:01.0	225 804.838137	16103.597136	8.560655	0.933313	0.758307
2022-05-04_11-00-32	**00:01.0**	**233 912.744260**	**15430.937306**	**5.425270**	**0.932760**	**0.746752**
2	00:02.0	233 912.744649	15430.937194	5.448041	0.932749	0.746721
5	00:03.0	233 912.744260	15430.937306	5.425270	0.932746	0.746707
10	00:05.0	233 912.747335	15430.937047	5.513921	0.932760	0.746752
-	00:01.0	242 777.483397	14779.310096	6.799587	0.927212	0.740272
2022-05-04_11-16-11	**00:01.0**	**235 129.444509**	**15334.011606**	**5.521202**	**0.935052**	**0.751658**
2	00:02.0	235 129.444509	15334.011606	5.521202	0.932763	0.746747
5	00:03.0	235 129.445740	15334.011534	5.582094	0.932742	0.746691
10	00:05.0	235 129.446245	15334.011469	5.605280	0.932746	0.746700
-	00:01.0	235 537.614900	15334.006019	6.820441	0.935052	0.751658
5	**00:01.0**	**1 461 789.038040**	**10298.055632**	**6.273266**	**0.957762**	**0.798949**
2022-05-04_08-37-43	**00:01.0**	**1 970 973.173441**	**7096.694449**	**8.311893**	**0.941439**	**0.708931**

	2	00:02.0	1 970 973.182629	7096.694348	8.544419	0.939679	0.702349
	5	00:04.0	1 970 973.174554	7096.694449	8.323434	0.939695	0.702403
	10	00:05.0	1 970 973.173441	7096.694447	8.311893	0.939690	0.702399
	-	00:01.0	1 976 597.302512	7096.694327	9.692402	0.941439	0.708931
	2022-05-04_09-00-37	**00:01.0**	**2 685 829.040619**	**4775.501725**	**10.387496**	**0.921250**	**0.656831**
	2	00:02.0	2 685 829.051505	4775.501685	10.656260	0.920179	0.654098
	5	00:04.0	2 685 829.040619	4775.501715	10.387496	0.920202	0.654185
	10	00:05.0	2 685 829.043425	4775.501725	10.457897	0.920215	0.654187
	-	00:01.0	2 687 869.797025	4775.501570	12.575494	0.921250	0.656831
	2022-05-04_09-20-38	**00:01.0**	**1 461 789.038040**	**10298.055632**	**6.273266**	**0.957762**	**0.798949**
	2	00:02.0	1 461 789.040071	10298.055631	6.297105	0.956772	0.794686
	5	00:04.0	1 461 789.038040	10298.055632	6.273266	0.956806	0.794815
	10	00:05.0	1 461 789.039795	10298.055596	6.312006	0.956776	0.794722
	-	00:01.0	1 465 058.497545	10134.495235	7.021585	0.957762	0.798949
	2022-05-04_09-39-14	**00:01.0**	**2 262 891.568676**	**5971.646831**	**9.881952**	**0.932526**	**0.693842**
	2	00:02.0	2 262 891.572108	5971.646793	9.963457	0.932505	0.693742
	5	00:03.0	2 262 891.568676	5971.646798	9.881952	0.932503	0.693778
	10	00:05.0	2 262 891.570713	5971.646831	9.921029	0.932526	0.693842
	-	00:01.0	2 296 764.995766	5865.236665	11.589610	0.930680	0.692902
	2022-05-04_09-55-59	**00:01.0**	**2 253 419.346630**	**6100.463776**	**9.209748**	**0.932846**	**0.689615**
	2	00:02.0	2 253 419.346630	6100.463729	9.209748	0.932826	0.689534
	5	00:04.0	2 253 419.348509	6100.463768	9.270452	0.932846	0.689615
	10	00:05.0	2 253 419.347573	6100.463776	9.266811	0.932845	0.689609
	-	00:01.0	2 297 052.561517	5937.607465	11.553707	0.930671	0.685913
	2022-05-04_10-12-39	**00:01.0**	**2 417 091.650964**	**5487.068041**	**8.685604**	**0.929630**	**0.674972**
	2	00:02.0	2 417 091.662644	5487.068021	8.952860	0.929630	0.674972
	5	00:04.0	2 417 091.656761	5487.068041	8.891731	0.929453	0.674501
	10	00:05.0	2 417 091.650964	5487.068018	8.685604	0.929479	0.674563
	-	00:01.0	2 672 226.046972	5031.993582	10.208030	0.923353	0.663380
	2022-05-04_10-28-09	**00:01.0**	**1 917 042.698423**	**7342.019232**	**7.205726**	**0.944464**	**0.746714**
	2	00:02.0	1 917 042.698423	7342.019203	7.355783	0.943744	0.745041
	5	00:04.0	1 917 118.965774	7342.019232	7.205726	0.943744	0.745019
	10	00:05.0	1 917 112.721301	7342.019213	7.259982	0.943742	0.744991
	-	00:01.0	1 920 179.125356	7342.018969	8.668768	0.944464	0.746714
	2022-05-04_10-45-34	**00:01.0**	**2 438 231.495295**	**5425.402531**	**9.241087**	**0.929591**	**0.686959**
	2	00:02.0	2 438 231.495295	5425.402531	9.241087	0.929558	0.686863
	5	00:04.0	2 438 231.500982	5425.402521	9.416629	0.929591	0.686959
	10	00:05.0	2 438 231.503471	5425.402512	9.488232	0.929581	0.686923
	-	00:01.0	2 475 250.530153	5325.033186	11.223729	0.926699	0.681699
	2022-05-04_11-00-32	**00:01.0**	**1 718 217.319335**	**8379.581222**	**7.118295**	**0.946066**	**0.718684**
	2	00:02.0	1 718 217.320159	8379.581205	7.187833	0.946040	0.718581
	5	00:04.0	1 718 217.319335	8379.581206	7.118295	0.946056	0.718651
	10	00:05.0	1 718 217.321180	8379.581222	7.135796	0.946066	0.718684
	-	00:01.0	1 809 582.095166	8073.156672	8.129865	0.944118	0.714690
	2022-05-04_11-16-11	**00:01.0**	**1 841 764.847567**	**7708.508637**	**7.203599**	**0.945548**	**0.737616**
	2	00:02.0	1 841 764.850754	7708.508618	7.254714	0.945548	0.736659
	5	00:04.0	1 841 764.847567	7708.508605	7.203599	0.945527	0.736611
	10	00:05.0	1 841 764.848535	7708.508637	7.270068	0.945531	0.736623
	-	00:01.0	1 888 609.238381	7543.034854	8.372769	0.945072	0.737616
6		**00:02.0**	**8 161 059.002974**	**6768.693619**	**10.100550**	**0.946914**	**0.756775**
	2022-05-04_08-37-43	**00:02.0**	**10 140 703.653395**	**5130.220744**	**12.693119**	**0.930684**	**0.647510**
	2	00:04.0	10 140 703.672376	5130.220738	14.088811	0.926675	0.633037
	5	00:06.0	10 140 703.656735	5130.220718	13.874534	0.926591	0.632668
	10	00:08.0	10 140 703.653395	5130.220744	13.631185	0.926922	0.634059
	-	00:02.0	10 147 727.517857	5130.220641	12.693119	0.930684	0.647510
	2022-05-04_09-00-37	**00:02.0**	**10 059 420.153454**	**5184.798817**	**10.936059**	**0.934015**	**0.682489**
	2	00:04.0	10 059 420.155862	5184.798804	11.373198	0.931689	0.675150
	5	00:06.0	10 059 420.153454	5184.798817	11.411489	0.931635	0.674831
	10	00:09.0	10 059 420.154825	5184.798810	11.386848	0.931707	0.675181
	-	00:02.0	10 064 634.909462	5184.798789	10.936059	0.934015	0.682489
	2022-05-04_09-20-38	**00:02.0**	**11 746 142.342013**	**4207.096536**	**12.755277**	**0.923154**	**0.652177**
	2	00:04.0	11 746 142.342013	4207.096536	13.560375	0.920502	0.644776
	5	00:06.0	11 746 182.934996	4207.096525	13.349578	0.920091	0.643167
	10	00:09.0	11 746 250.327362	4207.096519	13.215953	0.920185	0.643648
	-	00:02.0	11 757 110.258502	4207.096489	12.755277	0.923154	0.652177
	2022-05-04_09-39-14	**00:02.0**	**8 754 711.449849**	**6299.633677**	**10.617209**	**0.943685**	**0.744646**
	2	00:04.0	8 754 711.458342	6299.633629	10.815860	0.943618	0.741899
	5	00:06.0	8 754 711.449849	6299.633639	10.829103	0.943349	0.740685
	10	00:08.0	8 754 711.456471	6299.633677	10.807926	0.943473	0.741365

	-	00:02.0	8 977 373.076968	6216.394150	10.617209	0.943685	0.744646
	2022-05-04_09-55-59	**00:02.0**	**9 317 318.557594**	**5727.140708**	**12.017392**	**0.941305**	**0.748301**
	2	00:04.0	9 317 318.570176	5727.140648	12.593220	0.940263	0.742989
	5	00:06.0	9 317 318.565645	5727.140681	12.231965	0.940564	0.744257
	10	00:09.0	9 317 318.557594	5727.140708	12.309092	0.940274	0.742984
	-	00:02.0	9 360 886.296122	5694.381124	12.017392	0.941305	0.748301
	2022-05-04_10-12-39	**00:02.0**	**12 397 587.601137**	**3900.597339**	**13.956128**	**0.916183**	**0.625366**
	2	00:04.0	12 397 587.615669	3900.597325	14.911881	0.913105	0.614579
	5	00:06.0	12 397 587.601137	3900.597339	14.696558	0.913252	0.615218
	10	00:09.0	12 397 587.603575	3900.597338	14.841384	0.913043	0.614206
	-	00:02.0	12 399 477.830381	3900.597318	13.956128	0.916183	0.625366
	2022-05-04_10-28-09	**00:02.0**	**10 055 505.880358**	**5187.449878**	**11.880505**	**0.926373**	**0.628273**
	2	00:04.0	10 055 505.888753	5187.449854	12.191128	0.926179	0.611798
	5	00:06.0	10 055 505.880358	5187.449859	12.035092	0.926339	0.612269
	10	00:08.0	10 055 505.882264	5187.449878	12.129977	0.926211	0.611765
	-	00:02.0	10 217 391.886294	5087.675760	11.880505	0.926373	0.628273
	2022-05-04_10-45-34	**00:02.0**	**8 161 059.002974**	**6768.693619**	**10.100550**	**0.946914**	**0.756775**
	2	00:05.0	8 161 377.637328	6768.693599	10.917713	0.946247	0.750872
	5	00:06.0	8 161 059.002974	6768.693619	10.580985	0.946528	0.752226
	10	00:09.0	8 161 059.006972	6768.693601	10.449215	0.946914	0.753842
	-	00:02.0	8 282 972.474057	6660.506463	10.100550	0.946832	0.756775
	2022-05-04_11-00-32	**00:02.0**	**10 961 231.932342**	**4624.578754**	**12.511611**	**0.920015**	**0.606206**
	2	00:04.0	10 961 231.943305	4624.578738	13.374918	0.919610	0.600238
	5	00:06.0	10 961 231.932701	4624.578754	13.306417	0.919419	0.599459
	10	00:08.0	10 961 231.932342	4624.578748	13.090745	0.919855	0.601284
	-	00:02.0	11 176 168.346854	4517.139559	12.511611	0.920015	0.606206
	2022-05-04_11-16-11	**00:02.0**	**11 269 549.856394**	**4453.630653**	**11.656255**	**0.927320**	**0.671835**
	2	00:04.0	11 269 726.314530	4453.630620	12.266441	0.924905	0.663378
	5	00:06.0	11 269 549.862332	4453.630637	12.167621	0.924706	0.662824
	10	00:08.0	11 269 549.856394	4453.630653	12.220794	0.924418	0.661614
	-	00:02.0	11 299 760.596803	4444.148951	11.656255	0.927320	0.671835
7		**00:02.0**	**4 157 463.790502**	**18917.553312**	**7.839878**	**0.951747**	**0.710882**
	2022-05-04_08-37-43	**00:02.0**	**5 299 418.657003**	**13972.552316**	**10.298382**	**0.936706**	**0.648174**
	2	00:04.0	5 299 418.657003	13972.552316	10.368361	0.936683	0.648134
	5	00:07.0	5 299 418.658720	13972.552228	10.394843	0.936697	0.648155
	10	00:11.0	5 299 418.657606	13972.552282	10.298382	0.936691	0.648132
	-	00:02.0	5 300 337.087510	13972.552066	12.505198	0.936706	0.648174
	2022-05-04_09-00-37	**00:02.0**	**5 971 496.959792**	**11946.351061**	**12.402428**	**0.926582**	**0.577071**
	2	00:05.0	5 971 496.961471	11946.351033	12.733982	0.926582	0.577069
	5	00:07.0	5 971 496.959792	11946.351061	12.402428	0.926573	0.577031
	10	00:11.0	5 971 496.959992	11946.351045	12.483566	0.926582	0.577071
	-	00:02.0	5 971 771.192954	11946.350899	15.626095	0.926541	0.576846
	2022-05-04_09-20-38	**00:02.0**	**5 211 642.939512**	**14275.765017**	**10.861246**	**0.937369**	**0.644366**
	2	00:04.0	5 211 642.942152	14275.764966	11.071460	0.937369	0.642965
	5	00:07.0	5 211 642.939925	14275.765017	10.861246	0.937363	0.642946
	10	00:11.0	5 211 642.939512	14275.764990	10.896948	0.937364	0.642949
	-	00:02.0	5 266 474.212934	14105.844183	14.081280	0.936743	0.644366
	2022-05-04_09-39-14	**00:02.0**	**4 157 463.790502**	**18917.553312**	**7.839878**	**0.951747**	**0.710882**
	2	00:04.0	4 157 463.793762	18917.553287	8.040914	0.951736	0.710838
	5	00:06.0	4 157 463.790903	18917.553310	7.839878	0.951747	0.710882
	10	00:11.0	4 157 463.790502	18917.553312	7.925829	0.951744	0.710866
	-	00:02.0	4 209 499.603319	18672.172644	10.546057	0.950820	0.709033
	2022-05-04_09-55-59	**00:02.0**	**5 041 540.519180**	**15021.066987**	**11.553544**	**0.942200**	**0.686104**
	2	00:04.0	5 041 540.520483	15011.652994	11.553544	0.942199	0.686099
	5	00:07.0	5 041 540.519180	15011.652970	11.596602	0.942197	0.686089
	10	00:12.0	5 041 540.522043	15011.652978	11.780254	0.942200	0.686104
	-	00:02.0	5 129 051.003062	15021.066987	13.947483	0.941805	0.685692
	2022-05-04_10-12-39	**00:02.0**	**5 194 158.721426**	**14337.386317**	**10.485848**	**0.936306**	**0.628683**
	2	00:04.0	5 194 158.723186	14337.386317	10.567718	0.936306	0.628683
	5	00:07.0	5 194 158.722257	14337.386225	10.502510	0.936297	0.628654
	10	00:11.0	5 194 158.721426	14337.386261	10.485848	0.936296	0.628654
	-	00:02.0	5 216 496.720564	14262.070160	13.168194	0.935814	0.627987
	2022-05-04_10-28-09	**00:02.0**	**5 016 946.576872**	**14986.187506**	**10.403737**	**0.940635**	**0.662577**
	2	00:04.0	5 016 946.576872	14986.187504	10.403737	0.940632	0.662572
	5	00:06.0	5 016 946.577702	14986.187506	10.510283	0.940633	0.662567
	10	00:11.0	5 016 946.577638	14986.187493	10.607200	0.940635	0.662577
	-	00:02.0	5 026 369.396179	14951.307651	12.370109	0.940315	0.662064
	2022-05-04_10-45-34	**00:02.0**	**5 857 239.267817**	**12355.321863**	**10.698947**	**0.928369**	**0.615413**
	2	00:04.0	5 857 239.267817	12355.321847	10.851232	0.928369	0.615413

