Anomaly Detection in Distributed Dataflow Systems

Master Thesis
by
Xuan Truong NGUYEN

Submitted to the Faculty IV, Electrical Engineering and Computer Science Database Systems and Information Management Group in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
as part of the ERASMUS MUNDUS programme IT4BI
at the
TECHNISCHE UNIVERSITÄT BERLIN
July 31, 2016

Thesis Advisor:
Behrouz DERAKHSHAN

Thesis Supervisor:
Prof. Dr. Volker MARKL

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Berlin, July 31, 2016

Xuan Truong NGUYEN

Abstract

Anomaly Detection in Distributed Dataflow Systems

by

Xuan Truong NGUYEN

Anomalies (also known as outliers) are unexpected and abnormal patterns in data. They are often very different from normal data. Anomaly detection or outlier detection is an important data mining task because of its applications in diverse fields. Over the past decades, a number of outlier detection algorithms have been proposed. As data are getting bigger and bigger, identifying outliers becomes troublesome for conventional algorithms due to high computation cost. Outlier detection in big data is a challenging task. However, it can benefit from distributed dataflow engines and become more efficient when dealing with massive amount of data. In the scope of this thesis, we surveyed different approaches to detect anomalies. A variant of k-means clustering algorithm for outlier detection which is capable of handling big data was selected to study in depth. Strengths and weaknesses of the algorithm were also discussed. The experimental results on various synthesis and real datasets demonstrate that the algorithm has competitive performances. We also proposed a distributed parallel implementation of the algorithm and the experiments show that the parallel implementation is able to handle large data with smaller computation time compared to the single-machine implementation.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Behrouz Derakhshan for his support throughout this thesis. His patient guidance, constructive feedback, and continued support contribute greatly to the completion of my thesis.

Besides, I would like to thank IT4BI committee and everyone involved for creating a high-quality master program and giving me the opportunity to pursue the master degree. I would like to express my special thanks to Dr. Ralf Kutsche, Prof. Dr. Volker Markl and all researchers and staff at DIMA Group, TU Berlin where I spend the second year of the master program.

I want to thank all of my IT4BI classmates program the friendship and their help during the last two years.

Finally, I would like to give many thanks to my family for raising, teaching and loving me. They are always beside me whenever I need help or advice. This thesis would not have been completed without their support.

Contents

1 INTRODUCTION **1**
- 1.1 Motivations and Contributions . . . 2
- 1.2 Outline of the thesis . . . 3

2 BACKGROUND **4**
- 2.1 Anomaly detection problem . . . 4
- 2.2 Dataflow systems . . . 7
  - 2.2.1 Apache Hadoop . . . 7
  - 2.2.2 Apache Spark . . . 8
  - 2.2.3 Apache Flink . . . 8

3 RELATED WORK **10**
- 3.1 Distribution-based methods . . . 10
- 3.2 Depth-based methods . . . 10
- 3.3 Distance-based methods . . . 11
- 3.4 Density-based methods . . . 14
- 3.5 Clustering-based methods . . . 15
- 3.6 Different feature types . . . 15
- 3.7 Distributed algorithms . . . 15

4 IMPLEMENTATION **17**
- 4.1 The *k-means* clustering algorithm . . . 17
- 4.2 The *k-means--* algorithm . . . 19
- 4.3 Implementations on a single machine . . . 19
  - 4.3.1 Implementation of *k-means--* on a single machine . . . 19
  - 4.3.2 Implementation of the *k-meansOD* algorithm on a single machine . . . 22
- 4.4 Implementations in Flink . . . 22
  - 4.4.1 Flink implementation of *k-means--* . . . 23
  - 4.4.2 Flink implementation of *k-meansOD* . . . 25

5 EVALUATION **26**
- 5.1 Local evaluation . . . 26
  - 5.1.1 Synthesis datasets . . . 26
  - 5.1.2 Real datasets . . . 34
- 5.2 Cluster Evaluation . . . 41
  - 5.2.1 Local implementations versus Flink implementations . . . 42
  - 5.2.2 Varying parallelism . . . 43
  - 5.2.3 Varying number of clusters . . . 44
  - 5.2.4 Varying size of datasets . . . 45

5.2.5 Summary 46

6 CONCLUSION AND FUTURE WORK **47**

Bibliography **49**

List of Figures

5.1 *DS1* dataset and results of *k-means--* . . . 29
5.2 *DS2* dataset and results. Outliers are red points . . . 30
5.3 Visualization of *DS3*, *DS4* and *DS5* datasets. Outliers are red points . . . 31
5.4 Number of true outliers detected for varying number of clusters . . . 33
5.5 Number of outliers true detected for varying k . . . 34
5.6 Average number of true outliers detected of three algorithms: *k-means--*, *k-means* and *kNN* on 5 synthesis datasets . . . 35
5.7 Comparison between *k-means--* and *k-means* on *DS5* dataset . . . 36
5.8 Results on *Breast Cancel* dataset . . . 37
5.9 Results on *Glass identification* dataset . . . 38
5.10 Results on *NSL-KDD* dataset . . . 38
5.11 Running time on *KDD Cup 1999 small* dataset . . . 39
5.12 Running time on *NSL-KDD* dataset . . . 39
5.13 Running times of *k-means--* on *KDD Cup 1999* and *KDD Cup 1999 small* datasets . . . 43
5.14 Running times of *Flink k-means--* and *Flink k-meansOD* for varying parallelism . . . 44
5.15 Running times of *Flink k-means--* and *Flink k-meansOD* for varying number of clusters . . . 45
5.16 Running time of *Flink k-means--* for varying size of datasets . . . 46

List of Tables

5.1 Synthesis datasets in local experiments 27
5.2 Results of 10 runs on *DS2* 28
5.3 Real datasets for local experiments 36
5.4 Average sum of squared distances of *k-means--* and *k-means* 41
5.5 Datasets for cluster evaluation 42

Chapter 1

INTRODUCTION

Anomalies (also known as outliers) are unexpected and abnormal patterns in data. They are often rare and notably distinct from the rest of the data. However, it is not easy to describe concretely what are anomalies and how they differ from normal patterns. One of the general definitions which is broadly accepted was raised by Hawkins [1]:

"An outlier is an observation that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism."

The definition above assumes that data are generated by a certain process. Outliers occur when we can see observations that do not conform to some expected characteristics of normal data. The concrete definition, however, varies and depends on the domains, datasets and purposes of outlier studies. For example, an outlier can be a bank transaction in which high amount of money is spent on the credit card in a country different from the one of the card holder. An outlier can be the abnormal high incoming traffic to a particular machine in a computer network. An outlier might be a product's defect in a production line. Although outliers are often rare compared to the remainder of the data, they are present in almost of the available data. They can come from different sources such as human errors or machinery faults.

There are many reasons for detecting outliers. First, they are often indicators of strange and undesired problems. Therefore, identifying outliers helps early pinpoint the causes of the issues before they become severe. Second, outliers can negatively affect the outcomes of studies on the data in which they appear. Hence, cleaning outliers results in more accurate results. Third, outliers may reveal some interesting patterns and valuable hidden information. One may be interested in the rare and abnormal behavior than the normal data.

Outlier and detection methods have been researched for a long time in the field of statistics since the 19th century [2]. Most of the early studies are based on statistical approach [3]. Nowadays, detection of outliers is an important task because of its diverse applications in different domains. Over the past decades, a variety of methods and techniques with various approaches have been formulated to detect anomalies. Some of

them are designed specifically for a domain of application such as healthcare [4], network intrusion [5], and image processing while others are more general and can apply to different fields. Some algorithms require prior information and data to build a model for identifying outliers while others do not need. Some of them only work with numerical data while others can handle categorical attributes.

Outlier detection can be broadly categorized into supervised, unsupervised and semi-supervised approaches. Algorithms in supervised and semi-supervised categories require training datasets. Supervised methods need information about outliers and normal observations while semi-supervised only need either normal or anomalous observations. They all build a model based on the given data and are tested against unseen data. In contrast, unsupervised algorithms do not need any additional information about the input data. Outlier detection algorithms can be further divided into smaller groups such as statistical-based, distance-based, density-based, clustering-based and classification based techniques.

1.1 Motivations and Contributions

Outlier detection is a difficult task because of various challenges. It is hard to distinguish clearly between outliers and normal data observations. Outliers detection algorithms often require manually chosen parameters to deal with different datasets. There exist no best algorithms. The performances of them depend on the input parameters and the datasets. Designing an efficient algorithm for a particular field is a time-consuming task. It also requires domain-specific knowledge. Besides, some algorithms require labeled data which is nontrivial to obtain because outliers are rare. Moreover, the costs for data storage have decreased significantly in the past decades, allowing more data to be collected and processed. As data are getting bigger and bigger, identifying outliers becomes troublesome for conventional algorithms due to high computation cost. Fortunately, the introduction of distributed data processing platforms such Hadoop[1], Flink[2] and Spark[3] can help processing larger amount of data than before in a reasonable time.

In the scope of this thesis, we only focus on the unsupervised distance-based outlier detection. Specifically, we are interested in the problem of identifying top l outliers from a given dataset. First, we survey different widely used outlier detection techniques. We implement a variant of *k-means* clustering algorithm namely *k-means--* [6] and present the detailed analysis of different aspects of the algorithms. The experiments are conducted on different synthesis and real datasets in comparison with different distance-based outlier algorithms. Finally, we propose a distributed implementation of the *k-means--* on

[1] http://hadoop.apache.org/
[2] https://flink.apache.org/
[3] http://spark.apache.org/

Apache Flink which is scalable and could handle big amount of data compared to its implementation on a single machine.

1.2 Outline of the thesis

For the rest of the thesis, we organize it in different chapters. Chapter 2 is dedicated for background information where we describe the anomaly detection problem and dataflow systems. In Chapter 3, we review different outlier detection techniques. In chapter 4, we present the *k-means* algorithm together with its variants for anomaly detection. We describe in details how we implement those algorithms on both a single machine and Apache Flink. Chapter 5 presents the empirical results of the algorithms described in Chapter 4. We conduct experiments on a single computer as well as on a Flink cluster. Chapter 6 is for summary and potential future work.

Chapter 2

BACKGROUND

In this section, we will discuss the expected input and output of a typical anomaly detection algorithm. A taxonomy of anomaly detection algorithms is also introduced with their advantages and disadvantages. Finally, we will present some dataflow systems.

2.1 Anomaly detection problem

Since anomaly detection has broad applications in various domains, the format of the inputs is also different. For example, they can be textual logs from a website, raw TCP-IP requests to a computer or information about credit card transactions of a bank. In order to apply data mining techniques to identify anomalies from those inputs, it is essential to extract relevant features from the raw inputs and transform it into more generic and standardized format. Feature selection is an important task and has the influence on the result of anomaly detection algorithms; however, in the scope of this thesis, we will not focus on how to select good features from raw inputs. We assume that we are given the input with all necessary features. The input and output of the a typical anomaly detection algorithm are defined as follow

Input: An input dataset contains n data observations. Each data observation has the same number of features. The number of features is also known as the dimension or attributes of data. We also use the term data observation, data instance and data point interchangeably with the same meaning.

Output: Anomaly detection algorithms may produce different outputs depending on the selected algorithms. There are two common types of output for anomaly detection algorithms are often reported [7, 8].

- **Labels:** Algorithms identify for each data instance in the input dataset a categorical label indicating whether the instance is an outlier or not. In the case of when unsupervised algorithms are used, the label is binary i.e. either outlier or non-outlier. However, in supervised or semi-supervised methods, there could be multiple labels for outliers. For example, considering five label $\{x_1, x_2, y_1, y_2, y_3\}$, if x_i is the label of anomalies, we have two labels for anomalies and three labels for normal data.

- **Scores:** Algorithms compute for each data instances a numerical score representing the outlierness of the instance. Outliers are data observations which have high outlierness scores. Furthermore, algorithms which are interested in getting top l outliers, benefit from this type of output as it is easy to extract l instances with highest outlierness scores.

There are many criteria can be used to categorize anomaly detection algorithms. For example, based on the characteristics of the input data, one can divide the algorithms three groups for handling datasets with categorical features, numerical features and both categorical and numerical features. Based on the data mining techniques applied, the algorithms can be divided into different groups of approaches such as statistical approach, distance-based approach, density-based approach, classification approach, clustering approach. It is possible to divide algorithms in each group into smaller subgroups. For instance, statistical algorithms can be classified to univariate and multivariate algorithms. In this section, we broadly divide anomaly detection algorithms into three different groups based the characteristics of the machine learning algorithms employed to detect outliers.

- **Supervised methods:** In this approach, algorithms are given labeled training data to build a model in order to predict labels of unseen data observations. In the training data, each data observation has a label indicating if it is anomalous or normal observation. There could be multiple different labels indicating anomalous and normal observations. For example, in a dataset with ten different labels, there are three labels connected with anomalies and seven labels linked to normal data observations. Outlier detection in the supervised scenario could be viewed as a special case of the classification problem in which a predictive model is built based on the given training data. The model is then used to classify the unlabeled data. There are many techniques can be employed to train predictive models such as decision trees, support vector machines, and neural networks.

  Advantages

  - There are two phases in the supervised anomaly detection methods. In the first phase, a model is constructed using a training dataset. The computational complexity of this phase depends on the classification techniques used. In the second phase, unlabeled datasets are tested against the model to find the anomalies. The second phase is often fast because the model is already computed. Moreover, once the model is built, it can be used to identify outliers in different input datasets without having to re-train.
  - There are many powerful and robust classification methods which can be utilized.

Drawbacks

- The main drawback of supervised anomaly detection approach is that it requires the labeled data which could be expensive to build manually. Good predictive models require the training datasets to be large enough.
- Normally, the number of outliers are much smaller than normal data that can be troublesome for many algorithms to build an effective model.
- It is hard to rank the outliers based on their probability of being anomalies. Supervised methods only assign the anomalous or normal label to each of the data observations.

- **Semi-supervised methods:** In this approach, training dataset only has information about normal labels or anomalous labels. In the real world, the training data often has information about normal labels because they are more popular and easier to obtain than the anomalous labels. Similar to the supervised approach, a model of normal data (if only normal labels are available) or anomalous data (if only anomalous labels are available) is built based the training dataset and tested against the data without labels to detect anomalies. The advantage of semi-supervised methods is they do not need the information about both normal and anomalous labels as supervised methods. However, they suffer from the problem of how to model outliers and normal data observations correctly. The amount of available labels is limited, and a model might not cover all normal or anomalous scenarios.

- **Unsupervised methods:** This approach does not require labeled datasets, therefore is more applicable and popular than the supervised and semi-supervised methods. Techniques used in unsupervised settings often assume that the number of outliers in a dataset is much smaller than the number of normal instances. Unsupervised methods receive a dataset as input and produce a list of outliers from the dataset. The common techniques used for detection outliers under unsupervised settings are clustering approaches, density-based approaches, and distance-based approaches.

  Advantages

  - One advantage of the unsupervised methods is that they do not require labeled datasets.
  - Some unsupervised methods compute outlierness score for each data observation. Therefore, they are possible to determine the probability of being outliers of data observations. The outlier scores are also useful to rank or get some top outliers.

Drawbacks

- Computational complexity is often a problem with unsupervised approaches. The outliers are identified with respect to other observations in the input data. Therefore, it is not possible to build a common model and apply it to multiple input datasets as in supervised methods.
- Results of some clustering algorithms are sensitive to the initial setup, and parameters. The accuracy of outlier detection depends on how the algorithms are initialized.

