



**TRANSILVANIA University of Braşov**  
**Faculty of Mathematics and Computer Science**  
**Specialization in Internet Technologies**

# Master Thesis

Author:  
Supervisor:

**Păpară Mădălina**  
**Lecturer Lucian Sasu, Ph.D.**  
**Grigoraş Cosmin**

Braşov  
July 2014

TRANSILVANIA University of Braşov  
Faculty of Mathematics and Computer Science  
Specialization in Internet Technologies

Master thesis

# **Pattern classification using Hadoop**

*Author:*

Păpară Mădălina

*Supervisor:*

Lucian Sasu  
Grigoraş Cosmin

Braşov  
July 2014

# Contents

<b>Introduction</b>	<b>5</b>
<b>1 Problem Statement</b>	<b>6</b>
<b>2 Technologies</b>	<b>10</b>
2.1 Apache Hadoop . . . . .	10
2.1.1 Map Reduce . . . . .	10
2.1.2 Apache Hadoop . . . . .	11
2.2 Apache Maven . . . . .	15
<b>3 Classification Algorithms</b>	<b>16</b>
3.1 K-Nearest Neighbors . . . . .	16
3.2 Naive Bayes . . . . .	17
<b>4 Implementation</b>	<b>19</b>
4.1 Data Preprocessing . . . . .	19
4.1.1 Aggregate duplicate measurements . . . . .	19
4.1.2 Reduce frequency in file . . . . .	21
4.1.3 Merge and format results . . . . .	22
4.2 K-Fold Validation for KNN . . . . .	23
4.3 K-Fold Validation for Naive Bayes . . . . .	25
4.3.1 K-Fold validation framework . . . . .	25
4.3.2 Naive Bayes implementation . . . . .	28
<b>5 Results</b>	<b>30</b>
5.1 Execution time . . . . .	30
5.2 Accuracy . . . . .	33
<b>6 Validation of Results</b>	<b>38</b>
<b>7 Developer Manual</b>	<b>42</b>
<b>Conclusions</b>	<b>46</b>
<b>Bibliography</b>	<b>47</b>



# Introduction

Big Data technologies have gained a lot of attention and interest in recent years. Lots of data coming from logs, sensors, transactions gathered over the years **must now be analyzed and interpreted.**

An example is Gas sensor arrays in open sampling settings, with a machine learning benchmark available for download [3]. The data set consists of measurements coming from 72 gas sensors collected in 16 months. The problem is gas discrimination: to identify correctly between 10 potential dangerous gases under complex circumstances.


This paper intends to propose a solution for the classification problem in addition to the one proposed by the donors of the data set [17].

The structure of this paper is as follows: chapter 1 describes in detail the subject of the thesis, chapter 2 briefly presents the technologies used in the implementation part, chapter 3 presents the algorithms implemented for the classification problem, chapter 4 shows step-by-step the implementation part, chapter 5 describes the progress and the intermediate and final results, chapter 6 represents a user manual in order to run the Hadoop jobs accordingly.

The growth of Big Data sector requires efficient algorithms for processing and analyzing. It **is interesting** that Big Data not only refers to the size but also to the diversity of the data sets. **And** learning from data means that we can make better decisions. The goals are to learn better and faster.

# Chapter 1

## Problem Statement

The data set used in this work was gathered during 16 months using a wind tunnel test-bed facility and was donated to UCI Machine Learning Repository in May, 2013 by Alexander Vergara, Jordi Fonollosa, Marco Trincavelli, Nikolai Rulkov and Ramon Huerta. The measurements come from 72 gas sensors integrated in one sensor array. The sensor array was divided into 9 modules **so each module had 8 gas sensors.** The measures were recorded during 260 seconds at a rate of 100 Hz (samples per second). In addition to the 72 measurements the data set contains time information, temperature and humidity resulting into 75 values 

The sensor array can be placed in six different location P1-P6 orthogonal to the wind direction. The ventilator positioned at the end of the test section can have one of the three rotational speeds: 1500 rpm, 3900 rpm or 5500 rpm<sup>2</sup>. Besides the position of the array sensor and wind speed of the exhaust fan there is also the temperature involved. There are 5 predefined temperature values set to the heating elements of the sensors: 4-6 volts with a resolution of 0.5V.

The red dot from figure 1.2 represents the chemical source. Ten different

---

<sup>1</sup>The following description is a reproduction of the process of gathering the measurements from the article written by the donors

<sup>2</sup>Rotation per minute

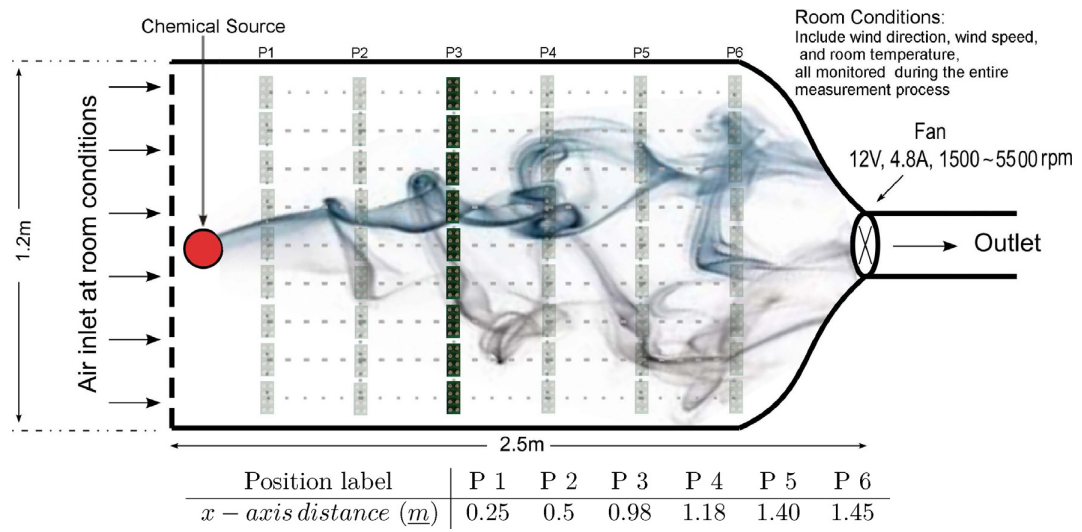


Figure 1.1: Wind tunnel test bed facility [17]

potential dangerous gases are being released using it:

Gas Name	Concentration
Acetone	2500ppm <sup>3</sup>
Acetaldehyde	500ppm
Ammonia	10000ppm
Butanol	100ppm
Ethylene	500ppm
Methane	1000ppm
Methanol	200ppm
Carbon monoxide	1000ppm / 4000ppm
Benzene	200ppm
Toluene	200ppm

For Carbon monoxide there are two concentrations. The authors start measuring Carbon monoxide with 1000ppm but in order to obtain cleaner response they switch to Carbon monoxide with 4000ppm. In the data set

archive exists two folder, one for each concentration but for the first one the measurements are not complete. Following the authors suggestion the folder containing the instances for Carbon monoxide with 1000ppm concentration is ignored.

For each of the given situation 20 replicas where collected so there are 18000 measurements: 3 different wind speeds  $\times$  5 sensor temperature  $\times$  10 gases  $\times$  6 location in the wind tunnel  $\times$  20 replicas.

From the 260 seconds used in recording, the first 20 were reserved to the preparation of the wind tunnel: set the predefined operating temperature, induce the turbulent airflow in the wind tunnel test bed facility. For the next 3 minutes one of the 10 gases is released into the tunnel. After this step, for a minute the test bed facility is ventilated with clean air.

The data set is organized into 10 folders, one for each gas. Each of the

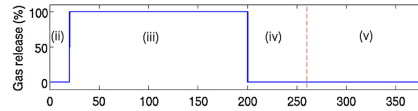


Figure 1.2: Gas release [17]

folders contains 6 sub-folders for each position. Each position folder contains 300 files: 3 wind speeds  $\times$  5 temperature values  $\times$  20 replicas. The file name represents the situation used for the measurements.

The first set in the file name figure 1.3 represents the date and time of the measurement (for figure 1.3: 3 June 2011, 02:33 am). The next block of 19 characters (board\_setPoint\_400V) indicate the voltage value applied to the array sensor. The value of the last 4 digits can be: 400, 450, 500, 550 or 600. The next block of 16 (fan\_setPoint\_100) characters represents the rotation

201106030233\_board\_setPoint\_400V\_fan\_setPoint\_100\_mfc\_setPoint\_Ammonia\_10000ppm\_p5

Figure 1.3: File Name Example



speed of the motor-driven exhaust fan; 000 for 1500 rpm, 060 for 3900 rpm and 100 for 5500 rpm. The next set of 14 characters indicates the gas used (Ammonia in the figure 1.3) and the gas concentration (10000ppm<sup>4</sup> in figure 1.3). Finally the last set represents location in the wind tunnel (p5 for the example).

The problem is a ten class gas discrimination so this paper's objective is to propose a new solution to correctly identify the gas. When a gas is released through the chemical source the goal is to detect the gas type based on previous measures.

---

<sup>4</sup>Parts per million

## Chapter 2

# Technologies

### 2.1 Apache Hadoop

#### 2.1.1 Map Reduce

MapReduce is a programming **model** and an associated implementation for processing and generating large data **sets**[8]. The model is designed for parallel and distributed computation on a cluster. The main idea of the MapReduce model is to hide details of parallel execution and allow users to focus only on data processing strategies[11]. As the model name implies the user have to define two function: map and reduce. Every job has at least one of this functions implemented.