5	00:07.0	5 857 239.270379	12355.321863	10.698947	0.928365	0.615399
10	00:11.0	5 857 239.270431	12355.321850	10.820005	0.928366	0.615403
-	00:02.0	5 860 012.515040	12258.011617	14.257338	0.927874	0.613672
2022-05-04_11-00-32	**00:02.0**	**5 288 528.042683**	**14009.628347**	**10.424709**	**0.938824**	**0.677738**
2	00:04.0	5 288 528.050768	14009.628270	10.897722	0.938814	0.677697
5	00:07.0	5 288 528.042683	14009.628347	10.424709	0.938820	0.677716
10	00:11.0	5 288 528.044672	14009.628329	10.507318	0.938824	0.677738
-	00:02.0	5 295 772.538433	13985.102257	12.671692	0.938516	0.677385
2022-05-04_11-16-11	**00:02.0**	**5 140 543.068400**	**14528.963791**	**9.668918**	**0.936850**	**0.634861**
2	00:04.0	5 140 543.068400	14528.963765	9.676001	0.936850	0.634861
5	00:07.0	5 140 543.071501	14528.963791	9.668918	0.936845	0.634840
10	00:11.0	5 140 543.071145	14528.963779	9.691591	0.936844	0.634840
-	00:02.0	5 184 830.884806	14411.069239	12.278674	0.936186	0.633509
8	**00:02.0**	**3 844 251.083423**	**36470.049497**	**15.838654**	**0.945253**	**0.734193**
2022-05-04_08-37-43	**00:02.0**	**4 105 362.723070**	**33766.067327**	**17.684982**	**0.941313**	**0.721434**
2	00:05.0	4 105 362.723070	33766.067278	17.684982	0.941297	0.721361
5	00:09.0	4 105 362.723261	33766.066916	17.782715	0.941306	0.721398
10	00:15.0	4 105 362.723529	33766.067327	17.806058	0.941302	0.721388
-	00:02.0	4 105 362.743651	33766.067096	22.230566	0.941313	0.721434
2022-05-04_09-00-37	**00:02.0**	**4 642 874.751938**	**28442.188645**	**17.103341**	**0.937166**	**0.724344**
2	00:05.0	4 642 874.753524	28442.188519	17.334490	0.937159	0.724319
5	00:10.0	4 642 874.751938	28442.188645	17.234673	0.937160	0.724312
10	00:15.0	4 642 874.751977	28442.188432	17.103341	0.937166	0.724344
-	00:02.0	4 725 948.606274	27865.857337	20.529482	0.935031	0.721156
2022-05-04_09-20-38	**00:03.0**	**4 545 172.505300**	**29228.251534**	**18.554064**	**0.933998**	**0.674987**
2	00:06.0	4 545 172.506854	29228.251466	18.823716	0.933974	0.674899
5	00:09.0	4 545 172.507641	29228.251408	19.011965	0.933998	0.674987
10	00:16.0	4 545 172.505300	29228.251508	18.554064	0.933990	0.674964
-	00:03.0	4 545 172.529296	29228.251534	22.298304	0.933984	0.674929
2022-05-04_09-39-14	**00:02.0**	**6 240 127.100095**	**19082.029332**	**24.705985**	**0.909192**	**0.578679**
2	00:05.0	6 240 127.103906	19082.029307	25.159899	0.909179	0.578633
5	00:09.0	6 240 127.100095	19082.029332	24.705985	0.909182	0.578648
10	00:15.0	6 240 127.100759	19082.029314	24.791827	0.909192	0.578679
-	00:02.0	6 250 775.149600	19035.857935	30.581977	0.908475	0.578410
2022-05-04_09-55-59	**00:02.0**	**5 429 860.030681**	**23142.125855**	**23.764094**	**0.918791**	**0.619481**
2	00:06.0	5 429 860.031821	23142.125659	23.908315	0.918791	0.619481
5	00:10.0	5 429 860.032485	23142.125855	24.023200	0.918784	0.619446
10	00:15.0	5 429 860.030681	23142.125594	23.764094	0.918787	0.619452
-	00:02.0	5 468 296.570288	22944.076521	29.820653	0.917317	0.616658
2022-05-04_10-12-39	**00:02.0**	**3 857 171.466735**	**36470.049497**	**16.601285**	**0.942300**	**0.695944**
2	00:06.0	3 857 171.466735	36470.049363	16.602576	0.942287	0.695899
5	00:09.0	3 857 171.467103	36470.049147	16.601285	0.942285	0.695878
10	00:16.0	3 857 171.467514	36470.049497	16.638434	0.942300	0.695944
-	00:02.0	3 887 247.133208	35891.088704	20.883658	0.942076	0.695499
2022-05-04_10-28-09	**00:02.0**	**3 844 251.083423**	**36039.034506**	**16.390678**	**0.945253**	**0.734193**
2	00:06.0	3 844 251.083992	36039.034415	16.390678	0.945233	0.734124
5	00:09.0	3 844 251.083423	36039.034506	16.398735	0.945253	0.734193
10	00:15.0	3 844 251.083694	36039.034504	16.526707	0.945228	0.734102
-	00:02.0	3 844 251.100139	36039.034344	19.785717	0.945239	0.734160
2022-05-04_10-45-34	**00:02.0**	**4 944 669.918839**	**26210.273587**	**21.268840**	**0.927573**	**0.672771**
2	00:05.0	4 944 669.923793	26210.273516	22.025627	0.927573	0.672771
5	00:09.0	4 944 669.921395	26210.273587	21.575842	0.927568	0.672742
10	00:16.0	4 944 669.918839	26210.273501	21.268840	0.927556	0.672698
-	00:02.0	4 944 669.949933	26210.273377	27.403875	0.927556	0.672694
2022-05-04_11-00-32	**00:02.0**	**4 557 793.846184**	**29594.624738**	**15.838654**	**0.939357**	**0.716815**
2	00:06.0	4 557 793.847977	29594.624738	15.979823	0.939346	0.716782
5	00:09.0	4 557 793.846184	29594.624467	15.838654	0.939357	0.716804
10	00:15.0	4 557 793.847206	29594.624693	15.952476	0.939349	0.716780
-	00:02.0	4 557 793.868676	29594.624558	20.142119	0.939357	0.716815
2022-05-04_11-16-11	**00:02.0**	**4 521 547.353100**	**29423.426421**	**19.497147**	**0.932976**	**0.669810**
2	00:06.0	4 521 547.355264	29423.426124	19.883279	0.932974	0.669799
5	00:10.0	4 521 547.354064	29423.426261	19.598499	0.932976	0.669810
10	00:15.0	4 521 547.353100	29423.426421	19.497147	0.932966	0.669763
-	00:02.0	4 521 547.383556	29423.426218	24.990183	0.932966	0.669763
9	**00:03.0**	**1 495 531.057293**	**97489.798735**	**12.380740**	**0.953718**	**0.824819**
2022-05-04_08-37-43	**00:03.0**	**1 495 531.057293**	**97489.798735**	**14.674612**	**0.953718**	**0.824819**
2	00:08.0	1 495 531.059143	97489.798735	15.047848	0.953663	0.824666
5	00:15.0	1 495 531.057983	97489.798312	14.852498	0.953673	0.824699
10	00:24.0	1 495 531.057293	97489.798073	14.674612	0.953681	0.824727