2.2 Dataflow systems

Dataflow is a software paradigm which enables a software application's execution to be modeled as a directed graph. Nodes in the graph can be considered as black box data processing units. Each node in the graph receives its input data from its direct predecessor nodes, applies a transformation on the data and pass the results to the direct successor nodes. Nodes link to other nodes through directed edges which define the data flow on the graph. One of the key features of dataflow architecture is that each node in the execution graph can be triggered to executed independently as soon as it receives necessary input from other nodes. This characteristic of dataflow paradigm implicitly enables the parallel computation. Therefore, programs under dataflow model can be efficiently deployed and executed in distributed dataflow systems building on top of shared-nothing computer clusters.

The need of efficiently processing massive amount of data has resulted in the emergence of many dataflow systems. In this section, we discuss three big open-source players in the subject of dataflow systems.

2.2.1 Apache Hadoop

Apache Hadoop[1] is a scalable platform for distributed processing large amount of data. Since its introduction, it has become a very popular framework for parallel programming. The underlying programming model of Hadoop is the MapReduce dataflow [9] paradigm. MapReduce provides an expressive abstraction for parallel programming through second order Map and Reduce functions. Every program which could be decomposed into Map and Reduce functions is able to run under MapReduce. One big advantage of the MapReduce is that it does not require powerful computers. Instead, it is able to deploy on commodity computer clusters. Map and Reduce are second order functions having two input parameters. One parameter is the user-defined function, and

[1] http://hadoop.apache.org

the other parameter is data to be processed by the user-defined function. Data are fed to Map and Reduce functions in key/pair format. The Map function receives a subset of input data, applies a user-defined function and writes the intermediate results to disk before sending to the Reduce function. The Reduce function groups received data into subsets of the same key and executes the user-defined function on each of the subsets.

2.2.2 Apache Spark

Apache Spark[2] is a data processing framework which originates from research work in University of California, Berkeley. Spark is built based on the Resilient Distributed Datasets (RDDs) [10] with the aim is to help alleviate shortcomings of MapReduce for iterative and data mining algorithms such as k-means, PageRank and Expectation–maximization. Those algorithms often require revisiting either the input data or the intermediate results multiple times. In MapReduce, the intermediate results are written to disk that makes the cost of revisiting the results expensive due to slow I/O operations. In Spark, data and intermediate results are stored under the abstraction RDDs. RDDs allows users to cache intermediate results in memory, thus, improves the performance of parallel iterative and data mining algorithms. Although Spark is a batch processing engine, it offers the stream processing ability by divided input data into small chunks and applies batch operation on each of the chunks. Spark also provides users with different APIs for programming languages. Currently, it supports Scala, Java, and Python. Additionally, it comes with many extensions to cope with various types of problems and data. For example, Spark SQL allows SQL or HiveQL to be executed under Spark. GraphX is designed for graph processing, and MLlib is developed for machine learning algorithms.

2.2.3 Apache Flink

Apache Flink[3] is an open-source data analysis platform. It is formerly known under the name Stratosphere [11]. Apache Flink implements the PACTs [12] programming model which could be seen as the generalization of MapReduce. Apache Flink supports parallel computation for both batch and stream processing. Flink and Spark have many similar features. Both of them support iterative algorithms. Both of them execute programs lazily, i.e. data transformations are not executed immediately after called, instead, execution graphs are created and the transformations are triggered by signals from users. However, unlike Spark, Flink does support stream processing natively that let users create and process windows of data in a more flexible way. Flink comes with both conventional Java and power modern Scala APIs. Flink is also shipped with the internal optimizer which works similar to the one of traditional Relational Database Management Systems. The

[2] http://spark.apache.org
[3] https://flink.apache.org/

optimizer in Flink generates different execution graph candidates and pick the one with least execution cost [13] based on some cost models.

Chapter 3

RELATED WORK

In this section, we report the related work in the field of anomaly detection. There are many algorithms designed for outlier detection. We present some common approaches focusing mainly on the unsupervised and distance-based methods.

3.1 Distribution-based methods

Outliers have been studied for a long time in statistics with various methods relying on statistical tests. They assume that normal data conform to a statistical distribution and estimate the distribution's parameters from the input training datasets. Outliers are data points which do not fit the distribution. Although the distribution-based methods have strong mathematics background [3], they present many drawbacks that have been outlined in the literature [7, 8, 14]. First, it is not trivial to obtain prior knowledge about the distribution of the normal data. Second, it can not be applied to data with mixed distributions, i.e. the normal data only fit one distribution [8]. Third, many methods only focus on univariate data [7]. Fourth, estimating parameters for a statistical distribution is complicated and time-consuming especially when the size of the dataset is massive. Moreover, obtaining training data with correct normal labels is not trivial. Fifth, it is troublesome when dealing with high dimensional data [8, 15].

3.2 Depth-based methods

To overcome shortcomings of distribution-based methods, depth-based methods was introduced. The main idea of depth-based methods is to arrange data into different layers in data space based on a particular depth concept [8]. The depth-based methods assume that normal data points lie in the centered layers while outliers often lie on border layers. For example, a depth concept for a 2-dimensional dataset could be defined as follow [8]:

- The data observations which lie on the convex hull of the input dataset have the depth 1

– Removing all depth-1 observations from the dataset and compute the convex hull for the rest of the dataset. The data observations which lie on the convex hull have the depth of 2.

– Similarly, it is possible to compute the depth for all data observations in the dataset.

After calculating the depth values for the dataset, outliers can be identified by setting a depth threshold k. Any data points which locates outside of the depth-k convex hull is considered as an outlier. The depth-based anomaly detection methods do not assume the distribution of the dataset. However, the major problem with this approach is that it is suitable for the low dimensional dataset with small number of observations. When the size of the dataset is big, it is expensive to compute the depth-k convex hull.

3.3 Distance-based methods

Similar to the depth-based approach, distanced-based algorithms do not assume the prior knowledge of the distribution of the data. The distanced-based methods rely on the distance measures to define distances among data observations. There are many such distance measures can be used. One of the common used is the Euclidean distance. Suppose two data points x and y of d dimensions. The Euclidean distance is defined as follow:

$$d(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

One of the first definitions for distanced-based outliers was proposed by Knorr and Ng [16]:

"An object O in a dataset T is a $DB(p, D)-outlier$ if at least fraction p of the objects in T lies greater than distance D from O."

There exists an equivalent variation of the definition above. Instead of declaring p as a fraction, it replaced p by an integer number k: "A point p in a data set is an outlier with respect to parameters p and d if no more than k points in the data set are at a distance of d or less from p" [17]. Knorr and Ng [16] introduced several methods together with optimizations using indexes and data tree structures to detect outliers. In summary, the authors concluded that the proposed methods do not scale well for high dimensional datasets. They are only suitable for datasets whose number of dimensions is less than or equal to 4. For the datasets with larger number of features, the simple nested loop join algorithm is the most appropriate one. It has the computational complexity in the worst case of $\mathcal{O}(dn^2)$ where d is the number of features and n is the number of data observations. Angiulli and Fassetti recently proposed DOLPHIN algorithm which could

deal with big datasets [18]. The algorithm requires two scans of the input data, one for building a special index data structure and the other to detect outliers.

Ramaswamy et al. [17] described some drawbacks of the aforementioned outlier definition. It is not easy to define the distance parameter D. It is efficient only when the dimension of data is low. Moreover, it does not provide a mechanism to rank the outliers based on their outlierness (the probability of being outliers). Hence, Ramaswamy et al. proposed a new definition of distanced-based outliers. Suppose $D^k(p)$ is the distance from a data point p to its k^{th} nearest neighbor. The outlier is defined as follow:

"Given an input data set with N points, parameters n and k, a point p is a D_n^k outlier if there are no more than $n-1$ other points p' such that $D^k(p') > D^k(p)$."

The intuitive idea is that outliers are points with high $D^k(p)$ distances. Outliers can be detected by picking l points which have highest $D^k(p)$ values. The authors of [17] developed a partition-based algorithm to identify top l outliers. This method does not require any distance threshold as in [16], but it requires to specify the number of outliers l and parameter k. The partitioned-based algorithm employs BIRCH clustering [19] to divide the input data set into partitions before computing the upper bound and lower bound values for the $D^k(p)$ in each partition. From that information, the anomaly detection algorithm is able to effectively prune non-outlier points; thus, reducing the computational time. During the execution, the algorithm, requires accessing data from neighbor partitions to prune non-outlier candidates from a particular partition. Therefore, it is hard to parallelize it in a shared nothing distributed system.

Bay and Schwabacher [20] used the similar definition of distance-based outliers to the one introduced in [17]. They proposed a solution called ORCA to combine pruning and randomization to speed up the simple nested-loop join algorithm. They theoretically analyzed the algorithm and show that its computation complexity in the worst case can go up $O(dn^2)$ where n is the number of data points, and d is the number of dimensions. However, the experimental results show that the proposed algorithm runs very fast in practice. The running time is nearly linear to the size of the data.

Ghoting et al. [21] proposed RBRP (Recursive Binning and Re-Projection), a two-phase algorithm to enhance the pruning technique described by Bay and Schwabacher [20]. In the first phase, data is partitioning into different bins. In each bin, the principle component analysis (PCA) is used to derive the principle component for all data points by projecting them to the axis with maximum variance. Data points within a bin are sorted base on their principle component. In the second phase, k nearest neighbors of a point are approximated by searching data points within the same bin. If the bin has less than k points, the search expands to the neighbor bins until k nearest neighbors are found. The algorithm is expected to have the complexity of $O(n \log nd)$ on a d-dimension dataset of n points in practice. However, the worst-case complexity is still $O(n^2 d)$.

In order to speed-up the computation of distanced-based outlier algorithms, some sampling-based techniques have been proposed with the idea to reduce the size of the dataset when computing the distance of a data point to its k^{th} nearest neighbors. Wu and Jermaine [22] introduced an iterative sampling method to approximate the results of the algorithm presented in [17]. Instead of comparing distances between a point and every other data points in the dataset to capture the k^{th} nearest neighbor distance, Wu and Jermaine create, for each data point p, a small dataset sampled from the input dataset. The distance to the k^{th} nearest neighbor is computed from p and its sampled dataset. Sugiyama and Borgwardt [23] proposed a one-time sampling method to detect outliers rapidly without drawing for each of the data points a sample dataset. The algorithm samples a dataset $DSample$ once, and the k^{th} nearest neighbor distance of every point is calculated against $DSample$ set.

Another definition variant of a distance-based outlier was presented by Angiulli and Pizzuti in [24]. The definition is similar to the one in [17]. The only difference is that instead of taking the distance to the k^{th} nearest neighbor as the measure to rank outliers as in [17], the sum of distances to k nearest points are used. The authors also proposed the HilOut algorithm to detect top l outliers with regard to their definition.

Chawla and Gionis [6] proposed a unified algorithm namely *k-means--* to cluster data and detect outliers simultaneously. The algorithm is based on *k-means* clustering algorithm [25]. The algorithm receives the parameters similar to *k-means* and one additional parameter l - the number of desired outliers. In each iteration of *k-means--*, after assigning points to their clusters, l points which have farthest distances to their centroids will be excluded before re-computing the centroids. At the end of the clustering process, top l points with distances farthest to their centroids are pointed as outliers. The proposed algorithm also suggests a rank on outlierness of the outliers based on distances to their nearest centroids.

Outlier definitions based on a distance metric suffers from the curse of dimensionality when applying to high dimensional data. When the number of dimensions of data becomes higher, distance measures become less effective to gauge how far from a data point to another data point [26]. In order to reduce the effect of the dimensionality curse, Kriegel et al. proposed a new approach to identify outliers namely Angle-Based Outlier Detection (ABOD) [27]. The angle of a data point A to a pair of other points B and C is computed by the scalar product of two vectors AB and AC weighted by lengths of the two vectors. The outlier definition of ABOD relies on the observation that the angle spectra of outliers are wider than those of normal data points. Thus, The algorithm has to compute for each data point an angle spectrum to every other pair of two points. The variance of the computed angles is then calculated and sorted. Outliers are data points with high values of angle variance. The computational complexity of ABOD is $O(n^3)$

which becomes very expensive if there is a large number of data points. The authors also proposed FastABOD algorithm which reduces the complexity down to $O(n^2 + nk^2)$. Instead of computing angles of a data point A to all possible pairs of other data points, $FastABOD$ only calculates angles of A to all pairs of data points within k nearest points of A. It is worth noting that the computational complexity in ABOD and FastABOD ignore the time for computing the angle between two vectors. It will take $O(d)$ to compute an angle (d is the dimension of data), resulting in the computational complexity of $O(dn^3)$ for ABOD and $O(dn^2 + dnk^2)$ for FastABOD. Pham and Pagh [28] proposed an improved version of ABDO which estimates the variance of an angle spectrum with a small probability of error. The experiments demonstrate that the proposed estimation algorithm has a competitive accuracy compared to the original ABOD while the running time is much faster and is almost independent of on size of the dimension.

3.4 Density-based methods

The assumption of the density-based approach is that an outlier is a point whose surrounding area has very few or no data points. The area around a point is formulated as a factor indicating how dense the area is. There are several algorithms have been proposed in the literature. Motivated by the drawbacks of the distance-based outlier definition proposed in [16] where it could not capture the outliers when data is divided into different clusters of different density. Breunig et al. [29] proposed a method called LOF (identifying density-based local outliers). The algorithm uses LOF factor to measure the outlierness of data points. The algorithm computes the LOF of each data point based on its distances to the nearest $MinPts$ neighbors ($MinPts$ is a given parameter). LOFs are the relative densities between data points and their neighbors. The outliers are points which have LOFs differ significantly from 1 which is the expected value for LOF of normal points. The running time of the algorithm depends on how to set the parameter $MinPts$ and how to execute the $MinPts$ nearest neighbor queries. As reported in the paper [29], the execution time is high even when the nearest neighbor queries are materialized to speed up the algorithm especially when the number of dimensions is greater than 10. The reason is that index structures could not handle very well the $MinPts$ nearest neighbor queries for high dimensional data. Tang et al. [30] presented Connectivity-based outlier factor (COF) as an improvement for LOF. The difference between COF and LOF is how to choose the nearest neighbors for a given point. LOF selects the k nearest neighbors based on distances to the point. COF incrementally builds the nearest neighbors set by adding points one by one. The point is chosen to add in each step is the one whose distance to the current nearest neighbors set is the smallest. Distance from a point to a set is minimum distance from the point any member of the set [7]. Another variant

of LOF is LOCI [26] which can reduce the number of input parameters by estimating the equivalent values from the input data itself.

3.5 Clustering-based methods

Clustering is an unsupervised technique to divide data into clusters of similar objects. Outliers which are very different from the rest of data are prone to affect the clustering quality. Therefore, many clustering algorithms are designed with the ability to detect outliers. BIRCH [19] is one of those algorithms. It reserves a certain amount of disk space for detecting outliers. ROCK algorithm [31] filters out outliers which have no links or very few links to other data points. DBSCAN [32] natively supports finding outliers. After the clustering process finishes, outliers are points which locate in a sparse region and do not belong to any cluster. *K-means--* proposed by Chawla and Gionis [6] described in section 3.3 is also a clustering-based algorithm. The general idea of detecting outliers using clustering approach is that outliers are calculated with respect to the clusters [7]. If a data point is different from all clusters, it will be identified as an outlier. The quality of clustering has significant influence on the accuracy of outlier detections.

