Fuzzy Rough Set Approximations in Large Scale Information Systems

Hasan Asfoor

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science and Systems

University of Washington

2015

Committee:

Martine De Cock, Chair

Ankur Teredesai

Matthew Tolentino

Chris Cornelis

Program Authorized to Offer Degree:
Institute of Technology

University of Washington

Abstract

Fuzzy Rough Set Approximations in Large Scale Information Systems

Hasan Asfoor

Chair of the Supervisory Committee:
Associate Professor Martine De Cock
Institute of Technology

Rough set theory is a popular and powerful machine learning tool. It is especially suitable for dealing with information systems that exhibit inconsistencies, i.e. objects that have the same values for the conditional attributes but a different value for the decision attribute. In line with the emerging granular computing paradigm, rough set theory groups objects together based on the indiscernibility of their attribute values. Fuzzy rough set theory extends rough set theory to data with continuous attributes, and detects degrees of inconsistency in the data. Key to this is turning the indiscernibility relation into a gradual relation, acknowledging that objects can be similar to a certain extent. In very large datasets with millions of objects, computing the gradual indiscernibility relation (or in other words, the soft granules) is very demanding, both in terms of runtime and in terms of memory. It is however required for the computation of the lower and upper approximations of concepts in the fuzzy rough set analysis pipeline. In this thesis, we present a parallel and distributed solution implemented on both Apache Spark and Message Passing Interface (MPI) to compute fuzzy rough approximations in very large information systems. Our results show that our parallel approach scales with problem size to information systems with millions of objects. To the best of our knowledge, no other parallel and distributed solutions have been proposed so far in the literature for this problem. We also present two distributed prototype selection approaches that are based on fuzzy rough set theory and couple them with our distributed implementation of the well known weighted k-nearest neighbors machine

learning prediction technique to solve regression problems. In addition, we show how our distributed approaches can be used on the State Inpatient Data Set (SID) and the Medical Expenditure Panel Survey (MEPS) to predict the total healthcare expenses of patients.

TABLE OF CONTENTS

Page

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to the University of Washington Tacoma, where I had the opportunity to pursue my interests in Computer Science and work with fellow aspiring and professional data scientists, researchers, and professors in the field. Furthermore, I would like to thank Professor Martine De Cock, Professor Matthew Tolentino and Professor Ankur Teredesai for their guidance and encouragement throughout this program.

DEDICATION

to my dear wife, son and parents

Chapter 1

INTRODUCTION

Rough set theory [53], introduced in the 1980s, is a powerful machine learning tool that has applications in many data mining and machine learning techniques such as instance selection, feature selection and data prediction.

Rough set theory deals with information systems that contain data inconsistencies, such as two patients who have the same symptoms but different diseases. A rough set approximates a crisp set by two other sets that give a lower and upper approximation of the crisp set. In rough set analysis, data is expected to be discrete. So, a continuous numeric attribute is required to be discretized. Fuzzy rough set theory [36] is an extension of rough set theory that deals with continuous numerical attributes. It can solve the same problems that rough set can solve and also can handle both numerical and discrete data.

The importance of fuzzy rough set theory is clearly seen in several applications, especially in the area of machine learning. For instance, [40] and [41] use fuzzy rough set theory to improve the accuracy of the well-known k Nearest Neighbour (k-NN) algorithm. In addition, it has been also used in [33] to refine search on the WWW, as well as to reduce the dimensionality of data sets in [43].

In very large information systems, the computation of the lower and upper approximation sets is a demanding process both in terms of processing time and memory utilization. Current implementations of this computation can not be used for large data sets. We experimentally observed that the "RoughSets" package [45] in R is not able to handle a data set with 30000 instances and 10 attributes on a capable computing machine with 48 CPU cores and 62GB of memory. This raises the need for a scalable implementation that makes efficient use of memory, CPU cores and multiple compute nodes. Such an implementation can become an enabler for fuzzy rough set based algorithms to scale.

There are four objectives in thesis. The first objective is to provide a scalable imple-

mentation of the fuzzy rough set approximation computation on a state-of-the-art cluster computing framework, Apache Spark. We also provide an alternative implementation on the well-known Message Passing Interface (MPI) and carry out a performance comparison between the two implementations. The second objective is to build a scalable implementation of a weighted k-NN algorithm in Apache Spark. The third objective is to develop a scalable fuzzy rough set based prototype selection approach that can be used, with the weighted k-NN implementation, to solve regression problems. The fourth objective is to use the k-NN and prototype selection implementations from the second and third objectives to predict the future cost of patients from real-life historical health care cost data.

The structure of this thesis is organized into six chapters. The first chapter is this current chapter which forms an introduction to the thesis. The second chapter provides background information of concepts that are needed in the subsequent chapters. It mainly covers basics of fuzzy logic, rough set and fuzzy rough set theory. In addition, it gives an overview of the big data frameworks and data sets that are used in this thesis. The third chapter describes the scalable implementation of the fuzzy rough set approximation computation on both Spark[1] and MPI[2]. It also shows experimental scalability results and a comparison of both implementations. The fourth chapter describes the distributed implementation of both the weighted k-NN algorithm and the fuzzy rough set based prototype selection algorithm on Apache Spark. It also includes scalability experiments and results. In the fifth chapter, we provide an accuracy analysis of our fuzzy rough set based prototype selection and k-NN approaches, presented in chapter three, by using them to predict the future cost of patients based on their past medical expenses and medical conditions. The sixth chapter is the conclusion chapter.

[1]The source code of all Spark implementations in all chapters can be found on `https://github.com/WebDataScience/FRS-KNN-Spark`

[2]The source code of the MPI implementation can be found on `https://github.com/WebDataScience/ParallelFuzzyRoughSetApproximationonMPI`

Chapter 2

BACKGROUND

In this chapter, we give an overview of fuzzy rough set theory. We particularly explain some basic concepts that are related to fuzzy logic, fuzzy sets, rough sets and fuzzy rough sets. We also describe the distributed processing frameworks as well as the data sets that we are going to use in this thesis.