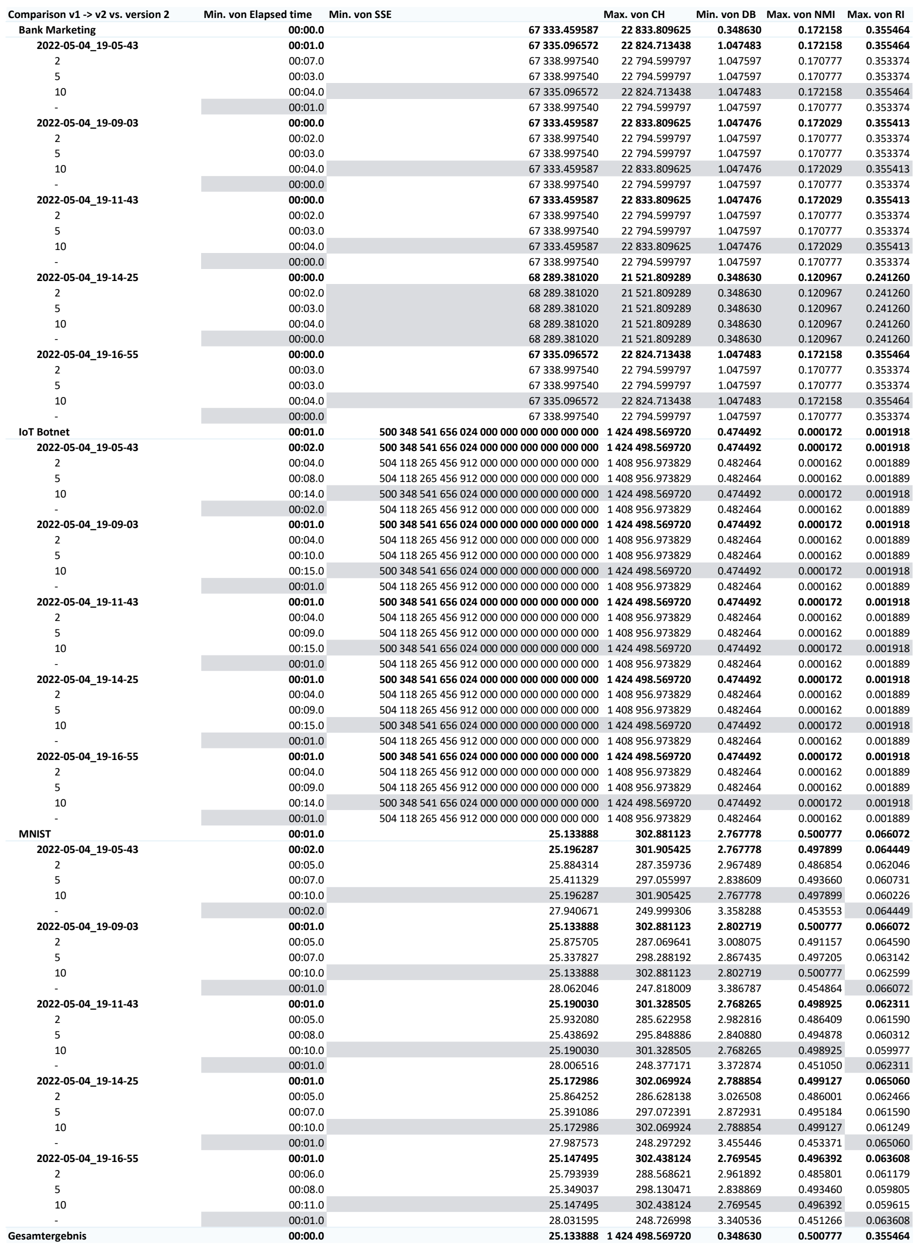
-	00:03.0	1 495 531.082518	97489.797016	18.856730	0.953718	0.824819
2022-05-04_09-00-37	**00:04.0**	**2 767 083.441563**	**45101.358775**	**20.638266**	**0.910625**	**0.654871**
2	00:08.0	2 767 083.443294	45101.358762	20.899534	0.909715	0.653730
5	00:15.0	2 767 083.441563	45101.358482	20.638266	0.909740	0.653835
10	00:24.0	2 767 083.441770	45101.358775	20.865828	0.909721	0.653780
-	00:04.0	2 767 094.832276	45101.357752	26.838593	0.910625	0.654871
2022-05-04_09-20-38	**00:03.0**	**2 852 046.635843**	**43265.790747**	**22.922711**	**0.908925**	**0.661624**
2	00:08.0	2 852 046.635843	43265.790747	22.981274	0.908925	0.661624
5	00:15.0	2 852 046.635936	43265.790320	22.922711	0.908923	0.661603
10	00:24.0	2 852 046.636497	43265.790415	23.017435	0.908916	0.661592
-	00:03.0	2 852 046.686343	43265.789601	30.870697	0.908924	0.661590
2022-05-04_09-39-14	**00:03.0**	**1 509 878.160568**	**96406.565232**	**12.380740**	**0.952416**	**0.806745**
2	00:08.0	1 509 878.161147	96406.564979	12.498334	0.952396	0.806693
5	00:16.0	1 509 878.160568	96406.565232	12.380740	0.952403	0.806717
10	00:24.0	1 509 878.161689	96406.565142	12.607579	0.952410	0.806739
-	00:03.0	1 509 878.187979	96406.562940	16.621131	0.952416	0.806745
2022-05-04_09-55-59	**00:03.0**	**2 863 288.681164**	**43031.103047**	**24.623941**	**0.908242**	**0.657728**
2	00:09.0	2 863 288.684588	43031.101956	25.044822	0.908197	0.657574
5	00:17.0	2 863 288.681251	43031.102511	24.681500	0.908203	0.657597
10	00:25.0	2 863 288.681164	43031.103047	24.623941	0.908199	0.657580
-	00:03.0	2 863 288.737221	43031.100734	33.998820	0.908242	0.657728
2022-05-04_10-12-39	**00:03.0**	**2 864 771.079086**	**44433.690199**	**22.257458**	**0.906793**	**0.633611**
2	00:08.0	2 864 771.083708	44433.689988	22.733482	0.906793	0.633611
5	00:15.0	2 864 771.080682	44433.690199	22.457323	0.906770	0.633571
10	00:25.0	2 864 771.079086	44433.689809	22.257458	0.906772	0.633581
-	00:03.0	2 903 287.331343	43000.260501	30.755637	0.906285	0.632922
2022-05-04_10-28-09	**00:03.0**	**2 827 659.749638**	**43781.391728**	**22.612678**	**0.907717**	**0.646635**
2	00:08.0	2 827 659.753465	43781.391728	23.067466	0.907717	0.646635
5	00:15.0	2 827 659.750124	43781.390581	22.718715	0.907709	0.646616
10	00:23.0	2 827 659.749638	43781.391642	22.612678	0.907717	0.646608
-	00:03.0	2 834 293.461535	43641.561599	30.781416	0.906872	0.646098
2022-05-04_10-45-34	**00:03.0**	**2 810 959.375263**	**44139.618600**	**23.697906**	**0.907186**	**0.645196**
2	00:08.0	2 810 959.376824	44139.618086	23.869010	0.907158	0.645112
5	00:15.0	2 810 959.376955	44139.618600	23.953762	0.907177	0.645173
10	00:24.0	2 810 959.375263	44139.618516	23.697906	0.907173	0.645166
-	00:03.0	2 810 959.420955	44139.617895	31.053075	0.907186	0.645196
2022-05-04_11-00-32	**00:04.0**	**2 169 607.662106**	**62069.548877**	**21.103061**	**0.931650**	**0.741456**
2	00:08.0	2 169 607.663338	62069.548877	21.346179	0.931585	0.741232
5	00:15.0	2 169 607.662106	62069.548385	21.103061	0.931626	0.741387
10	00:24.0	2 169 607.664147	62069.548010	21.340898	0.931628	0.741382
-	00:04.0	2 169 607.703149	62069.547010	28.458820	0.931650	0.741456
2022-05-04_11-16-11	**00:03.0**	**2 146 283.607461**	**65021.111720**	**15.632418**	**0.937348**	**0.757801**
2	00:09.0	2 146 283.611114	65021.111322	15.928945	0.936380	0.756504
5	00:15.0	2 146 283.607461	65021.111720	15.632418	0.936379	0.756482
10	00:24.0	2 146 283.609358	65021.111635	15.813568	0.936363	0.756444
-	00:03.0	2 158 664.402191	62923.533386	21.075902	0.937348	0.757801
10	**00:02.0**	**157 699.374092**	**236010.026353**	**4.024494**	**0.950067**	**0.796300**
2022-05-04_08-37-43	**00:02.0**	**157 699.374092**	**236010.026353**	**4.024494**	**0.950067**	**0.796300**
2	00:05.0	157 699.376943	236010.026353	4.054371	0.950067	0.796300
5	00:09.0	157 699.374092	236010.024895	4.024494	0.950064	0.796290
10	00:15.0	157 699.374092	236010.024895	4.024494	0.950064	0.796290
-	00:02.0	157 699.443102	236009.909669	5.050677	0.950046	0.796221
2022-05-04_09-00-37	**00:02.0**	**343 987.735617**	**90309.227867**	**6.372139**	**0.884896**	**0.530584**
2	00:05.0	343 987.735617	90309.227867	6.372139	0.884862	0.530509
5	00:09.0	343 987.738066	90309.226686	6.399286	0.884855	0.530487
10	00:16.0	343 987.737287	90309.226907	6.386921	0.884856	0.530491
-	00:02.0	343 987.832305	90309.189773	7.751646	0.884896	0.530584
2022-05-04_09-20-38	**00:02.0**	**325 734.742041**	**97220.931037**	**6.676479**	**0.893905**	**0.603490**
2	00:05.0	325 734.746370	97220.926113	6.745471	0.893795	0.603105
5	00:09.0	325 734.743399	97220.929857	6.706474	0.893778	0.603052
10	00:16.0	325 734.742041	97220.931037	6.676479	0.893786	0.603051
-	00:02.0	325 734.833179	97220.885574	8.104898	0.893905	0.603490
2022-05-04_09-39-14	**00:02.0**	**209 885.435586**	**169115.379847**	**4.457223**	**0.937278**	**0.729864**
2	00:05.0	209 885.435586	169115.378743	4.457223	0.937268	0.729842
5	00:09.0	209 885.437062	169115.379847	4.471377	0.937269	0.729849
10	00:15.0	209 885.436487	169115.378978	4.460062	0.937270	0.729850
-	00:02.0	209 885.499936	169115.315199	5.292776	0.937278	0.729864
2022-05-04_09-55-59	**00:02.0**	**279 642.930059**	**118689.472838**	**5.028249**	**0.905455**	**0.569548**
2	00:05.0	279 642.932524	118689.472400	5.057238	0.905425	0.569441

5	00:10.0	279 642.930059	118689.472838	5.028249	0.905432	0.569460
10	00:16.0	279 642.931020	118689.472471	5.032016	0.905428	0.569449
-	00:02.0	279 642.999489	118689.441656	6.091427	0.905455	0.569548
2022-05-04_10-12-39	**00:02.0**	**299 000.131606**	**108867.168372**	**4.968464**	**0.907973**	**0.581788**
2	00:05.0	299 000.131937	108867.168358	4.970189	0.907973	0.581788
5	00:10.0	299 000.131718	108867.167570	4.969909	0.907967	0.581774
10	00:16.0	299 000.131606	108867.168372	4.968464	0.907968	0.581777
-	00:02.0	308 914.989179	104534.163115	6.074496	0.904099	0.577451
2022-05-04_10-28-09	**00:02.0**	**290 507.254065**	**113015.475075**	**6.237891**	**0.910966**	**0.675083**
2	00:05.0	290 507.254065	113015.475075	6.237891	0.910966	0.675045
5	00:09.0	290 507.254453	113015.474143	6.260377	0.910960	0.675026
10	00:16.0	290 507.255535	113015.473375	6.271152	0.910956	0.675011
-	00:02.0	290 507.346300	113015.427952	7.739872	0.910966	0.675083
2022-05-04_10-45-34	**00:02.0**	**268 861.354051**	**124773.717466**	**5.315204**	**0.923777**	**0.720970**
2	00:05.0	268 861.356757	124773.717466	5.338860	0.923717	0.720782
5	00:09.0	268 861.354051	124773.715603	5.315204	0.923667	0.720593
10	00:16.0	268 861.355622	124773.717193	5.325940	0.923660	0.720575
-	00:02.0	268 861.439335	124773.677045	6.567466	0.923777	0.720970
2022-05-04_11-00-32	**00:02.0**	**361 457.615309**	**84348.122049**	**6.038805**	**0.893812**	**0.587295**
2	00:05.0	361 457.617737	84348.121489	6.073706	0.893724	0.587012
5	00:09.0	361 457.615443	84348.122049	6.038805	0.893729	0.587020
10	00:16.0	361 457.615309	84348.121799	6.039351	0.893729	0.587021
-	00:02.0	361 457.720620	84348.085356	7.810757	0.893812	0.587295
2022-05-04_11-16-11	**00:02.0**	**233 957.144370**	**148316.651434**	**5.006457**	**0.929235**	**0.722012**
2	00:05.0	233 957.145064	148316.646809	5.012420	0.929128	0.721672
5	00:09.0	233 957.145367	148316.649282	5.017514	0.929128	0.721677
10	00:16.0	233 957.144370	148316.651434	5.006457	0.929139	0.721715
-	00:02.0	233 957.222360	148316.590686	6.041127	0.929235	0.722012
11	**00:02.0**	**4 167 574.959160**	**38157.865712**	**8.234709**	**0.953533**	**0.744248**
2022-05-04_08-37-43	**00:02.0**	**5 539 975.779089**	**27070.867706**	**9.289851**	**0.931416**	**0.565343**
2	00:05.0	5 539 975.780041	26919.952201	9.338467	0.931416	0.565033
5	00:10.0	5 539 975.779146	26919.952126	9.308564	0.931416	0.565034
10	00:19.0	5 539 975.779089	26919.952037	9.289851	0.931416	0.565030
-	00:02.0	5 567 184.343954	27070.867706	11.460414	0.931281	0.565343
2022-05-04_09-00-37	**00:02.0**	**5 458 282.017992**	**27664.077464**	**9.604580**	**0.939115**	**0.652884**
2	00:06.0	5 458 282.020546	27664.077464	9.738025	0.939110	0.652855
5	00:10.0	5 458 282.017992	27664.077407	9.662237	0.939109	0.652853
10	00:20.0	5 458 282.018493	27664.077365	9.604580	0.939115	0.652884
-	00:02.0	5 588 163.077029	26966.844751	12.182741	0.937172	0.647335
2022-05-04_09-20-38	**00:02.0**	**5 621 756.642980**	**26202.158441**	**11.313551**	**0.934056**	**0.634690**
2	00:05.0	5 621 756.646372	26202.158441	11.458661	0.934056	0.634690
5	00:11.0	5 621 756.642980	26202.158229	11.313551	0.934054	0.634679
10	00:19.0	5 621 756.643609	26202.158186	11.362346	0.934054	0.634679
-	00:02.0	5 665 727.118951	25965.229971	13.353248	0.933091	0.632833
2022-05-04_09-39-14	**00:02.0**	**4 699 556.194898**	**32925.857676**	**9.565811**	**0.945942**	**0.695595**
2	00:05.0	4 699 556.196757	32925.857571	9.565811	0.945942	0.695595
5	00:10.0	4 699 556.194898	32925.857676	10.034191	0.945935	0.695554
10	00:19.0	4 699 556.195531	32925.857561	10.197260	0.945935	0.695555
-	00:02.0	4 717 713.460348	32786.415372	11.434712	0.944757	0.691221
2022-05-04_09-55-59	**00:02.0**	**5 939 950.553616**	**24366.686701**	**11.847252**	**0.928523**	**0.607027**
2	00:06.0	5 939 950.564302	24366.686701	12.360909	0.928523	0.607027
5	00:11.0	5 939 950.554633	24366.686587	11.888081	0.928516	0.606988
10	00:19.0	5 939 950.553616	24366.686609	11.847252	0.928514	0.606982
-	00:02.0	5 989 210.227972	24119.831632	15.473164	0.927289	0.606786
2022-05-04_10-12-39	**00:02.0**	**6 787 485.550365**	**20809.016005**	**13.325443**	**0.919522**	**0.558932**
2	00:06.0	6 787 485.551530	20809.015953	13.350980	0.919519	0.558910
5	00:10.0	6 787 485.550467	20809.015904	13.325443	0.919519	0.558923
10	00:19.0	6 787 485.550365	20809.016005	13.326198	0.919522	0.558932
-	00:02.0	6 840 905.102444	20512.328439	14.942172	0.918238	0.557643
2022-05-04_10-28-09	**00:02.0**	**4 523 993.279862**	**34516.483773**	**9.350791**	**0.948489**	**0.734475**
2	00:06.0	4 523 993.283964	34516.483558	9.558599	0.948480	0.734430
5	00:10.0	4 523 993.279925	34516.483715	9.481851	0.948484	0.734449
10	00:19.0	4 523 993.279862	34516.483773	9.350791	0.948489	0.734475
-	00:02.0	4 523 993.331955	34516.483522	11.272946	0.948471	0.734378
2022-05-04_10-45-34	**00:02.0**	**4 167 574.959160**	**38157.865712**	**8.394200**	**0.953533**	**0.744248**
2	00:06.0	4 167 574.960409	38157.865530	8.394200	0.953532	0.744239
5	00:11.0	4 167 574.959160	38157.865712	8.588586	0.953527	0.744218
10	00:19.0	4 167 574.959831	38157.865570	8.548614	0.953533	0.744248
-	00:02.0	4 173 449.118254	38157.865379	9.774933	0.953527	0.744225