3.6 Different feature types

So far, most of the approaches we introduced can only work with numerical attributes. However, in the real world, there many scenarios where the data have only categorical attributes or both numerical and categorical attributes. A number of algorithms with different approaches have been proposed to cope with the non-numeric features. He et al. [33] proposed a local search local-search heuristic (LSA) algorithm which is based on the information entropy of input data. An enhanced version of LSA with greedy search technique was also introduced in [34]. Otey et al. [35] presented a distributed algorithm which could work on datasets with mixed attributes. The algorithm relies on the frequency of categorical attributes and covariance matrix of numerical attributes. Attribute Value Frequency(AVF) [36] is another algorithm to detect outliers on categorical datasets based on the frequency of attributes. Akoglu et al. [37] used a dictionary compression technique to identify outliers. The algorithm is free of parameters except the number of desired outliers.

3.7 Distributed algorithms

A number of algorithms have been proposed in distributed settings in order to speed up the detection of outliers on large datasets. Some algorithms are based on the approaches

introduced previous sections. Koufakou et al. [38] propose the distributed version of AVF [36] called MR-AVF which employs the power of Hadoop/MapReduce. The algorithm only works with categorical datasets. The algorithm devised by Otey et al. [35] defines outliers based on the support concept in frequent itemset mining. It is implemented using Message Passing Interface (MPI) for communication between processors. PENL algorithm was proposed by Hung et al. [39] to detect outliers which are defined in [16]. The definition does not provide a mechanism to calculate scores for outliers. Therefore, it is not suitable for getting top l outliers. Moreover, the implementation of PENL requires communication between parallel processors. Lozano et al. [40] presented parallel implementations for ORCA [20] and LOF [29]. Both algorithms require data to be centralized and use the MPI to support parallel programming. The algorithms also require communications between processors involving in the computation. Dutta et al. [41] proposed a distributed algorithm for detecting outliers using Principal Component Analysis. The algorithm focuses specifically on the astronomy domain. Angiulli et al. [42] presented a distributed algorithm for mining top l outliers. The algorithm is based on the outlier definition described in [24]. The algorithm needs to compute distances from all points to their nearest k^{th} neighbors. it approximates the results by sampling a solving set from the input dataset. The solving set is then used to calculate parallelly the k^{th} nearest neighbors for all data points in parallel. There is a supervisor node which coordinates and communicates with involving processors through TCP sockets. Each processor computes its local k^{th} nearest neighbors and sends the result back to the supervisor node.

Chapter 4

IMPLEMENTATION

The *k-means--* algorithm is formulated based on the *k-means* clustering algorithm. Therefore, in this section, we first describe the *k-means* algorithm followed by the formal definition of *k-means--* algorithm. We also introduce a simple and intuitive algorithm to detect outliers based on *k-means* called *k-meansOD*. The algorithm is used as the baseline to compare with *k-means--* algorithm in the evaluation section. Finally, we discuss both implementations on a single machine and distributed implementations on Flink of *k-means--* and *k-meansOD* algorithms.

4.1 The *k-means* clustering algorithm

Clustering is one of the fundamental problems in data mining. It refers to the unsupervised task of partitioning similar objects in an input dataset into different clusters. The task can be accomplished by various algorithms among which *k-means* is one of the most popular clustering methods [43]. The *k-means* algorithm divides the input data points into k groups of similar points. Each group has a representative centroid which is the mean of all group's members. The goal of the *k-means* algorithm is to minimize the sum of squared distances from all points in the input dataset to their respective centroids. Although the algorithm could not guarantee the global optimal answer, *k-means* can find a local optimal solution by implementing the local search technique proposed by Lloyd [44] which is also known as Lloyd's algorithm. The *k-means* we are referring here is the realization of the Lloyd's algorithm. It can theoretically have exponential computational complexity [45, 46] in the worst case. However, the algorithm is very fast in practice.

Algorithm 1 shows the pseudo-code of *k-means*. The algorithm receives multiple input parameters. Points in the dataset D have the same number of dimensions as initial centroids in the set C. The eps and $maxIterations$ define the algorithm's stopping criteria. The algorithm will stop if the difference of the sum of squared distances from all points to their nearest centroids is smaller than eps or $maxIterations$ iterations have been already executed. In each iteration, there are two main computations. The first computation starts from line 10. For each point, the squared distances from it to all centroids

Algorithm 1 k-means algorithm

Input: Input dataset D of points with d dimensions, a set C of k initial centroids, a stopping threshold eps, maximum number of iterations $maxIterations$

Output: A set of final centroids

1: $prevSumDistances \leftarrow eps + 1$
2: $sumDistances \leftarrow 0$
3: $iteration \leftarrow 0$
4: $clusters \leftarrow array(|D|)$ ▷ array size |D| of 2-tuple($\mathbb{R}^d, \mathbb{N}$)
5: **while** $prevSumDistances - sumDistances >= eps\ AND\ iteration < maxIterations$ **do**
6: $iteration \leftarrow iteration + 1$
7: $prevSumDistances \leftarrow sumDistances$
8: $sumDistances \leftarrow 0$
9: reset all element of $clusters$ to ($\vec{0}$, 0)
10: **for** $p_i \in D$ **do** ▷ Assign points to the nearest respective clusters
11: $minDist \leftarrow +\infty$
12: $cid \leftarrow 1$
13: **for** $c_j \in C$ **do**
14: **if** $squaredDist(p_i, c_j) < minDist$ **then**
15: $minDist = squaredDist(p_i, c_j)$
16: $cid \leftarrow j$
17: **end if**
18: **end for**
19: $clusters[cid].sum \leftarrow clusters[cid].sum + p_i$
20: $clusters[cid].size \leftarrow clusters[cid].size + 1$ ▷ add one point to the cluster
21: $sumDistances \leftarrow sumDistances + minDist$
22: **end for**
23: **for** $j \in \{1, ..., k\}$ **do** ▷ Re-computing centroids
24: reset all dimensions of c_j to 0
25: $c_j \leftarrow \dfrac{clusters[j].sum}{clusters[j].size}$
26: **end for**
27: **end while**

are calculated. It is then assigned to a cluster whose centroid is the nearest. The $clusters$ variable stores information about all clusters. The $clusters[i].sum$ is the cumulative sum of point members within $cluster_i$ and $clusters[i].size$ is the number of points in $cluster_i$. In the second main computation, centroids need to be re-computed due to the new assignment of points in the first computation. The $centroid_i$ is easily obtained by dividing the sum $cluster[i].sum$ by number of points $cluster[i].size$ as shown in line 25.

4.2 The *k-means--* algorithm

The *k-means--* algorithm can be considered as the generalization of *k-means*. The difference between the two algorithms is that *k-means--* does both clustering and detecting outliers simultaneously. The algorithm receives additional parameter l compared to *k-means*. The parameter defines how many outliers should be extracted from the input dataset. The *k-means--* algorithm minimizes the sum of squared distances from all points except l outliers to their nearest centroids. When l is set to 0, *k-means--* becomes the classic *k-means* algorithm. *K-means--* detects outliers based on the assumption that outliers are points lying far from the clusters. Similar to *k-means*, the algorithm does not guarantee the global optimal solution, but it has been proved to be able to find a local optimal solution [6]. Given similar input parameters as *k-means* plus parameter l, the *k-means--* can be described in following steps:

1. Assign all points to their nearest clusters.
2. Sort all points in the decreasing order of distances to their nearest clusters.
3. Remove first l points from the sorted list in step 2.
4. Recompute the centroids of all clusters based on the remaining points after step 3.
5. Repeat step 1 to 4 until reaching stopping criteria.

The *k-means--* algorithm works similarly to the *k-means*. In the first step, it assigns all points in the input dataset to their nearest clusters. After step 2 and 3, the algorithm removes l points which have highest distances to their centroids. In step 4, it re-computes the centroids based on the remaining points. Note that in step 1 and 2, all points in the input dataset (including those are removed in step 3 from the previous iteration) are used. The l points eliminated in step 3 in the last iteration will become outliers.

4.3 Implementations on a single machine

4.3.1 Implementation of *k-means--* on a single machine

In this section, we describe the detailed implementation of *k-means--* on a single machine. From the definition and steps of *k-means--* presented in section 4.2, we observe that:

– In each iteration, the algorithm removes top l points from the sorted list of distances in step 3. It could happen that the list has duplicate values. Therefore, the list of remaining points might have points whose distances to their nearest centroids equal to those distances of l removed points. We think that if two points have

the same distances to their centroids, they should be treated similarly i.e. either removing them or keeping them from the sorted list of distances.

– Sorting all points requires the complexity of $O(n \log n)$ where n is the number of points. If the algorithm runs in i iterations, the total additional complexity for sorting will be $O(in \log n)$. This is expensive if both i and n are large.

We suggest to make a modification that could handle well both of those issues. Instead of strictly removing l points before computing centroids, we compute a distance threshold to filter out potential outliers in each iteration. After sorting all points in decreasing order of distances to their respective centroids, the distance threshold is equal to the l^{th} highest value in the sorted list. All points whose distances are greater than or equal to the threshold will be filtered out. The remaining points are used for computing centroids. It is evident that at least l points will be removed. Furthermore, points which have same distances to their centroids will be either removed or kept for computing centroids. Regarding the computational complexity, our modification exploits a min heap to keep track of l^{th} highest distance. Given n is the number of points in the datasets, in each iteration, there will be at most n insert operations into the min heap. Therefore, the worst case complexity to find l^{th} highest distances is $O(n \log l)$. However, in practice, the number of operations are much lower because distances are only added to the min heap if they are greater than the heap's min value and the heap only keeps at most l elements.

The pseudo code of the modified version is presented in Algorithm 2. Similar to *k-means*, it consists of two main phases. In the first step, points are assigned to their nearest clusters. During the assignment starting at line 5, the squared distance from a point p_i to its nearest cluster is also stored in the $pointInfo[i].minDist$ variable. The distance is then added to a min heap of size l. The distance is inserted only if the heap has less than l elements or the distance is greater than the current minimum value of the heap. After adding the distance, if the heap's size exceeds l, the minimum value is removed. After all points are assigned to their clusters. It is clear that the heap's minimum value is the l^{th} biggest value among all nearest distances from points to their centroids. The minimum value of the heap is used as the distance threshold to filter out points with big distances. The second phase begins at line 30. Points whose distances to their nearest centroids are small than the distance threshold are used to compute the centroids. The rest of points are marked as the outliers for the current iteration. The algorithm stops when it reaches one of the stopping criteria. The first one is when the number of executed iterations exceeds a constant $maxIterations$. The second one is when the difference between the sum of squared distances of all points to their clusters between two consecutive iterations is smaller than the threshold eps. The points which are filtered out in the last iteration become the final outliers.

Algorithm 2 *k-means--* algorithm

Input: Input dataset D of points $\in \mathbb{R}^d$, a set C of k initial centroids, a stopping threshold eps, maximum number of iterations $maxIterations$, number of outliers l
Output: A set O of l outliers

$pointInfo \leftarrow array(|D|)$ ▷ array size |D| of 3-tuple($\mathbb{R}^d, \mathbb{N}, \mathbb{R}$,)
while stopping criteria are not satisfied **do**
  reset all element of $clusters$ to ($\vec{0}$, 0, 0)
  $minHeap \leftarrow$ new empty min heap
  for $p_i \in D$ **do** ▷ Assign points to the nearest respective clusters
    $minDist \leftarrow +\infty, cid \leftarrow 1$
    for $c_j \in C$ **do**
      if $squaredDist(p_i, c_j) < minDist$ **then**
        $minDist = squaredDist(p_i, c_j)$
        $cid \leftarrow j$
      end if
    end for
    $pointInfo[i].point \leftarrow p_i$
    $pointInfo[i].centroidId \leftarrow cid$
    $pointInfo[i].minDist \leftarrow minDist$
    if $minHeap.size < l \; OR \; minHeap.min < minDist$ **then**
      $minHeap.add(minDist)$
    end if
    if $minHeap.size > l$ **then** ▷ Keep only l highest values
      remove min element in $minHeap$
    end if
  end for
  $threshold \leftarrow minHeap.min$
  for $c_j \in C$ **do**
    reset all dimensions of c_j to 0
  end for
  $numPoints \leftarrow array(\mathbb{N})$ ▷ Array of k integer
  reset all element of $numPoints$ to 0
  $O \leftarrow \{\}$ ▷ empty set of outliers
  for $point \in pointInfo$ **do** ▷ Re-computing centroids
    if $point.minDist < threshold$ **then**
      $j \leftarrow point.centroidId$
      $c_j \leftarrow c_j + point.point$
      $numPoints_j \leftarrow numPoints_j + 1$
    else
      $O = O \cup point.point$ ▷ Current point is an outlier
    end if
  end for
  for $c_j \in C$ **do** ▷ Final divisions to compute mean values
    $c_j = \dfrac{c_j}{numPoints_j}$
  end for
end while

4.3.2 Implementation of the *k-meansOD* algorithm on a single machine

In this section, we introduce an algorithm for outlier detection based on the original *k-means* algorithm. In order to distinguish with *k-means*, we call the algorithm for identifying outliers *k-meansOD* (*k-means* for outlier detection). The algorithm relies on the very simple and intuitive heuristic: outliers are the farthest points from their centroids after clustering using *k-means*. The algorithm consists of two phases. The first one is clustering the input data using *k-means*, the second phase is computing outliers. The *k-meansOD* is the combination of the *k-means* and *k-means--* algorithms. It uses the clustering technique of *k-means* and computes the outliers in the same manner as *k-means--* does. The *k-meansOD* algorithm receives similar input parameters as the *k-means--* and outputs top l outliers. The algorithm's steps are described as follow:

1. Clustering the input data points using *k-means* algorithm.
2. Compute the distance threshold as l^{th} biggest value from distances of all points to their centroids.
3. Outliers are points whose distances to their centroids are greater than or equal to the distance threshold.

It is trivial to implement the *k-meansOD* based on the implementations of the *k-means* and *k-means--*. In the first phase, we can use the implementation of *k-means* to cluster the input data. After having the centroids, we can use the implementation of *k-means--* to run one iteration to detect outliers.

4.4 Implementations in Flink

In this section, we describe our implementations of *k-means--* and *k-meansOD* in Apache Flink. We have introduced in section 2.2.3 that Apache Flink is a suitable dataflow system for iterative algorithms. In an iterative Flink program, it defines the step function as the piece of code which is repeatedly executed. The step function is wrapped into a special iteration operator which executes the step function until the program reaches a stopping criterion. A stopping criterion is either maximum number of iterations or a custom convergence criterion registered by users. Flink supports two type of iteration operators whose names are *Bulk Iteration* and *Delta Iteration*[1]. In our context, the *k-means--* can be implemented using the *Bulk Iteration*.

[1] https://ci.apache.org/projects/flink/flink-docs-release-1.0/apis/batch/iterations.html

4.4.1 Flink implementation of *k-means--*

The parallel implementation of *k-means--* is based on the technique described by Zhao et al. in [47]. Although the method was proposed specifically for *k-means* under MapReduce, it is adaptable for *k-means--* because both of the algorithms share many similar calculations. Our implementation of *k-means--* in Flink is inspired by the example implementation of *k-means* in Flink[2]. The bottleneck when implementing *k-means--* in Flink is computing the distance threshold which is later used to filter points before re-computing centroids. In Flink, a *Bulk Iteration* program only guarantees the correct results upon finishing all iterations. Moreover, Flink currently does not support materializing intermediate results after each iteration. Therefore we were not able to compute the distance threshold and store it in a primitive variable. Fortunately, we have a workaround by keeping the threshold under a Flink DataSet[3] variable which contains only one value. We can broadcast the variable to successor operators to filter points with highest distances to their centroids. The Flink implementation of *k-means--* consists of two main functions. COMPUTECENTROID function is for computing the centroids of clusters and COMPUTEOUTLIERS function detects final outliers given that the centroids have been calculated. The implementation of the algorithm is shown in Algorithm 3.