The data set is divided into splits. Each split is assigned to a mapper and the map function runs for each block of the split. The output of this function is a key-value pair generated for each block and will serve as the input for the reducers.

A reducer will take all the values associated with the same key and run the reduce function on this set. The output will be submitted using key-value pairs.

The user can also define a combine function as optional step. The role of

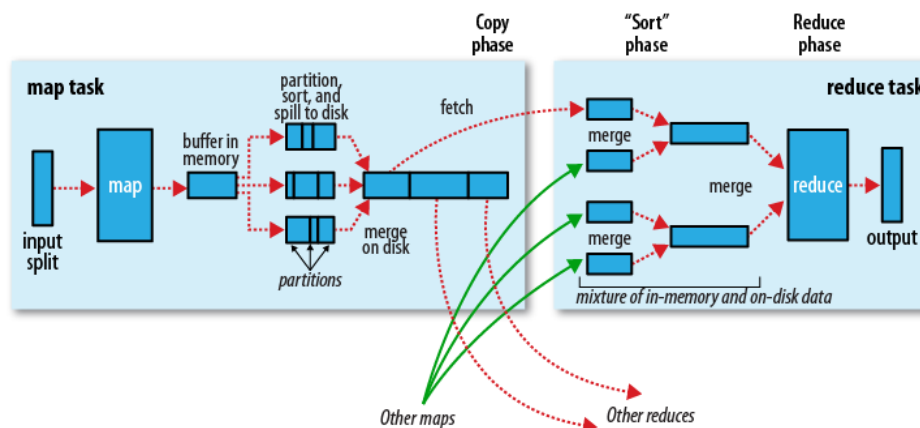


Figure 2.1: MapReduce workflow [19]

this function is to minimize the output of the map function and reduce the bandwidth.

In addition to the user defined function there are a couple of hidden steps in the MapReduce workflow. While the mappers are still running, a partition function determines which reducer will receive each pair. Each partition will do an in-memory sort by key. Periodically a combiner function might be called for each mapper. The intermediate output is committed to the disk. After the mappers finish their jobs, the sort phase begins. In this phase all the pairs are merged keeping the order of the keys (already sorted) and sent to the reducer.

### 2.1.2 Apache Hadoop

The best known Map Reduce implementation is the one provided by the Apache Software Foundation: Apache Hadoop. For storage purpose Hadoop comes with HDFS (Hadoop Distributed File System). Even if HDFS **stores** data it must not be confused with a Relational Database Management System (RDBMS). Hadoop's goal is to process in parallel large amount of data and HDFS it is a way to keep such unstructured data. Hadoop is best used

as a write-once, read-many-times type of data store [10] while a relational database is good for datasets that are continually updated [19].

The objectives of HDFS:

- **Scalable** - adding new servers when data grows.
- **Fault tolerance and self-healing** - recover automatically from failures.
- **Data Replication** - multiple copies of the same block are stored over the cluster (default 3).
- **Balancing** - place data intelligently for maximum efficiency and utilization [4].

HDFS also has the concept unit of information called block. A block has 64 MB by default. If a file is larger than the block size it is broken into block size units. The actual data is stored on DataNodes (the slaves) which report to a NameNode (the master). The NameNode stores metadata: the directory tree of the filesystem, where a data file is stored and replicated across the cluster. If the NameNode fails “all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes.” [19]. In the previous releases of Hadoop the NameNode was a single point of failure. All the cluster was inaccessible if the machine running the NameNode was unavailable. In order to prevent such case a from 2.2.0 version of Hadoop a Secondary Name Node will take the responsibilities of the Name Node.

*Data locality* is the feature that points out the relation between HDFS and MapReduce. Instead of moving data around the cluster, Hadoop distributes data and logic to process it in parallel on nodes where data is located [9].

YARN (Yet Another Resource Negotiator) or MapReduce 2 splits the job execution: a part for the Resource Manager and a part for Application Master.

The first releases had a single jobtracker who has the two responsibilities: job scheduling and task progress. The resource manager has to manage the use of resources across the cluster while the application master manages the lifecycle of applications running on the cluster[19].

One of HDFS limitation is the small files problem. The data set described in Chapter 1 has a lot of files (around 17700) which are smaller than a block. Each file size is about 6KB. HDFS is designed for streaming access of large files. Reading through small files normally causes lots of seeks and lots of hopping from datanode to datanode to retrieve each small file [18]. Besides the seeks a lot of small files can use valuable memory on the NameNode. If FileInputFormat is used by default for a job so each map takes a file as input. If there are a lot of small files in HDFS a lot of time is wasted scheduling and instantiating a lot of mappers. It is faster to have fewer files but larger than a HDFS block.

There are a couple of solutions to solve this problem. One of them is to create an archive with all these files [16, 18]. Hadoop Archives (HAR files) group small files using a lot less memory on NameNode. The HAR file is no longer access using `hdfs://` instead the client uses `har://`.

Another way to solve the small problem file is using CombineFileInputFormat[19, 15]. The goal of this input format is to pack a couple of files in order to be analyzed by a single mapper. CombineFileInputFormat is an abstract class and needs to be implemented according to this article [15]. If the current file name or the current file path is required then it can be kept just like in the listing 2.1. Also the size of the combine file split can be set programmatically in the configuration object using `mapreduce.input.fileinputformat.split.maxsize` property.

```

public class CombinedInputFormat extends CombineFileInputFormat<
    LongWritable, Text> {

    @Override
    public RecordReader<LongWritable, Text>
        createRecordReader(InputSplit split, TaskAttemptContext
            context)
        throws IOException {
        //Implement createRecordReader method
    }

    public static class CustomCombineFileRecordReader extends
        RecordReader<LongWritable, Text> {

        private static String currentPath;
        private static String fileName;
        //other parameters

        @Override
        public void initialize(InputSplit inputSplit, TaskAttemptContext
            context)
            throws IOException, InterruptedException {

            CombineFileSplit split = (CombineFileSplit) inputSplit;
            //get file info based on file index in CombinedFileInput
            FileSplit fileSplit = new FileSplit(split.getPath(index),
                split.getOffset(index),
                split.getLength(index),
                split.getLocations());
            //set file path and file name
            currentPath = fileSplit.getPath().toString();
            fileName = fileSplit.getPath().getName();
        }
        //Other methods, getters and setters
    }
}

```

Listing 2.1: File name in CombineFileInputFormat

```
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-mapreduce-client-core</artifactId>
  <version>2.2.0</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-mapreduce-client-jobclient</artifactId>
  <version>2.2.0</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-common</artifactId>
  <version>2.2.0</version>
</dependency>
```

Listing 2.2: Maven dependencies

## 2.2 Apache Maven

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information[1]. Maven helps the dependency management. A local repository will be created and all the dependencies will be downloaded from a central repository. Also Maven will build the project into a jar and can make it executable. The dependencies (see listing 2.2 for Apache Hadoop 2.2.0) and the build options will be configured in pom.xml file.

## Chapter 3

# Classification Algorithms

### 3.1 K-Nearest Neighbors

K-Nearest Neighbors is one of the simplest of all machine learning algorithms[5] and it is used for classification and regression. KNN classifiers are lazy learners[20] so there is no explicit training phase. The goal is to find K instances that are closest to the test instance and classify it using the majority of classes of its neighbors (figure 3.1).

The training set consists of a set of points in an N dimensional space and each point has a class associated. In order to classify a test instance a distance function must be computed between the test instance and every

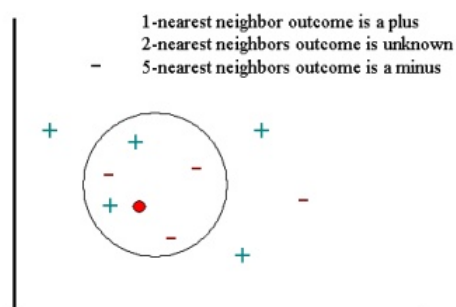


Figure 3.1: KNN classification



instance in the training data. Often used distance functions:

$$\begin{aligned} \textit{Euclidean} \quad d(x, y) &= \sqrt{\sum_{i=1}^n (x_i - y_i)^2} \\ \textit{Manhattan} \quad d(x, y) &= \sum_{i=1}^n |x_i - y_i| \\ \textit{Minkowski} \quad d(x, y) &= \left( \sum_{i=1}^n (|x_i - y_i|)^q \right)^{1/q} \end{aligned}$$

The value of K determines how many train instances impact the classification. Cross validation can determine the optimal value of K and usually K is an odd **number in order to avoid ties.**

## 3.2 Naive Bayes

In machine learning, Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes theorem [6] with the “naive” assumption that the features are mutually independent.