2022-05-04_11-00-32	**00:02.0**	**5 660 250.329460**	**26175.278938**	**8.234709**	**0.934924**	**0.643807**
2	00:06.0	5 660 250.329460	26175.278910	8.234709	0.934924	0.643807
5	00:10.0	5 660 250.329916	26175.278919	8.269942	0.934916	0.643783
10	00:18.0	5 660 250.330811	26175.278938	8.274487	0.934920	0.643804
-	00:02.0	5 660 306.014852	26175.278528	10.486534	0.934890	0.643680
2022-05-04_11-16-11	**00:02.0**	**4 468 911.190421**	**35319.424775**	**9.027130**	**0.947403**	**0.677430**
2	00:06.0	4 468 911.191342	35319.424660	9.367083	0.947396	0.677402
5	00:10.0	4 468 911.190792	35319.424775	9.243363	0.947395	0.677398
10	00:19.0	4 468 911.190421	35319.424744	9.027130	0.947403	0.677430
-	00:02.0	4 479 310.670878	35220.637162	10.571383	0.947059	0.676893
12	**00:03.0**	**3 201 373.595013**	**89980.187186**	**12.908943**	**0.954936**	**0.781942**
2022-05-04_08-37-43	**00:03.0**	**3 201 373.595013**	**89980.187186**	**12.908943**	**0.954936**	**0.781942**
2	00:09.0	3 201 373.595612	89980.187186	13.115068	0.954936	0.781942
5	00:15.0	3 201 373.595013	89980.186507	12.908943	0.954921	0.781870
10	00:28.0	3 201 373.595331	89980.185928	12.935451	0.954934	0.781919
-	00:03.0	3 201 373.617720	89980.186270	16.364888	0.954935	0.781936
2022-05-04_09-00-37	**00:04.0**	**5 105 649.035981**	**50357.961488**	**17.621603**	**0.918333**	**0.559950**
2	00:08.0	5 105 649.038306	50357.961488	17.756110	0.918317	0.559887
5	00:15.0	5 105 649.035981	50357.960977	17.621603	0.918317	0.559882
10	00:27.0	5 105 649.036806	50357.960639	17.727668	0.918318	0.559896
-	00:04.0	5 105 649.076119	50357.960203	23.549862	0.918333	0.559950
2022-05-04_09-20-38	**00:03.0**	**3 777 365.064428**	**73781.240653**	**14.647059**	**0.948412**	**0.770988**
2	00:08.0	3 777 365.068712	73781.240229	15.244865	0.948396	0.770921
5	00:16.0	3 777 365.064830	73781.240291	14.684459	0.948391	0.770903
10	00:27.0	3 777 365.064428	73781.240653	14.647059	0.948412	0.770988
-	00:03.0	3 801 246.831076	73233.649025	21.459105	0.947173	0.768512
2022-05-04_09-39-14	**00:03.0**	**4 248 624.684131**	**63794.573178**	**14.939142**	**0.934979**	**0.652720**
2	00:08.0	4 248 624.686747	63794.573178	15.069611	0.934944	0.652589
5	00:15.0	4 248 624.684575	63794.572723	14.972712	0.934974	0.652698
10	00:27.0	4 248 624.684131	63794.573099	14.939142	0.934979	0.652720
-	00:03.0	4 275 390.226114	63302.222948	19.739260	0.933632	0.651081
2022-05-04_09-55-59	**00:03.0**	**4 448 988.727272**	**60189.572012**	**18.442194**	**0.933570**	**0.678268**
2	00:09.0	4 448 988.728006	60189.572012	18.463059	0.933561	0.678257
5	00:16.0	4 448 988.727272	60189.571587	18.465502	0.933546	0.678181
10	00:28.0	4 448 988.727342	60189.571928	18.442194	0.933568	0.678268
-	00:03.0	4 448 988.761780	60189.571575	24.284252	0.933570	0.678261
2022-05-04_10-12-39	**00:03.0**	**5 920 920.787466**	**41186.075827**	**21.501713**	**0.916772**	**0.644742**
2	00:08.0	5 920 920.791374	41186.075264	21.920466	0.916772	0.644732
5	00:15.0	5 920 920.791044	41186.075744	21.649998	0.916772	0.644736
10	00:27.0	5 920 920.787466	41186.075827	21.501713	0.916769	0.644727
-	00:03.0	5 920 920.835836	41186.075358	28.422880	0.916770	0.644742
2022-05-04_10-28-09	**00:03.0**	**4 937 835.086028**	**52621.775464**	**18.978293**	**0.929966**	**0.687385**
2	00:08.0	4 937 835.087050	52621.775252	19.045558	0.929962	0.687385
5	00:15.0	4 937 835.088895	52621.775464	19.139857	0.929956	0.687357
10	00:27.0	4 937 835.086028	52621.775021	18.978293	0.929961	0.687384
-	00:03.0	4 937 835.122945	52621.775007	24.926914	0.929966	0.687372
2022-05-04_10-45-34	**00:03.0**	**4 463 158.214129**	**59946.912001**	**15.414404**	**0.941305**	**0.742057**
2	00:08.0	4 463 158.214586	59946.911715	15.575150	0.941292	0.742006
5	00:15.0	4 463 158.214129	59946.912001	15.414404	0.941304	0.742057
10	00:28.0	4 463 158.215035	59946.911697	15.520168	0.941298	0.742029
-	00:03.0	4 463 158.248658	59946.910945	20.534779	0.941305	0.742050
2022-05-04_11-00-32	**00:03.0**	**4 156 390.110687**	**66628.591654**	**15.584688**	**0.944068**	**0.747885**
2	00:08.0	4 156 390.116238	66628.591319	16.938191	0.944068	0.747885
5	00:15.0	4 156 390.112892	66628.590742	15.957813	0.944063	0.747867
10	00:26.0	4 156 390.110687	66628.591654	15.584688	0.944062	0.747850
-	00:03.0	4 207 149.532535	65570.929908	23.140438	0.943657	0.746955
2022-05-04_11-16-11	**00:03.0**	**4 138 457.951480**	**66988.851113**	**15.683230**	**0.944749**	**0.746597**
2	00:08.0	4 138 457.952011	66988.850882	15.683230	0.944732	0.746552
5	00:15.0	4 138 457.951480	66988.851029	16.073736	0.944735	0.746568
10	00:28.0	4 138 457.953349	66988.851113	16.312886	0.944727	0.746528
-	00:03.0	4 138 457.980049	66988.850897	20.729667	0.944749	0.746597
Gesamtergebnis	**00:01.0**	**670.753411**	**236010.026353**	**1.346372**	**0.957762**	**0.858327**

A.6.4. Part 3: Verify Results With Real Data Sets

The results of the third part of the experiment are summarized in the included document 2.

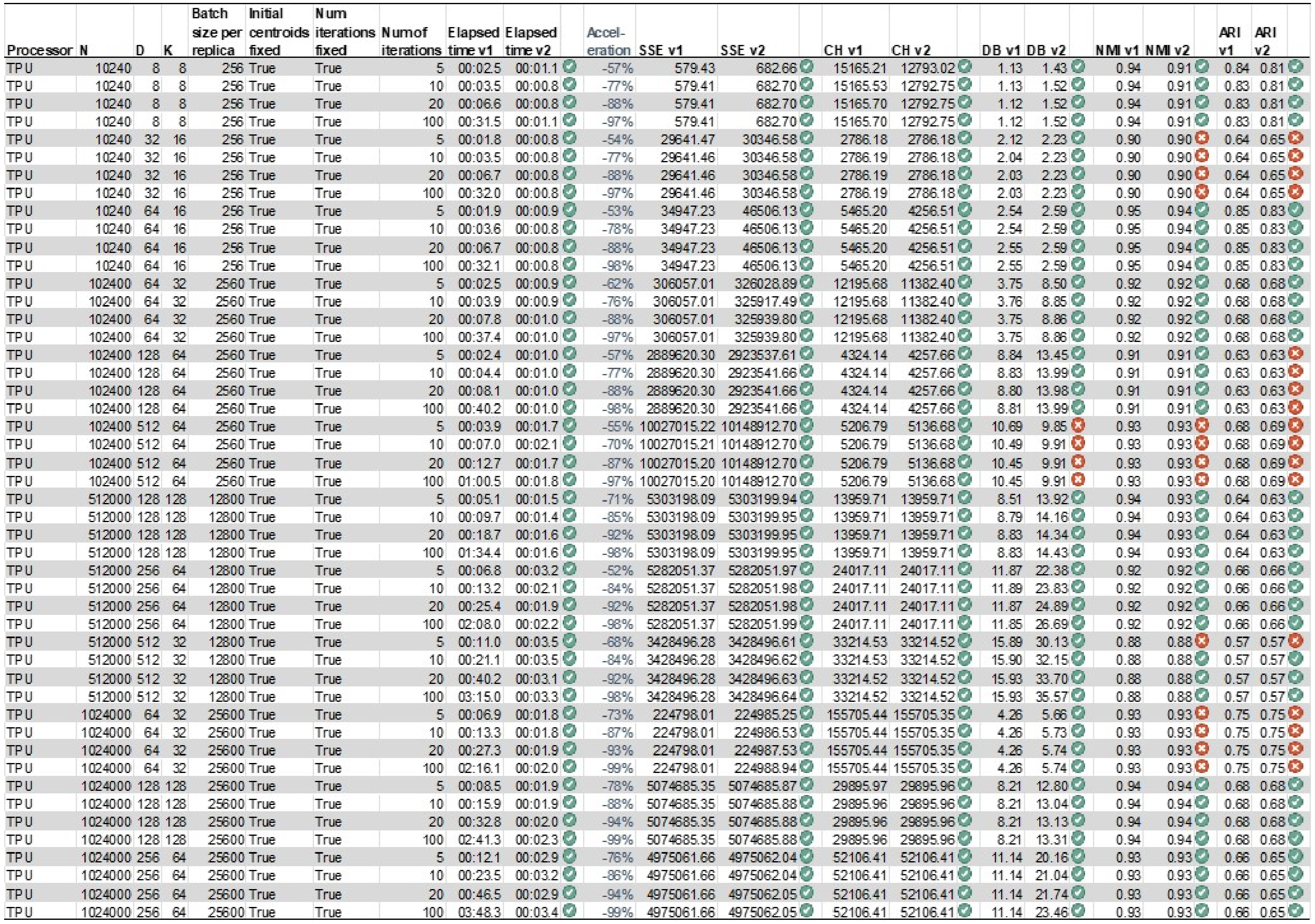


Comparison v1 -> v2 vs. version 2	Min. von Elapsed time	Min. von SSE	Max. von CH	Min. von DB	Max. von NMI	Max. von RI
Bank Marketing	**00:00.0**	**67 333.459587**	**22 833.809625**	**0.348630**	**0.172158**	**0.355464**
2022-05-04_19-05-43	**00:01.0**	**67 335.096572**	**22 824.713438**	**1.047483**	**0.172158**	**0.355464**
2	00:07.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
5	00:03.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
10	00:04.0	67 335.096572	22 824.713438	1.047483	0.172158	0.355464
-	00:01.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
2022-05-04_19-09-03	**00:00.0**	**67 333.459587**	**22 833.809625**	**1.047476**	**0.172029**	**0.355413**
2	00:02.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
5	00:03.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
10	00:04.0	67 333.459587	22 833.809625	1.047476	0.172029	0.355413
-	00:00.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
2022-05-04_19-11-43	**00:00.0**	**67 333.459587**	**22 833.809625**	**1.047476**	**0.172029**	**0.355413**
2	00:02.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
5	00:03.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
10	00:04.0	67 333.459587	22 833.809625	1.047476	0.172029	0.355413
-	00:00.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
2022-05-04_19-14-25	**00:00.0**	**68 289.381020**	**21 521.809289**	**0.348630**	**0.120967**	**0.241260**
2	00:02.0	68 289.381020	21 521.809289	0.348630	0.120967	0.241260
5	00:03.0	68 289.381020	21 521.809289	0.348630	0.120967	0.241260
10	00:04.0	68 289.381020	21 521.809289	0.348630	0.120967	0.241260
-	00:00.0	68 289.381020	21 521.809289	0.348630	0.120967	0.241260
2022-05-04_19-16-55	**00:00.0**	**67 335.096572**	**22 824.713438**	**1.047483**	**0.172158**	**0.355464**
2	00:03.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
5	00:03.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
10	00:04.0	67 335.096572	22 824.713438	1.047483	0.172158	0.355464
-	00:00.0	67 338.997540	22 794.599797	1.047597	0.170777	0.353374
IoT Botnet	**00:01.0**	**500 348 541 656 024 000 000 000 000 000 000**	**1 424 498.569720**	**0.474492**	**0.000172**	**0.001918**
2022-05-04_19-05-43	**00:02.0**	**500 348 541 656 024 000 000 000 000 000 000**	**1 424 498.569720**	**0.474492**	**0.000172**	**0.001918**
2	00:04.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
5	00:08.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
10	00:14.0	500 348 541 656 024 000 000 000 000 000 000	1 424 498.569720	0.474492	0.000172	0.001918
-	00:02.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
2022-05-04_19-09-03	**00:01.0**	**500 348 541 656 024 000 000 000 000 000 000**	**1 424 498.569720**	**0.474492**	**0.000172**	**0.001918**
2	00:04.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
5	00:10.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
10	00:15.0	500 348 541 656 024 000 000 000 000 000 000	1 424 498.569720	0.474492	0.000172	0.001918
-	00:01.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
2022-05-04_19-11-43	**00:01.0**	**500 348 541 656 024 000 000 000 000 000 000**	**1 424 498.569720**	**0.474492**	**0.000172**	**0.001918**
2	00:04.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
5	00:09.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
10	00:15.0	500 348 541 656 024 000 000 000 000 000 000	1 424 498.569720	0.474492	0.000172	0.001918
-	00:01.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
2022-05-04_19-14-25	**00:01.0**	**500 348 541 656 024 000 000 000 000 000 000**	**1 424 498.569720**	**0.474492**	**0.000172**	**0.001918**
2	00:04.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
5	00:09.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
10	00:15.0	500 348 541 656 024 000 000 000 000 000 000	1 424 498.569720	0.474492	0.000172	0.001918
-	00:01.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
2022-05-04_19-16-55	**00:01.0**	**500 348 541 656 024 000 000 000 000 000 000**	**1 424 498.569720**	**0.474492**	**0.000172**	**0.001918**
2	00:04.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
5	00:09.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
10	00:14.0	500 348 541 656 024 000 000 000 000 000 000	1 424 498.569720	0.474492	0.000172	0.001918
-	00:01.0	504 118 265 456 912 000 000 000 000 000 000	1 408 956.973829	0.482464	0.000162	0.001889
MNIST	**00:01.0**	**25.133888**	**302.881123**	**2.767778**	**0.500777**	**0.066072**
2022-05-04_19-05-43	**00:02.0**	**25.196287**	**301.905425**	**2.767778**	**0.497899**	**0.064449**
2	00:05.0	25.884314	287.359736	2.967489	0.486854	0.062046
5	00:07.0	25.411329	297.055997	2.838609	0.493660	0.060731
10	00:10.0	25.196287	301.905425	2.767778	0.497899	0.060226
-	00:02.0	27.940671	249.999306	3.358288	0.453553	0.064449
2022-05-04_19-09-03	**00:01.0**	**25.133888**	**302.881123**	**2.802719**	**0.500777**	**0.066072**
2	00:05.0	25.875705	287.069641	3.008075	0.491157	0.064590
5	00:07.0	25.337827	298.288192	2.867435	0.497205	0.063142
10	00:10.0	25.133888	302.881123	2.802719	0.500777	0.062599
-	00:01.0	28.062046	247.818009	3.386787	0.454864	0.066072
2022-05-04_19-11-43	**00:01.0**	**25.190030**	**301.328505**	**2.768265**	**0.498925**	**0.062311**
2	00:05.0	25.932080	285.622958	2.982816	0.486409	0.061590
5	00:08.0	25.438692	295.848886	2.840880	0.494878	0.060312
10	00:10.0	25.190030	301.328505	2.768265	0.498925	0.059977
-	00:01.0	28.006516	248.377171	3.372874	0.451050	0.062311
2022-05-04_19-14-25	**00:01.0**	**25.172986**	**302.069924**	**2.788854**	**0.499127**	**0.065060**
2	00:05.0	25.864252	286.628138	3.026508	0.486001	0.062466
5	00:07.0	25.391086	297.072391	2.872931	0.495184	0.061590
10	00:10.0	25.172986	302.069924	2.788854	0.499127	0.061249
-	00:01.0	27.987573	248.297292	3.455446	0.453371	0.065060
2022-05-04_19-16-55	**00:01.0**	**25.147495**	**302.438124**	**2.769545**	**0.496392**	**0.063608**
2	00:06.0	25.793939	288.568621	2.961892	0.485801	0.061179
5	00:08.0	25.349037	298.130471	2.838869	0.493460	0.059805
10	00:11.0	25.147495	302.438124	2.769545	0.496392	0.059615
-	00:01.0	28.031595	248.726998	3.340536	0.451266	0.063608
Gesamtergebnis	**00:00.0**	**25.133888**	**1 424 498.569720**	**0.348630**	**0.500777**	**0.355464**