Algorithm 3 *k-means-- in Flink*

```
1:  function ComputeOutliers(points, centroids, maxIterations, l)
2:      centroids ←ComputeCentroid(points, centroids, maxIterations, l)
3:      assign ← points.map().withBroadcastSet(centroids)
4:      threshold ← assign.mapPartition().groupReduce()
5:      outliers ← assign.filter().withBroadcastSet(threshold)
6:      return outliers
7:  end function

8:  function ComputeCentroid(points, centroids, maxIterations, l)
9:      finalCentroids ← centroids.iterate(maxIterations) {
10:         assign ← points.map().withBroadcastSet(centroids)
11:         threshold ← assign.mapPartition().groupReduce()
12:         remaningPoints ← assign.filter().withBroadcastSet(threshold)
13:         newCentroids ← remaningPoints.groupBy(centroid).reduce()
14:     }
15:     return newCentroids
16: end function
```

[2] https://github.com/apache/flink/blob/master/flink-examples/flink-examples-batch/src/main/scala/org/apache/flink/examples/scala/clustering/KMeans.scala

[3] https://ci.apache.org/projects/flink/flink-docs-master/api/java/org/apache/flink/api/java/DataSet.html

In the COMPUTECENTROID function, the program executes not more than $maxIterations$ iterations. In each iteration the following operations are executed in order:

- In line 10: Flink creates different parallel map tasks and assigns them to available task slots. Tasks are executed parallelly depending on the parallelism value. The centroids are broadcasted to all tasks and each task process different portion of data points. Each task receives a point as the parameter, computes its nearest centroid and outputs a tuple with information about the nearest centroid, the point and the distance from the point to the centroid. After this step, all points are assigned to their nearest clusters.

- In line 11, we use a $mapPartition$ and a $groupReduce$ operator to compute the l^{th} biggest value of distances from all points to their centroids.

  - Each $mapPartition$ task receives a list of tuples (computed in line 10) and extracts top l largest distances. We use a min heap to keep at most top l distances similar to what we do with the min heap of the single implementation in section 4.3.1. Distances are inserted into the heap one by one. The minimum value will be removed from the heap if its size exceeds l. Finally, the heap contains l highest distances which are the results of the $mapPartition$ task.
  - The outputs of all $mapParition$ tasks are fed to the $groupReduce$ task. There is only one $groupReduce$ task because we do not group by any key. We again use a min heap to store top l biggest distances. The output of the $groupReduce$ task is the single value - the minimum distance in the heap. This is exactly the value of l^{th} biggest values from distances of all points to their centroids. After this step, we have the distance threshold to filter points in the next step.

- In line 12 we employ the $filter$ operator to eliminate all points whose distances to their centroids are greater than or equal to the threshold we have computed in the previous step.

- In line 13 The remaining points from the previous step are grouped based on their centroids into different clusters. All points in a cluster must have the same centroid and all points have the same centroids must belong to one cluster. The $reduce$ operator is exploited to recompute the centroid for each of the clusters.

In the COMPUTEOUTLIERS function, the execution flow is as follow

- The final centroids are calculated by calling the function COMPUTECENTROID

- The map operator is employed to assign all points to their clusters. The map tasks work exactly similar to what has been described at line 10 in the COMPUTECENTROID function.

- The $mapPartition$ and $groupReduce$ are used to compute the distance threshold. This step is similar to what happens at line 11 in the COMPUTECENTROID function.
- The outliers are identified by a $filter$ operator. Because we are detecting the outliers, we keep all points whose distances to their centroids are greater than or equal to the distance threshold.

It is worth mentioning that Apache Flink implicitly parallelizes its programs based on the parallelism parameter. If the parallelism is set to X, the operators in our implementation such as map, $reduce$, $filter$ and $mapPartition$ will create X different independently tasks which could be executed in concurrently. That is how our implementation benefits from the parallel computation.

4.4.2 Flink implementation of *k-meansOD*

In this section, we will describe how we implement *k-meansOD* in Apache Flink. Similar to the implementation of *k-means--*, we also have two functions to calculate the centroids and identify outliers. For the first function shown in Algorithm 4, it is simpler than the one of *k-means--*. We only need two steps of assigning points to their clusters and recomputing centroids. Both of them work exactly similar to the corresponding parts in the COMPUTECENTROID function of the Flink implementation of *k-means--*. For the second function for detecting outliers, we can completely re-use the code of COMPUTEOUTLIERS function in Algorithm 3.

Algorithm 4 *Function to compute centroids of k-meansOD in Flink*

```
1: function COMPUTECENTROID(points, centroids, maxIterations)
2:     finalCentroids ← centroids.iterate(maxIterations) {
3:         assign ← points.map().withBroadcastSet(centroids)
4:         newCentroids ← assign.groupBy(centroid).reduce()
5:     }
6:     return newCentroids
7: end function
```

Chapter 5

EVALUATION

In this section, we will evaluate the performance of the *k-means--* algorithm. We set up two environments for testing. The experiments in the local environment are designed to examine the efficiency while experiments in the cluster environment investigate the scalability of the algorithm. We also compare *k-means--* with different distance-based outlier detection algorithms. In this section, we use Euclidean distance (also known as $L^2 - norm$) as the distance measure.

5.1 Local evaluation

The local evaluation is designed to test how well *k-means--* can identify outliers. The datasets in the experiments are small enough to be able to run on a single computer. First, we test the algorithm with the synthesis datasets and discuss some aspects of the algorithm. We also conduct the experiments to check how the algorithm performs on some real datasets from UCI Machine Learning Repository [48]. Finally, we investigate the computation time and the clustering ability of *k-means--*. All experiments in this section are conducted on a single laptop machine with the following specifications:

- Intel Core i7-4510U CPU @2.00GHz 2.6GHz
- 8GB of RAM
- HDD 5400RPM
- Softwares: Oracle Java 8, Scala 2.10.4

5.1.1 Synthesis datasets

We use a data generator implemented in R [49] to generate four synthesis datasets namely *DS1*, *DS2*, *DS3* and *DS4*. The R package implements the algorithm proposed in [50]. The R package also contains a visualization method for generated data in 2-dimensional space which we employ to plot some figures in this thesis. The data generator is able to generate random clusters with different parameters among which we are interested in:

- *sepVal*: The degree of separation among clusters of points. This parameter controls how the generated clusters are far from others. The value of $sepVal$ is in the range (-1, 1). If $sepVal$ is high, clusters are far from others. If $sepVal$ is small, clusters are overlapped.
- *numNonNoisy*: The number of dimensions (features) of data points
- *numOutliers*: Number of outliers. The outliers are generated following a uniform distribution on each of the features.
- *rangeN*: The range of cluster population. Each cluster has its population within the given range. In the experiments, we generate clusters with equal population.
- *numClusters*: The number desired clusters

We notice that shapes of the clusters generated by the data generator are convex. Therefore, we use an additional *DS5* dataset whose shapes of its clusters are circular and half-circular. We employed the available API from *scikit-learn* [51] to create the *DS5* dataset. The outliers in the dataset are uniformly distributed.

The details of synthesis datasets we use in the local experiments are summarized in Table 5.1. In all experiments, the number of outliers returned by the *k-means--* algorithm is equal to the number of true outliers of the datasets. The eps parameter of *k-means--* is set to 1E-9.

TABLE 5.1: Synthesis datasets in local experiments

Dataset	#clusters	#outliers	#features	sepVal	#points
DS1	3	6	2	-0.7	309
DS2	5	20	2	0.3	1023
DS3	5	20	2	-0.03	1021
DS4	5	20	2	0.3	1022
DS5	4	20	2	-	1020

5.1.1.1 Clusters with low degree of separation

The aim of this experiment is to test how the algorithm works in the case clusters are not clearly separated. Figure 5.1 visualizes the *DS1* dataset and the results produced by *k-means--* algorithm on *DS1*. The outliers are plotted by red color in all three figures. We ran the algorithm multiple times, each with different random sets of three centroids sampled from the original points in the dataset. We observed that the algorithm either detected two or three true outliers in each run after ten runs. When clusters are not distinctly separated, *k-means--* can only identify outliers near the border of the dense region where

TABLE 5.2: Results of 10 runs on *DS2*

Run	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**
True outliers detected	12	1	12	12	12	1	12	12	12	12

data points locate. The outliers which are inside a cluster or in the middle of some clusters are miss-classified as clusters' members by the *k-means--* algorithm.

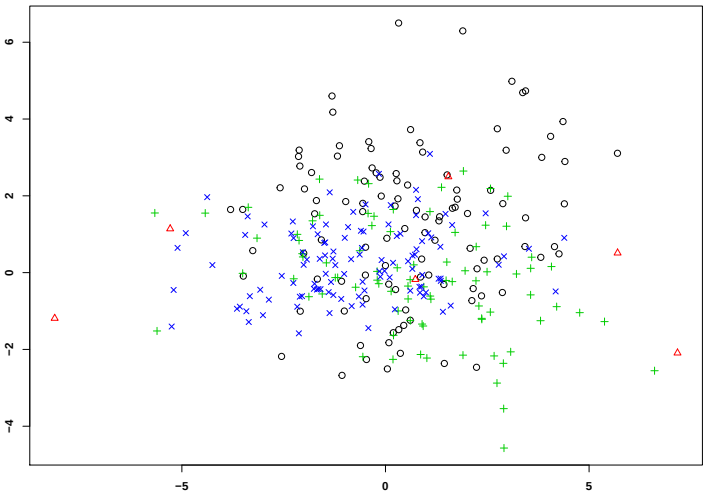
5.1.1.2 Clusters with high degree separation

In this test, we generated the *DS2* whose clusters are far from others. The dataset is plotted in Figure 5.2. Similar to the experiment on *DS1*, we ran the *k-means--* algorithm on the *DS2* dataset ten consecutive times and reported the number of true outliers detected in Table 5.2. The algorithm identified only one real outlier over the total of 20 twice while it had eight times successfully finding 12 true outliers. The results are illustrated in Figure 5.2. In the case when only one outlier was detected, the result clusters were not the optimal clusters. The poor choice of initial centroids led to the wrong splitting of one cluster into two and the merging of two remote clusters into one. This is a well-known problem to the *k-mean* clustering algorithm and *k-means--* has the same issue. In the case when data points are clustered correctly by *k-means--*, majority of the outliers were detected and they all located near the border of the clusters. This experiment has shown that the *k-means--* works generally well in the case where clusters are highly separated, however, the results are sensitive to the selection of the initial centroids.

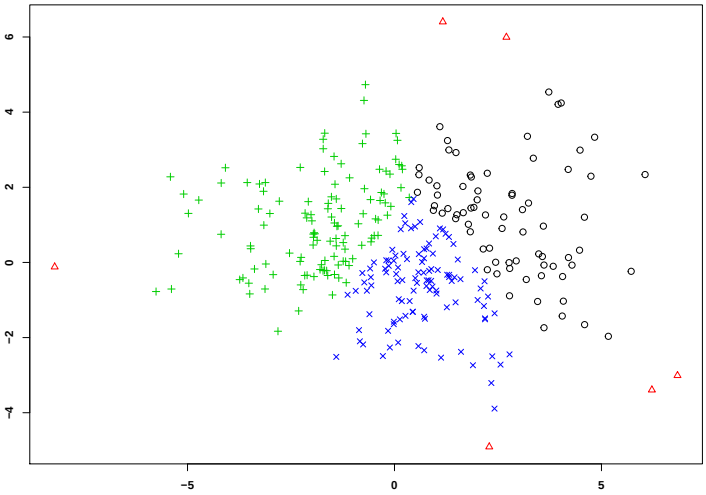
5.1.1.3 Different number of clusters

In this section, we experiment the *k-means--* under different values of initial number of clusters. We employ the same procedure as previous experiments where we ran the algorithm ten times and recorded the number of correctly recognized outliers. We experimented on five datasets presented in Table 5.1. The *DS1* and *DS3* datasets have low degrees of separation while *DS2* and *DS4* datasets have high degrees of separation. The *DS5* dataset consists of four concave clusters. The clusters in both *DS3* and *DS4* datasets have both spherical and elliptical shapes compared to mostly spherical clusters as in *DS1* and *DS2* datasets. The original distribution of points into clusters of the *DS3*, *DS4* and *DS5* datasets are shown in Figure 5.3.

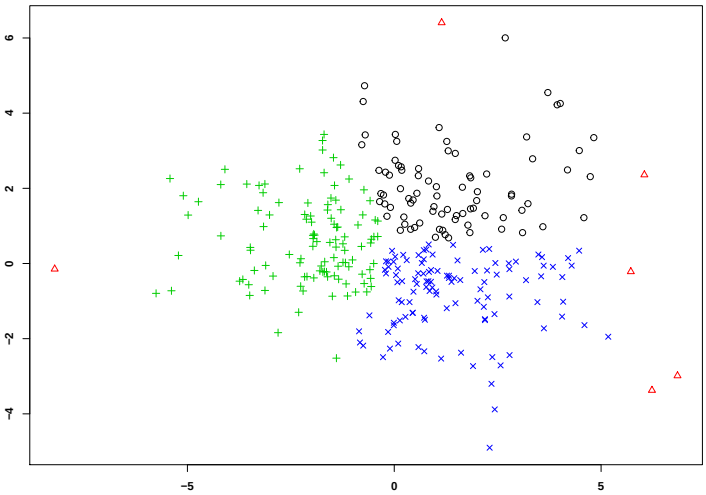
For this experiment, we ran the *k-means--* with different numbers of clusters. For each number of clusters, we reported the average number of true outliers detected after ten runs. The initial centroids were randomly drawn from the input datasets. The results are plotted in Figure 5.4.