2.1 Fuzzy Rough Set Theory

2.1.1 Fuzzy Logic & Fuzzy Sets

Binary logic is discrete and has only two logic values which are true and false, or 1 and 0. In real life, however, things are true to a certain degree. For example, we can say that a patient is sick but we can also say that he is very sick or starting to get sick. So, the patient is not either sick or not sick, rather the patient is sick with a certain intensity. Fuzzy logic [21] extends binary logic by adding this intensity range of values to specify the extent to which something is true. In this case, the range of truth values is between 0 and 1 (inclusive). The closer the truth value of a statement to 1 is, the more true the statement is. For instance, a very sick patient can have a sickness degree of 0.9 to indicate that he is very sick. On the contrary, he can have a sickness degree of 0.1 indicating that he is almost recovered from his illness. A fuzzy set [8] is a set whose elements belong to the set with a degree of membership. For instance, suppose that we have two fuzzy sets, one representing old people and another representing young people. Then the larger the age of a person is, the higher his degree of membership to the old set is and the lower his degree of membership to the young set is. Since fuzzy logic extends binary logic, it also extends its logical operations. There are many fuzzy logical operators. However, we will only refer to the operators that will be used in this thesis which are the fuzzy logical t-norm, implicator and t-conorm [10] which extend the binary logical conjunction, implication and disjunction.

There are several t-norm fuzzy operators but the one that will be used in this thesis is the minimum t-norm operator T_M of x and y (also denoted by $x \tilde{\wedge} y$) which is defined by Equation 2.1. Figure 2.1 gives a visualization of the minimum t-norm.

$$\forall x, y \in [0,1] : x \tilde{\wedge} y = \min(x, y) \tag{2.1}$$

We will use the fuzzy logical Kleene-Dienes implicator I of x and y (also denoted by $x \tilde{\rightarrow} y$) which is defined by Equation 2.2. Figure 2.2 gives a visualization of the Kleene-Dienes implicator.

$$\forall x, y \in [0,1] : x \tilde{\rightarrow} y = \max(1 - x, y) \tag{2.2}$$

Finally, an example of a t-conorm is the maximum t-conorm operator S_M defined by Equation 2.3 and visualized in Figure 2.3.

$$\forall x, y \in [0,1] : S_M(x, y) = max(x, y) \tag{2.3}$$

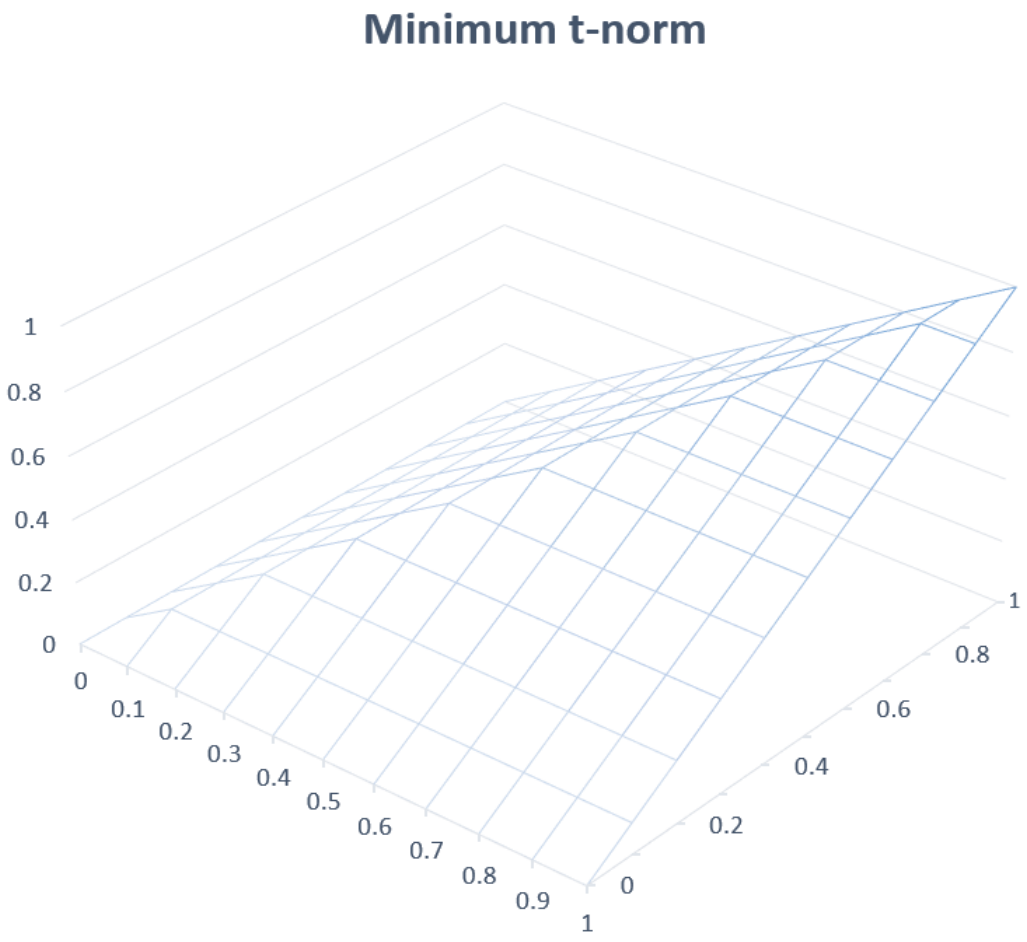

Figure 2.1: This figure is a 3D visualization of a fuzzy logical conjunction, the minimum t-norm.