Figure A.15 shows the results in more detail, comparing the results to the benchmark result per initial centroid per data set.

A.7. Additional Data Experiment 5

Figure A.16 shows the runtime comparison of Experiment 5 in detail. Figure A.17 shows the clustering quality comparison of Experiment 5 in detail.



Processor	N	D	K	Batch size per replica	Initial centroids fixed	Num iterations fixed	Numof iterations	Elapsed time v1	Elapsed time v2		Accel-eration	SSE v1	SSE v2		CH v1	CH v2		DB v1	DB v2		NMI v1	NMI v2		ARI v1	ARI v2	
TPU	10240	8	8	256	True	True	5	00:02.5	00:01.1	✓	-57%	579.43	682.66	✓	15165.21	12793.02	✓	1.13	1.43	✓	0.94	0.91	✓	0.84	0.81	✓
TPU	10240	8	8	256	True	True	10	00:03.5	00:00.8	✓	-77%	579.41	682.70	✓	15165.53	12792.75	✓	1.13	1.52	✓	0.94	0.91	✓	0.83	0.81	✓
TPU	10240	8	8	256	True	True	20	00:06.6	00:00.8	✓	-88%	579.41	682.70	✓	15165.70	12792.75	✓	1.12	1.52	✓	0.94	0.91	✓	0.83	0.81	✓
TPU	10240	8	8	256	True	True	100	00:31.5	00:01.1	✓	-97%	579.41	682.70	✓	15165.70	12792.75	✓	1.12	1.52	✓	0.94	0.91	✓	0.83	0.81	✓
TPU	10240	32	16	256	True	True	5	00:01.8	00:00.8	✓	-54%	29641.47	30346.58	✓	2786.18	2786.18	✓	2.12	2.23	✓	0.90	0.90	✗	0.64	0.65	✗
TPU	10240	32	16	256	True	True	10	00:03.5	00:00.8	✓	-77%	29641.46	30346.58	✓	2786.19	2786.18	✓	2.04	2.23	✓	0.90	0.90	✗	0.64	0.65	✗
TPU	10240	32	16	256	True	True	20	00:06.7	00:00.8	✓	-88%	29641.46	30346.58	✓	2786.19	2786.18	✓	2.03	2.23	✓	0.90	0.90	✗	0.64	0.65	✗
TPU	10240	32	16	256	True	True	100	00:32.0	00:00.8	✓	-97%	29641.46	30346.58	✓	2786.19	2786.18	✓	2.03	2.23	✓	0.90	0.90	✗	0.64	0.65	✗
TPU	10240	64	16	256	True	True	5	00:01.9	00:00.9	✓	-53%	34947.23	46506.13	✓	5465.20	4256.51	✓	2.54	2.59	✓	0.95	0.94	✓	0.85	0.83	✓
TPU	10240	64	16	256	True	True	10	00:03.6	00:00.8	✓	-78%	34947.23	46506.13	✓	5465.20	4256.51	✓	2.54	2.59	✓	0.95	0.94	✓	0.85	0.83	✓
TPU	10240	64	16	256	True	True	20	00:06.7	00:00.8	✓	-88%	34947.23	46506.13	✓	5465.20	4256.51	✓	2.55	2.59	✓	0.95	0.94	✓	0.85	0.83	✓
TPU	10240	64	16	256	True	True	100	00:32.1	00:00.8	✓	-98%	34947.23	46506.13	✓	5465.20	4256.51	✓	2.55	2.59	✓	0.95	0.94	✓	0.85	0.83	✓
TPU	102400	64	32	2560	True	True	5	00:02.5	00:00.9	✓	-62%	306057.01	326028.89	✓	12195.68	11382.40	✓	3.75	8.50	✓	0.92	0.92	✓	0.68	0.68	✓
TPU	102400	64	32	2560	True	True	10	00:03.9	00:00.9	✓	-76%	306057.01	325917.49	✓	12195.68	11382.40	✓	3.76	8.85	✓	0.92	0.92	✓	0.68	0.68	✓
TPU	102400	64	32	2560	True	True	20	00:07.8	00:01.0	✓	-88%	306057.01	325939.80	✓	12195.68	11382.40	✓	3.75	8.86	✓	0.92	0.92	✓	0.68	0.68	✓
TPU	102400	64	32	2560	True	True	100	00:37.4	00:01.0	✓	-97%	306057.01	325939.80	✓	12195.68	11382.40	✓	3.75	8.86	✓	0.92	0.92	✓	0.68	0.68	✓
TPU	102400	128	64	2560	True	True	5	00:02.4	00:01.0	✓	-57%	2889620.30	2923537.61	✓	4324.14	4257.66	✓	8.84	13.45	✓	0.91	0.91	✓	0.63	0.63	✗
TPU	102400	128	64	2560	True	True	10	00:04.4	00:01.0	✓	-77%	2889620.30	2923541.66	✓	4324.14	4257.66	✓	8.83	13.99	✓	0.91	0.91	✓	0.63	0.63	✗
TPU	102400	128	64	2560	True	True	20	00:08.1	00:01.0	✓	-88%	2889620.30	2923541.66	✓	4324.14	4257.66	✓	8.80	13.98	✓	0.91	0.91	✓	0.63	0.63	✗
TPU	102400	128	64	2560	True	True	100	00:40.2	00:01.0	✓	-98%	2889620.30	2923541.66	✓	4324.14	4257.66	✓	8.81	13.99	✓	0.91	0.91	✓	0.63	0.63	✗
TPU	102400	512	64	2560	True	True	5	00:03.9	00:01.7	✓	-55%	10027015.22	10148912.70	✓	5206.79	5136.68	✓	10.69	9.85	✗	0.93	0.93	✗	0.68	0.69	✗
TPU	102400	512	64	2560	True	True	10	00:07.0	00:02.1	✓	-70%	10027015.21	10148912.70	✓	5206.79	5136.68	✓	10.49	9.91	✗	0.93	0.93	✗	0.68	0.69	✗
TPU	102400	512	64	2560	True	True	20	00:12.7	00:01.7	✓	-87%	10027015.20	10148912.70	✓	5206.79	5136.68	✓	10.45	9.91	✗	0.93	0.93	✗	0.68	0.69	✗
TPU	102400	512	64	2560	True	True	100	01:00.5	00:01.8	✓	-97%	10027015.20	10148912.70	✓	5206.79	5136.68	✓	10.45	9.91	✗	0.93	0.93	✗	0.68	0.69	✗
TPU	512000	128	128	12800	True	True	5	00:05.1	00:01.5	✓	-71%	5303198.09	5303199.94	✓	13959.71	13959.71	✓	8.51	13.92	✓	0.94	0.93	✓	0.64	0.63	✓
TPU	512000	128	128	12800	True	True	10	00:09.7	00:01.4	✓	-85%	5303198.09	5303199.95	✓	13959.71	13959.71	✓	8.79	14.16	✓	0.94	0.93	✓	0.64	0.63	✓
TPU	512000	128	128	12800	True	True	20	00:18.7	00:01.6	✓	-92%	5303198.09	5303199.95	✓	13959.71	13959.71	✓	8.83	14.34	✓	0.94	0.93	✓	0.64	0.63	✓
TPU	512000	128	128	12800	True	True	100	01:34.4	00:01.6	✓	-98%	5303198.09	5303199.95	✓	13959.71	13959.71	✓	8.83	14.43	✓	0.94	0.93	✓	0.64	0.63	✓
TPU	512000	256	64	12800	True	True	5	00:06.8	00:03.2	✓	-52%	5282051.37	5282051.97	✓	24017.11	24017.11	✓	11.87	22.38	✓	0.92	0.92	✓	0.66	0.66	✓
TPU	512000	256	64	12800	True	True	10	00:13.2	00:02.1	✓	-84%	5282051.37	5282051.98	✓	24017.11	24017.11	✓	11.89	23.83	✓	0.92	0.92	✓	0.66	0.66	✓
TPU	512000	256	64	12800	True	True	20	00:25.4	00:01.9	✓	-92%	5282051.37	5282051.98	✓	24017.11	24017.11	✓	11.87	24.89	✓	0.92	0.92	✓	0.66	0.66	✓
TPU	512000	256	64	12800	True	True	100	02:08.0	00:02.2	✓	-98%	5282051.37	5282051.99	✓	24017.11	24017.11	✓	11.85	26.69	✓	0.92	0.92	✓	0.66	0.66	✓
TPU	512000	512	32	12800	True	True	5	00:11.0	00:03.5	✓	-68%	3428496.28	3428496.61	✓	33214.53	33214.52	✓	15.89	30.13	✓	0.88	0.88	✗	0.57	0.57	✗
TPU	512000	512	32	12800	True	True	10	00:21.1	00:03.5	✓	-84%	3428496.28	3428496.62	✓	33214.53	33214.52	✓	15.90	32.15	✓	0.88	0.88	✓	0.57	0.57	✓
TPU	512000	512	32	12800	True	True	20	00:40.2	00:03.1	✓	-92%	3428496.28	3428496.63	✓	33214.52	33214.52	✓	15.93	33.70	✓	0.88	0.88	✓	0.57	0.57	✓
TPU	512000	512	32	12800	True	True	100	03:15.0	00:03.3	✓	-98%	3428496.28	3428496.64	✓	33214.52	33214.52	✓	15.93	35.57	✓	0.88	0.88	✓	0.57	0.57	✓
TPU	1024000	64	32	25600	True	True	5	00:06.9	00:01.8	✓	-73%	224798.01	224985.25	✓	155705.44	155705.35	✓	4.26	5.66	✓	0.93	0.93	✗	0.75	0.75	✗
TPU	1024000	64	32	25600	True	True	10	00:13.3	00:01.8	✓	-87%	224798.01	224986.53	✓	155705.44	155705.35	✓	4.26	5.73	✓	0.93	0.93	✗	0.75	0.75	✗
TPU	1024000	64	32	25600	True	True	20	00:27.3	00:01.9	✓	-93%	224798.01	224987.53	✓	155705.44	155705.35	✓	4.26	5.74	✓	0.93	0.93	✗	0.75	0.75	✗
TPU	1024000	64	32	25600	True	True	100	02:16.1	00:02.0	✓	-99%	224798.01	224988.94	✓	155705.44	155705.35	✓	4.26	5.74	✓	0.93	0.93	✗	0.75	0.75	✗
TPU	1024000	128	128	25600	True	True	5	00:08.5	00:01.9	✓	-78%	5074685.35	5074685.87	✓	29895.97	29895.96	✓	8.21	12.80	✓	0.94	0.94	✓	0.68	0.68	✓
TPU	1024000	128	128	25600	True	True	10	00:15.9	00:01.9	✓	-88%	5074685.35	5074685.88	✓	29895.96	29895.96	✓	8.21	13.04	✓	0.94	0.94	✓	0.68	0.68	✓
TPU	1024000	128	128	25600	True	True	20	00:32.8	00:02.0	✓	-94%	5074685.35	5074685.88	✓	29895.96	29895.96	✓	8.21	13.13	✓	0.94	0.94	✓	0.68	0.68	✓
TPU	1024000	128	128	25600	True	True	100	02:41.3	00:02.3	✓	-99%	5074685.35	5074685.88	✓	29895.96	29895.96	✓	8.21	13.31	✓	0.94	0.94	✓	0.68	0.68	✓
TPU	1024000	256	64	25600	True	True	5	00:12.1	00:02.9	✓	-76%	4975061.66	4975062.04	✓	52106.41	52106.41	✓	11.14	20.16	✓	0.93	0.93	✓	0.66	0.65	✓
TPU	1024000	256	64	25600	True	True	10	00:23.5	00:03.2	✓	-86%	4975061.66	4975062.04	✓	52106.41	52106.41	✓	11.14	21.04	✓	0.93	0.93	✓	0.66	0.65	✓
TPU	1024000	256	64	25600	True	True	20	00:46.5	00:02.9	✓	-94%	4975061.66	4975062.05	✓	52106.41	52106.41	✓	11.14	21.74	✓	0.93	0.93	✓	0.66	0.65	✓
TPU	1024000	256	64	25600	True	True	100	03:48.3	00:03.4	✓	-99%	4975061.66	4975062.05	✓	52106.41	52106.41	✓	11.14	23.46	✓	0.93	0.93	✓	0.66	0.65	✓