(A) The generated clusters and outliers

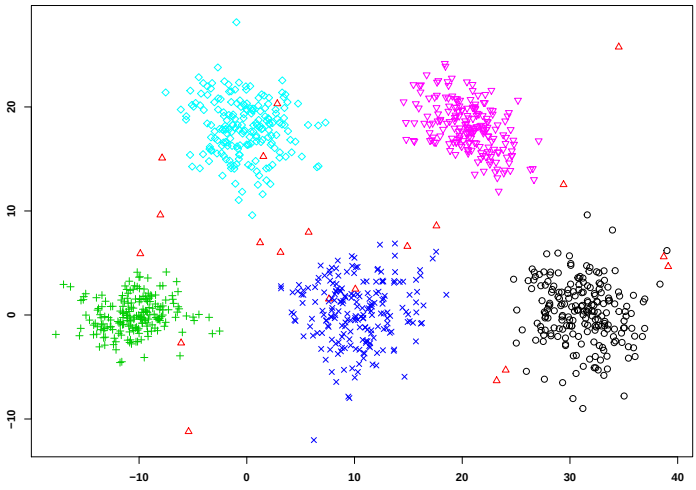


(B) 2 true outliers detected

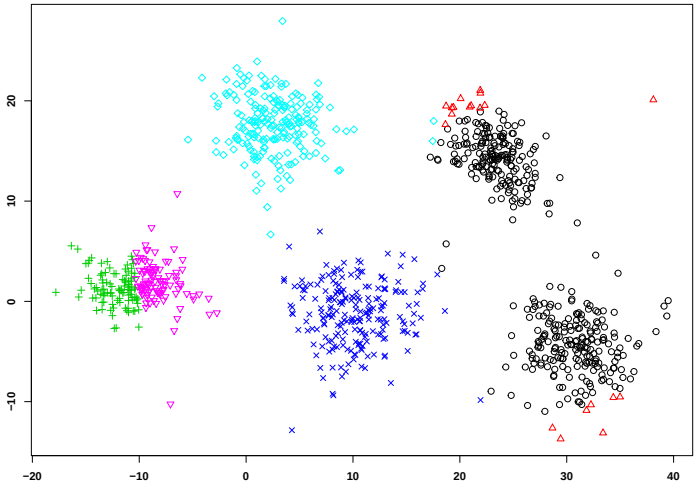


(C) 3 true outliers detected

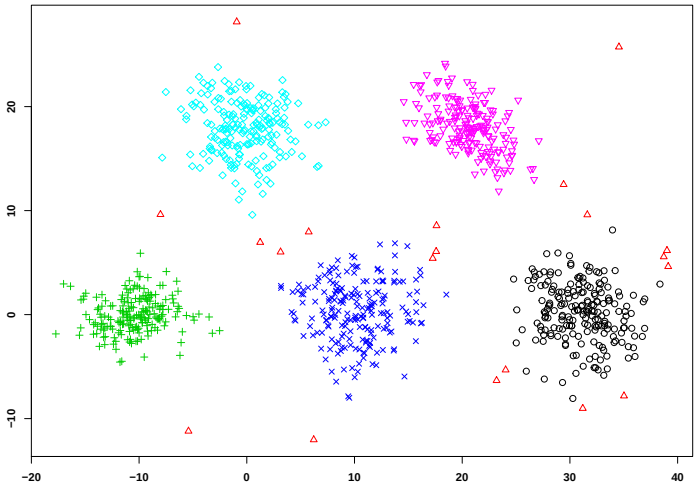
FIGURE 5.1: *DS1* dataset and results of *k-means--*



(A) The generated clusters and outliers

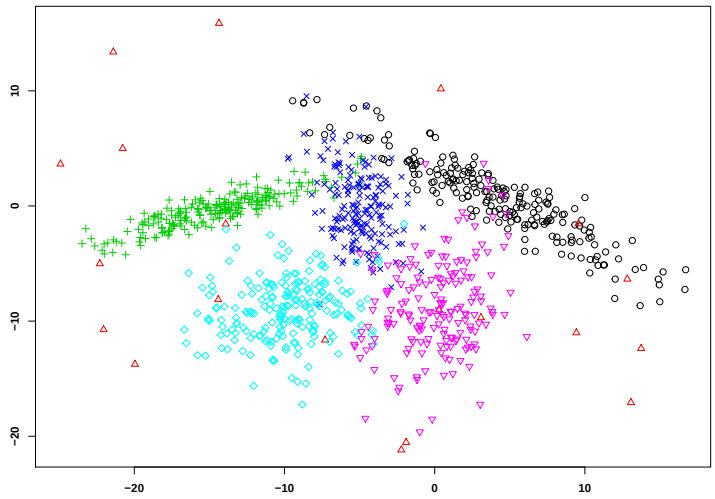


(B) 1 true outlier detected

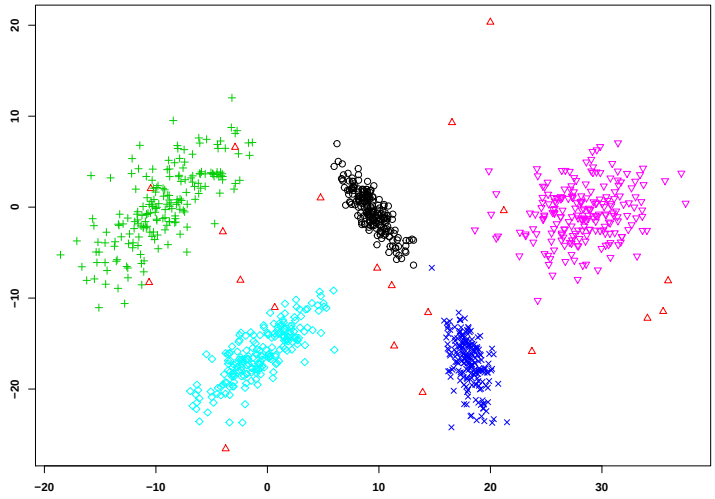


(C) 12 true outliers detected

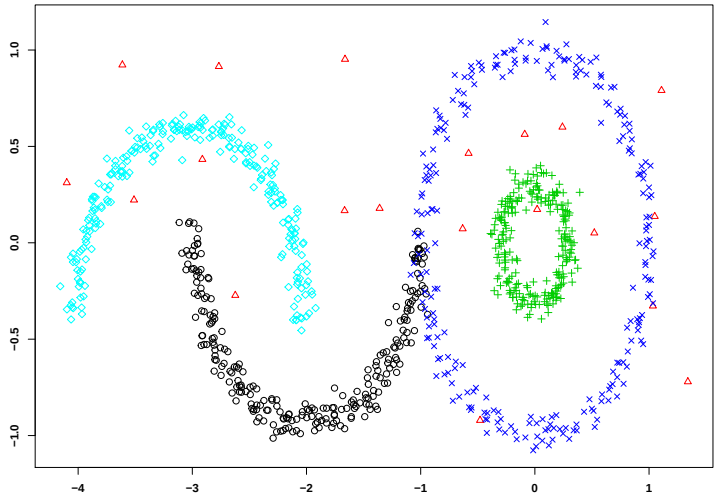
FIGURE 5.2: *DS2* dataset and results. Outliers are red points



(A) *DS3*



(B) *DS4*



(C) *DS5*

FIGURE 5.3: Visualization of *DS3*, *DS4* and *DS5* datasets. Outliers are red points

– For the *DS1* and *DS3* datasets with small degrees of separation: We can see that the number of correctly recognized outliers are almost stable to the increase in the number of clusters. This can be explained by the fact that when the clusters are closer to others, they create a dense region of points. Outliers are points locates far from the dense area. Therefore, when increasing the number of clusters, the dense region are divided into different clusters, but the centroids still lie within the dense region. The distance between a point in the dense region and its nearest centroid becomes smaller; Hence, the point is less likely to be marked as outliers. In other words, it is unlikely that more points are marked as outliers; Therefore, the number of detected outliers are stable.

– For the *DS2* and *DS4* datasets with high degrees of separation: The number of real outliers detected increases as the number of clusters get bigger. On the one hand, when the number of clusters is small, some clusters are merged to make larger clusters. Since the true clusters are highly separated, it could happen the case where the centroids of the large clusters locate in spare areas of points in the middle of some true clusters. Therefore, the centroids are nearer to the true outliers than the normal points. On the other hand, when the number of clusters increases, some clusters are divided into smaller clusters with their centroids locate in the middle of the true clusters. Distances from outliers which are in spare areas of points among clusters to their nearest centroids become larger. Thus, more true outliers are identified.

– For the *DS5* dataset with clusters having concave shapes: The result was bad even if we set the number of clusters twice as large as the number of true clusters. However, when the number of clusters was set to adequately big, the number of true outliers detected increased. The reason is that when the number of clusters is small, the centroids are likely to be in spare regions of points. This makes the distances between outliers in the spare regions to their centroids closer than distances from the centroids to the respective members of the clusters. Therefore, the outliers are not detected. When the number of clusters is sufficiently high, many small clusters are formed. In this case, the centroids of the smaller clusters have high chances to lie in dense regions of points, making more outliers in spare regions to be detected.

The results of the experiment show that the number of initial clusters has less influence to the performance of *k-means--* algorithm on the datasets with low degrees of separation. For other datasets, the higher value of number of clusters was, the better results the algorithm achieved. In general, one can choose high number of clusters to achieve good number of true detected outliers by *k-means--*.

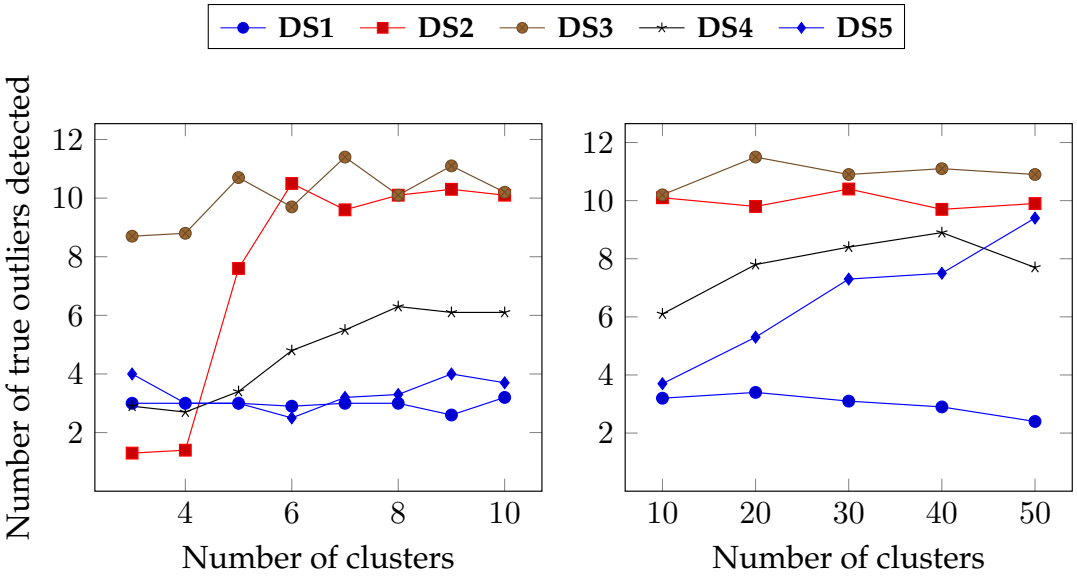


FIGURE 5.4: Number of true outliers detected for varying number of clusters

5.1.1.4 Comparing with other distance-based outlier detection algorithms

In this experiment, we compare the *k-means--* algorithm with two different distance-based algorithms. Both of them are able to define how many outliers that should be returned.

- **kNN** algorithm [17] computes the distance from each point to its k^{th} nearest neighbor. Top l outliers are points which have highest distances to their k^{th} nearest neighbors.

- **k-meansOD** algorithm uses a simple and intuitive outlier detection method based on *k-means* clustering algorithm. The *k-meansOD* algorithm requires the similar set of parameters to the *k-means--* algorithm. In the experiments, we used the same parameters for both *k-means--* and *k-meansOD*. The *k-meansOD* is described in section 4.3.2.

The *kNN* algorithm requires two parameters k and l. The parameter k indicates the nearest neighbor to be considered, and l is the number of desired outliers. We set parameters l similar to the number of desired outliers of *k-means--* algorithm. Because the result of *kNN* algorithm depends on the selection of k, we did an experiment with various values of k to find the suitable values. The results of the experiment shown in Figure 5.5 suggest that k should not be chosen too high as it would reduce the number of correct outliers detected. For this experiment, in order to compare to *k-means--* and *k-meansOD*, we averaged number of correct outliers detected in five runs for five smallest values of k ($k \in \{3, 4, 5, 6, 7\}$).

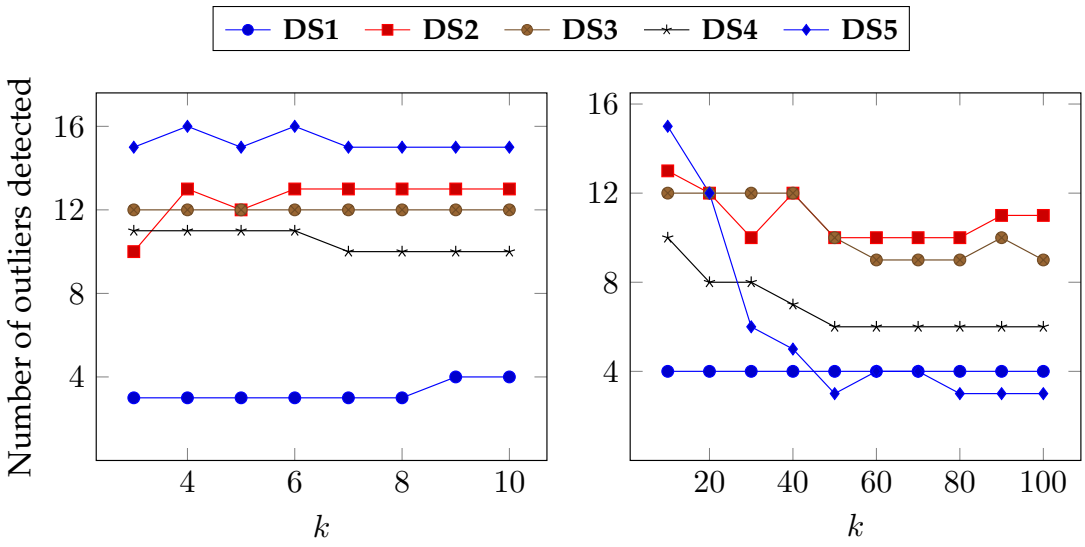


FIGURE 5.5: Number of outliers true detected for varying k

For the *k-means--* algorithm, we choose a high value for the number of clusters as suggested empirical results in our previous experiments. We set the number of clusters to 50 for all datasets presented in Table 5.1. Figure 5.6 shows the comparison results of three algorithms: *k-means--*, *k-meansOD* and *kNN*. For all five datasets, the *kNN* was able to detect more true outliers than both *k-means--* and *k-means*. The *k-means--* was better than the *k-means* algorithm in four datasets and only worse on *DS5*. We inspected deeper into the result of *k-means--* on *DS5* and noticed that its performance could be better if we set the number of clusters higher than 50. For example, the average number of true outliers detected could go up to 13.2 when we set the number of clusters to as high as 100. For the *k-meansOD* algorithm, the average number of true outliers detected started decreasing when the number of clusters was greater than 110. The detailed comparison between *k-means--* and *k-meansOD* is shown in Figure 5.7. The results of the experiment demonstrate that *k-means--* is more stable than *k-meansOD* when varying the number of clusters. Moreover, the number of outliers detected of *k-means--* is generally better.

5.1.2 Real datasets

In this section, we experiment the *k-means--* algorithm with some real datasets. We used some classification datasets from UCI Machine Learning Repository [48] because they have information about classes to which data points belong. Therefore, we could measure the correctness of the *k-means--* and compare it with the aforementioned *k-meansOD* and *kNN* algorithms. Additionally, we also report the running time of the three algorithms and compare the clustering quality of *k-means--* and the classic k-means clustering algorithm. We assume that, for each dataset, classes having the majority of points are

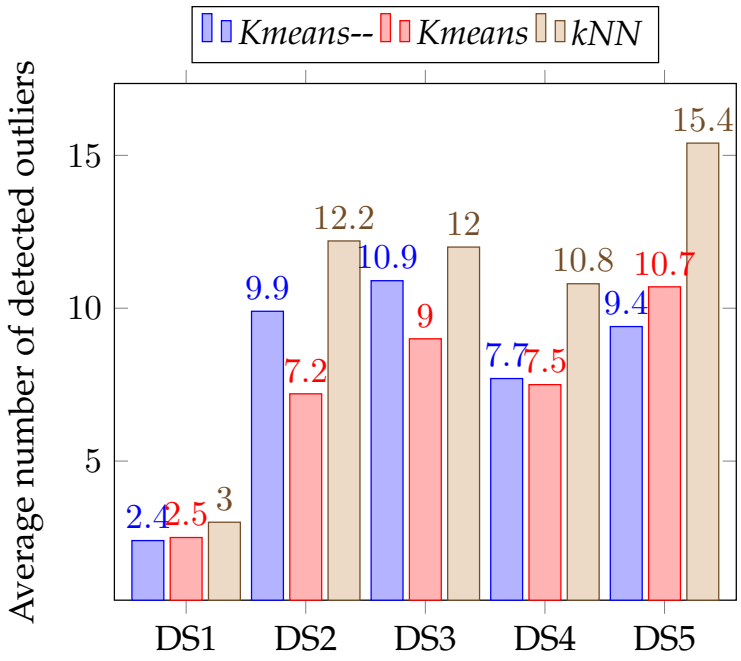


FIGURE 5.6: Average number of true outliers detected of three algorithms: *k-means--*, *k-means* and *kNN* on 5 synthesis datasets

normal and classes with a small number of points are outliers. All datasets have all or almost all attributes are numerical. When pre-processing the datasets, we removed points with categorical or missing attributes. A summary of the all real datasets is presented in Table 5.3.