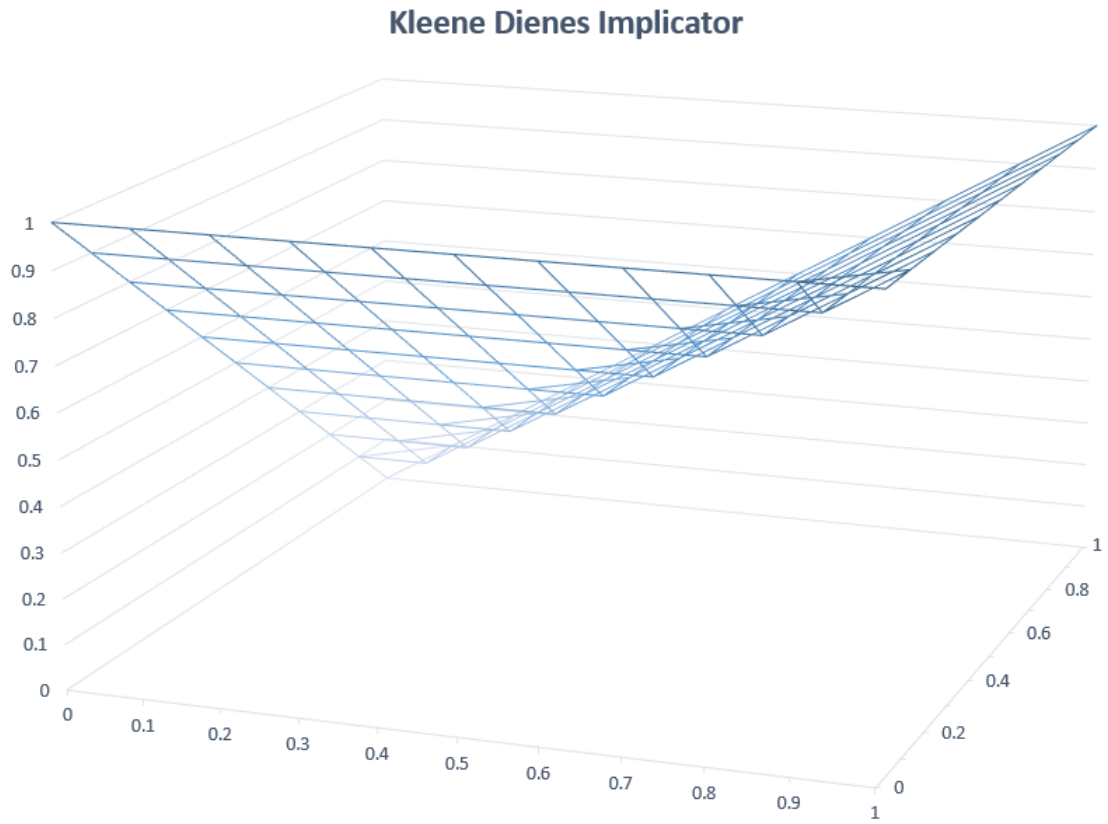

Figure 2.2: This figure is a 3D visualization of the Kleene-Dienes fuzzy logical implicator.

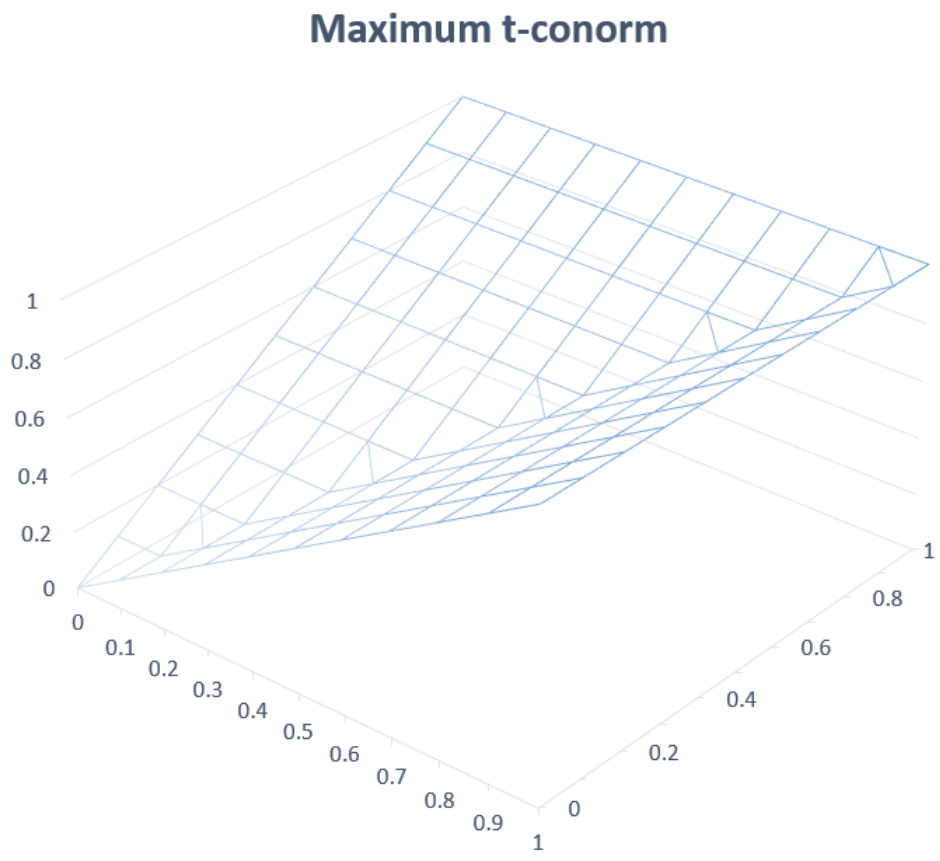

Figure 2.3: This figure is a 3D visualization of the maximum t-conorm

2.1.2 Rough Sets

Rough set theory [54] [30] is a discipline of computer science that deals with information systems with data inconsistencies, such as two patients with the same symptoms but a different underlying disease. It forms a powerful machine learning tool that can be used in many data mining tasks such as feature selection, prediction, instance selection and decision making. It also has applications in many fields such as medical data analysis, image processing, finance and many other real-life problems [52]. A rough set approximates a certain set of elements with two other subsets called upper and lower approximations. For example, Figure 2.4 lists a set of patients that had flu, namely $\{P1, P2\}$, and those that did not have flu, $\{P3, P4\}$, given their diagnosed conditions. The upper approximation of

	Attributes			Decision
	Temperature	Cough	Headache	Flu
P1	High	Y	Y	Y
P2	High	Y	N	Y
P3	High	Y	Y	N
P4	low	N	N	N

Figure 2.4: A small decision system representing a set of patients diagnosed to have (or do not have) flu given their temperature, coughing and headache conditions.

the set of patients that had flu is the set of all patients that had flu in addition to patients that had the same symptoms as the ones that had flu. For example, $P1$ and $P2$ in Figure 2.4 both had flu. $P3$ is a patient that had the same symptoms as $P1$ even though he didn't have flu. Thus, the upper approximation is the set $\{P1, P2, P3\}$. The lower approximation is the set of patients that had flu such that there are no other patients with the same symptoms who did not have flu. For example, there are no patients in Figure 2.4 that had the same symptoms as $P2$ but did not have flu. Thus, the lower approximation is the set $\{P2\}$. Note that even though $P1$ had flu, he doesn't belong to the lower approximation set because he had the same symptoms as $P3$, which didn't have flu. So, if we want to classify a new patient that has the same symptoms as $P1$ and $P3$ (high temperature, coughing and

headache), then we are not certain if he should belong to the set of patients that have flu because patients with these symptoms do not always belong to the set of patients that have flu. On the other hand, if the new patient has symptoms similar to $P2$ (high temperature, coughing and no headache), then we have enough information to classify him as having flu because any patient, in the decision system in Figure 2.5, with these symptoms always belongs to the set of patients that have flu.