Figure A.1.: Runtime And Quality Comparison of version 1 *distribute_ repeat* and version 2 *distribute_ mean* (Fixed Number of Iterations)

Data Set	***SSE v1***	***SSE v2***	***Rel.***	***CH v1***	***CH v2***	***Rel.***	***DB v1***	***DB v2***	***Rel.***
1	579.43	682.69	-18%	15165.21	12792.84	-16%	1.13	1.49	-32%
2	29641.46	30346.58	-2%	2786.19	2786.18	0%	2.03	2.23	-10%
3	34947.23	46506.13	-33%	5465.20	4256.51	-22%	2.53	2.59	-2%
4	306057.01	325939.80	-6%	12195.68	11382.40	-7%	3.75	8.86	-136%
5	2889620.30	2923541.66	-1%	4324.14	4257.66	-2%	8.81	13.95	-58%
6	10027015.20	10148912.70	-1%	5206.79	5136.68	-1%	10.45	9.91	5%
7	5303198.09	5303199.95	0%	13959.71	13959.71	0%	8.85	14.42	-63%
8	5282051.37	5282051.99	0%	24017.11	24017.11	0%	11.85	26.69	-125%
9	3428496.28	3428496.64	0%	33214.52	33214.52	0%	15.93	35.57	-123%
10	224798.01	224985.45	0%	155705.44	155705.35	0%	4.26	5.71	-34%
11	5074685.35	5074685.88	0%	29895.97	29895.96	0%	8.19	13.31	-62%
12	4975061.66	4975062.05	0%	52106.41	52106.41	0%	11.14	23.46	-111%

Table A.2.: Quality Comparison Internal Metrics (Variable Number of Iterations)

Data Set	***NMI v1***	***NMI v2***	***Rel.***	***ARI v1***	***ARI v2***	***Rel.***
1	0.94	0.91	-3%	0.84	0.81	-3%
2	0.90	0.90	0%	0.64	0.65	0%
3	0.95	0.94	-2%	0.85	0.83	-2%
4	0.92	0.92	0%	0.68	0.68	-1%
5	0.91	0.91	0%	0.63	0.63	0%
6	0.93	0.93	0%	0.68	0.69	0%
7	0.94	0.93	0%	0.63	0.63	0%
8	0.92	0.92	0%	0.66	0.66	0%
9	0.88	0.88	0%	0.57	0.57	0%
10	0.93	0.93	0%	0.75	0.75	0%
11	0.94	0.94	0%	0.68	0.68	0%
12	0.93	0.93	0%	0.66	0.65	0%

Table A.3.: Quality Comparison External Metrics (Variable Number of Iterations)

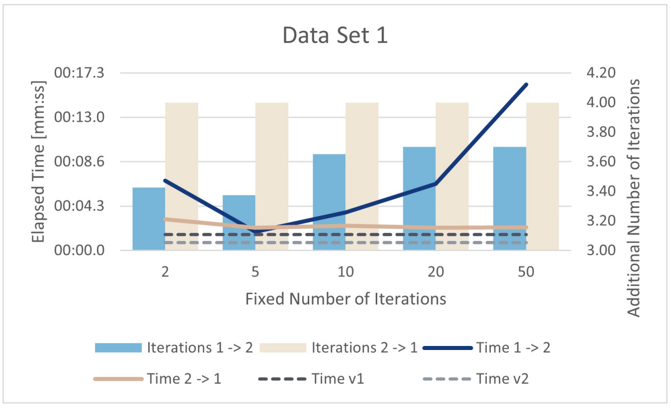


Figure A.2.: Runtime Comparison of the Combinations for Data Set 1

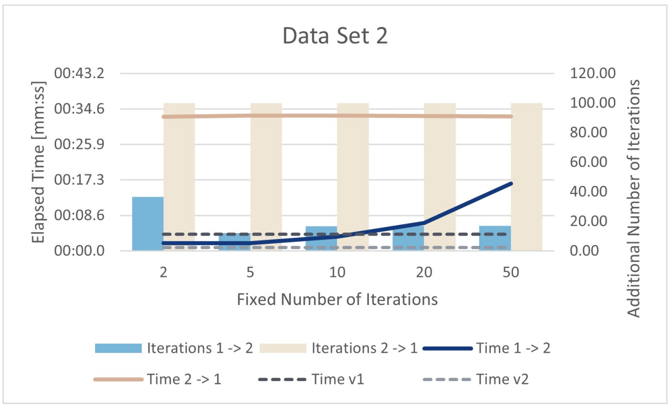


Figure A.3.: Runtime Comparison of the Combinations for Data Set 2

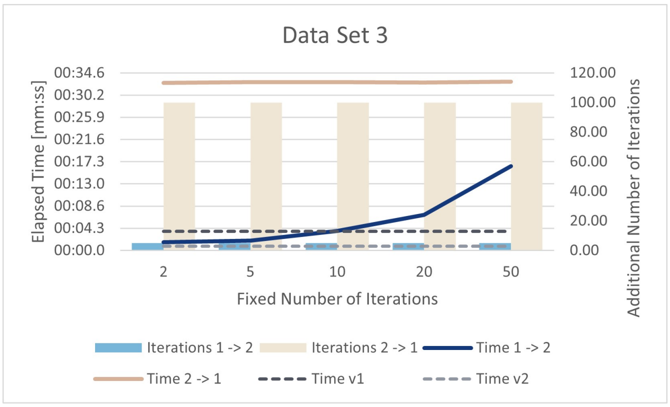


Figure A.4.: Runtime Comparison of the Combinations for Data Set 3

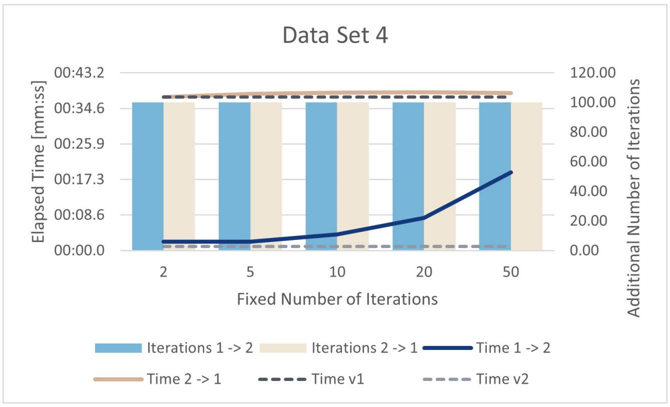


Figure A.5.: Runtime Comparison of the Combinations for Data Set 4

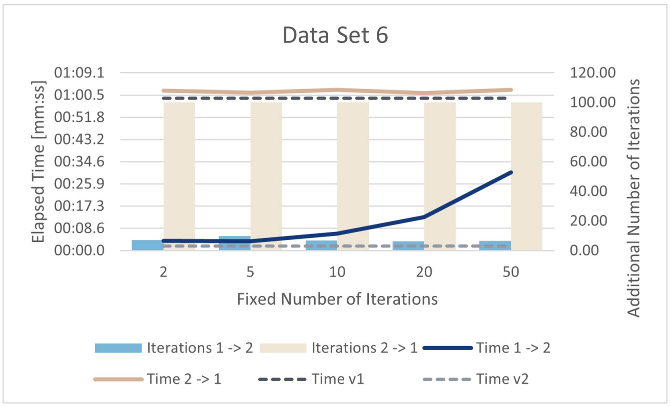


Figure A.6.: Runtime Comparison of the Combinations for Data Set 6

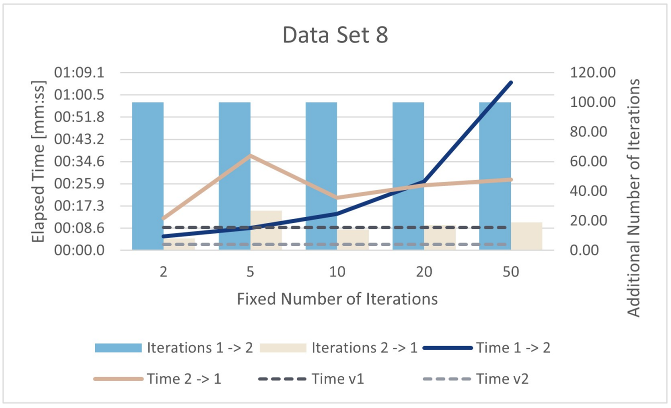


Figure A.7.: Runtime Comparison of the Combinations for Data Set 8

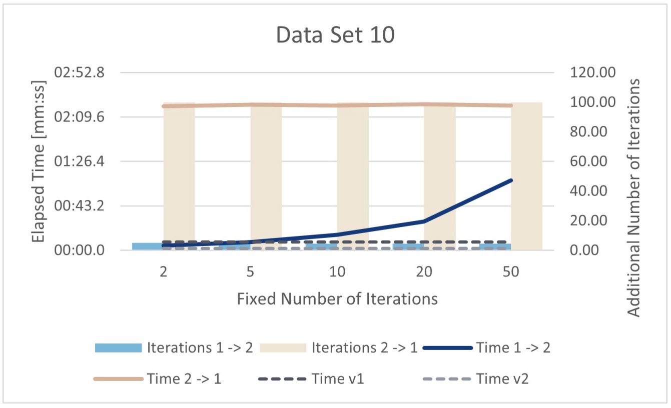


Figure A.8.: Runtime Comparison of the Combinations for Data Set 10

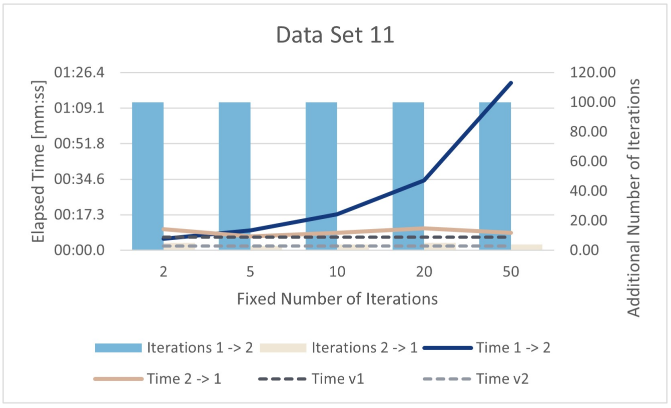


Figure A.9.: Runtime Comparison of the Combinations for Data Set 11

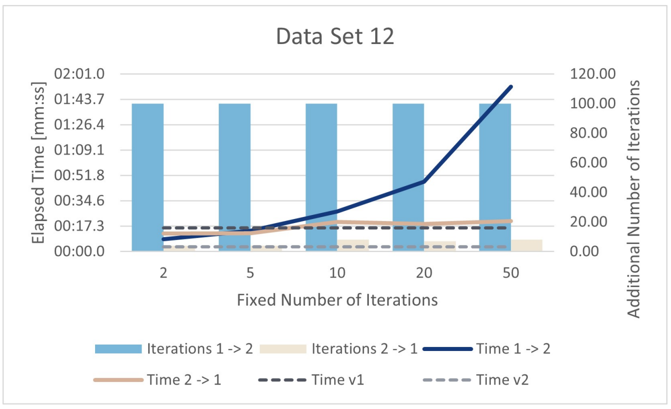


Figure A.10.: Runtime Comparison of the Combinations for Data Set 12

	CH											
Data Set	1 → 2 (2)	1 → 2 (5)	1 → 2 (10)	1 → 2 (20)	1 → 2 (50)	2 → 1 (2)	2 → 1 (5)	2 → 1 (10)	2 → 1 (20)	2 → 1 (50)	v1	v2
1	15164.08	15164.24	15164.38	15164.45	15164.45	15165.19	15165.20	15165.19	15165.19	15165.19	15165.21	12792.84
2	2786.18	2786.19	2786.19	2786.19	2786.19	3483.73	3483.72	3483.72	3483.72	3483.72	2786.19	2786.18
3	5465.19	5465.20	5465.20	5465.20	5465.20	6307.18	6307.18	6307.18	6307.18	6307.18	5465.20	4256.51
4	12195.67	12195.67	12195.67	12195.67	12195.67	12195.68	12195.68	12195.68	12195.68	12195.68	12195.68	11382.40
5	4324.14	4324.14	4324.14	4324.14	4324.14	4393.92	4393.92	4393.92	4393.92	4393.92	4324.14	4257.66
6	5206.79	5206.79	5206.79	5206.79	5206.79	5655.92	5655.92	5655.92	5655.92	5655.92	5206.79	5136.68
7	13959.71	13959.71	13959.71	13959.71	13959.71	13959.71	14070.53	13959.71	14070.53	13959.71	13959.71	13959.71
8	24017.11	24017.11	24017.11	24017.11	24017.11	24017.11	24017.11	24017.11	24017.11	24017.11	24017.11	24017.11
9	33214.52	33214.52	33214.52	33214.52	33214.52	33214.53	33214.52	33214.53	33214.52	33214.53	33214.52	33214.52
10	155705.41	155705.42	155705.41	155705.41	155705.41	160895.76	160895.76	160895.76	160895.76	160895.76	155705.44	155705.35
11	29895.96	29895.96	29895.96	29895.96	29895.96	29895.96	29895.96	29895.96	29895.96	29895.96	29895.97	29895.96
12	52106.41	52106.41	52106.41	52106.41	52106.41	52106.41	52106.41	52106.41	52106.41	52106.41	52106.41	52106.41