Given a training set  $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$  where  $x^{(i)} \in \mathbb{R}^n$  and  $y^{(i)}$  is a discrete label the goal is to associated a new  $x_{test}$  vector of features with a label. Bayes theorem says that

$$P(y_k | x_{test}) = \frac{P(x_{test} | y_k) P(y_k)}{P(x_{test})}$$

The probability of  $x_{test}$  belonging to class  $y_k$  is the probability of class  $y_k$  (known as prior) multiplied by the probability of generating instance  $x_{test}$  given the class  $y_k$  divided by the probability of instance  $x_{test}$  occurring (which is constant for the classes and can be ignored). Since  $x_{test}$  is **a n** dimensional vector and the intent is to choose the class where the instance

most probably belongs the formula is:

$$\hat{y} = \underset{y_k}{\operatorname{argmax}} P(Y = y_k) \prod_{i=1}^n P(x_i | y_k)$$

However the features described in chapter 1 are continuous so a typical assumption is that the continuous values associated with each class are distributed according to a Gaussian distribution [6] also known as Normal Distribution. If  $x^{(i)}$  is distributed according to a normal/gaussian distribution with some parameter  $\mu$ , mean and the variance  $\sigma^2$  :  $x \sim \mathcal{N}(\mu, \sigma^2)$  then

$$\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)} \quad \text{and} \quad \sigma^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)^2$$

The normal distribution is computed as follows:

$$p(x, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma}} \exp^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Using the Normal Distribution formula for Naive Bayes classifier, the label will be determined using the formula:

$$\hat{y} = \underset{y_k}{\operatorname{argmax}} P(Y = y_k) \prod_{i=1}^n \frac{1}{\sqrt{2\pi\sigma_i}} \exp^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}} \quad (3.1)$$

## Chapter 4

# Implementation

### 4.1 Data Preprocessing

#### 4.1.1 Aggregate duplicate measurements

As shown in chapter 1 in one situation (same temperature value, wind speed and location) the data set contains 20 duplicate measurements in 20 different files. The first step in the preprocessing phase is to aggregate these measurements. Also the values from the first 20 seconds and the last 60 seconds are ignored and discarded. We take all the values recorded at the same moment and for each sensor we compute the trimmed mean. The result will be a vector for each moment instead of 20.

This is the first Hadoop job that takes as arguments: the input path (from HDFS) of the data set, output path for the result and the discarding percentage used by the trimmed mean (15% by default). Other useful job configuration is to get recursively all the files in the input path.

The **map** function has three arguments. The first is the offset of the line, the second is an actual line containing an instance and the third is the job context. In this situation the first one is worthless. The third one is used just for writing the key-value pair for the reducer. A PairWritable object

will be used in order to send the key and a Text object to send the value. The key will be composed of the file name (after removing the time stamp) and the moment of the measurement. The file name is retrieved using the `CombinedFileInput` (shown in listing 2.1). By removing the time stamp from the file name and overriding the `hashCode()` method of the `PairWritable` we make sure that the instances recorded in the same situation at the same moment in the experiment are sent to the same reducer. Each line contains tab-separated values. The first value is the moment in which the instance was collected; the first moment is zero and the last one will be 260000 (260 seconds  $\times$  100 measures per second).

The **reduce** function has three arguments. The first is the a `PairWritable` object which represents the key, the second one is a list of instances assigned to the key and the third one is the job context. Hadoop makes sure that all the objects that have the same key are grouped and assigned to the same reducer. Using the list of objects the result is computed using the discard percentage for trimmed mean.

The first idea was to keep the hierarchy of the input. Using a `MultipleOutputs` object each instance will be written according to the file name from the key. Trying to keep the hierarchy of the folders and files could trigger errors due to the ‘Small File Problem’ of Hadoop (described in chapter 1). One of the suggestion of dealing with this problem is to change the “feeder” software so it does not produce small files[7]. Dropping the `MultipleOutputs` and using the job context to write the key-value pairs will decrease the number of files. The output of the reducer will be the file name and the result vector. Using the default `OutputFileFormat` these two will be written on the same line and will be separated by a tab character.

### 4.1.2 Reduce frequency in file

The second Hadoop job will reduce the frequency of the measures for each situation. Since there are 100 measurements per second the values recorded by the sensors cannot vary too much in such a short interval. For this job is also computed the trimmed mean of the values. The arguments used by the job are: the input path, the output path for the result, the discard percentage and the value of the interval measured in second. The result will be a single vector for each interval in a given file. If the trimmed mean percentage is not mentioned by default it will be 15% and the default value for reducing sample frequency will be 1. The input ised will be the files resulted after the first job.

The Aggregate Duplicate Measurements job and this job will resemble very much since the idea is the same: to average some values.

The **map** function has three arguments: the offset of the line, the actual line and the context of the job. Since the input comes from the first job, the first value in line will be the original file name and the second will be the moment of the recording. Using the frequency sample argument, the instance will be associated with an interval and a key composed of file name and interval will be sent to the reducer. The value will be the line dropping the file name from it.

The **reduce** function has also three arguments: a key, the values associated with that key and the job context. The instances from the same interval from a file will be grouped in the same reducer. For each of this keys the trimmed mean will be computed for each set of values of one sensor. The result will be a single vector for each interval of a given situation (one temperature value, same rotational speed of the ventilator and same location in the wind tunnel). The key-value pair for the reducer will be the file name as key and the vector as value.

### 4.1.3 Merge and format results

The files produced by Reduce frequency in file job will serve as input for this job. This job will take all the instances resulting from the second job and will format (comma-separated values) and group them accordingly. The grouping can be configured in one of these three ways:

- **ONE\_FILE\_PER\_COMBINATION\_PER\_GAS**: for each gas it will produce one file per situation. Therefore, in one folder gas there will be 90 files (3 different wind speeds  $\times$  5 sensor temperature  $\times$  6 location in the wind tunnel).
- **ONE\_FILE\_PER\_COMBINATION**: the result will be 90 files. For each instance will be added a gas identifier. The last value in each line will be a gas id. Usually these files will be used by the classification algorithms.
- **ONE\_FILE** the result will be just one file. Each instance will have the gas identifier, the wind speed, the temperature value and the position (in this order).

The output files name will have the following template: `Wx_Ty_Lz.csv` where the values of x, y and z will be replaced by the wind speed, temperature value and position/location.

The **map** function of this job has three arguments. The first is the offset of the line followed by the actual line and the job context. The line will be parsed and the original file name will be selected. Using it (example Figure 4.1) the output file name will be generated. This will represent the key for the reducer. The value will be the line. The **reduce** function parameters

`board_setPoint_400V_fan_setPoint_100_mfc_setPoint_Ammonia_10000ppm_p5`

Figure 4.1: File name example

are: the key, the values associated with that key and the job context. Since

there are a lot less files a MultipleOutputs can be safely used. In this situation the reducer will use the key to generate file names and will write the instances.

## 4.2 K-Fold Validation for KNN

The idea of cross validation is to split the data set into two subsets: a training set and a test set. The train set is used to train the model and validate it against the test set.

Cross-validation is used to evaluate or compare learning algorithms[14] by estimating how accurately a predictive model will perform in practice[2]. K-Fold validation is a form of cross validation. The data set is split into k folds. The cross validation algorithm is applied k times with each fold used one time as test set. k-Fold validation is also used for selecting the optimal parameters.

The problem was to design K Fold validation for KNN using Map Reduce model. The main idea is to divide the tasks of the algorithm into Map Reduce jobs. As suggested in the article [12] the first job splits the data set into several chunks. Since KNN has no explicit training phase, this second job was designed to find the optimal K parameter based on percent correctly classified (PCC). This job computes PCC for every  $k=1, 10$  and returns K for maximum value of PCC.

The input for K-Fold validation was a file resulted in the preprocessing phase described in section 4.1.1. The files used were ONE\_FILE\_PER\_COMBINATION where the class (the gas identifier) was the last value on each row.

The default value for K (K-fold validation) is 10 but it can be configured using the job parameters.

The first job consists only of the map function. The goal was to split the file; the first fold is considered test set the remaining is training set. in this manner the file will be replicated 10 times. (see figure 4.2). The second job

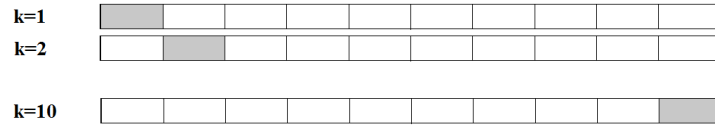


Figure 4.2: Split job

```

for (int i = 0; i < folds; i++) {
    String filePath = args[0] + i + "/train";
    MultipleInputs.addInputPath(job, new Path(filePath),
        TextInputFormat.class);
}

```

Listing 4.1: Using MultipleInputs class

intent is to compute PCC for  $k=\overline{1, 10}$  for each replica generated by the first job. For each of the subsets a mapper is instantiated. This is achieved using the MultipleInputs class for setting the input (see listing 4.1).

When a mapper is initialized the test instances from a subset are loaded into a list. The **map** function has three parameters: the offset of a line, the actual line corresponding to a train instance and the job context. Given a train instance, the euclidean distance is computed between this and every test instance loaded before. The distance is sent to the reducer along with the gas identifiers of the train instance and test instance. The mapper processed one subset so all the outputs of a mapper are sent to the same reducer. Therefore a mapper and a reducer are instantiated for each of the subsets.

The **reduce** function takes the values from a mapper. For each test instance  $k=\overline{1, 10}$  nearest neighbors are selected, the majority of classes in the neighbors is found and based on the test class, the percent correctly classified is computed (see figure 4.3). The output will be the highest PCC and the value of k for this fold (see figure 4.3).