– *Breast Cancel*: We first removed all data instances with missing attributes. The remainder of the dataset has two classes. One of them accounts for 65% of the whole dataset while the other accounts for 35%. We considered the class which has 35% population of the dataset as the abnormal class. We used this dataset to test *k-means--* in an extreme situation where the number of outliers is not small compared to the number of normal points.

– *Glass identification*: The dataset has six classes representing six different types of glasses. We considered two of them as outliers.

– *NSL-KDD*: The dataset is an improved version of KDD Cup 1999 dataset [52]. According to the authors, the dataset is more challenging than the KDD Cup 1999 dataset for data mining algorithms to predict different types network attacks correctly. We considered 25 classes with smallest numbers of data points as outliers.

– *KDD Cup 1999 small*: The dataset has 23 classes representing different types of network attacks. We assume three classes with highest numbers of data points to be

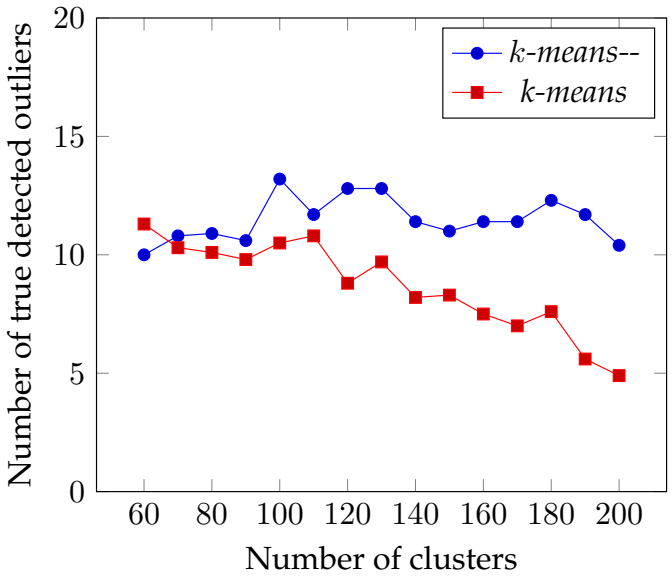


FIGURE 5.7: Comparison between *k-means--* and *k-means* on *DS5* dataset

normal. In total, the dataset has 42 categorical and numerical attributes. Because we are considering distance-based algorithms, we removed all categorical attributes and kept only numerical attributes. The dataset is 10% of the original *KDD Cup 1999* dataset. This dataset is used to evaluate the running time of *k-means---* with different distance-based outlier detection algorithms.

TABLE 5.3: Real datasets for local experiments

Dataset	#points	#features	#classes	#outliers	outliers
Breast cancel	683	9	2	239	35%
Glass identification	213	10	6	22	10.3%
NSL-KDD	21663	38	38	881	4%
KDD Cup 1999 small	494021	38	23	8752	1.78%

5.1.2.1 Correctness

For each dataset, we experimented *k-means--* and *k-means* with various values for number of clusters. Let $IC = \{c_1, c_2, ..., c_m\}$ is the set containing M different values of number of clusters. For each value in IC, we ran both *k-means--* and *k-means* p times and recorded the average number of outliers detected. Denote $A = \{a_1, a_2, ..., a_m\}$ is the set containing average numbers of true outliers detected for different values of $c_i \in IC$. For each value $c_i \in IC$, we define the precision as the correctness measurement for *k-means--* and *k-means* algorithms as

$$precision = \frac{a_i}{total\ number\ of\ true\ outliers}$$

For the *kNN* algorithm, on each dataset, we tried n different values of parameter $k \in K = \{k_1, ..., k_n\}$ and recorded the corresponding number of true outliers detected set $OD = \{p_1, ..., p_n\}$. Given a value k_i, we can compute the number of detected outliers p_i. The precision given k_i is defined as

$$precision = \frac{p_i}{total\ number\ of\ true\ outliers}$$

Figure 5.8 shows the results of all three algorithms on the *Breast cancel* dataset. All algorithms performed very well on this dataset. All algorithms showed that they are stable with different parameters. The difference in performance among the three algorithms are small. However, we can see that kNN is a little better than the two others and *k-meansOD* is slightly better than *k-means--*.

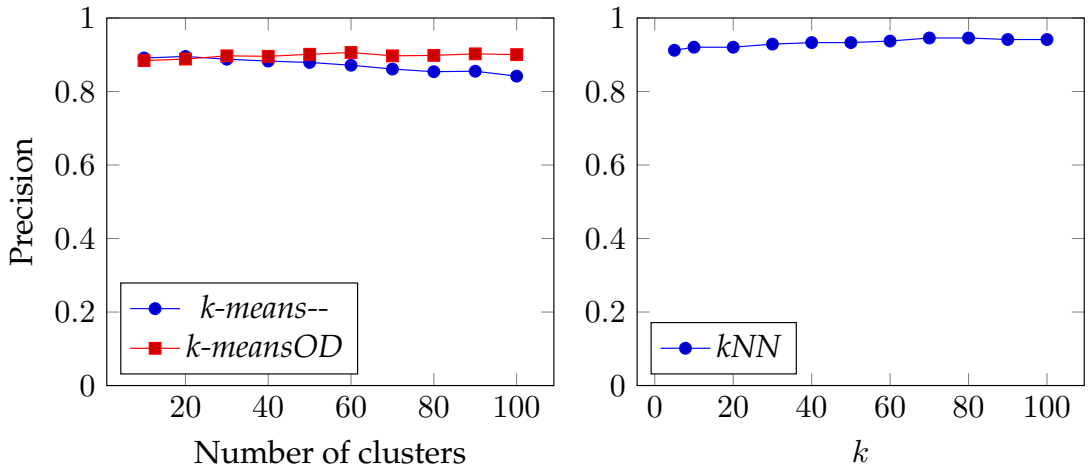


FIGURE 5.8: Results on *Breast Cancel* dataset

For the *Glass identification* dataset, the experiment results in Figure 5.9 show that *k-means--* and *k-meansOD* performed similarly as the number of clusters increases. The *kNN* showed its low accuracy when k increased. The behavior of *kNN* is consistent with what we have found in previous experiments. Overall, the performances of all three algorithms are competitive.

For the *NSL-KDD* dataset, the precision produced by all the three algorithms are low as shown in Figure 5.10. However, we can see that the precision lines reflect the interesting observation when experimenting on the synthesis datasets. *K-means--* works better and is more stable when increasing the number of clusters while *kNN* algorithm works well with small values of k. The *NSL-KDD* is a hard dataset for all three algorithms. It is even difficult for a machine learning classifier [52] to predict correct types of network

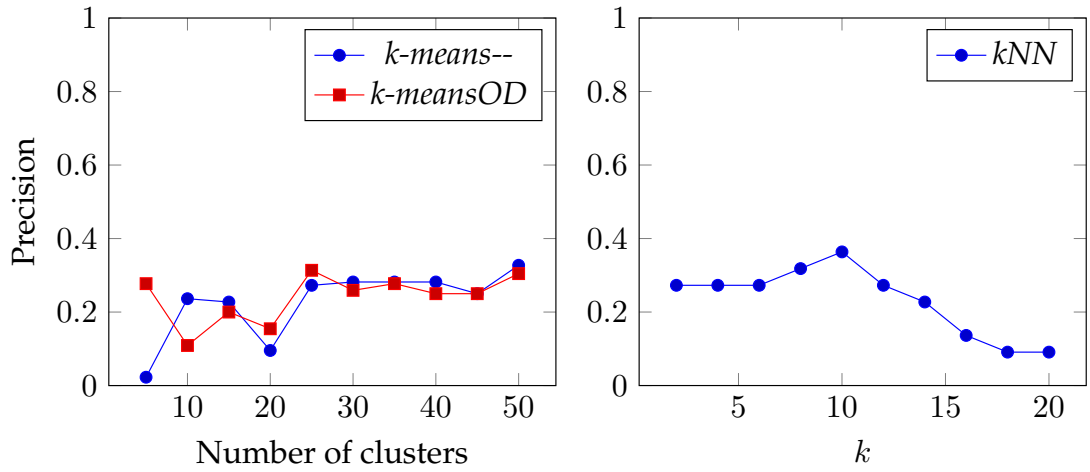


FIGURE 5.9: Results on *Glass identification* dataset

attacks due to the way the dataset is created. The attacks which are often wrongly classified by commons machine learning algorithms have relatively high proportion in the dataset. Therefore, it is understandable why the majority of outliers are not identified and the accuracies of all three algorithms are low.

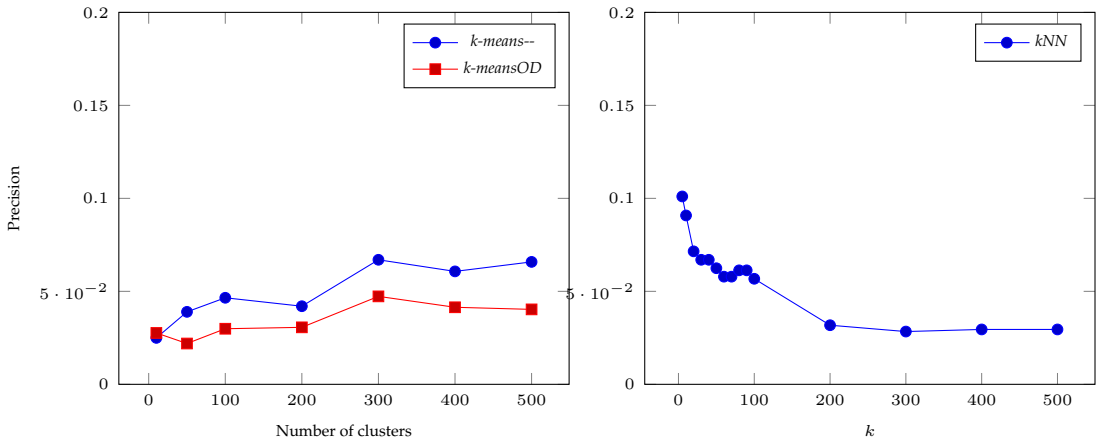


FIGURE 5.10: Results on *NSL-KDD* dataset

5.1.2.2 Running time

In this section, we study three distance-based outlier detection algorithms regarding running time. The running time measure we use is the wall clock time. For the *kNN* we implemented the speed-up technique proposed by Bay and Schwabacher in [20]. In the worst case, the problem has to compute n^2 distances between any pair of data points. However, The algorithm rarely reaches that complexity in practice because it can effectively avoid many computations by pruning non-outlier data points. *K-meansOD* has

the complexity of $O(nkdi)$ where k is the number of clusters, d is the dimension of data and i is the number of iterations executed. *K-means--* implemented in this thesis has the complexity of $O(i(nkd + n \log l))) = O(ni(kd + \log l)$ with the part $\log l$ is for finding the cutoff threshold distance to filter top l points before re-computing the centroids in each iteration. Since the $\log l$ is often smaller than kd, we expect the complexity of *k-means--* is $O(nkdi)$ which is similar to the complexity of *k-meansOD*. We implemented the same convergence criterion for both *k-means--* and *k-meansOD*. Specifically, those algorithms will stop if the difference of the sum of squared distances of all points to their nearest centroids between two consecutive iterations is smaller than 1E-9.

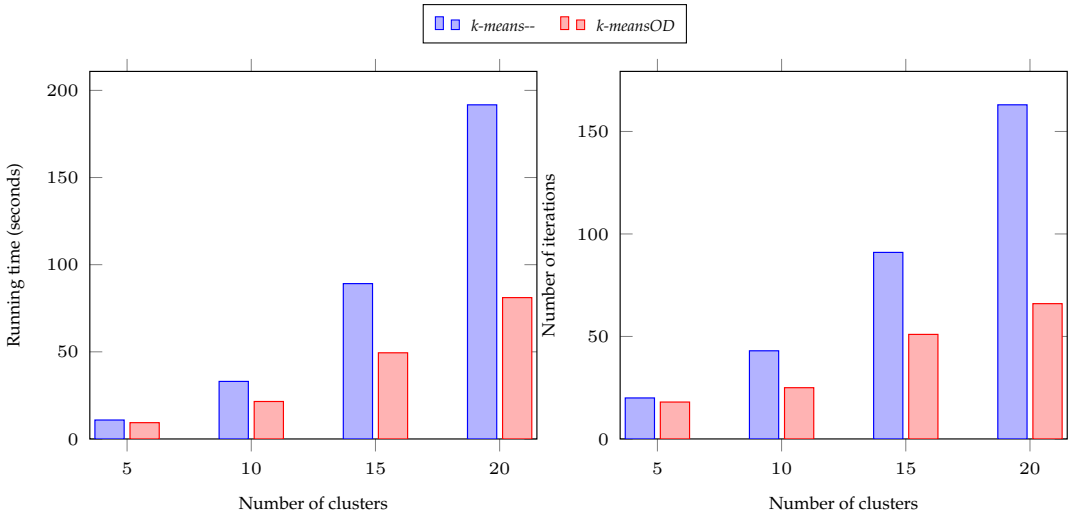


FIGURE 5.11: Running time on *KDD Cup 1999 small* dataset

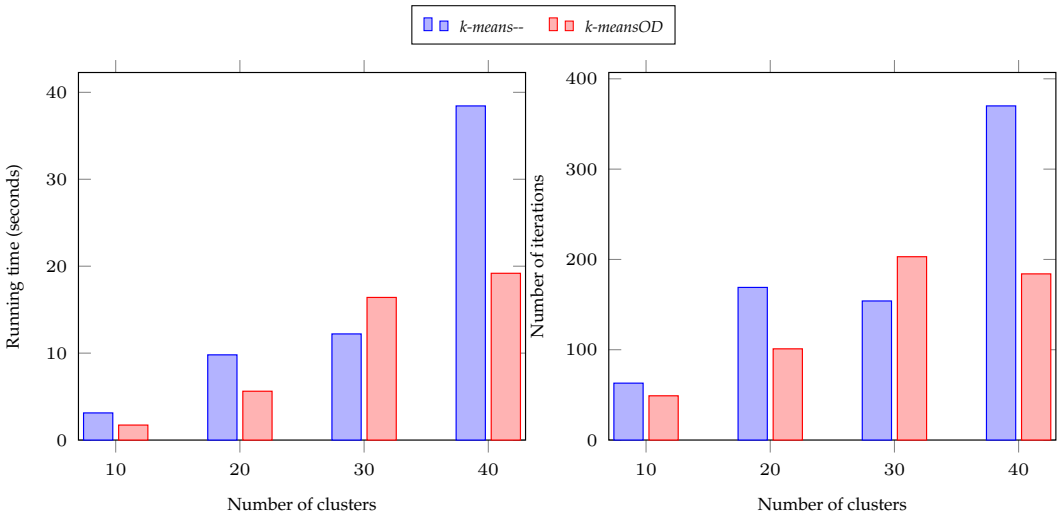


FIGURE 5.12: Running time on *NSL-KDD* dataset

Figure 5.11 and figure 5.12 show the running times and number of iterations executed for *k-means--* and *k-meansOD* when varying number of clusters for the *KDD Cup 1999 small* and *NSL-KDD* datasets, respectively. Note that we experimented on the same initial centroids for both algorithms. The *k-means--* was prone to require a higher number of iterations to reach the convergence criterion than the *k-meansOD*; thus, the running time was higher. We can also observe that the computation times increased proportionally to the rise of iterations. This observation verifies that the computational complexities of *k-means--* and *k-means* are almost the similar. Moreover, the running times increased as the numbers of clusters increased for both algorithms. This is expected behavior because they need to compute distances from the input data points to higher number centroids. For the *kNN* algorithm, it ran very slow on the *KDD Cup 1999 small* dataset, taking 3610 seconds to finish when we set the parameter $k = 3$. However, on the *NSL-KDD* dataset, it is much faster than both *k-means--* and *k-means*. The running was only 10 seconds and stable for different values of k. The *k-means--* and *k-means* are fast when the number of clusters is small. However, through experiments, we observed that for detecting outliers, the number of clusters should be set to relatively high. Therefore, we need a more scalable solution to deal with larger data.

