UNIVERSITY OF TARTU
Faculty of Mathematics and Computer Science
Institute of Computer Science

Sergei Laada

Suitability of the Spark framework for data classification

Master's thesis
30 EAP

Supervisor: Pelle Jakovits

Author: ..... "….." November 2014
Supervisor: ..... "….." November 2014
Permission for defence
Professor ..... "….." November 2014

TARTU 2014

Info page

Suitability of the Spark framework for data classification

The goal of this thesis is to show the suitability of the Spark framework when dealing with different types of classification algorithms and to show how exactly to adapt algorithms from MapReduce to Spark. To fulfill the goal three algorithms were chosen: k-nearest neighbor's algorithm, naïve Bayesian algorithm and Clara algorithm. To show the various approaches it was decided to implement those algorithms using two frameworks, Hadoop and Spark. To get the results, tests were run using the same input data and input parameters for both frameworks. During the tests varied parameters were used to show the correctness of the implementations. As a result charts and tables were generated for each algorithm separately. In addition parallel speedup charts were generated to show how well algorithm implementations can be distributed between the worker nodes. Results show that Spark handles easy algorithms, like k-nearest neighbor's algorithm, well, but the difference with Hadoop results is not very large. Naïve Bayesian algorithm revealed the special case with easy algorithms. The results show that with very fast algorithms Spark framework use more time for data distribution and configuration than for data processing itself. Clara algorithm results have shown that Spark framework handles more difficult algorithms noticeably better.

Keywords

Algorithm, Hadoop, Spark, framework, MapReduce, classification, parallel, k-nearest neighbor's, naïve Bayesian, Clara, cluster, Tartu University.

Spark raamistiku sobivus andmete klassifitseerimiseks

Selle lõputöö eesmärk on näidata Spark raamistiku sobivust erinevate klassifitseerimis algoritmite rakendamisel ja näidata kuidas täpselt algoritmid MapReduce-ist Spark-i üle viia. Eesmärgi täitmiseks said implementeertud kolm algoritmi: paralleelne k-nearest neighbor's algoritm, paralleelne naïve Bayesian algoritm ja Clara algoritm. Et näidata erinevaid lähenemisviise otsustati rakendada need algoritmid kasutades kahte raamistiku: Hadoop ja Spark. Et tulemusi kätte saada, jooksutati mõlema raamistiku puhul testid samade sisend-andmete ja parameetritega. Testid käivitati erinevate parameetritega et näidata realiseerimise korrektsust. Tulemustele vastavad graafikud ja tabelid genereeriti et näidata kui hästi on algoritmide käivitamisel töö hajutatud paralleelsete protsesside vahel. Tulemused näitavad et Spark saab hakkama lihtsamate algoritmidega, nagu näiteks k-nearest neighbor's, edukalt aga vahe Hadoop tulemustega ei ole väga suur. Naïve Bayesian algoritm osutus lihtsate algoritmide erijuhtumiks. Selle tulemused näitavad et väga kiire algoritmide korral kulutab Spark raamistik rohkem aega andmete jaotamiseks ning konfigureerimiseks kui andmete töötlemiseks. Clara algoritmi tulemused näitavad et Spark raamistik saab suurema keerukusega algoritmidega hakkama märgatavalt paremini kui Hadoop.

Märksõnad

Algoritm, Hadoop, Spark, raamistik, MapReduce, klassifikatsioon, paralleelne, k-nearest neighbor's, naïve Bayesian, kobar, Tartu Ülikool.

Table of Contents

Index of Figures

Index of Tables

Acknowledgements

First of all I thank my supervisor Pelle Jakovits for introducing MapReduce, Hadoop and Spark frameworks to me. He's help and suggestions helped me a lot while writing this thesis and getting familiar with large scale data processing. Secondly I would like to thank Tartu University for a great education it is providing. I learned a lot from the lectures I participated in. I would like to thank all the lecturers I met in Tartu University, they helped me to understand hard but, at the same time, very interesting aspects of the new knowledge I received during the studies.

Chapter 1

Introduction

Today Internet is becoming more and more popular and many people can`t even imagine their lives without the use of such services as Facebook or Twitter. Those services require to process large data sets in a short period of time. As a user of such a service one can`t imagine waiting even for 10 seconds to get the list of people you are interested in. One can say that today it is possible to find anything in the Internet using such a searching engines like Google or Yahoo, but it wouldn`t be possible without introducing MapReduce technology and its implementation named Hadoop.

In the last few years when dealing with cloud computing most likely one will hear the term "Hadoop" in the conjunction with "large-scale data processing". Hadoop has evolved into really huge and exiting project. This framework makes it possible to process large data sets using the cloud structure without having to worry about losing the data in case of network or hardware failures. Hadoop has proven itself as a stable and fault tolerant framework that is easy to use after it is installed and configured.

The reason why Hadoop became so popular is because it is easy to install and configure, also because it is freely available, open source, and proven to scale to hundreds or even thousands of machines. In the past two years the amount of different tutorials was growing very fast and today it is not a problem to install this framework even for a non-involved people. Additionally Hadoop is very customizable, for example one can create custom classes to parse the data in a way it is needed.

Hadoop also has some disadvantages. One of the problems with Hadoop will arise when dealing with iterative algorithms. Its approach will not be the most effective; the only way to deal with iterative algorithms (which require repeating a specific task many times) in Hadoop is to create a chain of MapReduce jobs. Another problem is that it is not possible to hold data in memory, that means it is needed to load the data with every additional MapReduce job. As a result if there is ten jobs to run, it is needed to read the data ten times, configure the tasks ten times and save resulting data ten times. Data is loaded and saved in HDFS, that is relatively slow file system. Another difficulty is in Hadoop complexity when writing some non-standard

programs. This will be not a problem for a people who are already familiar with this framework, but for people less involved it will be quite difficult to understand the logic and write custom classes.

Lately there have been other frameworks appearing that try to solve the problems Hadoop has with these kinds of iterative applications. Spark is one of the most interesting out of them because it allows to perform all computations in memory and simplify writing applications. In two last year's some publications were released comparing Spark and Hadoop frameworks, for example in "Fast and Interactive Analytics over Hadoop Data with Spark"[1] authors show the advantages of Spark framework comparing the running speed with Hadoop.

Authors of the Spark framework claim that Spark can process data hundred times faster than Hadoop does.[2] Even taking into account in-memory data processing ability of Spark framework it doesn`t sound plausible. The main goal of this thesis is to show that Spark framework is a good choice when dealing with implementation of the classification algorithms and to show how exactly to adapt algorithms from MapReduce to Spark. Additional objective is to show that Spark framework can handle all kind of classification algorithms in a better way than Hadoop framework does or at least is not worse.

To fulfill the goals of this thesis three algorithms were chosen and implemented using Spark and Hadoop frameworks. Implementing the same algorithms using different frameworks and running them using the same cluster configuration will show the differences between implementations and identify what solution is a better choice in certain circumstances.

In the scope of this thesis three algorithms will be implemented using two different frameworks, Hadoop and Spark. Deployed applications will run inside the same cluster and will share the same configuration. For that purpose different running parameters will be used, like different input data size and different number of working nodes. Different number of worker nodes were used for each approach to show how well the algorithm implementations can be parallelized. After each iteration results will be saved and presented as tables. In addition data classification results will be visualized to show the correctness of the implementations. Also parallel speedup diagrams will be generated to demonstrate the scalability of two frameworks.

The thesis structure is as follows. In the first section Apache Spark framework will be introduced. Examples of using Spark will be shown and described. In addition MapReduce and Hadoop Distributed File System will be briefly introduced. In the third chapter description of the algorithms selected will be shown. Basic algorithms background will be described, after that implementation of each algorithm using Spark framework will be delineated. In the end of the description of each algorithm the implementation using Spark framework will be compared to the implementation using Hadoop framework. In the last chapter results of the implementations will be presented. Difference in running time between Hadoop and Spark implementations will be indicated. In addition visualization of the results will be given in a form of charts. Also parallel speedup diagrams will be generated for each algorithm separately. Finally, in the last chapter conclusions will be presented where thesis goals and results will be described.

Chapter 2

State of the Art

In this section Apache Spark framework will be described. It will briefly introduce the MapReduce technology, the programming model for processing large data sets that was developed by Google Company in 2004, and how this technology is used by the Spark framework. This chapter will also give an overview of the differences between Hadoop and Spark implementations, the advantages and disadvantages of using those two frameworks in different situations. In addition Hadoop Distributed File System (HDFS) will be briefly introduced.