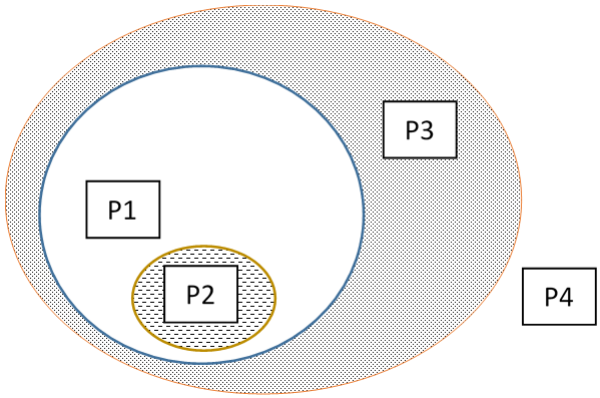

Figure 2.5: The white circle (the one with no texture) is the set of patients that had flu. The circle with dashed lines is the set of all patients that are guaranteed to have flu (lower approximation). The gray circle (the one with small dots) represents the set of all patients who have flu or could have flu (upper approximation).

In an information system with more than two possible values for the decision attribute, the upper and lower approximations can be computed for each of the decision values. For instance, if the decision column in Figure 2.4 was "illness type" and contained the set {flu, fever, bacterial infection} as possible decision values, then there will be upper and lower approximations for flu, upper and lower approximations for fever and upper and lower approximations for bacterial infection. Figure 2.5 visualizes the upper and lower approximations based on the decision system given in Figure 2.4.

Formally, the data is represented in rough set theory as an *information system* $(X, \mathcal{A})$, where $X = \{x_1, \ldots, x_n\}$ and $\mathcal{A} = \{a_1, \ldots, a_m\}$ are finite, non-empty sets of objects and attributes, respectively. The values of the attributes for the instances can be represented in

an $n \times m$ matrix Q where $q_{i,j}$ ($i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$) is the value of attribute a_j for instance x_i. The values of the attributes $a \in \mathcal{A}$ can only be discrete or categorical.

Given $(X, \mathcal{A})$, we define an equivalence relation R, also called the indiscernibility relation, as:

$$R = \{(x_i, x_j) \in X^2 \; and \; (\forall y \in \{1, \ldots, m\})(q_{i,y} = q_{j,y})\} \tag{2.4}$$

Its equivalence classes $[x]_R$ can be used to approximate classes, represented by subsets of X. Given a class $C \subseteq X$, its lower and upper approximations are defined by

$$R \downarrow C = \{x \in X | [x]_R \subseteq C\} \tag{2.5}$$

$$R \uparrow C = \{x \in X | [x]_R \cap C \neq \phi\} \tag{2.6}$$

The instances of $R \downarrow C$ can be with certainty classified as members of the class C based on the values of their attributes from A, while the instances of $R \uparrow C$ can be classified only as possible members of C. The set $X \setminus (R \downarrow C)$ consists of those instances that can be with certainty classified as not belonging to C. The boundary region $(R \uparrow C) \setminus (\mathrm{R} \downarrow C)$ thus consists of those instances that can not be decisively classified in or outside of C because of lack of information. A set C is said to be rough if the boundary region is nonempty.

2.1.3 Fuzzy Rough Sets

Rough set theory can only work with discrete values. If the information system contains numerical attributes then they have to be discretized. Fuzzy rough set [11] [15] theory is an extension of rough set theory that can handle numerical values in addition to discrete values. It is very similar to rough set theory but with the gradual values in mind. In rough set theory, an element can either belong to a set or it does not belong to that set. For example, a patient in Figure 2.4 can either completely belong to the set of patients that had flu or completely belong to the set of patients that didnt have flu. This means that the range of membership values is binary (either belong or doesn't belong). In fuzzy rough set theory, however, an element belongs to every possible set but with a certain degree of

membership that ranges between 0 and 1. For instance, a patient can belong to the set of patients that had flu with a certain degree of membership. Decisions in Figure 2.4 and Figure 2.5 could be having very bad flu with degree 0.9 or recovering from flu with degree 0.1 which indicates the severity level of the illness.

In fuzzy rough set analysis, the indiscernibility relation R is represented by an $n \times n$ matrix R. The value $r_{i,j}$ is the degree to which the objects x_i and x_j ($i, j \in \{1, \ldots, n\}$) are similar. The values $r_{i,j}$ range between 0 (x_i and x_j are completely dissimilar) and 1 (x_i and x_j are identical). Analogous to Equation 2.4, the fuzzy indiscernibility relation matrix r can be derived from the information system $(X, \mathcal{A})$ by aggregating per-attribute similarities of the objects:

$$r_{i,j} = \frac{1}{m} \sum_{y \in \{1,\ldots,m\}} f(q_{i,y}, q_{j,y}) \tag{2.7}$$

where f is a similarity function. For numeric attributes, we represent f with a similarity function that is based on the normalized Manhattan distance which is defined in Equation 2.8 with the range of a_y as defined in Equation 2.10. For nominal attributes, we represent f with the definition in Equation 2.9.

$$f(q_{i,y}, q_{j,y}) = 1 - \frac{|q_{i,y} - q_{j,y}|}{range(a_y)} \tag{2.8}$$

$$f(q_{i,y}, q_{j,y}) = \begin{cases} 1 & \text{if } q_{i,y} = q_{j,y} \\ 0 & \text{Otherwise} \end{cases} \tag{2.9}$$

$$range(a_y) = max\{q_{i,y} | i \in \{1, \ldots, n\}\} - min\{q_{i,y} | i \in \{1, \ldots, n\}\} \tag{2.10}$$

In addition to the information system $(X, \mathcal{A})$, a fuzzy class C is given in the form of an $n \times 1$ vector. For each $i \in \{1, \ldots, n\}$, the value c_i denotes the degree to which object x_i belongs to class C. There may be several such fuzzy classes to which an instance x_i belongs. However, the focus will be on only one class C for simplicity. The values c_i of C range between 1 (x_i completely belongs to the class C) and 0 (x_i does not belong at all to the class C).

The lower and upper approximations of a fuzzy class C are defined w.r.t. an implicator and a t-norm respectively, which are equivalent to the boolean implication and boolean conjunction operators. In the experimental evaluation part of this thesis we use the Kleene-Dienes implicator and the minimum t-norm, respectively defined as $x \tilde{\rightarrow} y = \max(1 - x, y)$ and $x \tilde{\wedge} y = \min(x, y)$, for x and y in $[0, 1]$.