5.1.2.3 Clustering quality

Because the *k-means--* algorithm does both clustering and detecting outliers at the same time, in this experiment, we test its clustering quality compared to the traditional *k-means* algorithm. The metric to compare clustering quality is the average sum of squared distances from data points to their nearest centroids. Denote $D = \{p_1, ..., p_n\}$ is the input dataset, O is the set of outliers detected by *k-means--*, $C = \{c_1, ..., c_k\}$ is the set of centroids after clustering, we define the nearest distance from a point to its centroid as

$$dist(p_i, C) = \arg\min_{c \in C}\{dist(p_i, c)\}$$

The average sum of squared distances of *k-means--* is defined as

$$avgSSEKmeans-- = \frac{\sum dist^2(p_i, C)[p_i \in D \setminus O]}{|D| - |O|}$$

The average sum of squared distances of *k-means* is defined as

$$avgSSEKmeans = \frac{\sum dist^2(p_i, C)[p_i \in D]}{|D|}$$

We ran both *k-means--* and *k-means* on all real datasets presented in Table 5.3 with the same set of parameters. For each dataset, we configure the number of clusters so that it equals to the number of normal classes. The initial centroids are sampled randomly

from the dataset and are similar for the two algorithms. Both of the algorithms ran until they reach a similar stopping criterion which is the sum of squared distances from all points to their centroids between two consecutive iterations is smaller than 1E-9. The results are recorded in Table 5.4. It can be seen that $avgSSEKmeans--$ is smaller than $avgSSEKmeans$ in every dataset. This suggests that the *k-means--*, by removing outliers, produces tighter clusters than the *k-means* which is well known for being sensitive to outliers.

TABLE 5.4: Average sum of squared distances of *k-means--* and *k-means*

Datasets	**number of clusters**	**avgSSEKmeans--**	**avgSSEKmeans**
Breast cancel	1	7.9	122.4
Glass	4	196.9	311.0
NSL-KDD	13	557462.0	4.9E10
KDD Cup 1999 small	3	515250.2	7.1E9

5.1.2.4 Summary

In this section we have conducted experiments on *k-means--* on the local environment, we have tested the *k-means--* with various of synthesis as well as real datasets. The algorithm demonstrates that it has the competitive performance compared to the *kNN* algorithm. We have also noticed some interesting observations about the *k-means--*. It performs stably on different types of datasets. Although the algorithm is sensitive to the initialization of centroids, it can detect high number outliers when the number of clusters is a sufficiently big value. This observation suggests a good hint to use *k-means--* effectively. The algorithm is based on the *k-means* algorithm; therefore, it can be expected to run relatively fast. Compared to the *k-meansOD*, we can see that *k-means--* often run longer before converging given that both algorithms use the same parameters. Regarding the clustering quality, *k-means--* produces tighter clusters than the *k-means*.

5.2 Cluster Evaluation

In this section, we test how well our implementation of *k-means--* works on a dataflow system. As described in Chapter 4, we implemented the *k-means--* in Apache Flink. All experiments with Flink are conducted in a Flink cluster with 9 machines. Each machine has 12 CPU cores and each core has 4 threads. One machine is dedicated to be the job manager and the rest are task managers. The job manager has 50GB of RAM. All task managers have 64GB of RAM. The version of Flink we used in experiments is 1.0.0.

From now on, in order to easily distinguish between algorithms' implementations in Flink and in a local single machine environment. We define

TABLE 5.5: Datasets for cluster evaluation

Dataset	#points	#features	#outliers
KDD Cup 1999 small	494021	38	8752
KDD Cup 1999	4898431	38	45717
dataset20M	20000000	15	10000
dataset30M	30000000	15	10000
dataset40M	40000000	15	10000
dataset50M	50000000	15	10000
dataset60M	60000000	15	10000

– *Local k-means--* as the implementation of *k-means--* on a single machine

– *Flink k-means--* as the implementation of *k-means--* in Flink

– *Local k-meansOD* as the implementation of *k-meansOD* on a single machine

– *Flink k-meansOD* as the implementation of *k-meansOD* in Flink

In this section, we focus on the running time of different implementations of algorithms. We used several real and synthesis datasets with a variety of sizes and features. For the real datasets, we experimented with *KDD Cup 1999* dataset. We used two versions of *KDD Cup 1999* dataset. The small version *KDD Cup 1999 small* was previously used in section 5.1. The full version *KDD Cup 1999* is ten times bigger than the small version. Because it is difficult to find real big datasets, we also generated some synthesis datasets using the data generator from ELKI Data Mining Framework [53]. Attributes of points in the synthesis datasets were generated based on Uniform, Gamma and Gaussian distributions. Parameters of the distributions were selected randomly. The outliers were added using the Uniform distribution. Details about all datasets in this section are shown in Table 5.5.

5.2.1 Local implementations versus Flink implementations

In this experiment, we first compare the computation time between the *Local k-means--* and the *Flink k-means--* to check if *Flink k-means--* does speed up the *Local k-means--*. We ran both algorithms on the same set of parameters i.e. the same number of desired outliers, the same number of clusters, the same initial centroids. Figure 5.13 shows the running times of *k-means--* on *KDD Cup 1999* and *KDD Cup 1999 small* datasets on both local and cluster environments. It can be seen that in both datasets, the Flink version is faster than the local version. This demonstrates that the Flink implementation, even with small dataset, indeed improves the running time of the local version.

For the *KDD Cup 1999 small* dataset, we set the number of clusters to 50. The number of iterations is 85 which is the number of required iterations before the *k-means--* reaches

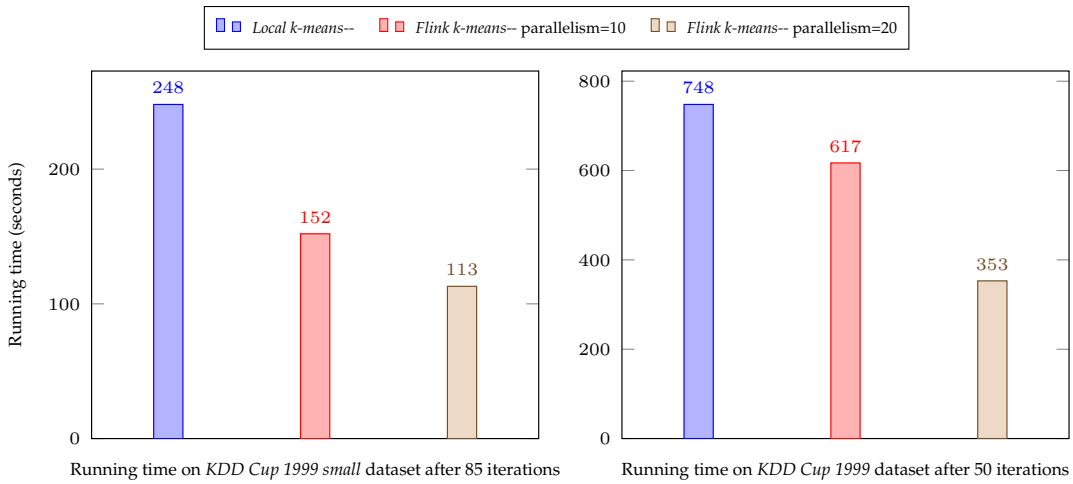


FIGURE 5.13: Running times of *k-means--* on *KDD Cup 1999* and *KDD Cup 1999 small* datasets

the convergence criterion. When the parallelism is 10, *Flink k-means--* is 1.6 times faster than the *Local k-means--*. Increasing the parallelism to 20 results in more than two times reduction in running time compared to the local version.

For the *KDD Cup 1999* dataset, we set the number of clusters to 20 and ran 50 iterations for both local and Flink versions. The number of points in the dataset is ten times bigger than the number of points in the *KDD Cup 1999 small*. When the parallelism value is small, the Flink version does not help improve much the running time. However, when the parallelism is a sufficiently high value, the computation time reduces considerably. This suggests that the parallelism should be big enough in order to make the *Flink k-means--* effective. One reason to explain the behavior is that when the parallelism is small, the cost of communication and data transferring among task managers could exceed the gains of parallel computation.

5.2.2 Varying parallelism

Flink's parallelism is an important parameter when executing a Flink program. A program in a Flink programs is divided into small tasks which then are allocated to task managers[1]. The parallelism defines the number of tasks should be executed concurrently. Normally, we expect increasing the parallelism will reduce the running time. In this part, we test with *dataset20M* dataset with different parallelism values to check how scalable our Flink implementation of *k-means--* is. We set the number of clusters to 20, use same initial centroids and record the running time for the first ten iterations for all tests. We

[1] https://flink.apache.org/faq.html#what-is-the-parallelism-how-do-i-set-it

also compare the *Flink k-means--* with *Flink k-meansOD* in order to compare the scalability of the two algorithms.

For both *Flink k-means--* and *Flink k-meansOD* algorithms, their running times decreases as number parallelism increases. However, the speed up degree is not linear to the increase of parallelism. The running time dropped quickly when parallelism values are small. From the parallelism of 60, the running times were decreasing slightly. It is because of the overhead costs of rising number of parallel tasks. High value of parallelism means more costs for communication, data transferring and data partitioning within the cluster. When the costs are significant, they will reduce the gains of parallel computation. Experimental results also reveal that *Flink k-means--* is slower than the *Flink k-meansOD*. It is the expected behavior because *Flink k-means--* requires additional calculation to remove top points with highest distances to their respective centroids in each iteration.

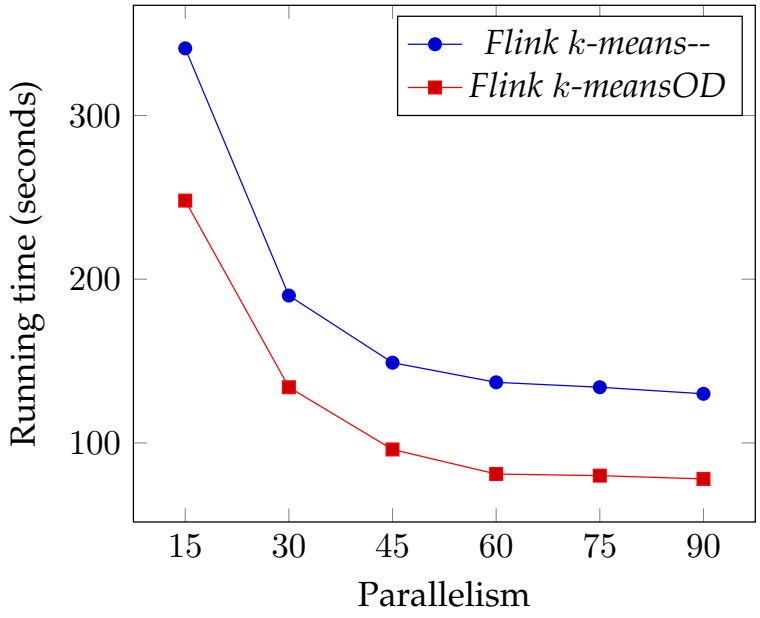


FIGURE 5.14: Running times of *Flink k-means--* and *Flink k-meansOD* for varying parallelism

5.2.3 Varying number of clusters

In the local version of *k-means--*, increasing the number of clusters results in nearly a linear increase of running time as shown in Figure 5.11. As we observed that *k-means--* could better detect outliers when the number of clusters is high, we want to see the effect of the number of clusters to the running time of *Flink k-means--*. We did the experiment on *dataset20M* dataset and set the number of clusters to 20, the parallelism to 40. Besides, we also ran the *Flink k-meansOD* to compare its behavior with *Flink k-means--*. We used

the same initial centroids for both algorithms and recorded the running time of first ten iterations.

The results of the experiment in Figure 5.15 verify that the number of clusters is one of the main factors that affects the running time of *k-means--*. The rise of running time is linear to the increase of the number of clusters in Flink version. This intrinsic property of *k-means--* is a challenge for the algorithm especially when in previous experiments, we showed that *k-means--* is more efficient when the number of clusters is high.

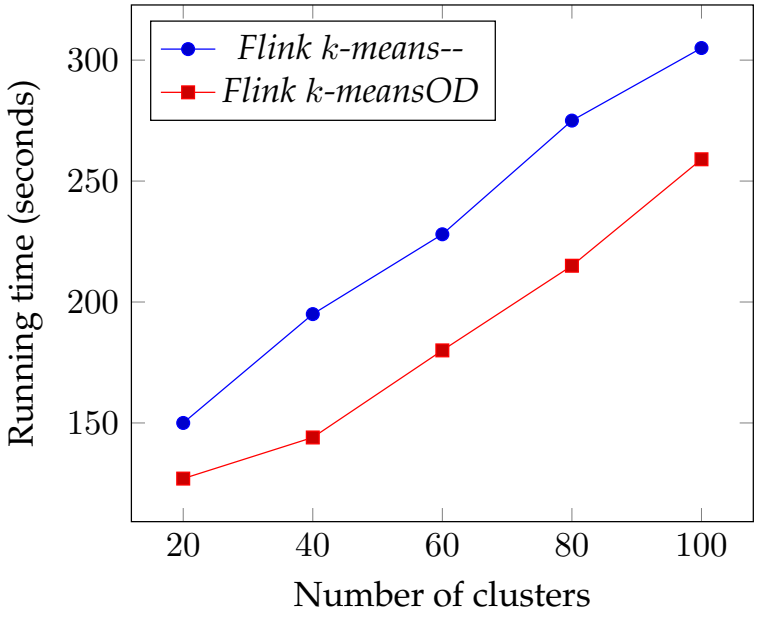


FIGURE 5.15: Running times of *Flink k-means--* and *Flink k-meansOD* for varying number of clusters

5.2.4 Varying size of datasets

Since the aim of our Flink implementation of *Flink k-means--* is to help detect outliers from large input data. In this section, we investigate how the *Flink k-means--* behaves correspond to the change in the size of input data. In this experiment, "size" means the number of points in the input data. In order to do this experiment, we use datasets with different numbers of data points. The datasets we use are *dataset20M*, *dataset30M*, *dataset40M*, *dataset50M* and *dataset60M*. They have different numbers of points ranging from 20 million to 60 million points. Their sizes on disk vary from 5.8 Gigabytes to 17.5 Gigabytes. We set the number of clusters to 20 and record the running time for the first ten iterations. We use 40 as the value of Flink's parallelism. The results of the experiment in Figure 5.16 show that the running time increases linearly to the size of datasets. These are the understandable results because the more data points are, the more computation

the algorithm requires. In each iteration, *Flink k-means--* has to iterate through all data points to assign them to their nearest clusters. Therefore, even if we use high value of parallelism, the computation time in each task manager of Flink is proportional to the number of the data points it processes i.e. for any value of parallelism, computational time is linear to the size of input data.