In this situation a **combiner** function is used. The reducer does not require all the distances between the test and the train instances. The combiner



objective is to filter the map output so the reducer will receive just a set of the nearest neighbors for each test instance. From the combiner are sent 10 neighbors since PCC will be computed using k nearest neighbors with  $k=\overline{1,10}$ ;

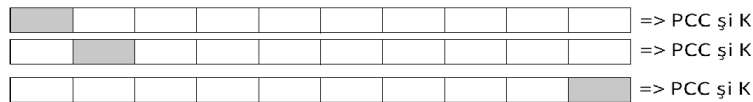


Figure 4.3: K-Nearest Neighbors job

## 4.3 K-Fold Validation for Naive Bayes

One disadvantage of the K-fold validation algorithm described above is that is specialized for K-nearest neighbors. Also having a lost of small jobs will increase the execution time since the Resource Manager will be busy assigning the jobs to the Application Masters. The idea is to create a framework for K-fold validation that uses Hadoop distributed facilities to run a classifier. A user with little or no knowledge of Hadoop can implement the classifier and run the job on a cluster.

### 4.3.1 K-Fold validation framework

The k-fold validation framework has a single Map Reduce job and uses an abstract class `AbstractClassifier` that represents the classifier. In order to run the job a user must implement this it.

The class has to abstract methods and a final method that starts the k-fold validation for the classifier. The methods that need to be implemented are:

- **void buildClassifier(Dataset set)** - this method must build the classifier using the training set received through the arguments.

```

/**
 * Start the job
 * @param args Arguments for the job
 *      [0] input_file
 *      [1] output_file
 *      [2] [-dDelimiter] attribute delimiter - default ,
 *      [3] [-cClassIndex] - column number of the class/label - F (
 *          first) L (last) or a number
 *      [4] [-kFold] - k for k-fold validation - default 10
 * @throws Exception
 */
public final void runKFoldValidation(String[] args) throws Exception{
    ClassifierJob job = new ClassifierJob();
    job.setClassifier(this);
    int exitCode = ToolRunner.run(job, args);
    System.exit(exitCode);
}

```

Listing 4.2: Start the classifier job

- **long classifyInstance(Instance instance)** - this method is used to classify a new test instance after the classifier is built.

The final method takes the arguments from the command line and starts the classifier job (see listing 4.2). The command line arguments are:

- **[0]:** the path from HDFS of the file that contains the instances
- **[1]:** the path from HDFS where the results will be written
- **[2]:** the delimiter between the attributes; since it is considered that is a comma-separated value file, the default will be a comma (,)
- **[3]:** the index of the label; must be numeric, L or F; if the label is the last value in a row this argument will be set to L; if the label is the first value in the row the argument will be set to F; otherwise the column number will be set. The default value is L (last)
- **[4]:** the number of folds to split the data set; the default value is 10.

The ClassifierJob will have a map function and a reduce function. The **map** replicate the data set **k** times and splits it into k folds. For each of the times another subset is considered the test set. The input of the function will be:

the offset of the line, the actual line representing an instance and the job context. The line will be replicated  $k$  times and sent to the reducer: one time is considered a test instance and  $k - 1$  times a train instance (see figure 4.4).

The output key will be the fold number of the test set ( $\text{fold}=\overline{0, k-1}$ ). The output value will be a pair that consists of the actual line and the type (test or train).

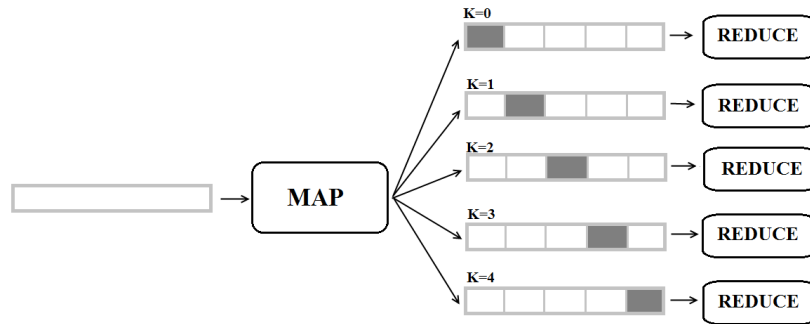


Figure 4.4: 5-fold validation workflow

The **reduce** function will receive all the pairs that have the same fold number so there will be  $k$  reducers running. The first step is to separate the test instances from the train instances and parse the lines to get the attributes. The parsing phase will use a formatter object defined by the delimiter and class index received from the the command line arguments. Then the method *buildClassifier* is called with the training set as argument. After this, for each instance in test set the method *classifyInstance* will be called and the predicted class will be compared with the actual class of the instance. If they are the same a variable that represents the correctly classified instances will be increased. The output of the reducer will be the the fold number of the test set and the percent correctly classified for that set.

```

FileSystem fs = FileSystem.get(configuration);
Path temp = new Path(Constant.DISTRIBUTED.CACHE_FILE);
ObjectOutputStream outputStream = new ObjectOutputStream(fs.create(
    temp));
outputStream.writeObject(classifier);
outputStream.close();
fs.deleteOnExit(temp);

//add classifier to cache file
job.addCacheFile(new Path(Constant.DISTRIBUTED.CACHE_FILE).toUri());

```

Listing 4.3: Add classifier to DistributedCache

```

Path[] paths = context.getLocalCacheFiles();

if(paths[0].toString().contains(Constant.DISTRIBUTED.CACHE_FILE)){
    ObjectInputStream is =
        new ObjectInputStream(new FileInputStream(paths[0].
            toString()));

    try {
        classifier = (AbstractClassifier) is.readObject();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
} else {
    throw new IOException("Classifier does not exists in the local
        cache files");
}

```

Listing 4.4: Restore classifier from DistributedCache

One challenge was to share among the reducers the class that implements the abstract classifier since it is not part of the framework. Hadoop has DistributedCache which offers the possibility to access rapidly files in read-only mode from all over the cluster. The AbstractClassifier must implement Serializable interface in order to add the class into a local file. The file is added to cache before the job starts and can be configured to be removed after the job finished (see listing 4.3). After a reducer is instantiated, the file is read and the classifier restored from it (see listing 4.4).

### 4.3.2 Naive Bayes implementation

In order to run a Naive Bayes classifier using the framework defined in section 4.3.1 the AbstractClassifier class must be extended. For this class

the *buildClassifier* and *classifyInstance* methods must be implemented.

Using the training set received as parameter the first method will build the classifier. As shown in section 3.2 in this step the mean  $\mu$  and variance  $\sigma^2$  must be computed using the instance attributes.

Given a new test instance, the second method will classify it. Using mean  $\mu$  and variance  $\sigma^2$  calculated in the previous step, this method will return the class prediction for the new instance. *classifyInstance* uses the equation 3.1 in order to determine the most probable class for an instance.

## Chapter 5

# Results

The Map Reduce jobs were executed using a Hadoop Cluster and a Single Node Hadoop. Both use Hadoop 2.2.0 version.

The cluster uses Hortonworks distribution and has 12 slaves machines, a machine for the Resource Manager, one for the Name Node and one for Secondary Name Node.

The cluster with one single node was installed on a virtual machine with Ubuntu 12.04.3 LTS following the steps described in the article *Setup newest Hadoop 2.x (2.2.0) on Ubuntu ??*.

### 5.1 Execution time

The initial data set has around 100GB and after the preprocessing phase all the result files have 50MB. All three jobs need around 20 minutes to finish successfully on the cluster. Since the Single Node instance runs on a virtual machine it was hard to measure the amount of time needed to preprocess all the data set on this. The input of the first job shown in figure 5.1 is a random subset that has around 600MB. The period of time was measured using the same job and the same subset. The second job will use the first job result files. As shown in figure 5.1 running the job on single node is almost 5

times slower than running it on a cluster with 12 slave nodes. The difference comes from the fact that on the cluster a lot of mappers and reducers are running at the same time.

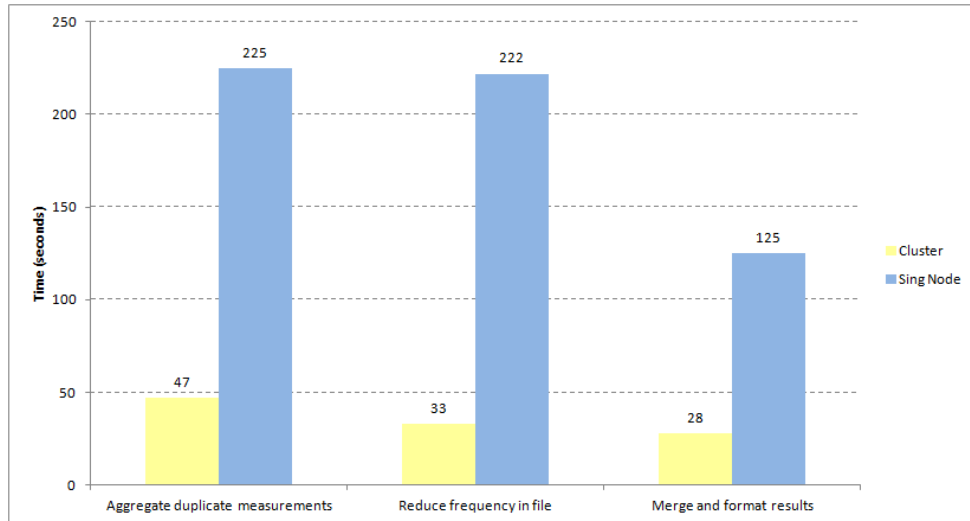


Figure 5.1: Execution time for preprocessing phase

Running the classification algorithms on Hadoop does not produce such a difference since the input is a lot smaller. Hadoop is designed for Big Data. For such a input a lot of useful time will be wasted for job management and between chain jobs. Also the algorithms are using the parallel functionalities more then the Map Reduce model. Figure 5.2 shows the difference between running **K-fold validation** for K-nearest Neighbors algorithm on a cluster and on the single node instance of Hadoop. The job uses as input one file resulted after the preprocessing phase (section 4.1.1) with instances measured in the same location, having the same rotational speed and the same temperature value.