For each instance x_j ($j \in \{1, \ldots, n\}$) its membership values to the lower and upper approximation of C (also denoted by l_j and u_j respectively) are respectively defined as:

$$l_j = \min_{i \in \{1,\ldots,n\}} (r_{j,i} \tilde{\rightarrow} c_i) \tag{2.11}$$

$$u_j = \max_{i \in \{1,\ldots,n\}} (r_{j,i} \tilde{\wedge} c_i). \tag{2.12}$$

2.1.4 Example: Fuzzy Rough Set Approximations

In this part of the chapter, we walk through an example to demonstrate the main steps to compute the fuzzy rough upper and lower approximations. For this purpose, we use the data set in Figure 2.6 which has four instances, three attributes and one class vector. We perform the following steps to compute the lower and upper approximations.

- First, we compute the similarity matrix R as defined in in Equation 2.7. In our example, $r_{1,2} = \frac{1}{3}((1 - \frac{|1-2|}{6-1}) + (1 - \frac{|5-4|}{5-1}) + (1 - \frac{|3-2|}{3-2})) = \frac{1}{3}(3 - (\frac{1}{5} + \frac{1}{4} + 1)) = \frac{1}{3}(2 - \frac{9}{20}) = 0.52$. The complete similarity matrix is shown in Figure 2.7.

- Second, we compute the upper approximation U for the class vector C using Equation 2.12. This equation can also be rewritten as $u_i = \max\{(r_{i,1} \tilde{\wedge} c_1), (r_{i,2} \tilde{\wedge} c_2), \ldots, (r_{i,n} \tilde{\wedge} c_n)\}$ where $x \tilde{\wedge} y$ is the minimum t-norm of x and y which is equal to $\min(x, y)$. In our example, $U_1 = \max\{\min(1, 1), \min(0.52, 1), \min(1, 0), \min(0, 0.5)\} = \max\{1, 0.52, 0, 0.5\} = 1$. The complete upper approximation vector is shown in Figure 2.8.

- Finally, we compute the lower approximation L for the class vector C using Equation 2.11. This equation can also be rewritten as $l_i = \min\{(r_{i,1} \tilde{\rightarrow} c_1), (r_{i,2} \tilde{\rightarrow} c_2), \ldots, (r_{i,n} \tilde{\rightarrow} c_n)\}$ where $x \tilde{\rightarrow} y$ is the Kleene Dienes implication of x and y which is equal to $\max(1 - x, y)$.

For instance, $L_1 = \min\{\max(1-1,1), \max(1-0.52,1), \max(1-1,0), \max(1-0,0.5)\} = \min\{1,1,0,1\} = 0$. The complete lower approximation vector is shown in Figure 2.9.

	Attributes			Class Vector C
	a_1	a_2	a_3	
x_1	1	5	3	1
x_2	2	4	2	1
x_3	1	5	3	0
x_4	6	1	2	0.5

Figure 2.6: An information system that is used as an example in this section.

	x_1	x_2	x_3	x_4
x_1	1	0.52	1	0
x_2	0.52	1	0.52	0.48
x_3	1	0.52	1	0
x_4	0	0.48	0	1

Figure 2.7: This is the similarity matrix computed from Figure 2.6.

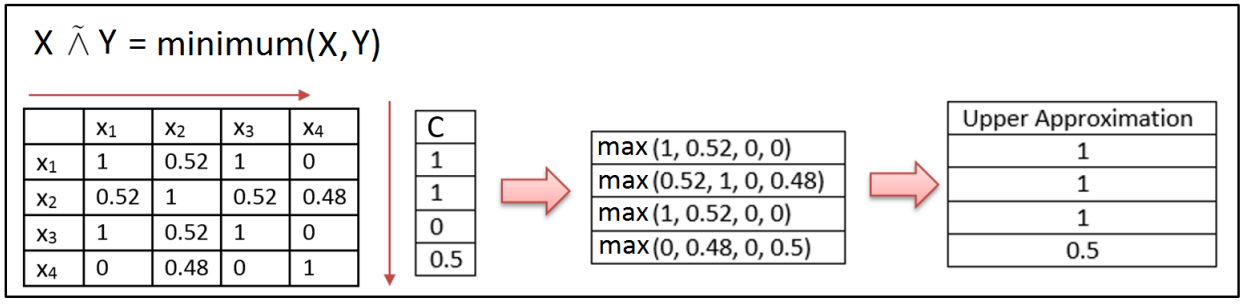

Figure 2.8: The steps to compute the upper approximation given a similarity matrix and a class vector. Note that the fuzzy conjunction used in this figure is the minimum t-norm.

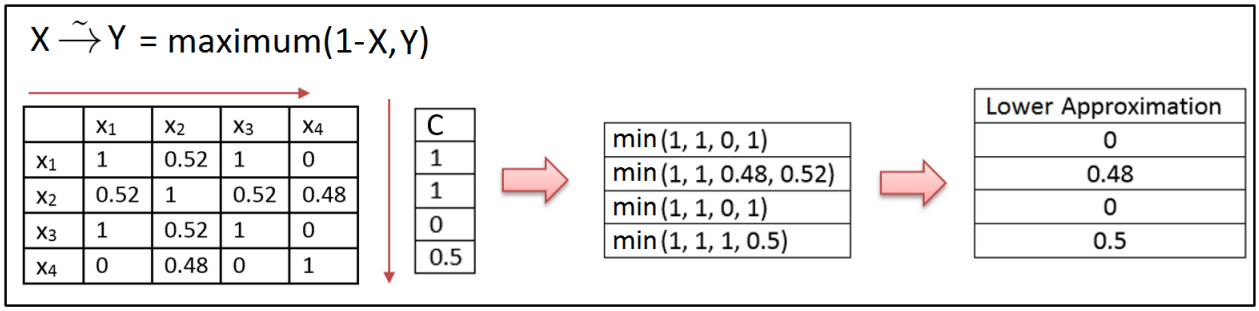

Figure 2.9: The steps to compute the lower approximation given a similarity matrix and a class vector. Note that the fuzzy implication used in this figure is the Kleene Dienes implication operator.