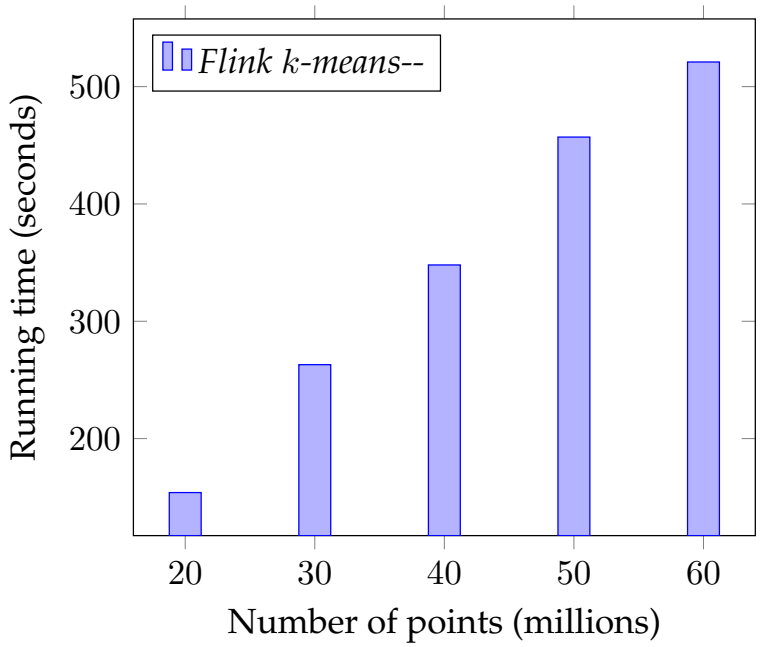


FIGURE 5.16: Running time of *Flink k-means--* for varying size of datasets

5.2.5 Summary

We have conducted the experiments for our implementation of *k-means--* in Apache Flink with different synthesis datasets. The computation time of the *Flink k-means--* algorithm increases linearly to the rise of the number of clusters and the number of data points. The running time of the *Flink k-means--* reduces when parallelism increases. Although the speed up is not proportional to the parallelism due to the overhead costs, the algorithm demonstrates that it can handle large datasets with faster computation time than when running the algorithm in a single machine. *Flink k-means--* is also a good solution when the datasets are big and could not fit the memory of a single computer.

Chapter 6

CONCLUSION AND FUTURE WORK

Outlier detection is an essential and challenging data mining task. It has a variety of applications on different domains. In this thesis, the goal is to study unsupervised distance-based outlier detection techniques. Specifically, we focus on the algorithms which assume that outliers are distant from the rest of the data. In order to measure how far outliers are, we use the Euclidean distance because it is simple and fast to compute. We also target on the distance-based algorithms which could handle large amount of data by leveraging modern distributed dataflow systems.

We started by surveying different approaches and algorithms in the domain of outlier detection. From the survey, we have an overview on current research in the field. Most of the algorithms focus on detecting outliers on centralized data and running on a single machine. There are some proposed improvement to speed up those algorithms when dealing with large amount of data by approximating the results. Some distributed solutions are also suggested. However, they often focus specifically on a particular domain or work with data where distance measures are not applicable. Some other distributed algorithms are implemented using MPI/OpenMP. Although they are designed for high-performance computing, they do not provide mechanisms for fault tolerance as in dataflow systems [54].

Simple algorithms are often ignored because they could not produce the best results. However, in the era of big data, it is very useful if they can quickly detect outliers from large amount of data with competitive results. In this thesis, we studied in detail a variant of *k-means* namely *k-means--*. We implemented the algorithm on a single machine and test its efficiency. The algorithm shows that it has competitive performance regarding detection of outliers compared to k-nearest neighbors approach. Its performance is also stable to the initial number of clusters. Regarding the clustering quality, it demonstrates that it can produce tighter clusters than classic *k-means* algorithm.

We also proposed a distributed implementation of the *k-means--* in Apache Flink to

test its capability of handling big data. The results show that the algorithm can significantly reduce the running time compared to its version on a single computer. Although the running time speed up in a Flink cluster is not linear to the number of parallelisms due to the overhead costs of data shuffling and transferring between task workers, the algorithm shows that it is capable of processing big datasets. The implementation of *k-means--* in Flink is also a good solution when data is larger than the memory of a single computer or when data is distributed.

Although *k-means--* is a good algorithm for detecting outliers and clustering. There are several aspects which can be considered to improve it in the future. First, we currently use the Euclidean distance measure which will become less meaningful to separate data points in high dimensional space. This is not the intrinsic problem of *k-means--*; however, it does affect the algorithm's results. Therefore, choosing a good distance measure will potentially improve the performance of *k-means--*. Second, *k-means--* can be applied to any data where a distance measure can be defined. Therefore, we plan to test it on data with categorical and mixed attributes by defining appropriate measures. Finally, even though the algorithm is quite stable to the number of clusters, it is still sensitive to the initial placement of centroids. A good initialization of centroids as presented in [55, 56] will improve the clustering quality of *k-means*. We expect the initialization will help *k-means--* produce better clustering quality and detection of outliers.

Bibliography

[1] D. M. Hawkins, *Identification of outliers*. Springer, 1980, vol. 11.

[2] F. Edgeworth, "Xli. on discordant observations", *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 23, no. 143, pp. 364–375, 1887.

[3] C. Aggarwal, *Outlier Analysis*. Springer New York, 2013, ISBN: 9781461463955.

[4] J. Hu, F. Wang, J. Sun, R. Sorrentino, and S. Ebadollahi, "A healthcare utilization analysis framework for hot spotting and contextual anomaly detection.", in *AMIA*, 2012.

[5] J. Zhang and M. Zulkernine, "Anomaly based network intrusion detection with unsupervised outlier detection", in *2006 IEEE International Conference on Communications*, IEEE, vol. 5, 2006, pp. 2388–2393.

[6] S. Chawla and A. Gionis, "K-means-: A unified approach to clustering and outlier detection", in *Proceedings of the 13th SIAM International Conference on Data Mining, May 2-4, 2013. Austin, Texas, USA.*, 2013, pp. 189–197.

[7] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey", *ACM Comput. Surv.*, vol. 41, no. 3, 15:1–15:58, Jul. 2009, ISSN: 0360-0300.

[8] A. Z. Hans-Peter Kriegel Peer Kröger, "Outlier detection techniques", *Tutorial at the 16th ACM International Conference on Knowledge Discovery and Data Mining*, 2010.

[9] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters", *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008, ISSN: 0001-0782.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing", in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12, San Jose, CA: USENIX Association, 2012, pp. 2–2.

[11] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics", *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014, ISSN: 1066-8888.

[12] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: A programming model and execution framework for web-scale analytical processing", in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, Indianapolis, Indiana, USA: ACM, 2010, pp. 119–130, ISBN: 978-1-4503-0036-0.

[13] K. Tzoumas, J.-C. Freytag, V. Markl, F. Hueske, M. Peters, M. Ringwald, and A. Krettek, "Peeking into the optimization of data flow programs with mapreduce-style udfs", in *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ser. ICDE '13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 1292–1295, ISBN: 978-1-4673-4909-3.

[14] T. Johnson, I. Kwok, and R. Ng, *Fast computation of 2-dimensional depth contours*, 1998.

[15] C. C. Aggarwal and P. S. Yu, "Outlier detection for high dimensional data", in *ACM Sigmod Record*, ACM, vol. 30, 2001, pp. 37–46.

[16] E. M. Knorr and R. T. Ng, "Algorithms for mining distance-based outliers in large datasets", in *Proceedings of the 24rd International Conference on Very Large Data Bases*, ser. VLDB '98, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 392–403, ISBN: 1-55860-566-5.

[17] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets", *SIGMOD Rec.*, vol. 29, no. 2, pp. 427–438, May 2000, ISSN: 0163-5808.

[18] F. Angiulli and F. Fassetti, "Dolphin: An efficient algorithm for mining distance-based outliers in very large datasets", *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 3, no. 1, p. 4, 2009.

[19] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: An efficient data clustering method for very large databases", *SIGMOD Rec.*, vol. 25, no. 2, pp. 103–114, Jun. 1996, ISSN: 0163-5808.

[20] S. D. Bay and M. Schwabacher, "Mining distance-based outliers in near linear time with randomization and a simple pruning rule", in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03, Washington, D.C.: ACM, 2003, pp. 29–38, ISBN: 1-58113-737-0.

[21] A. Ghoting, S. Parthasarathy, and M. E. Otey, "Fast mining of distance-based outliers in high-dimensional datasets", *Data Mining and Knowledge Discovery*, vol. 16, no. 3, pp. 349–364, 2008.

[22] M. Wu and C. Jermaine, "Outlier detection by sampling with accuracy guarantees", in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06, Philadelphia, PA, USA: ACM, 2006, pp. 767–772, ISBN: 1-59593-339-5.

[23] M. Sugiyama and K. Borgwardt, "Rapid distance-based outlier detection via sampling", in *Advances in Neural Information Processing Systems 26*, 2013, pp. 467–475.

[24] F. Angiulli and C. Pizzuti, "Outlier mining in large high-dimensional data sets", *IEEE transactions on Knowledge and Data engineering*, vol. 17, no. 2, pp. 203–215, 2005.

[25] J. Macqueen, "Some methods for classification and analysis of multivariate observations", in *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, 1967, pp. 281–297.

[26] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space", in *Database Theory — ICDT 2001: 8th International Conference London, UK, January 4–6, 2001 Proceedings*, J. Van den Bussche and V. Vianu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 420–434, ISBN: 978-3-540-44503-6.

[27] H.-P. Kriegel, M. S hubert, and A. Zimek, "Angle-based outlier detection in high-dimensional data", in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '08, Las Vegas, Nevada, USA: ACM, 2008, pp. 444–452, ISBN: 978-1-60558-193-4.

[28] N. Pham and R. Pagh, "A near-linear time approximation algorithm for angle-based outlier detection in high-dimensional data", in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12, Beijing, China: ACM, 2012, pp. 877–885, ISBN: 978-1-4503-1462-6.

[29] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: Identifying density-based local outliers", *SIGMOD Rec.*, vol. 29, no. 2, pp. 93–104, May 2000, ISSN: 0163-5808.

[30] J. Tang, Z. Chen, A. W.-C. Fu, and D. W. Cheung, "Enhancing effectiveness of outlier detections for low density patterns", in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, 2002, pp. 535–548.

[31] S. Guha, R. Rastogi, and K. Shim, "Rock: A robust clustering algorithm for categorical attributes", in *Data Engineering, 1999. Proceedings., 15th International Conference on*, IEEE, 1999, pp. 512–521.

[32] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise.", in *Kdd*, vol. 96, 1996, pp. 226–231.

[33] Z. He, S. Deng, and X. Xu, "An optimization model for outlier detection in categorical data", in *International Conference on Intelligent Computing*, Springer, 2005, pp. 400–409.

[34] Z. He, S. Deng, X. Xu, and J. Z. Huang, "A fast greedy algorithm for outlier mining", in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, 2006, pp. 567–576.

[35] M. E. Otey, A. Ghoting, and S. Parthasarathy, "Fast distributed outlier detection in mixed-attribute data sets", *Data Mining and Knowledge Discovery*, vol. 12, no. 2-3, pp. 203–228, 2006.

[36] A. Koufakou, E. G. Ortiz, M. Georgiopoulos, G. C. Anagnostopoulos, and K. M. Reynolds, "A scalable and efficient outlier detection strategy for categorical data", in *19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007)*, IEEE, vol. 2, 2007, pp. 210–217.

[37] L. Akoglu, H. Tong, J. Vreeken, and C. Faloutsos, "Fast and reliable anomaly detection in categorical data", in *Proceedings of the 21st ACM international conference on Information and knowledge management*, ACM, 2012, pp. 415–424.

[38] A. Koufakou, J. Secretan, J. Reeder, K. Cardona, and M. Georgiopoulos, "Fast parallel outlier detection for categorical datasets using mapreduce", in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, IEEE, 2008, pp. 3298–3304.

[39] E. Hung and D. W. Cheung, "Parallel mining of outliers in large database", *Distributed and Parallel Databases*, vol. 12, no. 1, pp. 5–26, 2002.

[40] E. Lozano and E Acufia, "Parallel algorithms for distance-based and density-based outliers", in *Fifth IEEE International Conference on Data Mining (ICDM'05)*, IEEE, 2005, 4–pp.

[41] H. Dutta, C. Giannella, K. D. Borne, and H. Kargupta, "Distributed top-k outlier detection from astronomy catalogs using the demac system.", in *SDM*, SIAM, 2007, pp. 473–478.

[42] F. Angiulli, S. Basta, S. Lodi, and C. Sartori, "Distributed strategies for mining outliers in large data sets", *IEEE transactions on knowledge and data engineering*, vol. 25, no. 7, pp. 1520–1532, 2013.

[43] P. Berkhin, "A survey of clustering data mining techniques", in *Grouping multidimensional data*, Springer, 2006, pp. 25–71.

[44] S. Lloyd, "Least squares quantization in pcm", *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[45] A. Vattani, "K-means requires exponentially many iterations even in the plane", *Discrete & Computational Geometry*, vol. 45, no. 4, pp. 596–616, 2011.

[46] D. Arthur and S. Vassilvitskii, "How slow is the k-means method?", in *Proceedings of the twenty-second annual symposium on Computational geometry*, ACM, 2006, pp. 144–153.

[47] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on mapreduce", in *IEEE International Conference on Cloud Computing*, Springer, 2009, pp. 674–679.

[48] M. Lichman, *UCI machine learning repository*, 2013. [Online]. Available: http://archive.ics.uci.edu/ml.

[49] H. J. Weiliang Qiu, *Random cluster generation (with specified degree of separation)*, 2015. [Online]. Available: https://cran.r-project.org/web/packages/clusterGeneration/index.html.

[50] W. Qiu and H. Joe, "Generation of random clusters with specified degree of separation", *Journal of Classification*, vol. 23, no. 2, pp. 315–334, 2006, ISSN: 1432-1343.

[51] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[52] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the kdd cup 99 data set", in *Proceedings of the Second IEEE International Conference on Computational Intelligence for Security and Defense Applications*, ser. CISDA'09, Ottawa, Ontario, Canada: IEEE Press, 2009, pp. 53–58, ISBN: 978-1-4244-3763-4.

[53] E. Schubert, A. Koos, T. Emrich, A. Züfle, K. A. Schmid, and A. Zimek, "A framework for clustering uncertain data", *PVLDB*, vol. 8, no. 12, pp. 1976–1979, 2015.

[54] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf", *Procedia Computer Science*, vol. 53, pp. 121–130, 2015.

[55] A. K. Jain, "Data clustering: 50 years beyond k-means", *Pattern recognition letters*, vol. 31, no. 8, pp. 651–666, 2010.

[56] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++", *Proceedings of the VLDB Endowment*, vol. 5, no. 7, pp. 622–633, 2012.