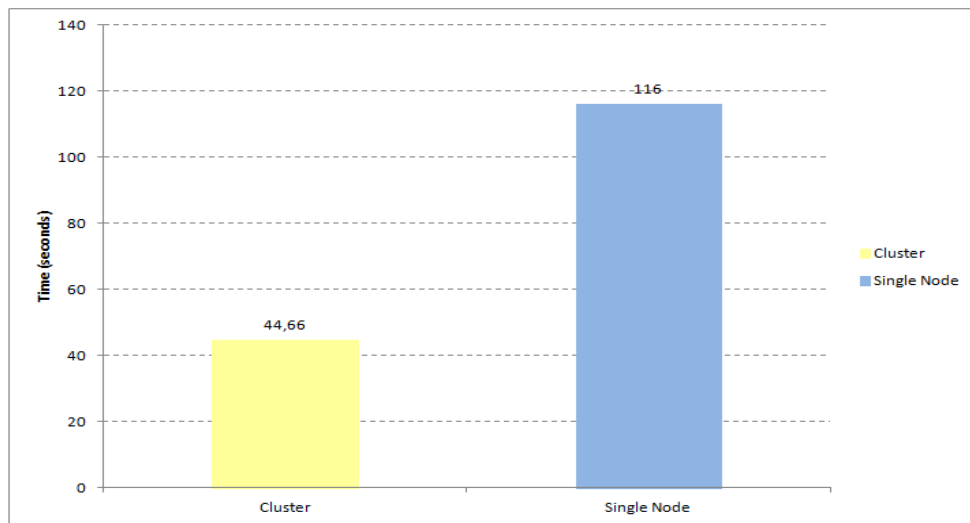


Figure 5.2: Execution time for K-fold validation for K-nearest neighbors

Same kind of files were used for K-fold validation for Naive Bayes. The elapsed times for the job are shown in figure 5.3. Running the job on a cluster takes half the time if it was submitted to the single node.

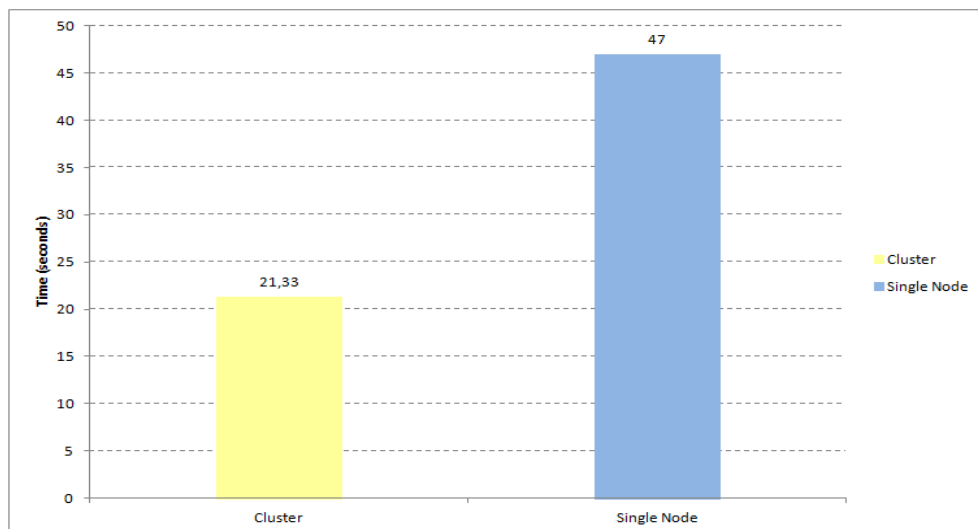


Figure 5.3: Execution time for K-fold validation for Naive Bayes



## 5.2 Accuracy

At first the classification algorithms were tested using instances recorded in the same situation (same position, at the same temperature and with the same rotational speed of the ventilator; one file resulted from the preprocessing phase). Figure 5.4 indicates that K-NN has the best accuracy with all the test instances classified correctly while Naive Bayes has 99% correctly classified test instances.

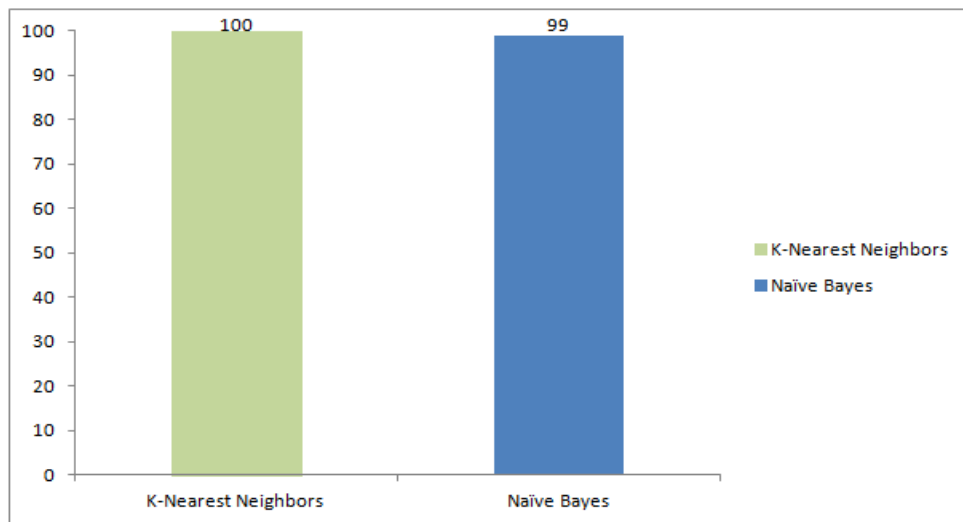


Figure 5.4: Percent correctly classified for KNN and Naive Bayes

The next step was to investigate how the position of the sensor array can influence the result. For a specific wind speed and one temperature value were collected instances from several positions in the wind tunnel. The aggregation of the instances was made similar to the tests presented in the article *On the performance of gas sensor arrays in open sampling systems using Inhibitory Support Vector Machines* [17]. Figure 5.5 presents the results for K-NN for the following situations:

- 1500rpm (W000) and 6V (T600) applied to the sensors heater
- 3900rpm (W060) and 4.5W (T450)
- 5500rpm (W100) and 5.5V (T550)

The instances were gathered:

- all the instances from each situation
- the instances collected when the sensor array was in the first and last location (L1 and L6)
- the instances from the second location and the fifth location (L2 and L5)
- the instances from the third and fourth location (L3 and L4)

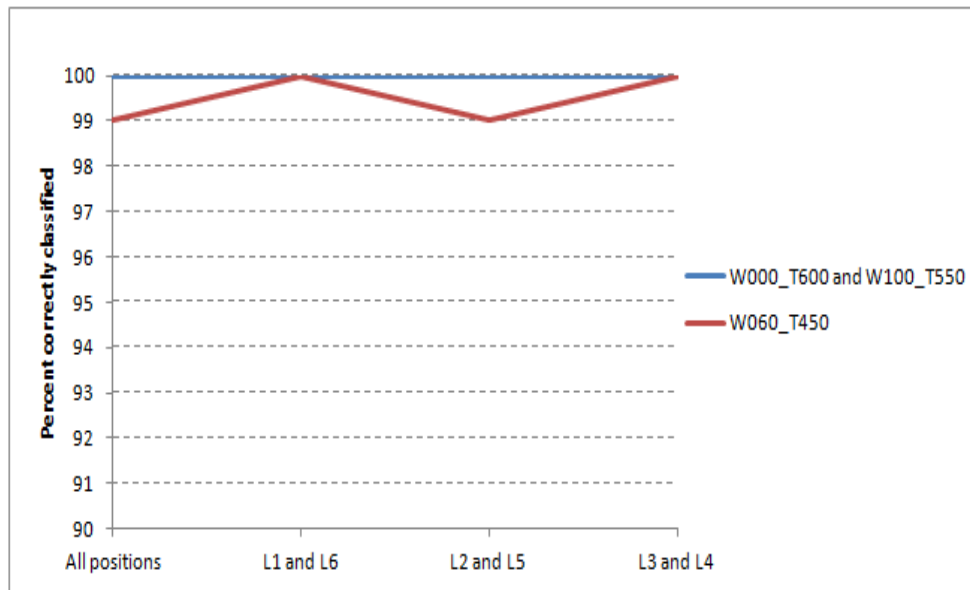


Figure 5.5: K-NN tested in multiple locations

The same subsets were used as input for K-fold validation for Naive Bayes. The results are displayed in figure 5.6.