2.2 Distributed Processing Frameworks

Data sets in many areas are getting bigger and bigger (exceeding Peta-bytes). This fact resulted in the manifestation of new techniques to manage such data sets and perform computations on them. One famous technique is the MapReduce [22] technique, invented by engineers at Google. The main components of MapReduce are two functions which are Map and Reduce. A Map function converts the data set into partitions of key/value pairs and distributes these partitions, over multiple nodes in a compute cluster. In some cases, the Map function applies some operations on the distributed partitions. A Reduce function is executed after one or more Map calls on a data set. It mainly aggregates the results from the Map functions and applies some operations on them. Between Map and Reduce calls, intermediate data gets sorted and shuffled between the compute nodes. Figure 2.10 shows an example MapReduce that counts each word in a document. First it splits the data set into partitions of words and sets each word to be a key and the value to be number 1. Then it shuffles the data partitions and finally aggregates them by word (key) and applies a sum operation on all the values associated with the same word. MapReduce made great success in several areas including machine learning, large-scale graph computations and large matrix processing. Two well-known open source frameworks that can be used to solve problems using the MapReduce technique are Hadoop [29] [1] and Apache Spark [37]. We discuss them in more details below.

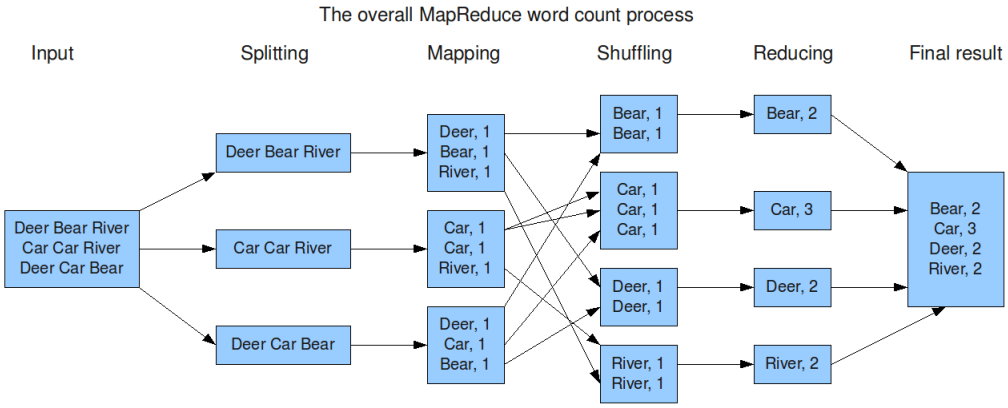

Figure 2.10: This is a demonstration of the word count example. MapReduce first splits the data line by line then maps each word to the number 1 where the word is the key. Then it shuffles the words by grouping the same words together and finally reduces pairs by key and applies summation on the values. This image is taken from `http://www.cs.uml.edu/~jlu1/doc/source/report/img/MapReduceExample.png` and last accessed on 11/12/2014.

.

2.2.1 Hadoop

Hadoop is a framework that provides a distributed large scale storage, called Hadoop File System (HDFS), and supports parallel processing on large data sets following the MapReduce model. A data set that is to be stored on Hadoop is split in parts that get replicated over multiple nodes of the Hadoop cluster. The number of replications is configurable by the system administrator. This way, Hadoop takes advantage of the locality of the data while processing a data set.

2.2.2 Spark

Apache Spark is a clustering and distributed processing system that is developed in Scala, a java based programming language, and implements the MapReduce model. To process a data set, Spark uses the resilient distributed data set (RDD) [38] object which splits the data set into smaller partitions and distributes them over multiple nodes. In addition, an RDD

object enables performing in-memory computations on a data set in a fault-tolerant way. It also provides the ability to cache the data set partitions in memory so that they can span multiple map and reduce steps. This makes it more efficient when executing an algorithm that is iterative in nature. Experiments showed that Spark can outperform Hadoop [37]. Spark is divided into three main components which are a driver program, a cluster manager and worker nodes. The driver program is the main program, which serves as the starting point of execution of an application on the Spark cluster. The cluster manager is a resource allocator for Spark applications. The worker nodes are the nodes responsible for doing the actual data processing in the form of tasks. Each application will have a set of processes called Executors that are responsible for executing the tasks. Figure 2.11 gives an overview of the Spark achitecture.

As we mentioned earlier, Spark partitions a data set using an RDD which enables transforming the data from one form to another through map calls. For instance, a text-based data set with each line representing a person's name, age and salary can be loaded using an RDD so that each element in the RDD is a line in the data set. Each element in the RDD then can be converted into an object of type person. This conversion is called RDD transformation and is done through calls to Map functions. The data in an RDD can also be aggregated through a call to a Reduce function. For example, a Reduce function call on an RDD can aggregate the salary of every person in the RDD and return the total salary of all people in the data set. The Reduce function is best used in an aggregation scenario. However in a scenario where the output is a list of elements, the Grouping function is used. One scenario of using the Grouping function is to find a list of people whose age is greater than 30.

Spark also provides the concepts of accumulator and broadcast variables. The accumulator variable can be used to apply an associative operation on the elements of a data set by passing it over each element in the RDD. A typical scenario for the accumulator variables is to implement counters. A broadcast variable, on the other hand, is a read-only value that is sent from the driver node to every worker node. A typical scenario for it is when the driver node broadcasts a list of constant values that are supposed to be used by every worker node.

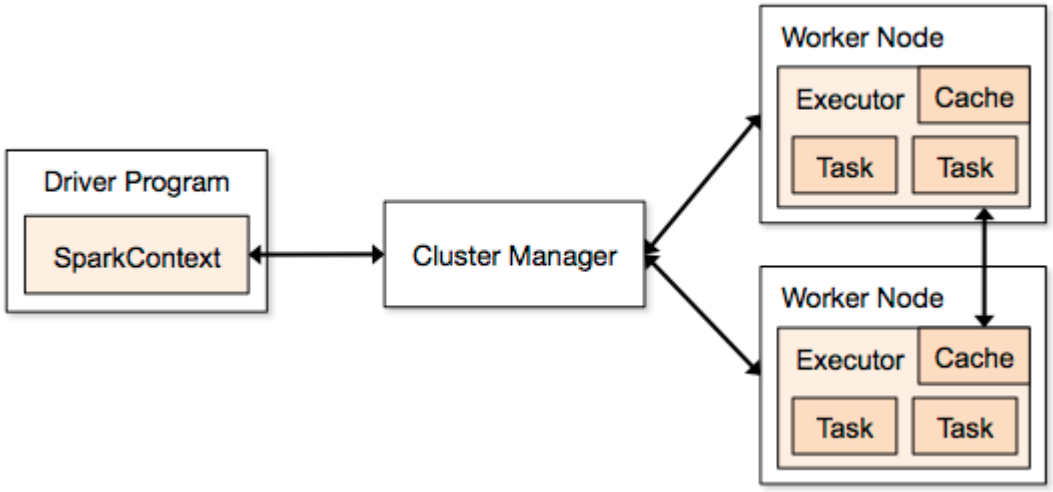

Figure 2.11: This figure shows the architecture of Apache Spark. This image is taken from `https://spark.apache.org/docs/latest/img/cluster-overview.png` and last accessed on 11/12/2014.