Figure A.11.: Quality Heatmap Based on Calinski-Harabasz Index

	DB											
Data Set	1 → 2 (2)	1 → 2 (5)	1 → 2 (10)	1 → 2 (20)	1 → 2 (50)	2 → 1 (2)	2 → 1 (5)	2 → 1 (10)	2 → 1 (20)	2 → 1 (50)	v1	v2
1	1.25	1.24	1.23	1.23	1.23	1.12	1.12	1.12	1.12	1.12	1.13	1.49
2	2.15	2.09	2.04	2.03	2.03	1.63	1.63	1.63	1.63	1.63	2.03	2.23
3	2.58	2.58	2.58	2.59	2.59	0.66	0.66	0.66	0.66	0.66	2.53	2.59
4	5.00	4.98	5.04	5.05	5.05	3.77	3.75	3.78	3.77	3.77	3.75	8.86
5	12.04	11.79	11.78	11.81	11.77	8.76	8.76	8.79	8.79	8.80	8.81	13.95
6	11.47	11.25	11.05	11.03	11.03	8.84	8.94	9.02	8.99	8.99	10.45	9.91
7	12.76	12.85	12.85	12.85	12.85	8.66	8.29	9.14	8.34	8.89	8.85	14.42
8	21.56	22.18	22.07	21.92	22.00	11.92	12.04	12.04	12.15	12.21	11.85	26.69
9	27.11	27.22	27.27	27.18	27.18	15.93	15.91	15.95	15.95	15.99	15.93	35.57
10	4.57	4.54	4.54	4.54	4.54	4.16	4.16	4.16	4.16	4.16	4.26	5.71
11	11.37	11.07	11.14	11.14	11.14	7.90	7.92	7.80	7.73	7.81	8.19	13.31
12	17.74	17.48	17.48	17.48	17.48	11.12	11.13	11.17	11.19	11.24	11.14	23.46

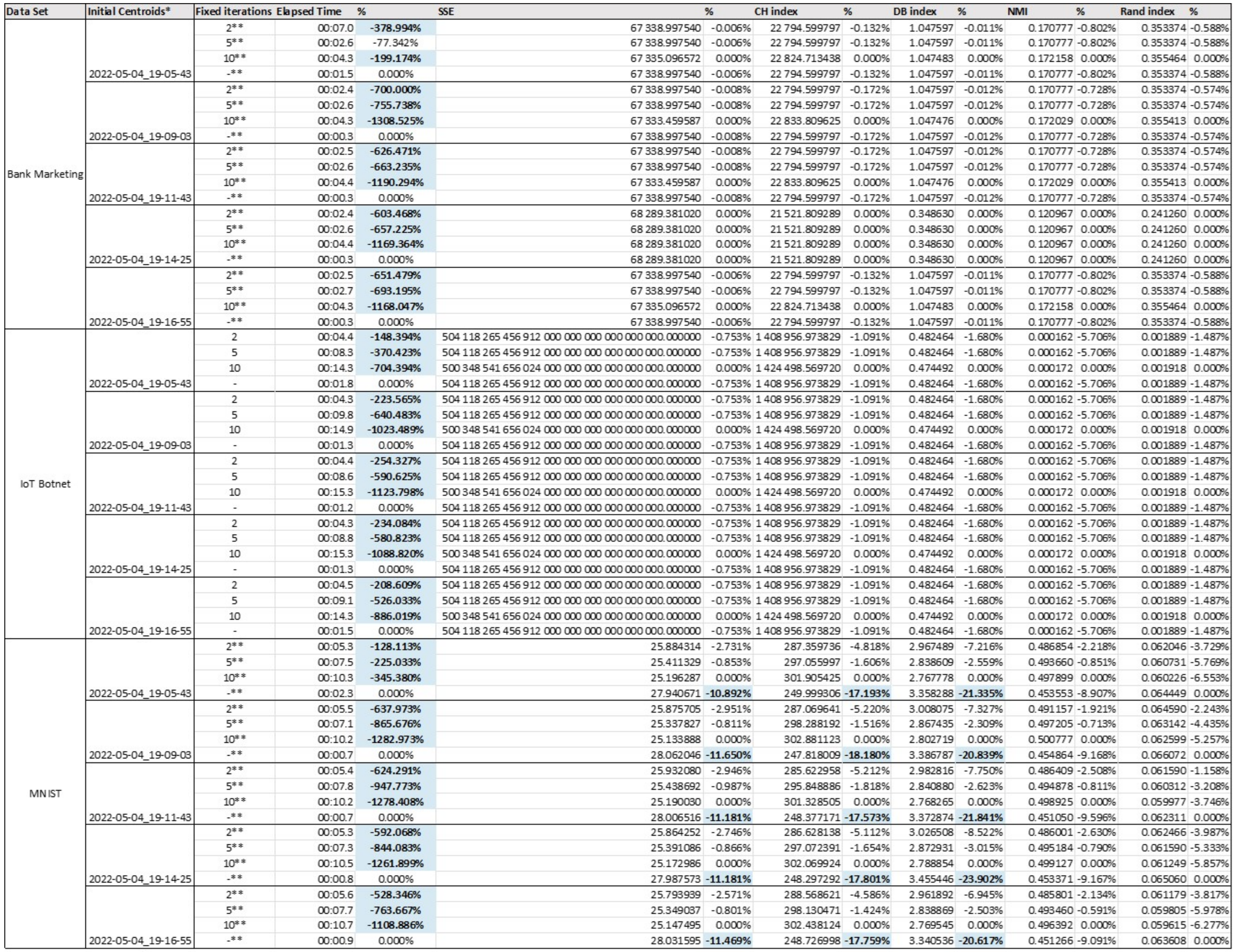
Figure A.12.: Quality Heatmap Based on Davies Boulding Index

	NMI											
Data Set	**1 → 2 (2)**	**1 → 2 (5)**	**1 → 2 (10)**	**1 → 2 (20)**	**1 → 2 (50)**	**2 → 1 (2)**	**2 → 1 (5)**	**2 → 1 (10)**	**2 → 1 (20)**	**2 → 1 (50)**	**v1**	**v2**
1	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.91
2	0.90	0.90	0.90	0.90	0.90	0.91	0.91	0.91	0.91	0.91	0.90	0.90
3	0.95	0.95	0.95	0.95	0.95	0.97	0.97	0.97	0.97	0.97	0.95	0.94
4	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92
5	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91
6	0.93	0.93	0.93	0.93	0.93	0.94	0.94	0.94	0.94	0.94	0.93	0.93
7	0.93	0.93	0.93	0.93	0.93	0.94	0.94	0.93	0.94	0.94	0.94	0.93
8	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92
9	0.88	0.88	0.88	0.88	0.88	0.88	0.88	0.88	0.88	0.88	0.88	0.88
10	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93
11	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94
12	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93

Figure A.13.: Quality Heatmap Based on Normalized Mutual Information

	ARI											
Data Set	**1 → 2 (2)**	**1 → 2 (5)**	**1 → 2 (10)**	**1 → 2 (20)**	**1 → 2 (50)**	**2 → 1 (2)**	**2 → 1 (5)**	**2 → 1 (10)**	**2 → 1 (20)**	**2 → 1 (50)**	**v1**	**v2**
1	0.84	0.83	0.83	0.83	0.83	0.84	0.84	0.84	0.84	0.84	0.84	0.81
2	0.64	0.64	0.64	0.64	0.64	0.65	0.65	0.65	0.65	0.65	0.64	0.65
3	0.85	0.85	0.85	0.85	0.85	0.88	0.88	0.88	0.88	0.88	0.85	0.83
4	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.68
5	0.62	0.62	0.62	0.62	0.62	0.63	0.63	0.63	0.63	0.63	0.63	0.63
6	0.68	0.68	0.68	0.68	0.68	0.69	0.69	0.69	0.69	0.69	0.68	0.69
7	0.63	0.63	0.63	0.63	0.63	0.63	0.64	0.63	0.64	0.63	0.63	0.63
8	0.66	0.66	0.66	0.66	0.66	0.66	0.66	0.66	0.66	0.66	0.66	0.66
9	0.57	0.57	0.57	0.57	0.57	0.57	0.57	0.57	0.57	0.57	0.57	0.57
10	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75
11	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.68
12	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.66	0.66	0.66	0.65