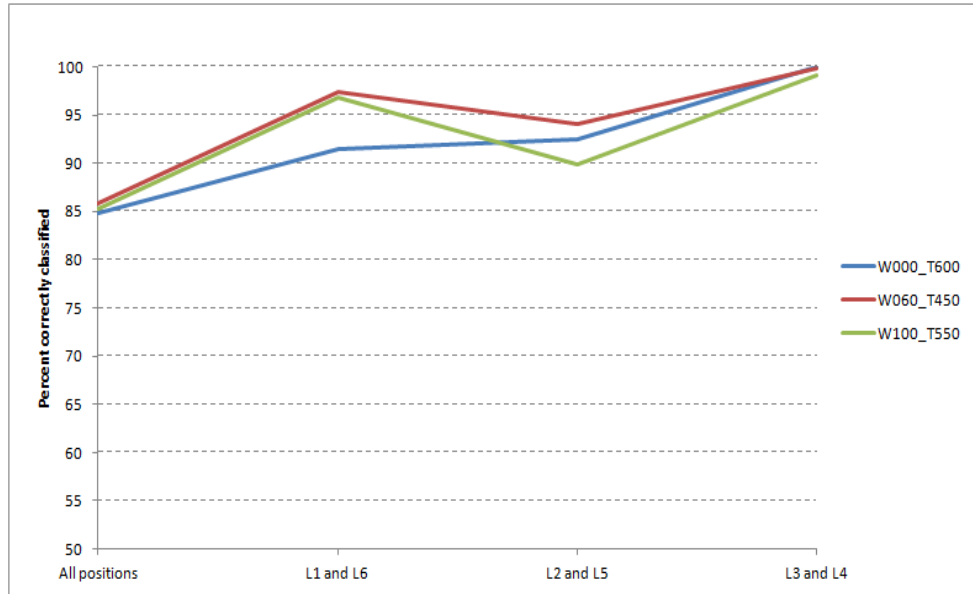


Figure 5.6: Naive Bayes tested in multiple locations

K-NN has better accuracy trained in multiple positions. The best locations to train Naive Bayes classifier are L3 and L4, the middle of the wind tunnel. Just like the result from K-NN, the process of training and testing the instances from L2 and L5 has poor accuracy.

The next step was to see how the rotational speed of the ventilator situated at the end of the wind tunnel can impact the precision of the result. Data collected at the same value of the ventilator was merged for each location in the wind tunnel. Figure 5.7 presents the result for K-NN using this data while figure 5.8 presents the result for Naive Bayes using the same input. K-fold validation for K-NN has almost 100% test instances correctly classified regardless the volts applied to the sensors heater. Training and testing the instances using Naive Bayes shows that positioning the ventilator at the end of the wind tunnel (nearest to L6 position) can influence the results. The accuracy has the lowest value for the instances recorded in L6 regardless the actual value of the ventilator. Also L4 seems the best position for the sensor

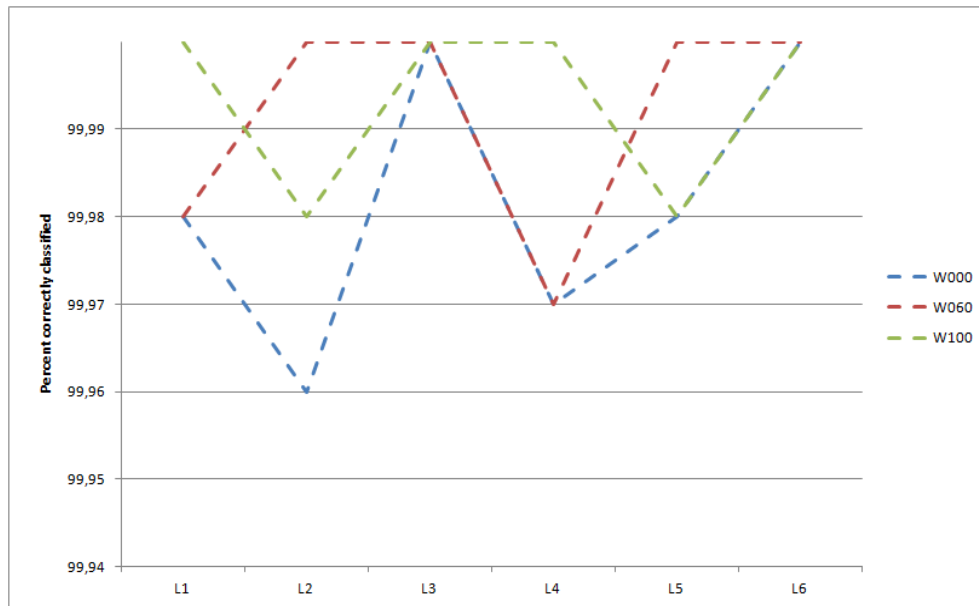


Figure 5.7: Influence of wind speed for K-NN

array having PCC above 90% for every value of the wind speed.

In all three steps K-nearest neighbors algorithm has better accuracy than Naive Bayes being the best choice for the classification problem from the data set (described in chapter 1). From each standpoint the percent correctly classified of test instances is above 98%.

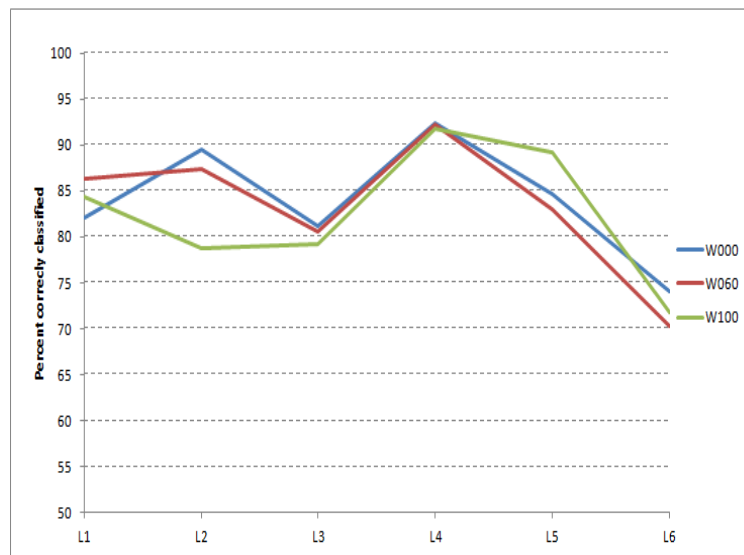


Figure 5.8: Influence of wind speed for Naive Bayes

## Chapter 6

# Validation of Results

The results shown in chapter 5 were validated using Rapid Miner version 5.3.015. As shown in figure 6.1 in Rapid Miner was used a X-Validation operator for k-fold validation with 10 folds. In the subprocess of the operator was used either K-NN or Naive Bayes. The model build by this classifier is then sent to the Apply Model operator along with the testing set. For accuracy was used the Performance operator.

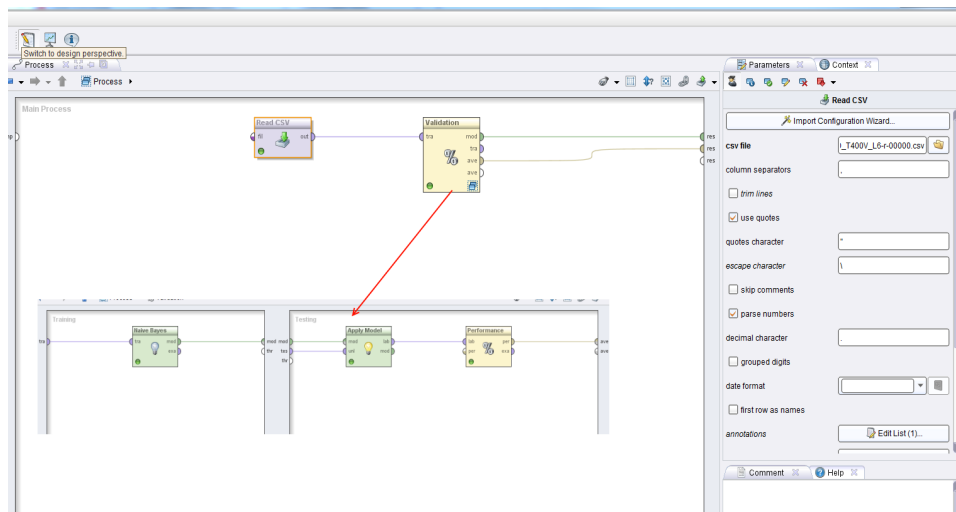


Figure 6.1: Rapid Miner operators used for validation

	true 0	true 4	true 5	true 3	true 7	true 2	true 1	true 9	true 8	true 6	class precision
pred 0	181	0	0	0	0	0	0	0	0	0	100.00%
pred 4	0	181	0	0	0	0	0	0	0	0	100.00%
pred 5	0	0	181	0	0	0	0	0	0	0	100.00%
pred 3	0	0	0	181	0	0	0	0	0	0	100.00%
pred 7	0	0	0	0	181	0	0	0	0	0	100.00%
pred 2	0	0	0	0	0	181	0	0	0	0	100.00%
pred 1	0	0	0	0	0	0	181	0	0	0	100.00%
pred 9	0	0	0	0	0	0	0	181	0	0	100.00%
pred 8	0	0	0	0	0	0	0	0	181	0	100.00%
pred 6	0	0	0	0	0	0	0	0	0	181	100.00%
class recall	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	

Figure 6.2: k-fold validation for K-NN

Figure 6.2 and 6.3 indicate the accuracy for K-NN and Naive Bayes having instances collected in single location, having the same voltage applied to the sensors heater and the same rotational speed set for the ventilator. In most of this situations k-fold validation for Naive Bayes algorithm returns 100% accuracy. Also the tables reveal the class precision and class recall. Precision is the probability that a (randomly selected) retrieved document

accuracy: 99.89%  $\pm$  0.02% (micro: 99.89%)

	true 0	true 6	true 4	true 3	true 9	true 1	true 7	true 5	true 8	true 2	class precision
pred. 0	181	0	0	0	0	0	0	0	0	0	100.00%
pred. 6	0	179	0	0	0	0	0	0	0	0	100.00%
pred. 4	0	0	181	0	0	0	0	0	0	0	100.00%
pred. 3	0	0	0	180	0	0	0	0	0	0	100.00%
pred. 9	0	0	0	0	181	0	0	0	0	0	100.00%
pred. 1	0	0	0	0	0	181	0	0	0	0	100.00%
pred. 7	0	0	0	0	0	0	181	0	0	0	100.00%
pred. 5	0	0	0	0	0	0	0	181	0	0	100.00%
pred. 8	0	2	0	0	0	0	0	0	181	0	98.91%
pred. 2	0	0	0	0	0	0	0	0	0	181	100.00%
class recall	100.00%	98.90%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	

Figure 6.3: k-fold validation for Naive Bayes

is relevant. Recall is the probability that a (randomly selected) relevant document is retrieved in a search. Or high recall means that an algorithm returned most of the relevant results. High precision means that an algorithm returned more relevant results than irrelevant[13]. For Naive Bayes, from 181 test instances having gas identifier 6, 2 instances were foreseen with gas identifier 8. The class recall is 98.9% while the class precision is 99.81%.