2.2.3 Message Passing Interface

The Message Passing Interface (MPI) is a set of standards that define a library of functions to be used in C, C++ and FORTRAN programs. It aims is to parallelize the implementation of an algorithm by distributing the load across multiple compute nodes or multiple cores of one compute node. A rank in MPI is a computer process created in a local or a remote compute node in a computing cluster. It is up to the MPI implementation to decide in which physical compute node to create a particular rank. MPI consists of several low level functions that can be used to write parallel code. There are many implementations of MPI available such as MPICH [49], Intel MPI [2] and OpenMPI [17].

MPI has been extensively used in scientific computing intensive applications [20], specially the ones that involve matrix operations. Since MPI and C are lower level than Spark and Scala (which is based on Java), it is more challenging to develop an MPI solution, more time consuming and more difficult to troubleshoot. A C program runs generally faster than a Java program [31] [32] [42]. So, it is expected that an MPI program runs faster than a Spark program. In this thesis, we conduct a performance comparison between the two frameworks

based on our implementation of the fuzzy rough set approximations computation.

2.3 Data Sets

Two types of data sets are used in this thesis. The first one is a group of synthetic data sets that are used to evaluate the scalability of our implementations of the fuzzy rough set approximation computations, k-NN algorithm and fuzzy rough set based prototype selection algorithm. The second is a health care data set that is used to evaluate the accuracy of our weighted k-NN implementation as well as our fuzzy rough set based prototype selection implementation.

2.3.1 Synthetic Data Set

In order to test the scalability of our implementations, their behavior needs to be observed as the data set characteristics change. Such characteristics include number of instances as well as the number of attributes. So, we have generated thirteen data sets as in Figure 2.12. All of these data sets are comma separated numeric values with the first column as a row identifier (serial number) the last column represents the class, or decision, vector and the columns in between are attributes. All attribute values are randomly generated numbers between 0 and 1000. The class vectors are generated in the same way, but with randomly generated values between 0 and 1.

2.3.2 Healthcare Data

a) *State Inpatient Database*

The Washington State Inpatient Database (SID) is a healthcare data set provided by the Healthcare Cost and Utilization Project (HCUP) which contains clinical and claims data, collected through a partnership between HCUP and the Washington State Department of Health. In this thesis, we use a preprocessed version of the original SID data set as detailed in [46] which is in a tabular csv format that has a total of 320395 rows with each row represents a patient with a total of 807 attributes. The attributes included in this data set are age, gender, race, 773 diagnosis attributes with integer values showing how many times

	Name	Instances	Attributes
1	10k-10	10000	10
2	100k-10	100000	10
3	1000k-10-1	1000000	10
4	1000k-10-2	1000000	10
5	10000k-10	10000000	10
6	10k-50	10000	50
7	100k-50	100000	50
8	1000k-50-1	1000000	50
9	1000k-50-2	1000000	50
10	10000k-50	10000000	50
11	10k-20	1000000	20
12	10k-30	1000000	30
13	10k-40	1000000	40

Figure 2.12: This is a list of all synthetic data sets that were generated for the scalability experiments.

a patient has been admitted with certain medical condition, 29 comorbidity attributes (with true or false values), the previous cost attribute which is the aggregated patient expenses over the first three quarters (January through September) of year 2010 and finally the future cost attribute which is the aggregated patient expenses over the fourth quarter (October through December) of the same year. Our objective is to predict the future cost attribute using the rest of the attributes.

b) *MEPS*

The Medical Expenditure Panel Survey (MEPS) data set is another medical data set that we are using in experiments in this thesis. It consists of data extracted from responses to panel surveys given to households and their employers, medical providers, and insurance providers over two year periods. The survey is intended to be representative of the national civilian non-institutionalized population of the United States. The data that we use represents the 2011-2012 period. We actually used an already preprocessed file from [46] which contains 14039 rows each representing a patient with a total of 570 numeric attributes. Each attribute, except the last, represents how many times a patients has been admitted with certain medical condition. The last attribute represents the total cost of patients in the

year 2011. Our task is to predict the total cost of patients in the year 2012.

We use this data set to measure the accuracies of our distributed fuzzy rough prototype selection algorithms (see Chapter 5).

Chapter 3

COMPUTING FUZZY ROUGH SET APPROXIMATIONS FOR LARGE DATA SETS

In this chapter, we provide a scalable approach for computing the fuzzy rough set upper and lower approximations. First, we describe how our approach differs from other related works. Then we go over the general idea of our approach. Next, we give details of the implementation of our approach on Apache Spark. After that, we describe how we implemented the same approach on MPI. Finally, we present scalability experiments and results for both implementations and also provide a comparison of MPI and Spark in light of our implementations.

3.1 Related Work

To the best of our knowledge, we are the first to present a distributed approach to calculate the fuzzy rough lower and upper approximations in an information system. In [47], a parallel attribute reduction algorithm based on fuzzy rough set theory was proposed, but this algorithm is based on mutual information, and does not calculate the fuzzy rough lower and upper approximations explicitly. Additionally, in [39, 50, 26] rough set approaches to handle big data are presented, without explicitly calculating the lower and upper approximations. In addition, these approaches only deal with non-fuzzy rough sets; a significant distinction.

The work of Zhang et al. is more closely related to the work presented in this thesis. These researchers have worked on parallel approaches to calculate the lower and upper approximations in traditional (non-fuzzy) rough set theory. For instance, in [25], the authors present a MapReduce based approach to compute the lower and upper approximations in a decision system, i.e. an information system with an additional categorical decision attribute (class) that is used to approach classification problems. Calculating the equivalence classes, both w.r.t. the conditional attributes and the decision attribute, is carried out in parallel, as well as calculating the final lower and upper approximations of all decision classes. In

[24], different runtime systems to use these approximations for knowledge acquisition are compared. In [51], an approach to calculate the (non-fuzzy) positive region, i.e. the union of the lower approximation of all classes is proposed.