2.1 Apache Spark

Apache Spark is an open source cluster computing environment that enables in-memory distributed datasets optimizing iterative process runs. Spark was developed at the University of California Berkeley, Algorithms Machines and People Lab to build large-scale and low-latency data analytics applications.[3]

Spark is implemented in the Scala language. Spark and Scala are tightly integrated, that provides Scala the ability to easily manipulate distributed datasets as local objects. Spark was designed for a specific type of jobs in cluster computing that reuse a working set of data across the parallel operations. As an optimization for these types of jobs, Spark developers introduced the concept of in-memory cluster computing, where it is possible to cache the datasets in memory to reduce their latency of access.[3]

Spark established an abstraction named resilient distributed datasets (RDD). Those datasets are read-only object groups that are allocated across the nodes. These collections are fault tolerant and can be restored if something unexpected happens and the dataset or only a part of it is lost. The restoring process of the dataset, or a part of it, uses fault-tolerant mechanism that preserve information that allows the portion of the dataset to be reconstructed based on the process from which that data was generated. Each RDD tracks the graph of transformations that was used to build it, called its lineage graph, and reruns these operations on initial data to recover any lost section. An RDD is represented as Scala object that can be

created from a file as a parallelized chunk (propagated across the nodes), as a mapping of another RDD or by changing the consistency of an existing RDD, for example caching it in memory.[3] One can create RDDs by applying operations called transformations, such as *map*, *filter* and *groupBy*, to the data in a stable storage system, such as the Hadoop Distributed File System (HDFS). As an example the following Spark code counts the words in a text file:

```
JavaRDD<String> file = spark.textFile("hdfs://...");
JavaRDD<String> words = file.flatMap(new FlatMapFunction<String, String>()
  public Iterable<String> call(String s) { return Arrays.asList(s.split(" "));}
});
JavaPairRDD<String, Integer> pairs = words.map(new PairFunction<String, String, Integer>()
  public Tuple2<String, Integer> call(String s) { return new Tuple2<String, Integer>(s, 1);}
});
JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new Function2<Integer, Integer>()
  public Integer call(Integer a, Integer b) { return a + b; }
});
counts.saveAsTextFile("hdfs://...");
```

The first line of code loads the text file from the Hadoop Distributed File System to be processed. Then *flatMap* function is used to get the words as a list, for that purpose text file is considered as a *String* and is split by ‘space’ delimiter. After that, *map* function is used to assign parameter ‘1’ to each word and return the pair of word and ‘1’ as a value. Finally *reduceByKey* is used to sum up ‘1’-s for each word. As a result each word will have some number attached to it and it will be possible to get the number of appearances for each word inside the loaded text file.

To sum up the main Spark features are listed below:[4]

- Spark supports three programming languages: Java, Scala and Python, providing necessary API-s
- Spark provides the ability to cache datasets in memory for interactive data analysis
- Reciprocal command line UI (in Scala and Python)
- Proven scalability to 2000 nodes in the research lab (EC2) and 1000 nodes in the production
- If data doesn`t fit into the memory, hard drive is used instead
- It is possible to replicate data in memory or on the hard drive
- Spark have additional operations for data processing like *groupBy* or *filter*

2.2 MapReduce

MapReduce is a programming model created to work on large-scale data sets. Map function is specified in order to process key/value pair and return modified key/value pairs as the result. Reduce function is meant to merge intermediate values produced by map function with specific intermediate key.[5]

Particularly map function receives an input as key/value pair and produces a number of intermediate key/value pairs. The MapReduce framework batches together values bounded with the same key 'K' and transfers them to the Reduce function. The Reduce function accepts the transferred key 'K' and a set of values associated with that key. Reduce function then groups together received values in such way, so it can generate a smaller set of values. In most cases no values or just one resulting value is returned by a single reduce function.[5] Figure 1 illustrating the MapReduce process can be seen below:

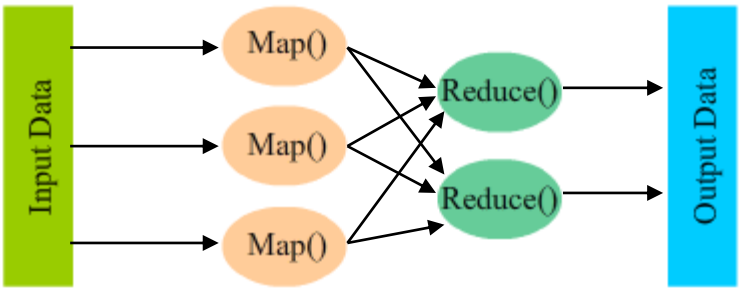

Figure 1: MapReduce process.

On this figure one can see that input data is split into the groups of separate data sets, then each group is processed by its map function. Finally the results of the map functions are combined inside the reduce functions and output data is returned to the end user.

Apache Spark uses MapReduce paradigm for large-scale data processing. The main concept lies in using two functions Map and Reduce. For the intermediate data it is possible to use additional functions available in Spark framework, like *filter*, *groupBy* or *count*. Due to RDD support, intermediate data can be cached in memory and reused in a fast manner. Also RDD's are fault tolerant and there is no need to worry about the data safety. Additionally it is possible to use HDFS to store and retrieve necessary data, for example as a text file.

2.3 Hadoop

Apache Hadoop is an open-source framework for storing and processing large-scale data sets. Hadoop was created by Doug Cutting and Mike Cafarella in 2005. The reason why it was

created is because Google didn`t allow others to use their MapReduce framework. The concepts of Hadoop is in loading the data, processing it using different worker nodes by applying the Map function and collecting the intermediate data from all worker nodes running the Reduce function.[6][7] This idea works well if there is no need to run the same process in a sequence. But if such a need will occur then this process will not be very effective.

In a simple Hadoop program user just defines Mapper and Reducer classes along with some additional parameters:

```
job.setMapperClass(PkNnMapper.class);
job.setReducerClass(PkNnReducer.class);
```

After that the framework will handle the rest of the work using the classes provided. The whole logic is located inside those classes. As a result there will be an output data stored, for example, inside HDFS. One possible Map class structure is shown below:[8]

```
public class Mapp extends Mapper<LongWritable, Text, Text, DoubleWritable>
{
        protected void map(LongWritable key, Text value, Context context)
        {
                //Code defined by the user
                //Write result to the context
                context.write(new Text(...), new DoubleWritable(...));
        }
}
```

2.4 Differences between Hadoop and Spark

The idea of the Spark framework is to enable iterative runs on the data sets. Apache Spark concept is slightly different from what Hadoop has. With Spark framework user loads the data and after that one can apply so much Map functions as it is needed, receiving intermediate data after each Map function. In the end of process it is possible to collect all the intermediate data and save it, for example, inside HDFS. Besides the map function Spark framework offers a bunch of other useful functions like *count*, *filter* or *groupByKey* that can be also applied to the intermediate results.

In a simple Spark program user defines each map function separately for each iteration. There can be tens of such a functions and the content of each of them will run in parallel. Here is a small example of Spark program:

```
JavaPairRDD<String, String> mapData = testData.map(new MapElement(mData);
JavaRDD<String> result = mapData.map(new SecondMapElement());
```

In this example one can see how multiple map functions are applied in a sequence. The process starts by applying *map* function of MapElement class on *testData* input. After that second *map* function located inside the SecondMapElement class is applied to the result of the previous step. This process can be repeated many times producing necessary intermediate results. Additional examples can be found in the publications released, the one of them is "Performance and Scalability of Broadcast in Spark"[9]. The author of this article gives an overview of Spark framework and shows some examples using this framework: text search, alternating least squares and logistic regression.

2.5 Hadoop Distributed File System

Both Hadoop and Spark frameworks use HDFS for storing the input/output data in a reliable manner. The Hadoop Distributed File System (HDFS) is a distributed file system created to run on qualifying hardware. HDFS is highly fault-tolerant and is meant to be deployed on hardware which cost is low. HDFS provides high throughput access to the data of the application and is fitting well for approaches that have large data sets.[10] HDFS implementation is based on master/slave structure. An HDFS cluster has one Name Node, this node fulfill the role of a master server that is responsible for operating on the file system namespace and controlling the access to files. HDFS cluster has also a number of Data Nodes, in most cases one per worker. Data nodes are responsible for managing the repository attached to them. HDFS exhibit a file system namespace and makes it possible for the user data to be saved in text files. Inside HDFS file is divided into several chunks, those chunks are saved on Data Nodes. The Name Node performs different operations on the file system namespace, for example: open, close and rename. Name Node also defines how blocks will be mapped on Data Nodes. The Data Nodes are processing read and write requests from the user of the file system. Data Nodes also create, delete and replicate blocks when specific commands are received from the Name Node.[10] Hadoop Distributed File System architecture can be seen on the Figure 2 below:

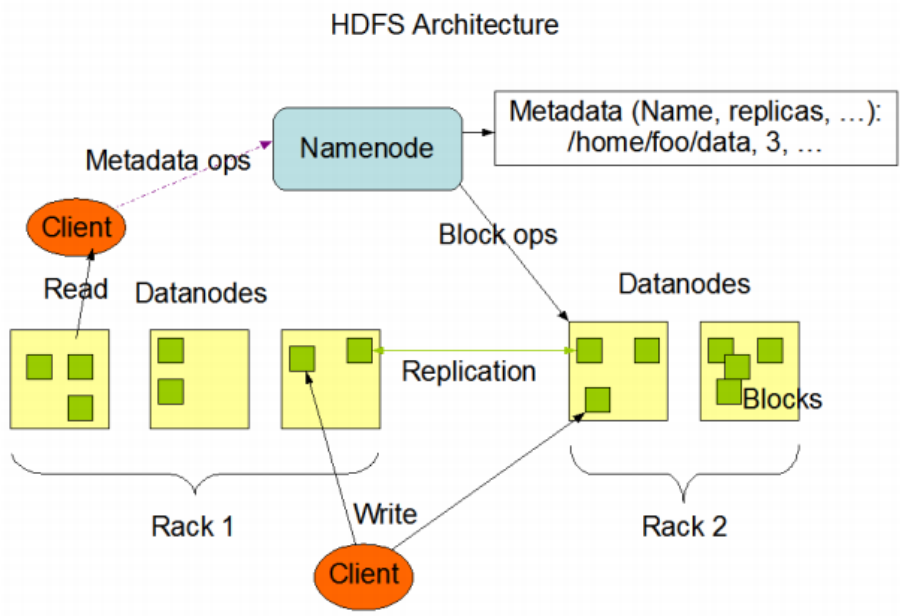

Figure 2: Hadoop Distributed File System architecture. Image is taken from [10] and [11]

On this figure one can see how the architecture of HDFS looks like. The green rectangles represent the data nodes and the blue rounded rectangle represents the name node. The figure also shows that stored data is replicated across the different racks. Two green rectangles on the right are located in the Rack 2. Three green rectangles on the left are the replication of the data stored inside the Rack 2. Racks are different machines, possibly in different networks. To reduce the possibility of losing any important data, records are distributed between different racks. In that case even if some machine or even network will stop working, data will not be lost completely and it will be relatively easy to restore it in a full size.

Section 3

Algorithms

In these section three algorithms chosen for the implementation using Spark framework will be described. Each algorithm will be studied separately. It will be shown how algorithms can be implemented using Spark and Hadoop frameworks and what the fundamental difference is.

3.1 Choice of algorithms

Three different algorithms were selected to illustrate the different approach of implementation using the Spark framework. To show the difference between implementations it was decided to implement the same algorithms, in addition, using Hadoop framework. Algorithms that were chosen are:

- K-nearest neighbors algorithm
- Naïve Bayesian algorithm
- CLARA algorithm
- Additionally PAM algorithm will be considered as a part of CLARA algorithm and described in the same section

3.2 Parallel k-nearest neighbor's algorithm

The k-nearest neighbor's algorithm (k-NN) is a method used for classifying the objects based on the closest training example. The k-nearest neighbor's algorithm is one of the simplest algorithms. The object is classified according to the 'k' nearest training examples. The new object will be assigned to the class of training example, if the number of training examples of that class is greater than the number of all other example classes in a range of 'k' near the new object. 'k' is a positive integer, typically small.[12]

An example of classification of the new object is shown on the Figure 3. One need to classify new object (poly star), which color is blue. If k = 3 then object will be classified as a green circle, if k = 5, then object will be classified as a red rectangle. The class depends from a number of classified objects. If k = 3, then the number of green circles is greater than the

number of red rectangles (2 green and 1 red). In a second case (k = 5), the number of red rectangles is greater (3 red and 2 green).[13]

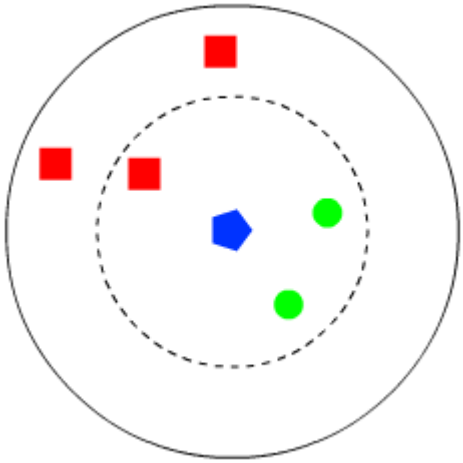

Figure 3. Example of the k-NN algorithm.

For the k-nearest neighbors algorithm, the computation of similarity between training and testing samples is independent, thus the algorithm can be easily extended to parallel mode. The input dataset can be split into 'N' number of blocks, which can be at the same time processed on the different nodes.[14] For parallel k-NN, the parameters include the number of nearest neighbor's 'k', the paths of the input and output files, the number of map tasks and reduce tasks. One of the most important parts of implementing parallel k-NN algorithm is the design of key/value pairs. For the map function, the input is the training data set and the sample "s" from the testing data set. Sample "s" includes three parts: the identification, the "x" coordinate and the "y" coordinate. Algorithm 1 gives the pseudo code of the map function for parallel k-NN implementation. Reduce function is redundant for this algorithms and can be skipped.

Algorithm 1. map (key, value)

Input: the training dataset 'tds' and the sample 's' from the testing dataset

Output: < key, value > pair, where key is id, value is the predicted group

```
parse the x sample value as < id, x, y >;
for each tds
        parse the tds sample value as < id', x', y', group >;
        compute the similarity S = sim(x, x', y, y');
        value = < S;group >
        save in values < key, value >
```

```
end for

sort values by distance

for number of neighbours
      find the group with maximum number of appearance
end for

output < key, value >
```

3.2.1 Parallel k-nearest neighbor's algorithm in Spark

The implementation of parallel k-NN algorithm in Spark is divided into two parts. In the first part data is loaded and parsed. In the second part data is processed. As an optimization it was decided to keep the whole logic inside the single map task. That way the implementation will run faster.

For the purpose of implementing the parallel k-NN algorithm it was decided to use a text file with point coordinates. That file will be located inside HDFS and loaded before starting the classification itself. After the file is loaded, its content will be parsed and distributed between map tasks for further processing. The file looks like that:

- 0;148;174
- 1;263;272
- 2;280;276
- 3;104;134
- 4;150;288
- …

Each point is separated by the new line, 'x' and 'y' coordinates are separated with semicolon ';'. In addition, the first number is the point id that is also separated with a semicolon. That way point vector appearance is: 'id;x;y'. This file is randomly generated. The limits for each point are from 0 to 1000. To make it possible to show the correctness of the algorithm additionally predefined points were generated. Those points were generated using certain ranges, for example from 100 to 300 that represents cluster with id 1 from the training data set. That way it will be possible to see on the figure that points are clustered in a correct way. The figures will be shown in the following sections.

To process the data and classify it, it is needed to generate train data set that will have point coordinates with cluster id attached to each of them. Train data file will be located inside HDFS and loaded before the test data file. This data will be passed to each map task to do all needed calculations. Train data file is very similar to the test data file, but in addition it has cluster id, separated from the coordinates with a semicolon. The following list represents the part of the train data file:

- 0;1;1;0
- 2;2;2;0
- 3;20;21;1
- 6;28;21;1
- 7;50;50;2
- 9;56;51;2
- …

This file was generated by using a simple "for" loop. The number of iterations is 1000. During each iteration point coordinates are randomly generated and cluster id is decided depending from the point location on the coordinate's grid. The criteria are as shown below:

- If x > 100 and y > 100 and x < 300 and y < 300 then cluster id = 1
- If x > 700 and y > 700 and x < 900 and y < 900 then cluster id = 2
- If x > 500 and y > 500 and x < 600 and y < 600 then cluster id = 3
- If x > 0 and y > 0 and x < 50 and y < 50 then cluster id = 4
- If x > 800 and y > 50 and x < 1000 and y < 150 then cluster id = 5

As a result train data file will be generated containing a list of points that are associated with cluster id from 1 to 5.

When train data set and test data set are generated, it is possible to classify the points from the test data set. The whole calculation process is done inside the single map function. Additional map function is used to parse each point string to appropriate class instance. When points are parsed, main map function is called. The point is received and the distance between this point and every point from the training data set is calculated. Distance with appropriate group taken

from the training data point is saved. The pseudo code representing that process can be seen below:

```
parse the point p

for each point p1 in training data set do
      calculate distance between p and p1
      add distance with p1 cluster id to the resulting list pList
end for
```

When first part of the map function has completed its work, the second part is started using the output of the previous section as an input. The data is sorted by the distance. As a result the list of cluster id-s with their distances is received. Then first specific number of values, defined by the user, is selected according to the number of neighbors. Finally the group with maximum appearance number from the selected values is chosen as a resulting cluster. The pair of value id and cluster id is returned as a result of this map function. The pseudo code describing this process can be seen below:

```
sort list of pairs pList by distance
select n number of points from pList, where n is a number of neighbors
select the point p that appear in the selected list more than all other
points
return pair of point id and cluster id of the selected point
```

As a result each point from the testing data set is associated with proper cluster id. Finally the list of pairs < point id, cluster id > is received from all worker nodes and saved inside HDFS.