Also the figures 6.4 and 6.5 validate the result shown in figures 5.5 and 5.6 from chapter 5. The instances are grouped by the rotational speed of the ventilator (W060 which represents 3900rpm) and temperature value (T450 which represents 4.5 volts applied to the heater).

accuracy: 99.99%  $\pm$  0.03% (micro: 99.99%)

	true 0	true 7	true 3	true 1	true 8	true 9	true 6	true 2	true 5	true 4	class precision
pred. 0	1085	0	0	0	0	0	0	0	1	0	99.91%
pred. 7	0	1086	0	0	0	0	0	0	0	0	100.00%
pred. 3	0	0	1086	0	0	0	0	0	0	0	100.00%
pred. 1	0	0	0	1086	0	0	0	0	0	0	100.00%
pred. 8	0	0	0	0	1086	0	0	0	0	0	100.00%
pred. 9	0	0	0	0	0	1086	0	0	0	0	100.00%
pred. 6	0	0	0	0	0	0	1086	0	0	0	100.00%
pred. 2	0	0	0	0	0	0	0	1086	0	0	100.00%
pred. 5	0	0	0	0	0	0	0	0	1086	0	100.00%
pred. 4	0	0	0	0	0	0	0	0	0	905	100.00%
class recall	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	99.91%	100.00%	

Figure 6.4: K-NN results for W060\_T450



Tools View Help

SimpleDistribution (Naive Bayes) PerformanceVector (Performance)

Text View Annotations

Multiclass Classification Performance Annotations

Table View Plot View

accuracy: 85.71% +/- 1.05% (mikro: 85.71%)

	true 0	true 7	true 3	true 1	true 8	true 9	true 6	true 2	true 5	true 4	class precision
pred. 0	1081	0	0	52	178	0	0	0	0	11	81.77%
pred. 7	0	1588	0	0	0	0	0	0	0	0	100.00%
pred. 3	0	0	1026	2	0	150	0	0	0	0	87.10%
pred. 1	0	0	0	887	0	0	0	0	0	0	100.00%
pred. 8	0	0	0	0	224	0	37	0	0	35	75.68%
pred. 9	4	0	43	26	0	900	0	0	66	0	86.62%
pred. 6	0	0	17	0	199	0	1008	0	0	35	80.06%
pred. 2	0	0	0	78	0	0	0	1088	0	0	93.30%
pred. 5	0	0	0	0	0	36	0	0	1020	0	96.59%
pred. 4	0	0	0	31	485	0	41	0	0	824	59.67%
class recall	99.63%	100.00%	94.48%	82.60%	20.63%	82.87%	92.82%	100.00%	93.92%	91.05%	

Figure 6.5: Naive Bayes results for W060\_T450

For K-NN the accuracy is 99.99% and for Naive Bayes is 85.71%. Further figure 6.6 displays the results for the same situation but only for the instances collected in L3 and L4 position.

Tools View Help

SimpleDistribution (Naive Bayes) PerformanceVector (Performance)

Text View Annotations

Multiclass Classification Performance Annotations

Table View Plot View

accuracy: 99.89% +/- 0.18% (mikro: 99.89%)

	true 1	true 0	true 2	true 5	true 9	true 4	true 6	true 3	true 8	true 7	class precision
pred. 1	362	0	0	0	0	0	0	0	0	0	100.00%
pred. 0	0	362	0	0	0	0	0	0	0	0	100.00%
pred. 2	0	0	362	0	0	0	0	0	0	0	100.00%
pred. 5	0	0	0	362	0	0	0	0	0	0	100.00%
pred. 9	0	0	0	0	362	0	0	0	0	0	100.00%
pred. 4	0	0	0	0	0	358	0	0	0	0	100.00%
pred. 6	0	0	0	0	0	0	361	0	0	0	100.00%
pred. 3	0	0	0	0	0	0	0	362	0	0	100.00%
pred. 8	0	0	0	0	0	4	0	0	362	0	98.91%
pred. 7	0	0	0	0	0	0	0	0	0	362	100.00%
class recall	100.00%	100.00%	100.00%	100.00%	100.00%	99.90%	100.00%	100.00%	100.00%	100.00%	

Figure 6.6: Naive Bayes results for W060\_T450 in L3 and L4 locations

## Chapter 7

# Developer Manual

In section 4.3 was described the k-fold validation framework. The goal was to create a framework that can run k-fold validation on a cluster using a newly implemented classification algorithm. The framework was packed into a jar file. In order to integrate it a new Java project must be created and the jar file must be added into the build path (see figure 7.1).

Then a class must be created. The class must extend `AbstractClassifier` and must have a main method that starts the job (see listing 7.1). Since the class extends `AbstractClassifier`, the methods *buildClassifier* and *classifyInstance* must be implemented. In the main method besides starting the job it is recommended to set the classifier name in order to distinguish it in the job history of Hadoop.

After implementing the methods the project must be packed into a jar in order to run it on a cluster. First a run configuration with the main class should be created (see figure 7.2). Then the project should be exported as a runnable jar such as the steps presented in figure 7.3.

At last the jar can be used as a Hadoop job and run on a cluster. The command to run a jar is:

```
hadoop jar <jarName>[mainClass] arguments
```

so in the case presented before it will be:

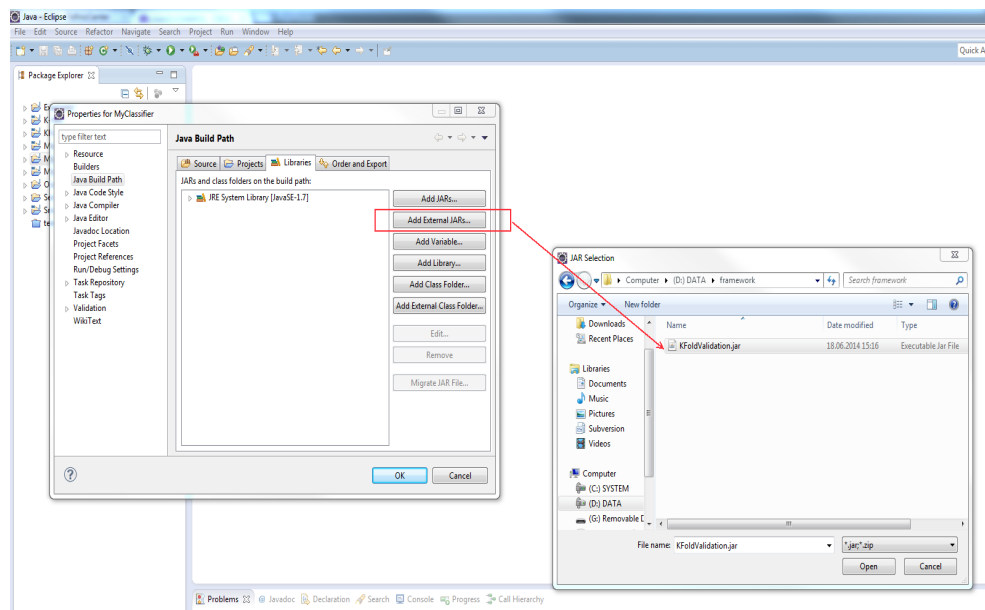


Figure 7.1: Add jar to build path

```
hadoop jar myclassifier.jar hdfsInputPath hdfsOutputPath
```

The main class can be ignored since it is a runnable jar and the main class was set through the configuration. Other useful arguments that can be set for k-fold validation framework were presented in listing 4.2.

```

public class MyClassifier extends AbstractClassifier {

    private static final long serialVersionUID = -8050623050936268107L;

    @Override
    public void buildClassifier(Dataset dataset) {

        // TODO Auto-generated method stub

    }

    @Override
    public long classifyInstance(Instance instance) {

        // TODO Auto-generated method stub
        return 0;
    }

    public static void main(String[] args) throws Exception {
        MyClassifier classifier = new MyClassifier();
        classifier.setName("MyClassifier");
        classifier.runKfoldValidation(args);
    }
}