Figure A.14.: Quality Heatmap Based on Adjusted Rand Index

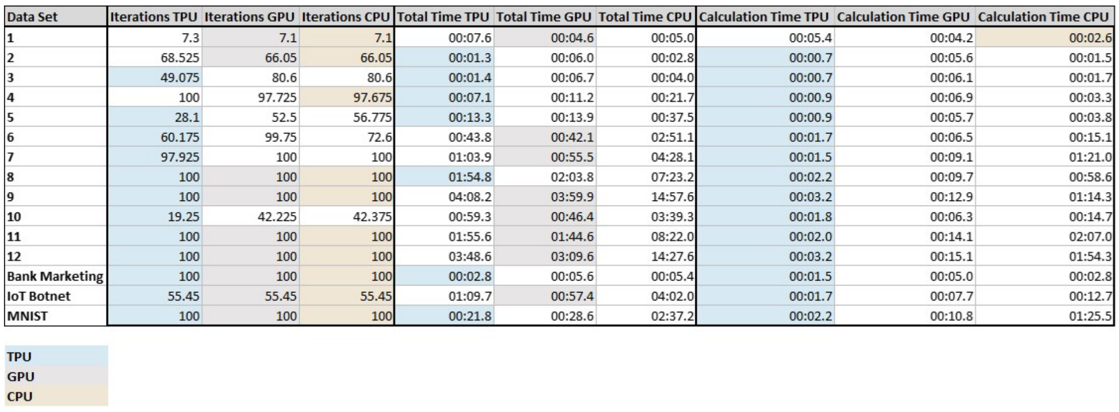


Data Set	Initial Centroids*	Fixed iterations	Elapsed Time	%	SSE	%	CH index	%	DB index	%	NMI	%	Rand index	%
Bank Marketing		2**	00:07.0	**-378.994%**	67 338.997540	-0.006%	22 794.599797	-0.132%	1.047597	-0.011%	0.170777	-0.802%	0.353374	-0.588%
		5**	00:02.6	-77.342%	67 338.997540	-0.006%	22 794.599797	-0.132%	1.047597	-0.011%	0.170777	-0.802%	0.353374	-0.588%
		10**	00:04.3	**-199.174%**	67 335.096572	0.000%	22 824.713438	0.000%	1.047483	0.000%	0.172158	0.000%	0.355464	0.000%
	2022-05-04_19-05-43	-**	00:01.5	0.000%	67 338.997540	-0.006%	22 794.599797	-0.132%	1.047597	-0.011%	0.170777	-0.802%	0.353374	-0.588%
		2**	00:02.4	**-700.000%**	67 338.997540	-0.008%	22 794.599797	-0.172%	1.047597	-0.012%	0.170777	-0.728%	0.353374	-0.574%
		5**	00:02.6	**-755.738%**	67 338.997540	-0.008%	22 794.599797	-0.172%	1.047597	-0.012%	0.170777	-0.728%	0.353374	-0.574%
		10**	00:04.3	**-1308.525%**	67 333.459587	0.000%	22 833.809625	0.000%	1.047476	0.000%	0.172029	0.000%	0.355413	0.000%
	2022-05-04_19-09-03	-**	00:00.3	0.000%	67 338.997540	-0.008%	22 794.599797	-0.172%	1.047597	-0.012%	0.170777	-0.728%	0.353374	-0.574%
		2**	00:02.5	**-626.471%**	67 338.997540	-0.008%	22 794.599797	-0.172%	1.047597	-0.012%	0.170777	-0.728%	0.353374	-0.574%
		5**	00:02.6	**-663.235%**	67 338.997540	-0.008%	22 794.599797	-0.172%	1.047597	-0.012%	0.170777	-0.728%	0.353374	-0.574%
		10**	00:04.4	**-1190.294%**	67 333.459587	0.000%	22 833.809625	0.000%	1.047476	0.000%	0.172029	0.000%	0.355413	0.000%
	2022-05-04_19-11-43	-**	00:00.3	0.000%	67 338.997540	-0.008%	22 794.599797	-0.172%	1.047597	-0.012%	0.170777	-0.728%	0.353374	-0.574%
		2**	00:02.4	**-603.468%**	68 289.381020	0.000%	21 521.809289	0.000%	0.348630	0.000%	0.120967	0.000%	0.241260	0.000%
		5**	00:02.6	**-657.225%**	68 289.381020	0.000%	21 521.809289	0.000%	0.348630	0.000%	0.120967	0.000%	0.241260	0.000%
		10**	00:04.4	**-1169.364%**	68 289.381020	0.000%	21 521.809289	0.000%	0.348630	0.000%	0.120967	0.000%	0.241260	0.000%
	2022-05-04_19-14-25	-**	00:00.3	0.000%	68 289.381020	0.000%	21 521.809289	0.000%	0.348630	0.000%	0.120967	0.000%	0.241260	0.000%
		2**	00:02.5	**-651.479%**	67 338.997540	-0.006%	22 794.599797	-0.132%	1.047597	-0.011%	0.170777	-0.802%	0.353374	-0.588%
		5**	00:02.7	**-693.195%**	67 338.997540	-0.006%	22 794.599797	-0.132%	1.047597	-0.011%	0.170777	-0.802%	0.353374	-0.588%
		10**	00:04.3	**-1168.047%**	67 335.096572	0.000%	22 824.713438	0.000%	1.047483	0.000%	0.172158	0.000%	0.355464	0.000%
	2022-05-04_19-16-55	-**	00:00.3	0.000%	67 338.997540	-0.006%	22 794.599797	-0.132%	1.047597	-0.011%	0.170777	-0.802%	0.353374	-0.588%
IoT Botnet		2	00:04.4	**-148.394%**	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		5	00:08.3	**-370.423%**	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		10	00:14.3	**-704.394%**	500 348 541 656 024 000 000 000 000 000 000.000000	0.000%	1 424 498.569720	0.000%	0.474492	0.000%	0.000172	0.000%	0.001918	0.000%
	2022-05-04_19-05-43	-	00:01.8	0.000%	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		2	00:04.3	**-223.565%**	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		5	00:09.8	**-640.483%**	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		10	00:14.9	**-1023.489%**	500 348 541 656 024 000 000 000 000 000 000.000000	0.000%	1 424 498.569720	0.000%	0.474492	0.000%	0.000172	0.000%	0.001918	0.000%
	2022-05-04_19-09-03	-	00:01.3	0.000%	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		2	00:04.4	**-254.327%**	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		5	00:08.6	**-590.625%**	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		10	00:15.3	**-1123.798%**	500 348 541 656 024 000 000 000 000 000 000.000000	0.000%	1 424 498.569720	0.000%	0.474492	0.000%	0.000172	0.000%	0.001918	0.000%
	2022-05-04_19-11-43	-	00:01.2	0.000%	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		2	00:04.3	**-234.084%**	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		5	00:08.8	**-580.823%**	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		10	00:15.3	**-1088.820%**	500 348 541 656 024 000 000 000 000 000 000.000000	0.000%	1 424 498.569720	0.000%	0.474492	0.000%	0.000172	0.000%	0.001918	0.000%
	2022-05-04_19-14-25	-	00:01.3	0.000%	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		2	00:04.5	**-208.609%**	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		5	00:09.1	**-526.033%**	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
		10	00:14.3	**-886.019%**	500 348 541 656 024 000 000 000 000 000 000.000000	0.000%	1 424 498.569720	0.000%	0.474492	0.000%	0.000172	0.000%	0.001918	0.000%
	2022-05-04_19-16-55	-	00:01.5	0.000%	504 118 265 456 912 000 000 000 000 000 000.000000	-0.753%	1 408 956.973829	-1.091%	0.482464	-1.680%	0.000162	-5.706%	0.001889	-1.487%
MNIST		2**	00:05.3	**-128.113%**	25.884314	-2.731%	287.359736	-4.818%	2.967489	-7.216%	0.486854	-2.218%	0.062046	-3.729%
		5**	00:07.5	**-225.033%**	25.411329	-0.853%	297.055997	-1.606%	2.838609	-2.559%	0.493660	-0.851%	0.060731	-5.769%
		10**	00:10.3	**-345.380%**	25.196287	0.000%	301.905425	0.000%	2.767778	0.000%	0.497899	0.000%	0.060226	-6.553%
	2022-05-04_19-05-43	-**	00:02.3	0.000%	27.940671	**-10.892%**	249.999306	**-17.193%**	3.358288	**-21.335%**	0.453553	-8.907%	0.064449	0.000%
		2**	00:05.5	**-637.973%**	25.875705	-2.951%	287.069641	-5.220%	3.008075	-7.327%	0.491157	-1.921%	0.064590	-2.243%
		5**	00:07.1	**-865.676%**	25.337827	-0.811%	298.288192	-1.516%	2.867435	-2.309%	0.497205	-0.713%	0.063142	-4.435%
		10**	00:10.2	**-1282.973%**	25.133888	0.000%	302.881123	0.000%	2.802719	0.000%	0.500777	0.000%	0.062599	-5.257%
	2022-05-04_19-09-03	-**	00:00.7	0.000%	28.062046	**-11.650%**	247.818009	**-18.180%**	3.386787	**-20.839%**	0.454864	-9.168%	0.066072	0.000%
		2**	00:05.4	**-624.291%**	25.932080	-2.946%	285.622958	-5.212%	2.982816	-7.750%	0.486409	-2.508%	0.061590	-1.158%
		5**	00:07.8	**-947.773%**	25.438692	-0.987%	295.848886	-1.818%	2.840880	-2.623%	0.494878	-0.811%	0.060312	-3.208%
		10**	00:10.2	**-1278.408%**	25.190030	0.000%	301.328505	0.000%	2.768265	0.000%	0.498925	0.000%	0.059977	-3.746%
	2022-05-04_19-11-43	-**	00:00.7	0.000%	28.006516	**-11.181%**	248.377171	**-17.573%**	3.372874	**-21.841%**	0.451050	-9.596%	0.062311	0.000%
		2**	00:05.3	**-592.068%**	25.864252	-2.746%	286.628138	-5.112%	3.026508	-8.522%	0.486001	-2.630%	0.062466	-3.987%
		5**	00:07.3	**-844.083%**	25.391086	-0.866%	297.072391	-1.654%	2.872931	-3.015%	0.495184	-0.790%	0.061590	-5.333%
		10**	00:10.5	**-1261.899%**	25.172986	0.000%	302.069924	0.000%	2.788854	0.000%	0.499127	0.000%	0.061249	-5.857%
	2022-05-04_19-14-25	-**	00:00.8	0.000%	27.987573	**-11.181%**	248.297292	**-17.801%**	3.455446	**-23.902%**	0.453371	-9.167%	0.065060	0.000%
		2**	00:05.6	**-528.346%**	25.793939	-2.571%	288.568621	-4.586%	2.961892	-6.945%	0.485801	-2.134%	0.061179	-3.817%
		5**	00:07.7	**-763.667%**	25.349037	-0.801%	298.130471	-1.424%	2.838869	-2.503%	0.493460	-0.591%	0.059805	-5.978%
		10**	00:10.7	**-1108.886%**	25.147495	0.000%	302.438124	0.000%	2.769545	0.000%	0.496392	0.000%	0.059615	-6.277%
	2022-05-04_19-16-55	-**	00:00.9	0.000%	28.031595	**-11.469%**	248.726998	**-17.759%**	3.340536	**-20.617%**	0.451266	-9.091%	0.063608	0.000%

* Reference to File

** Max iterations were neeeded

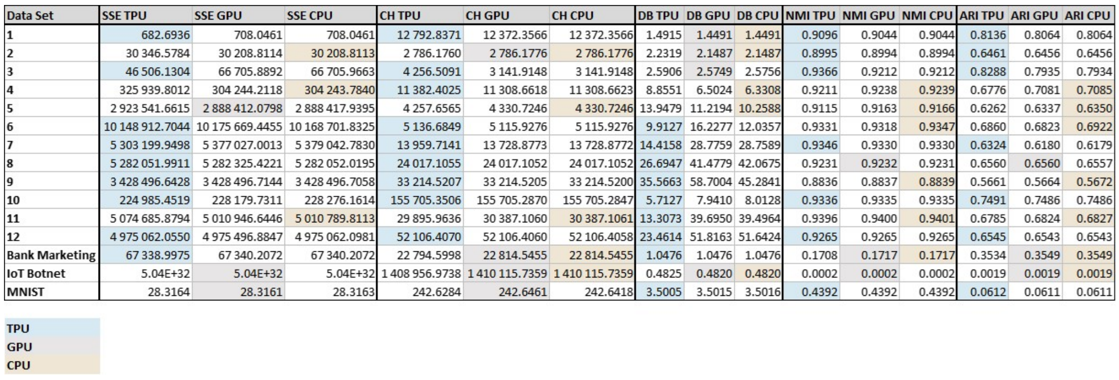
Figure A.15.: Benchmark Comparison of Runtime and Quality



Data Set	Iterations TPU	Iterations GPU	Iterations CPU	Total Time TPU	Total Time GPU	Total Time CPU	Calculation Time TPU	Calculation Time GPU	Calculation Time CPU
1	7.3	7.1	7.1	00:07.6	00:04.6	00:05.0	00:05.4	00:04.2	00:02.6
2	68.525	66.05	66.05	00:01.3	00:06.0	00:02.8	00:00.7	00:05.6	00:01.5
3	49.075	80.6	80.6	00:01.4	00:06.7	00:04.0	00:00.7	00:06.1	00:01.7
4	100	97.725	97.675	00:07.1	00:11.2	00:21.7	00:00.9	00:06.9	00:03.3
5	28.1	52.5	56.775	00:13.3	00:13.9	00:37.5	00:00.9	00:05.7	00:03.8
6	60.175	99.75	72.6	00:43.8	00:42.1	02:51.1	00:01.7	00:06.5	00:15.1
7	97.925	100	100	01:03.9	00:55.5	04:28.1	00:01.5	00:09.1	01:21.0
8	100	100	100	01:54.8	02:03.8	07:23.2	00:02.2	00:09.7	00:58.6
9	100	100	100	04:08.2	03:59.9	14:57.6	00:03.2	00:12.9	01:14.3
10	19.25	42.225	42.375	00:59.3	00:46.4	03:39.3	00:01.8	00:06.3	00:14.7
11	100	100	100	01:55.6	01:44.6	08:22.0	00:02.0	00:14.1	02:07.0
12	100	100	100	03:48.6	03:09.6	14:27.6	00:03.2	00:15.1	01:54.3
Bank Marketing	100	100	100	00:02.8	00:05.6	00:05.4	00:01.5	00:05.0	00:02.8
IoT Botnet	55.45	55.45	55.45	01:09.7	00:57.4	04:02.0	00:01.7	00:07.7	00:12.7
MNIST	100	100	100	00:21.8	00:28.6	02:37.2	00:02.2	00:10.8	01:25.5

🟦	TPU
⬜	GPU
🟫	CPU

Figure A.16.: Runtime Comparison of TPU, GPU and, CPU



Data Set	SSE TPU	SSE GPU	SSE CPU	CH TPU	CH GPU	CH CPU	DB TPU	DB GPU	DB CPU	NMI TPU	NMI GPU	NMI CPU	ARI TPU	ARI GPU	ARI CPU
1	682.6936	708.0461	708.0461	12 792.8371	12 372.3566	12 372.3566	1.4915	1.4491	1.4491	0.9096	0.9044	0.9044	0.8136	0.8064	0.8064
2	30 346.5784	30 208.8114	30 208.8113	2 786.1760	2 786.1776	2 786.1776	2.2319	2.1487	2.1487	0.8995	0.8994	0.8994	0.6461	0.6456	0.6456
3	46 506.1304	66 705.8892	66 705.9663	4 256.5091	3 141.9148	3 141.9148	2.5906	2.5749	2.5756	0.9366	0.9212	0.9212	0.8288	0.7935	0.7934
4	325 939.8012	304 244.2118	304 243.7840	11 382.4025	11 308.6618	11 308.6623	8.8551	6.5024	6.3308	0.9211	0.9238	0.9239	0.6776	0.7081	0.7085
5	2 923 541.6615	2 888 412.0798	2 888 417.9395	4 257.6565	4 330.7246	4 330.7246	13.9479	11.2194	10.2588	0.9115	0.9163	0.9166	0.6262	0.6337	0.6350
6	10 148 912.7044	10 175 669.4455	10 168 701.8325	5 136.6849	5 115.9276	5 115.9276	9.9127	16.2277	12.0357	0.9331	0.9318	0.9347	0.6860	0.6823	0.6922
7	5 303 199.9498	5 377 027.0013	5 379 042.7830	13 959.7141	13 728.8773	13 728.8772	14.4158	28.7759	28.7589	0.9346	0.9330	0.9330	0.6324	0.6180	0.6179
8	5 282 051.9911	5 282 325.4221	5 282 052.0195	24 017.1055	24 017.1052	24 017.1052	26.6947	41.4779	42.0675	0.9231	0.9232	0.9231	0.6560	0.6560	0.6557
9	3 428 496.6428	3 428 496.7144	3 428 496.7058	33 214.5207	33 214.5205	33 214.5200	35.5663	58.7004	45.2841	0.8836	0.8837	0.8839	0.5661	0.5664	0.5672
10	224 985.4519	228 179.7311	228 276.1614	155 705.3506	155 705.2870	155 705.2847	5.7127	7.9410	8.0128	0.9336	0.9335	0.9335	0.7491	0.7486	0.7486
11	5 074 685.8794	5 010 946.6446	5 010 789.8113	29 895.9636	30 387.1060	30 387.1061	13.3073	39.6950	39.4964	0.9396	0.9400	0.9401	0.6785	0.6824	0.6827
12	4 975 062.0550	4 975 496.8847	4 975 062.0981	52 106.4070	52 106.4060	52 106.4058	23.4614	51.8163	51.6424	0.9265	0.9265	0.9265	0.6545	0.6543	0.6543
Bank Marketing	67 338.9975	67 340.2072	67 340.2072	22 794.5998	22 814.5455	22 814.5455	1.0476	1.0476	1.0476	0.1708	0.1717	0.1717	0.3534	0.3549	0.3549
IoT Botnet	5.04E+32	5.04E+32	5.04E+32	1 408 956.9738	1 410 115.7359	1 410 115.7359	0.4825	0.4820	0.4820	0.0002	0.0002	0.0002	0.0019	0.0019	0.0019
MNIST	28.3164	28.3161	28.3163	242.6284	242.6461	242.6418	3.5005	3.5015	3.5016	0.4392	0.4392	0.4392	0.0612	0.0611	0.0611

🟦	TPU
⬜	GPU
🟫	CPU

Figure A.17.: Quality Comparison of TPU, GPU and, CPU