3.2.2 Differences with Hadoop implementation

The implementations of parallel k-NN algorithm in Hadoop and Spark frameworks are almost identical. The difference is in framework syntax and the way code is written. The advantage of Spark framework is in its simplicity. If non-involved person will see Hadoop and Spark code one will understand the code logic of Spark framework much quicker than Hadoop code. It can be seen right away where train data is passed, where parsed and where used. With Hadoop it is much harder to follow the code logic.

3.3 Parallel naive Bayesian algorithm

A naive Bayes classifier is a straightforward probabilistic classifier constructed upon applying Bayes theorem with strong independence assumptions. A naive Bayes classifier conjecture that the presence or absence of a particular property of a class is unrelated to the presence or absence of any other property of the specified class variable. For example, a fruit can be an apple, if it is green, round, and 3" in diameter. Even if those properties rely on each other a naive Bayes classifier takes them into consideration independently when deciding that this fruit is an apple.[15][16] Bayes' Theorem finds the probability of an event that will occur, using the probability of another event that has already occurred. If B appear for the dependent event and A represents the preceding event, Bayes' theorem can be specified like that: [17]

```
Prob(B given A) = Prob(A and B)/Prob(A)[17][18]
```

To calculate the probability of event B with event A specified, the formula counts the number of occurrences where A and B happen together and divides it by the number of occurrences where A happen alone.[17]

For better understanding of this algorithm, it will be described in details in the next sub-section. It will be described how train data is generated, what parameters are considered and what formulas used.

3.3.1 Parallel naive Bayesian algorithm in Spark

To implement parallel naïve Bayesian algorithm train data is needed, the similar way it was needed for parallel k-NN algorithm. It was decided to store train data in a text file that will be located in HDFS. In addition to generation, train data needs to be processed and intermediate results received.

To show how this algorithm can be implemented in Spark framework, male/female classification problem was considered. This is a problem where one needs to decide is person a male or female taking into the account the attributes provided. The attributes considered are: weight, foot size and height.

Initial data is located in a text file. This data represents person information that will be processed. Person information is divided with a new line, each person data is divided with semicolon ‘;’. The structure of the text file can be seen below:

- male;5.58;170;12
- male;5.92;165;10
- female;5;100;6
- female;5.5;150;8
- female;5.42;130;7
- …

The first parameter is male/female attribute, second is height, third weight and the last one is foot size. To apply Bayesian theorem additional attributes are needed, those are: mean height, mean weight, mean foot size, height variance, weight variance and foot size variance. To calculate additional attributes each line is considered separately for male and female parameters. First of all, means are calculated for each attribute separately for male and female parameters using the formula below:

$$\text{Mean} = \text{sum of elements} / \text{number of elements}$$ [19]

The next step is to calculate the variance in the same way using the formula below:

$$s^2 = \Sigma (x_i - x_m)^2 / (n - 1)$$ [19]

Where ‘s’ is variance, ‘x_i’ is current element, ‘x_m‘ is mean and ‘n’ is a number of elements.

As a result mean and variance of each attribute are saved with male/female attribute in the text file in HDFS. This file will be used to calculate naïve Bayesian probabilities. The resulting file will look as shown below:

- female 5.4175;0.097225;132.5;558.333333;7.5;1.666666
- male 5.855;0.03503;176.25;122.9166666;11.25;0.9166666

The first parameter is male/female attribute, second is variance height, third variance weight and the last one is variance foot size.

When train data is ready to be processed it is needed to generate test data set. It will be saved in a text file in HDFS and loaded by the java program. This file will have the following structure:

- 0;4.82;182.13;5.46
- 1;4.82;160.25;5.24
- 2;5.55;114.72;4.05
- 3;5.52;193.94;9.24
- 4;4.61;120.66;3.15
- …

Here the first number is data id, second is height, third weight and the last one is foot size. This file is generated randomly using proper limits for each attribute. Those limits are shown below:

- minimum height: 4
- maximum height: 6
- minimum weight: 100
- maximum weight: 250
- minimum foot size: 3
- maximum foot size: 13

To start with classification first of all data is parsed. Then probability values are calculated for each attribute from the training data set separately. When this is done, resulting label is decided. Finally the label with the greatest probability is chosen. The pair of sample key and resulting label is returned as a result. The pseudo code describing that process is shown below:

```
parse sample data

for each line data in train data do
      calculate the probability of the person to be male and female
end for

get the attribute with the greater probability (male or female)
return a pair of < data id, attribute >
```

As a result list of data is received containing the data pairs. This text file is then saved in HDFS.

3.3.2 Differences with Hadoop implementation

The implementations of parallel naïve Bayesian algorithm in Hadoop and Spark frameworks are almost identical. The same as it was with k-NN algorithm the difference is in framework syntax and the way code is written. The advantage of Spark framework is in its simplicity. If non-involved person will see Hadoop and Spark code one will understand the code logic of Spark framework much quicker than Hadoop code. It can be seen right away where train data is passed, where parsed and where used. With Hadoop it is much harder to follow the code logic.

3.4 Clara algorithm

Clustering Large Applications (CLARA) is the k-medoid clustering algorithm. It was designed by Kaufman and Rousseeuw to handle large data sets.[20] The difference between PAM and CLARA algorithms is that instead of finding representative objects for the entire data set, CLARA draws a sample of the data set and performs actions on it. If the sample is taken out in an enough occasional way, the medoids taken will represent roughly the medoids of the total data set.

The main idea of this algorithm is to divide objects in the dataset into ‘k’ clusters (classify them into ‘k’ classes) by iteratively applying k-medoid clustering on a random small subset of the data and choosing the best out of them as the result. First of all the number of experiments is determined and exact number of points is selected for each experiment from the entire data set in a random manner. The number of points selected can be, for example, 500 + 2k, where ‘k’ denotes the number of medoids. After that, PAM algorithm is applied for each experiment to find the set of the best medoids for each sample. The next step is to divide the points from the entire data set between received medoids for each experiment. This division is done by calculating the distance between each point and each medoid. The point then is associated with the cluster with the minimum distance between the point and medoid. After that cluster cost is calculated for each experiment by adding distances between points and their medoids together. The cluster with minimum cost is concerned as the best solution and returned as a result.[21][22]

Exact CLARA algorithm consists of the steps below:[23]

1. Select ‘E’ different samples from the data set ‘n’, ‘E’ is a number of experiments. The number of points inside each sample is 500 + 2k, where ‘k’ is a number of medoids.
2. For each sample ‘S1’ from the list of samples ‘S’ do:
   - Apply PAM to sample ‘S1’ to determine ‘m’ medoids
3. Associate each point from the data set ‘n’ with appropriate medoid
4. Determine the cost of the clustering by summing up the costs of each cluster
5. Choose medoids from the clustering with minimum cluster cost

3.4.1 PAM algorithm

PAM (Partitioning Around Medoids) was developed by Kaufman and Rousseeuw. To find ‘k’ number of clusters, it finds the random object for each cluster. This object is called a medoid. In the end it has to be the most centrally located object in the cluster. After that, when random objects are selected for each cluster, each non-selected object is matched with the proper medoid, for example it is closest to.[20][21]

To find ‘k’ number of medoids, PAM begins with an arbitrary selection of ‘k’ objects. Then in each step, a swap between a selected object ‘m’ and a non-selected object ‘p’ is made, as long as such swap would result in an improvement of the quality of the clustering.

Exact PAM algorithm consists of the steps below:[24]

1. Select ‘k’ points from the ‘n’ data set randomly to represent the medoids of each cluster
2. Repeat until there is no change in medoids:[25]
   - For each point ‘p1’ from the data set ‘n’ do:
     - Calculate distance between ‘p1’ and each of the chosen medoids
     - Select the closest medoid to the point and associate ‘p1’ with selected medoid
     - For each cluster ‘c’ do:
       - For each point ‘p’ from the cluster ‘c’ swap the point ‘p’ with currently selected medoid ‘m’ and calculate the cost of the cluster. If the cost is lesser than it was before, ‘p’ becomes the new medoid of the cluster ‘c’,

else iteration continues until the point with the minimum cluster cost is found.