```

Listing 7.1: Classifier class

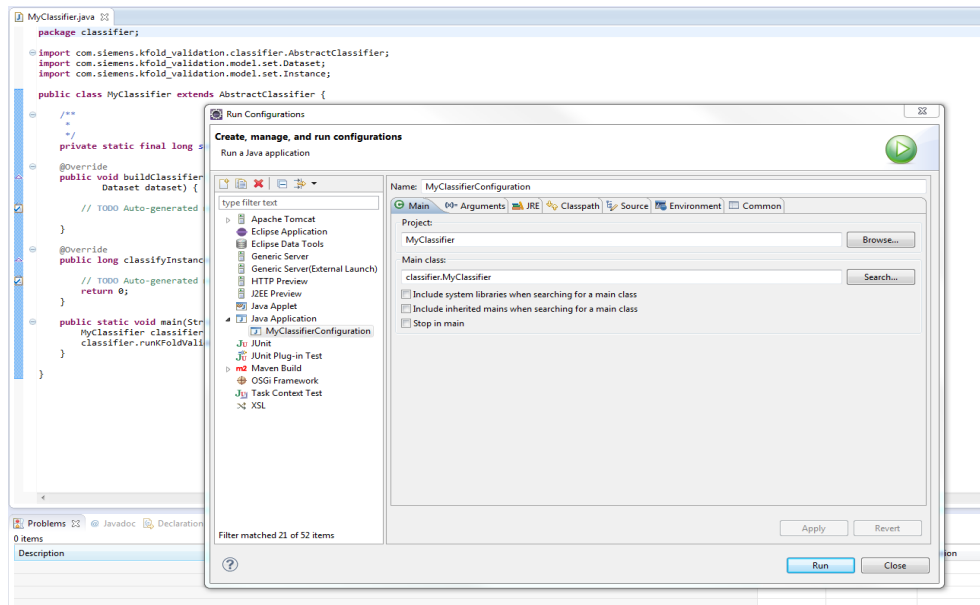


Figure 7.2: Creating run configuration for the classifier

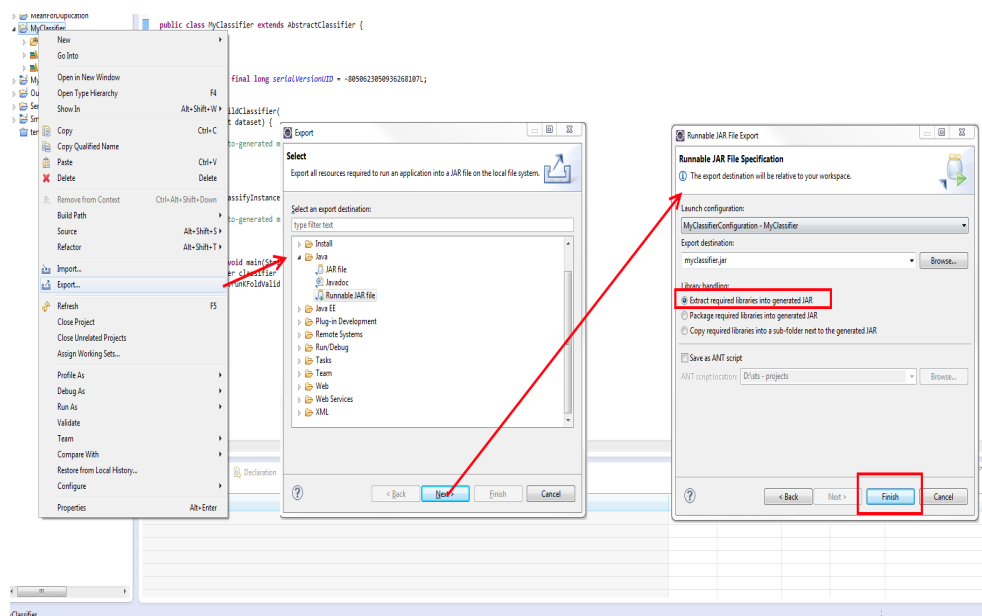


Figure 7.3: Export project as runnable jar

# Conclusions

The paper proposes a new solution to the one presented in the article *On the performance of gas sensor arrays in open sampling systems using Inhibitory Support Vector Machines* [17]. For the classification problem described in chapter 1 were proposed two algorithms: K-nearest neighbors and Naive Bayes. The performance measured in time execution and accuracy of the implementations showed that K-NN has better results for this data set. Also the results suggest that both of the solutions can represent an alternative for the one proposed by the donors of the data set.

Furthermore, new classification algorithms can be implemented using the parallel features of Apache Hadoop. The framework KFoldValidation packed as a jar can be used to run Hadoop jobs for k-fold validation for different algorithms on a cluster.

# Bibliography

- [1] Apache Maven Project. <http://maven.apache.org/>.
- [2] Cross-validation (statistics).
- [3] Gas sensor arrays in open sampling settings Data Set.  
<http://archive.ics.uci.edu/ml/datasets/Gas+sensor+arrays+in+open+sampling+settings>.
- [4] HDFS & MapReduce. <http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/hdfs-and-mapreduce.html>.
- [5] k-nearest neighbors algorithm. [http://en.wikipedia.org/wiki/K-nearest\\_neighbors\\_algorithm](http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm).
- [6] Naive Bayes classifier. [http://en.wikipedia.org/wiki/Naive\\_Bayes\\_classifier](http://en.wikipedia.org/wiki/Naive_Bayes_classifier).
- [7] Alex Dean. Dealing with Hadoop's small files problem.  
<http://snowplowanalytics.com/blog/2013/05/30/dealing-with-hadoops-small-files-problem/>.
- [8] Jeffrey Dean and Sanjay Ghemawa. MapReduce: Simplified Data Processing on Large Clusters. Google, Inc.

- [9] J. Jeffrey Hanson. An introduction to the Hadoop Distributed File System. <http://www.ibm.com/developerworks/library/wa-introhdfs/wa-introhdfs-pdf.pdf>.
- [10] Chuck Lam. *Hadoop IN ACTION*. Manning Publications Co., 2011.
- [11] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel Data Processing with MapReduce: A Survey. <http://www.cs.arizona.edu/~bkmoon/papers/sigmodrec11.pdf>.
- [12] Danilo Moret, Karin Breitman, Evelin Amorim, Jose Talavera, Ruy Milidui, and Jose Viterbo. Double Dip Map-Reduce for Processing Cross Validation Jobs.
- [13] Anto Jeson Raj. Sentiment Analysis with Rapidminer. <http://auburnbigdata.blogspot.dk/2013/02/sentiment-analysis-with-rapidminer.html>.
- [14] Payam Refailzadeh, Lei Tang, and Huan Liu. Cross-Validation.
- [15] Sujay Som. Process small, compressed files in Hadoop using CombineFileInputFormat. <http://www.ibm.com/developerworks/library/bd-hadoopcombine/bd-hadoopcombine-pdf.pdf>.
- [16] Tsz-Wo Nicholas Sze and Mahadev Konar. The Problem of Many Small Files. <https://developer.yahoo.com/blogs/hadoop/hadoop-archive-file-compaction-hdfs-461.html>.
- [17] Alexander Vergara, Jordi Fonollosa, Jonas Mahiques, Marco Trincavelli, Nikolai Rulkov, and Ramon Huerta. On the performance of gas sensor arrays in open sampling systems using Inhibitory Support Vector Machines. May 2013. ISSN 0925-4005, 10.1016/j.snb.2013.05.027.
- [18] Tom White. The Small Files Problem. <http://blog.cloudera.com/blog/2009/02/the-small-files-problem/>.



- [19] Tom White. *Hadoop: The definitive Guide*. O'Reilly, third edition, 2012.
- [20] X. Wu, V. Kunmar, J. Ross Quinlan, J. Ghosh, Q. Yang, and H. Motoda. Top 10 algorithms in data mining.

# List of Figures

1.1	Wind tunnel test bed facility [17] . . . . .	7
1.2	Gas release [17] . . . . .	8
1.3	File Name Example . . . . .	8
2.1	MapReduce workflow [19] . . . . .	11
3.1	KNN classification. Source <a href="http://www.statsoft.com/textbook/k-nearest-neighbors">http://www.statsoft.com/textbook/k-nearest-neighbors</a> . . . . .	16
4.1	File name example . . . . .	22
4.2	Split job . . . . .	24
4.3	K-Nearest Neighbors job . . . . .	25
4.4	5-fold validation workflow . . . . .	27
5.1	Execution time for preprocessing phase . . . . .	31
5.2	Execution time for K-fold validation for K-nearest neighbors . . . . .	32
5.3	Execution time for K-fold validation for Naive Bayes . . . . .	32
5.4	Percent correctly classified for KNN and Naive Bayes . . . . .	33
5.5	K-NN tested in multiple locations . . . . .	34
5.6	Naive Bayes tested in multiple locations . . . . .	35
5.7	Influence of wind speed for K-NN . . . . .	36
5.8	Influence of wind speed for Naive Bayes . . . . .	37
6.1	Rapid Miner operators used for validation . . . . .	39

6.2	k-fold validation for K-NN . . . . .	39
6.3	k-fold validation for Naive Bayes . . . . .	40
6.4	K-NN results for W060_T450 . . . . .	40
6.5	Naive Bayes results for W060_T450 . . . . .	41
6.6	Naive Bayes results for W060_T450 in L3 and L4 locations .	41
7.1	Add jar to build path . . . . .	43
7.2	Creating run configuration for the classifier . . . . .	44
7.3	Export project as runnable jar . . . . .	45