- End for each

- End for each

3. End repeat

The cost of the cluster is calculated from the sum of distances between medoid and each point from this cluster. Figure 4 describing the PAM algorithm is shown below:

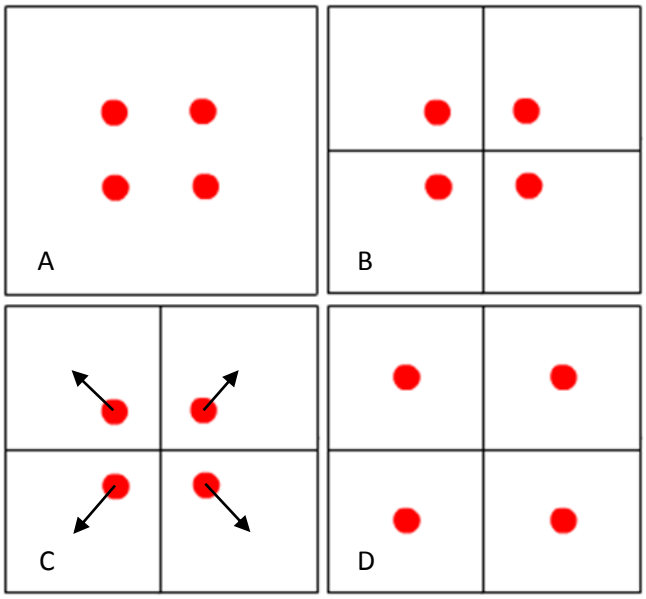

Figure 4: PAM algorithm. Select randomly 4 medoids. Divide data across medoids forming 4 different clusters. Recalculate medoid positions.

3.4.2 Clara algorithm in Spark

In a contrast to previous two algorithms Clara doesn`t require training data set. It is needed to have some input data represented by a text file stored in HDFS and loaded at runtime. The file structure can be seen below:

- 961;716
- 901;76
- 711;950
- 991;178
- 404;490
- …

The file is represented by the list of points, each point has ’x’ and ’y’ coordinates separated by semicolon ’;’. This file is generated randomly in two forms. First one has a lot of points in it generated from 0 to 1000, the second one has predefined list of points to show the correctness of the implementation. For the second form predefined limits are used, those are shown below:

- If x > 200 and y > 200 and x < 400 and y < 400 then cluster id = 1
- If x > 700 and y > 700 and x < 900 and y < 900 then cluster id = 2
- If x > 500 and y > 500 and x < 600 and y < 600 then cluster id = 3
- If x > 0 and y > 0 and x < 100 and y < 100 then cluster id = 4
- If x > 800 and y > 50 and x < 1000 and y < 150 then cluster id = 5

When files are generated the proper one is uploaded to HDFS. To show the correctness of the implementation the file with 5100 predefined points is used. The results are shown in the sections below. Randomly generated file is used to measure the time taken by the framework to process the large data set.

The implementation of the Clara algorithm can be split into 4 steps:

- Select ‘n’ number of samples from the whole data set, where ‘n’ is a number of experiments
- Apply PAM algorithm to each sample
- Calculate cluster cost by using medoids received after PAM algorithm
- Select medoids with the lowest cluster cost

The first step is to split the whole data set into ‘n’ groups of data. For that purpose each point gets assigned with exact number that is in a range 0 – n, where ‘n’ is a number of experiments. After that, points are grouped by the number assigned, that way ‘n’ number of samples is received.

The second step is to apply PAM algorithm to each of generated samples. For that purpose each sample is handled separately. This process can also be split into several steps:

1. Get the data sample

2. Randomly select ‘n’ number of points from the data sample, ‘n’ is predefined number of points.
3. Randomly select ‘m’ number of medoids from the selected ‘n’ points, ‘m’ is predefined number of medoids.
4. Assign each point to the medoid it is nearer to.
5. Recalculate medoid position
6. Repeat steps 4 to 5 as long as there is no change in medoids
7. Return the list of medoids

The third step is to calculate the cluster cost of each sample. This is done by summing up the costs of each medoid in a data sample calculated during the previous step. During the last step the sample with minimum cluster cost is selected and medoids of this sample are returned as a result of the whole process. The result is saved in a text file in HDFS.

When best medoid set is found it is very easy to classify new data input. When new point is received the only thing it is left to do is to calculate distances between this point and each medoid received. Medoid with minimum distance to the point is considered as the cluster the new point belongs to.

3.4.3 Differences with Hadoop implementation

The implementation of Clara algorithm using the Spark framework is slightly different from the implementation using Hadoop. Hadoop implementation has two jobs defined as grouper job and cost job. The grouper job map function divides the whole data into ‘n’ number of samples the same way it was done inside the Spark implementation. Grouper reduce function applies PAM algorithm to each sample received from the grouper map function. After that cost map function calculates distance between each point from the initial data set and each medoid received from the grouper reduce job. Finally grouper reduce function calculates the cluster cost for each medoid set. The resulting medoid set is determined inside the Main class by getting the set with lowest cluster cost.

Chapter 4

Validation and Benchmarking

In this chapter Hadoop and Spark frameworks will be compared through benchmarking the implementations of the chosen algorithms. Results will be presented in the form of tables. Additionally to show the correctness of the implementations results will be visualized and presented using the charts. In addition to show how well algorithm implementations are parallelized, parallel speedup charts will be shown for each algorithm. In the first part of this chapter cluster configuration will be described, that embrace cluster technical configuration, data input sizes and number of worker nodes used. Finally the results of each algorithm will be considered separately.

4.1 Cluster Setup

To test the implementation of the algorithms Tartu University Mobile & Cloud Laboratory local cloud was used. Tests were done with different number of worker nodes and different input file sizes. The sizes of the datasets used in classifications were:

- 1 000 000 data inputs
- 3 000 000 data inputs
- 5 000 000 data inputs
- 10 000 000 data inputs
- 100 000 000 data inputs (for naïve Bayesian algorithm)

Different input sizes were used to show how Spark framework can handle different amount of data. For the naïve Bayesian algorithm additional data input size was introduced because implementation runs very fast and it is difficult to show the scalability of the implementation.

To see how fast Spark framework can process the data, different number of worker nodes was used. The number of workers is: 1, 2, 4 and 8. Running the tests on different number of workers will show how scalable the algorithm implementation is and how good it can be parallelized across the worker nodes. For speed up diagrams generation the greatest amount of

data was used. The only exception is Clara algorithm where with 10 000 000 data and 1 worker node job failed. In that case 5 000 000 data results are taken into account.

The cluster has one master node that handles the workflow and a number of worker nodes that are processing the data. Master and worker nodes have the same configuration, configuration parameters are listed below:

- RAM: 2 GB
- Number of virtual CPU-s: 1
- Disc space: 20 GB

The Tartu University cluster consists from two HP ProLiant DL180 G6 servers, each one have:

- Two 4-core CPU's (Xeon E5606)
- 32 GB of RAM
- 2x2TB hard disks
- two gigabit NICs

Each server has two Xeon E5606 CPU's running at 2.13 GHz clock speed. It has four cores but it is running on hyper threading that makes it eight cores. The operating system is Ubuntu 12.04.1 LTS 64 bit.

4.2 Parallel k-nearest neighbor's algorithm

To test the algorithm the number of neighbors was chosen to be 5. Results received by running the implementation on Spark framework are listed inside a table below:

	1 worker	2 workers	4 workers	8 workers
1 000 000 data	41.186 s	25.768 s	20.921 s	24.446 s
3 000 000 data	99.474 s	55.879 s	34.742 s	24.62 s

5 000 000 data	158.742 s	85.646 s	51.065 s	32.295 s
10 000 000 data	313.629 s	185.905 s	88.79 s	53.126 s

Table 1: Parallel k-nearest neighbors Spark results

From the table it can be seen that the time taken for processing different amount of data grows with data input size. From the other hand the time is decreasing with increasing number of worker nodes used. That means the implementation of the algorithm is suitable for processing the data inside the cluster and with growing number of worker nodes the time taken for processing the data will become lesser with each additional worker.

4.2.1 K-nearest neighbor's result visualization

As it can be seen on the figure below, points are correctly clustered. For this figure special list of points was generated. The points were generated with specific ranges to show the correctness of the implementation. The number of points generated for this figure is 5200 points. 100 of them were generated in a range from 0 to 1000, to add some randomness. Other points were generated within specific ranges described in a section before. There are five clusters and as it can be seen, points received correct cluster id during the classification process.

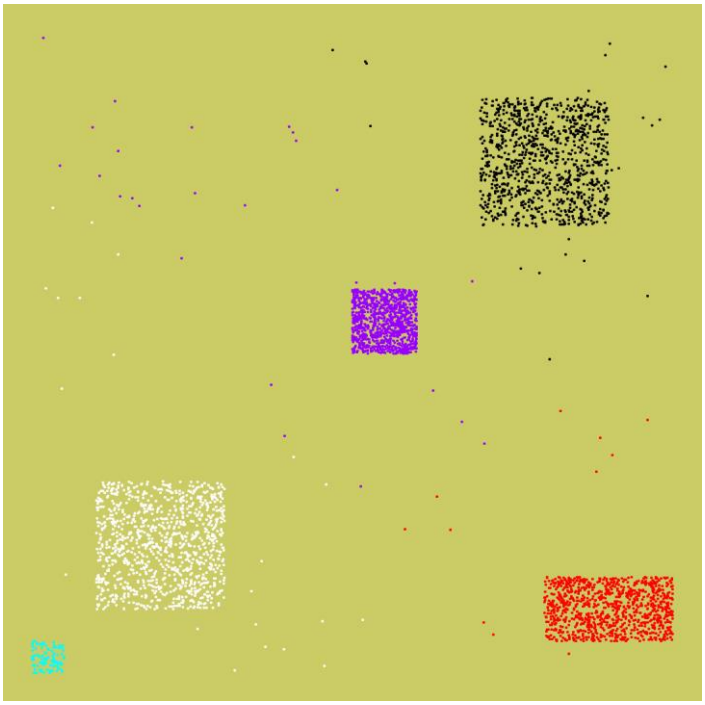

Figure 5: Points associated with proper cluster

4.2.2 Parallel k-nearest neighbor's algorithm Hadoop results

Results received by running the implementation on Hadoop framework are listed inside a table below:

	1 worker	2 workers	4 workers	8 workers
1 000 000 data	51.552 s	41.941 s	42.773 s	42.994 s
3 000 000 data	124.793 s	71.66 s	42.64 s	43.783 s
5 000 000 data	200.945 s	134.176 s	74.84 s	43.733 s
10 000 000 data	394.965 s	230.121 s	136.906 s	76.792 s

Table 2: Parallel k-nearest neighbors Hadoop results

From the Table 2 one can see that the running time of some cases (2 workers and 4 workers with 1 000 000 data) is very similar. This happens because of Hadoop optimization implementation. When data amount is small Hadoop runs only several map tasks, but if input size is large enough Hadoop starts more map tasks. In current situation the data input size for two and four workers is 1 000 000 and Hadoop starts only one map task, but for 3 000 000 input sizes Hadoop uses four map tasks. With increasing input data size this doesn`t happen and the difference in running time can be clearly seen. Figure 6 showing the difference in speed ups can be seen below:

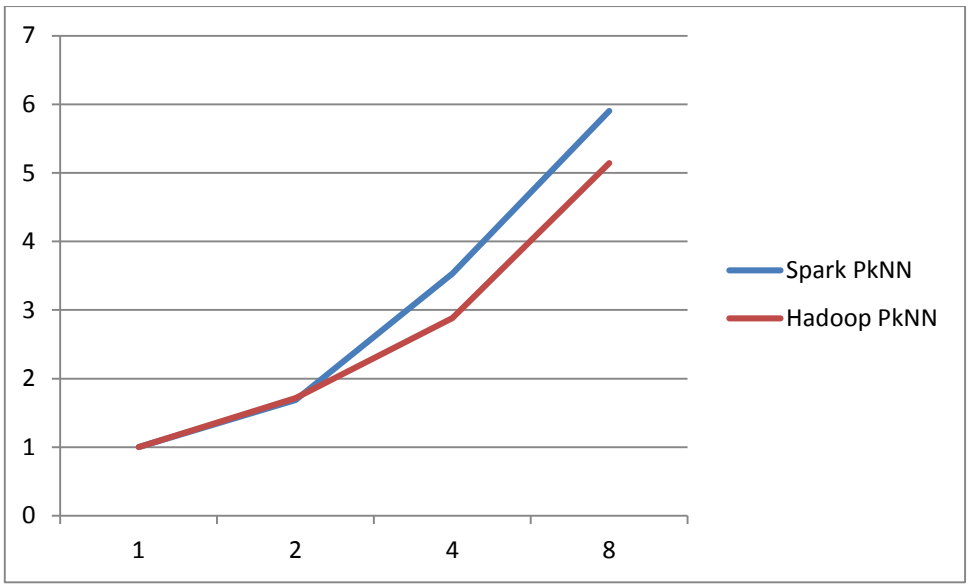

Figure 6: PkNN speed up diagram

From this figure one can see that the difference in speed ups is very small. Starting from the second worker Spark is showing more stable speed increase. From the other hand Hadoop isn`t left behind and goes side by side with Spark.

4.3 Parallel naive Bayesian algorithm

Results received by running the implementation on Spark framework are listed inside a table below:

	1 worker	2 workers	4 workers	8 workers
1 000 000 data	15.728 s	13.549 s	17.373 s	21.473 s
3 000 000 data	27.611 s	19.092 s	18.837 s	19.253 s
5 000 000 data	39.422 s	23.933 s	25.034 s	25.042 s
10 000 000 data	68.931 s	41.878 s	40.607 s	39.85 s
100 000 000 data	609.288 s	333.69 s	331.982 s	323.038 s

Table 3: Naïve Bayesian algorithm Spark results

One can notice that starting from using two worker nodes, time doesn`t differ much. The algorithm is very fast and one can think that this depends from the data input size. To check that 100 000 000 data input size was introduced, but as it can be seen, situation remains the same. Finally it was decided to try adding the complexity to the calculations to investigate whether the issue is related to the cost of data distribution and synchronization time in RDD's. As a result time received by single worker and 1 000 000 data was about 134 seconds, for two and four workers time was 71 and 46 accordingly. Having those results it was shown that time taken for data distribution between nodes and initialization takes more time than calculation itself. This means that Spark may not be suitable for algorithms with low computational complexity.

4.3.1 Parallel naive Bayesian algorithm Hadoop results

Results received by running the implementation on Hadoop framework are listed inside a table below:

	1 worker	2 workers	4 workers	8 workers
1 000 000 data	23.513 s	18.525 s	19.589 s	18.785 s
3 000 000 data	58.502 s	41.645 s	27.512 s	20.644 s
5 000 000 data	85.585 s	56.691 s	42.631 s	28.72 s
10 000 000 data	157.747 s	99.767 s	58.692 s	43.714 s
100 000 000 data	1435.005 s	876.661 s	458.944 s	247.453 s

Table 4: Naïve Bayesian algorithm Hadoop results

Figure 7, describing the speed up of naïve Bayesian algorithms can be seen below:

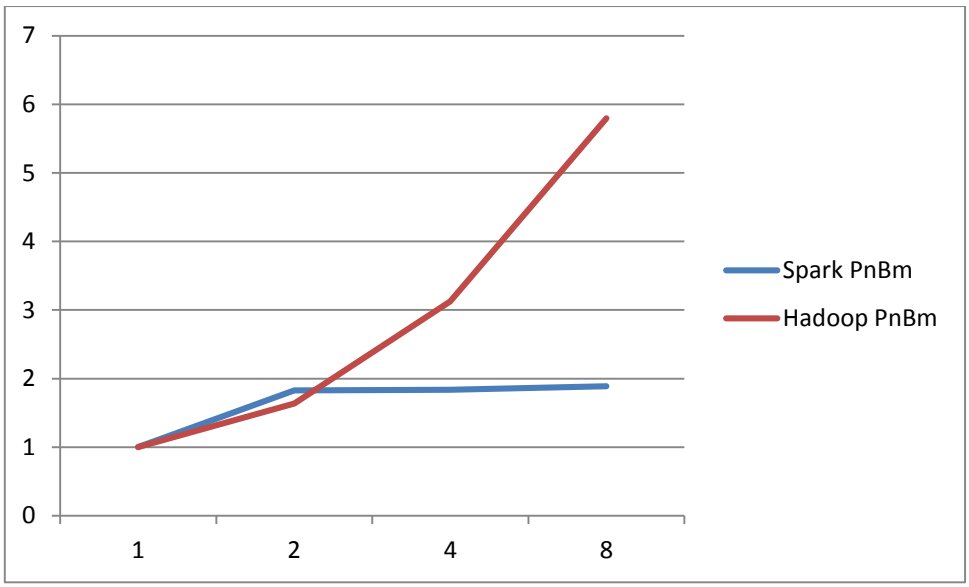

Figure 7: PnBm speed up diagram

As it can be seen on this figure Hadoop has shown the better speed up results than Spark framework. For the Spark the speed doesn`t grow up after addition of the new workers, but if the number of worker nodes is smaller than eight, then Spark is processing the data faster than Hadoop. From the other hand Hadoop has shown the stable speed up with every new worker added.

4.4 Clara algorithm

To run the implementation following parameters were used:

- Sample size: 512 inputs
- Number of medoids: 20
- Number of experiments: 100

Results received by running the implementation on Spark framework are listed inside a table below:

	1 worker	2 workers	4 workers	8 workers
1 000 000 data	171.724 s	95.368 s	59.551 s	35.33 s
3 000 000 data	489.22 s	253.926 s	132.286 s	88.246 s

5 000 000 data	877.86 s	412.43 s	209.059 s	116.895 s
10 000 000 data	Java heap space	860.567 s	423.058 s	218.22 s

Table 5: Clara algorithm Spark results

From the table it can be seen that the time taken for processing different amount of data grows with data input size. From the other hand the time is decreasing with increasing number of workers used. That means that implementation of the algorithm is suitable for processing the data inside the cluster and with growing number of nodes the time taken for processing will become lesser with each additional worker. In addition with 10 000 000 data input size Java heap space error was received, that means that a single worker node couldn`t handle the amount of data used. It was decided to compare the results of five million data elements for this algorithm.

4.4.1 Clara result visualization

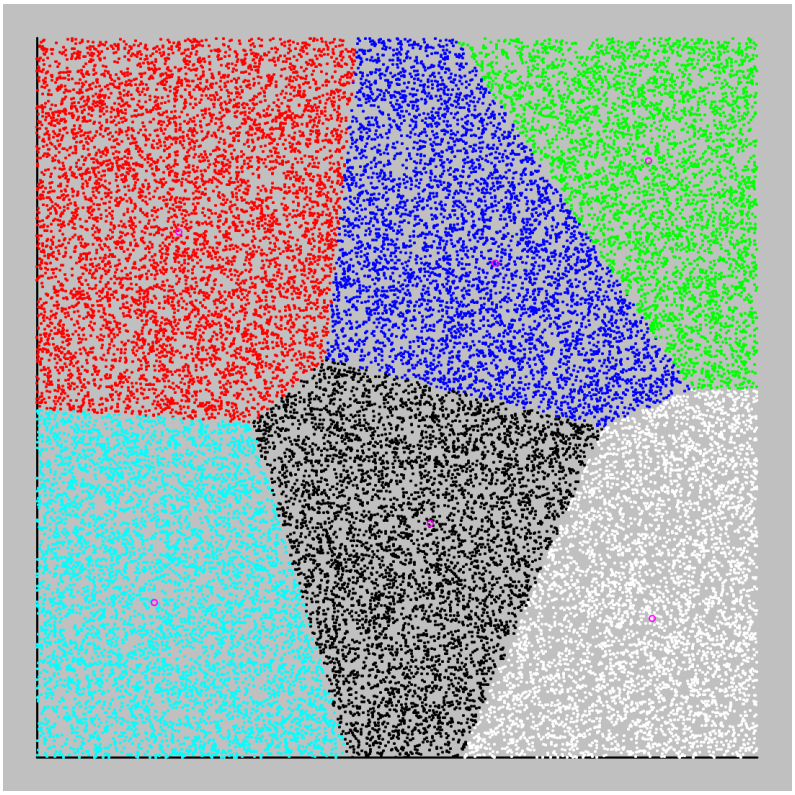

Figure 8: Large number of points associated with 6 medoids

On this figure it can be seen that data is clustered in a proper way and medoids are located in the center of each cluster.

To show the correctness of the implementation it was decided to reduce the number of data input, predefine location of the points and show the visualized output. The result can be seen on the figure below.

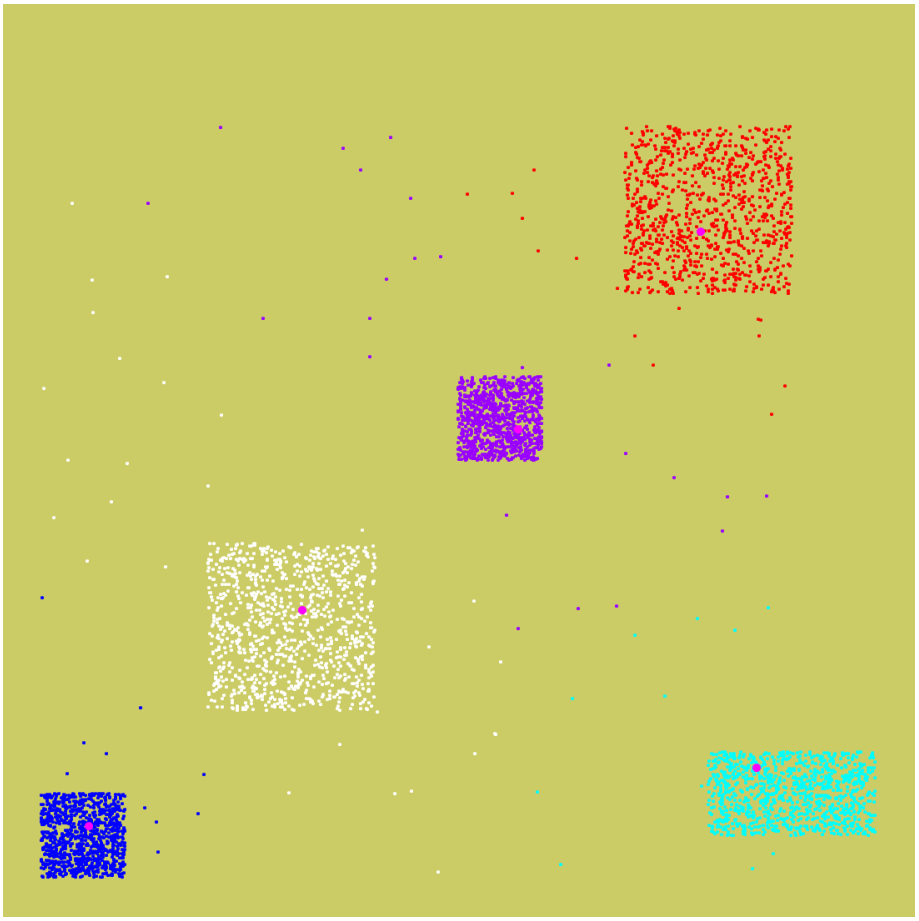

Figure 9: Predefined points associated with correct clusters

On this figure one can see that predefined input data is split into different clusters. After the processing of the data each point is assigned with specific medoid. As this figure illustrated Clara processed the data in a proper way and split the data between correct medoids. One can notice that medoids are not located exactly in the center of the clusters. This happened because of the additional points added outside the predefined regions. Clara is not very precise, but it is precise enough to split the main data correctly.

4.4.2 Clara algorithm Hadoop results

Results received by running the implementation on Hadoop framework are listed inside a table below:

	1 worker	2 workers	4 workers	8 workers
1 000 000 data	620.368 s	335.773 s	210.63 s	138.206 s
3 000 000 data	1733.69 s	1033.462 s	514.041 s	298.541 s
5 000 000 data	2840.191 s	1943.325 s	888.836 s	513.967 s
10 000 000 data	Failed	3733.413 s	2056.262 s	1088.805 s

Table 6: Clara algorithm Hadoop results

Figure 10 describing the speed up of Clara algorithm using Spark and Hadoop frameworks can be seen below. According to this figure Spark shows the better speed up with additional workers added. With 8 workers Hadoop showed only about 5.5 times speed increase, at the same time Spark could increase its speed for about 7.5 times from the initial speed with one worker.

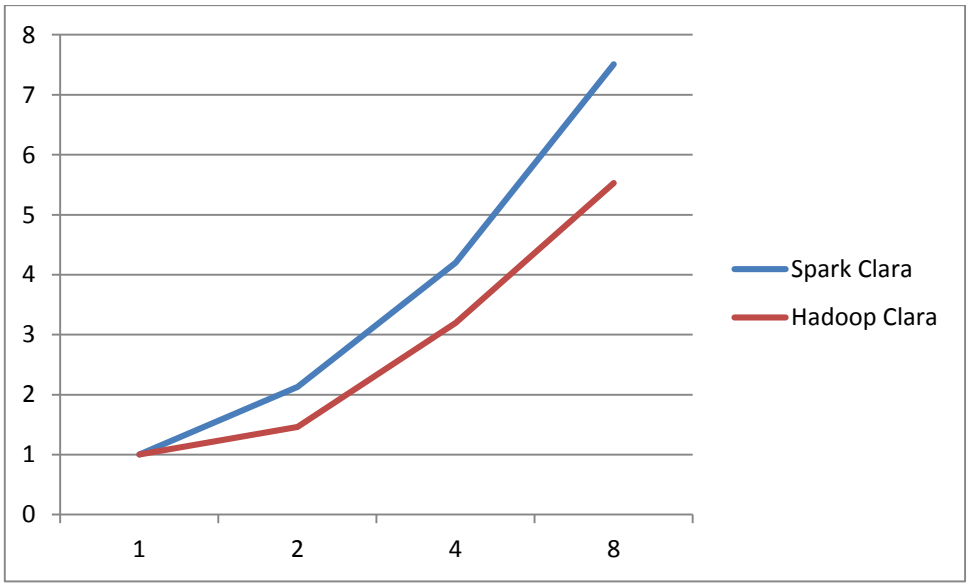

Figure 10: Clara speed up diagram

As it can be seen from the tables above, results received by Spark framework are better than the Hadoop implementation has shown. To reduce the running time of the Clara algorithm using Hadoop framework data was split into the smaller files. The number of files was the

amount of workers used. But even with this optimization Spark framework has shown the better results.

Chapter 5

Conclusion

In the scope of this thesis three algorithms were implemented using Hadoop and Spark frameworks: parallel k-nearest neighbor's algorithm, parallel naïve Bayesian algorithm and Clara algorithm. PAM algorithm was implemented as a part of the Clara algorithm. Those algorithms were chosen to show the suitability of Spark framework when dealing with different types of algorithms. Algorithms were implemented using two different frameworks to show the difference in running time and complexity of the implementation when using Hadoop and Spark frameworks.

As the result three algorithms were implemented and test input data was handled inside Tartu University cluster. The time taken by each algorithm with different input data sizes and different cluster configurations were gathered and saved inside the tables. Data input sizes were generated in a range from 1 000 000 to 10 000 000, special data input data size of 100 000 000 was generated for parallel naïve Bayesian algorithm. Different number of worker nodes was used to run the selected algorithm; this number is in a range from 1 to 8 nodes.

Results received show that Spark framework processes the data in a better way than Hadoop does. Results show that for parallel k-nearest algorithm results doesn`t differ much, but still Spark has shown the better running time. Additionally when dealing with Clara algorithm, Spark implementation has shown the better results than Hadoop framework even after splitting the data into smaller pieces for running time optimization. However it is not 10 or 100 times faster as the authors of Spark framework claim, at least for these algorithms. At the same time, these algorithms are not iterative.

Parallel naïve Bayesian algorithm should receive the special attention. The running time of this algorithm is very low, because of that it was strange that the running time doesn`t decrease with additional workers. During the tests it was found that the time taken is not the actual running time of this algorithm, but the time to distribute the data and initialization time. With this algorithm it was shown that Spark doesn`t handle well very fast algorithms. More precisely, the number of operations used on every data entity is very small. Because of that the addition of the new worker node didn`t afford any increase in data processing time.

Additionally speed up diagrams show the difference in running time with each additional worker added. For k-nearest neighbors algorithm the difference between Spark and Hadoop in speed ups is very small, that means that both frameworks are handling this algorithm pretty good. One can`t say the same for naïve Bayesian algorithm. The speed up of the Spark framework doesn`t almost change with additional workers, at the same time Hadoop is showing stable speed increase. From the other hand if the number of worker nodes is smaller than eight then Spark is processing the data faster than Hadoop does. For the Clara algorithm speed up diagram shows the better results for Spark framework, Hadoop couldn`t catch up with it.

The main goal of this thesis was to show that Spark framework is suitable for classification algorithms. The goal was fulfilled by implementing three different algorithms and showing the running time results. The second goal was to show that Spark is faster than Hadoop or at least not worst. With parallel k-nearest algorithm it was proven that Spark doesn`t run slower than Hadoop and even a little bit faster. With Clara algorithm it was shown that Spark can show better results than Hadoop when algorithm requires the initialization of multiple job runs. With naïve Bayesian algorithm it was shown that Spark doesn`t suite for all kinds of classification algorithms, it handles very fast algorithms in a worse way than Hadoop framework does. The only exception is if the number of worker nodes is less than eight, in this case the running time of Spark is higher than Hadoop could show.

References

1. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, Michael J. Franklin, Scott Shenker, and Ion Stoica, Fast and Interactive Analytics over Hadoop Data with Spark, August 2012
2. Apache Spark, http://spark.apache.org/ [04.11.2014]
3. Spark, an alternative for fast data analytics, http://www.ibm.com/developerworks/library/os-spark/ [04.11.2014]
4. Wikipedia, Apache Spark, http://en.wikipedia.org/wiki/Apache_Spark [04.11.2014]
5. MapReduce, http://searchcloudcomputing.techtarget.com/definition/MapReduce [04.11.2014]
6. Apache Hadoop, http://wiki.apache.org/hadoop/ [04.11.2014]
7. Hadoop, http://hadoop.apache.org/ [04.11.2014]
8. Hadoop : WordCount With Custom Record Reader Of TextInputFormat, http://bigdatacircus.com/2012/08/01/wordcount-with-custom-record-reader-of-textinputformat/ [04.11.2014]
9. Mosharaf Chowdhury, Performance and Scalability of Broadcast in Spark
10. Dhruba Borthakur, The Hadoop Distributed File System: Architecture and Design, 2007 The Apache Software Foundation
11. Running Hadoop on Ubuntu Linux (Single-Node Cluster), http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/ [04.11.2014]
12. Wikipedia, k-nearest neighbor's algorithm, http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm [04.11.2014]
13. K-Nearest Neighbor Classification, http://www.math.unipd.it/~aiolli/corsi/0708/IR/Lez12.pdf [04.11.2014]
14. Qing He, Fuzhen Zhuang, Jincheng Li, and Zhongzhi Shi: Parallel Implementation of Classification Algorithms Based on MapReduce, The Key Laboratory of Intelligent Information Processing, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China
15. Wikipedia, Naive Bayes classifier, http://en.wikipedia.org/wiki/Naive_Bayes_classifier [04.11.2014]
16. Based on Cloud Computing Environment, TELKOMNIKA IJEE Vol. 10, No. 5, September 2012,

http://iaesjournal.com/online/index.php/TELKOMNIKA/article/viewFile/1353/pdf [04.11.2014]

17. Naive Bayes, http://docs.oracle.com/cd/B28359_01/datamine.111/b28129/algo_nb.htm#DMCON018 [04.11.2014]
18. Parallel Naïve Bayesian Classifier, http://alitarhini.wordpress.com/2011/03/02/parallel-naive-bayesian-classifier/ [04.11.2014]
19. Standart derivation and variance, http://www.mathsisfun.com/data/standard-deviation.html [04.11.2014]
20. Raymond T. Ng, Department of Computer Science, University of British Columbia, Vancouver, B.C., V6T 1Z4, Canada, Efficient and Effective Clustering Methods for Spatial Data Mining, 20th VLDB Conference Santiago, Chile, 1994
21. Pelle Jakovits, Reducing scientific computing problems to MapReduce, Tartu, 2010
22. Maria Halkidi, Yannis Batistakis, Michalis Vazirgiannis, On Clustering Validation Techniques, Department of Informatics, Athens University of Economics & Business, Patision 76, 10434, Athens, Greece (Hellas), http://web.itu.edu.tr/sgunduz/courses/verimaden/paper/validity_survey.pdf [04.11.2014]
23. G.Kiran Kumar, P.Premchand, A Novel Hybrid Spatial Clustering Algorithm, International Journal of Engineering Research and Applications, Vol. 2, Issue 3, May-Jun 2012, pp.1746-1752, http://www.ijera.com/papers/Vol2_issue3/KL2317461752.pdf [04.11.2014]
24. The PAM Clustering Algorithm, http://www.cs.umb.edu/cs738/pam1.pdf [04.11.2014]
25. Lamiaa Fattouh Ibrahim, Manal Hamed Al Harbi, Using Modified Partitioning Around Medoids Clustering Technique in Mobile Network Planning, Department of Computer Sciences and Information System, Institute of Statistical Studies and Research, Cairo University Giza, Egypt, http://arxiv.org/ftp/arxiv/papers/1302/1302.6602.pdf [04.11.2014]

Appendices

Appendix A: Source code

Appendix B: Implemented algorithms

Appendix A

Source code

Source code is located on the external DVD or attached to the thesis, in the folder '/'. It includes the source code for each implemented algorithm and script codes for data generation. DVD is provided with this thesis.

Appendix B

Implemented algorithms

Jar files containing the compiled algorithms is located on external DVD or attached to the thesis in the folder ‘/bin’ of each algorithm project. DVD also contains the readme.txt where instruction how to run the jar files is described. DVD is provided with this thesis.

Non-exclusive licence to reproduce thesis and make thesis public

I, Sergei Laada,

1. Herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright, Suitability of the Spark framework for data classification, supervised by Pelle Jakovits.

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tallinn, 04.11.2014