Our work differs fundamentally from these approaches as calculating the fuzzy rough lower and upper approximations requires calculating the fuzzy similarity matrix, while the non-fuzzy lower and upper approximations are based on an equivalence relation, that can be calculated easily in one MapReduce iteration. Indeed, the maps divide chunks of objects in equivalence classes (groups with the same attribute values) and these equivalence classes are merged in the reduce step. Unfortunately, this strategy cannot be used for calculating the fuzzy similarity between objects, as all objects need to be compared against every other object for each attribute value. This poses additional challenges when calculating the fuzzy rough upper and lower approximations, as one MapReduce round does not suffice to calculate the fuzzy similarity matrix.

The approach in [47, 24] is extended in [27] for composite rough sets, where attributes from different types can be handled. This approach seems to be suitable for our purposes at first sight, as continuous attributes can be handled. However, these composite rough sets require crisp equivalence relations for each attribute type, which means that discretization is needed to handle the continuous attributes. Our approach uses a fuzzy similarity relation to handle continuous attributes, and as a result there is no information loss that is usually associated with discretization.

3.2 Approach

Our approach assumes the availability of a cluster with a master node and K slave nodes. It also assumes that the whole data set is accessible to every slave node. In addition, it is required that every slave node has enough memory to store $\lceil \frac{n}{K} \rceil$ of the n rows in the data set, upper and lower approximation vectors (each of size n) and the class vector. The general idea is to equally distribute the computational load over the available slave nodes such that each slave node performs a part of the fuzzy rough set approximations computation. Given an information system with n instances (rows), each slave node processes $\lceil \frac{n}{K} \rceil$ of the rows in the data set so that it computes $\lceil \frac{n}{K} \rceil$ columns of the $n \times n$ similarity

matrix. Using these computed columns and the class vector, each node generates a partial lower approximation vector and a partial upper approximation vector using Equations 2.11 and 2.12. Recall that a value u_i in the upper approximation can be expressed as $u_i = max\{(r_{i,1}\tilde{\wedge}c_1), (r_{i,2}\tilde{\wedge}c_2), \ldots, (r_{i,n}\tilde{\wedge}c_n)\}$. In this case, each node computes $\lceil \frac{n}{K} \rceil$ of the n $(r_{i,j}\tilde{\wedge}c_j)$ terms and takes their maximum for every row q_i in the information system Q. This forms one element u_i^k $(k \in \{1, \ldots, K\})$ in the partial upper approximation vector computed by one slave node. Now to compute the i^{th} value u_i in the final upper approximation, we aggregate the i^{th} values in the partial upper approximations of all slave nodes by taking their maximum as shown in Equation 3.2.

Similar steps are done to compute the lower approximation vector. Recall also that a value l_i in the lower approximation is equal to $l_i = \min\{(r_{i,1}\tilde{\rightarrow}c_1), (r_{i,2}\tilde{\rightarrow}c_2), \ldots, (r_{i,n}\tilde{\rightarrow}c_n)\}$. In this case, each node computes $\lceil \frac{n}{K} \rceil$ of the n $(r_{i,j}\tilde{\rightarrow}c_j)$ terms and takes their minimum for every row q_i in Q. This forms one element l_i^k $(k \in \{1, \ldots, K\})$ in the partial lower approximation vector computed by one slave node. Now to compute the i^{th} value l_i in the final lower approximation, we aggregate the i^{th} values in the partial lower approximations of all slave nodes by taking their minimum as shown in Equation 3.1.

$$l_i = min\{l_i^1, l_i^2, \ldots, l_i^K\} \tag{3.1}$$

$$u_i = max\{u_i^1, u_i^2, \ldots, u_i^K\} \tag{3.2}$$

The process of computing partial upper and lower approximations as well as computing the final approximation vectors is shown in Figure 3.1.

The flow of the overall approach is depicted in Figure 3.2. Computations happen on the master node (left) and on K slave nodes (right, with $K = 2$ in the picture). Each slave node k $(k \in \{1, \ldots, K\})$ is assigned a horizontal partition P_k of the data, for which it will do computations. However, it is assumed that the entire data is available to each of the K slave nodes. That is, the data is not partitioned over the slave nodes but the computations are. Each P_k partition has $\lceil \frac{n}{K} \rceil$ rows (except for the last slave node which has a partition with the remaining rows, which is possibly less than $\lceil \frac{n}{K} \rceil$).

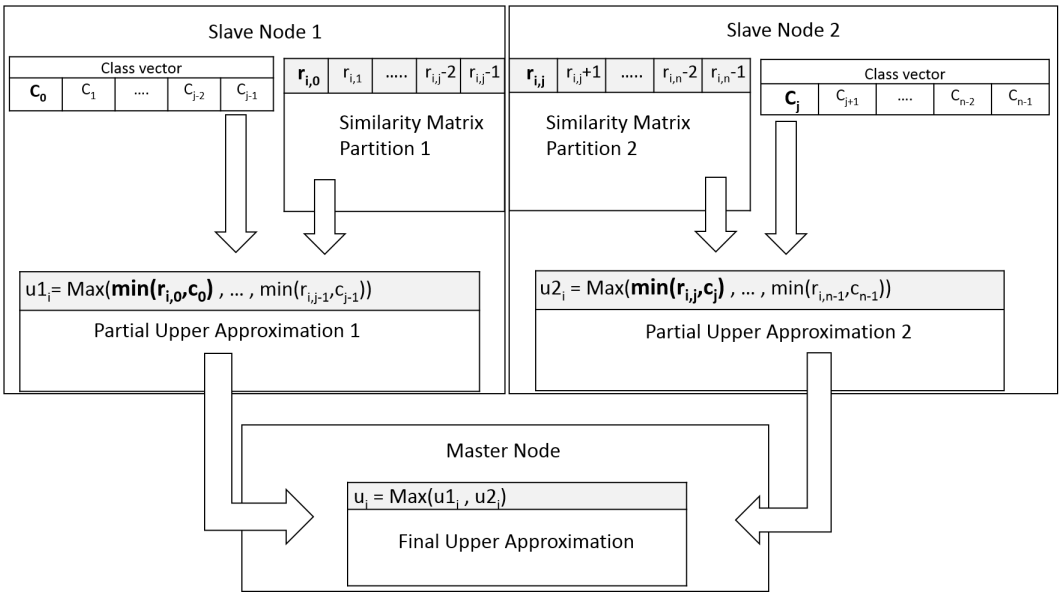

Figure 3.1: This figure shows how the upper approximation vectors are computed using two slave nodes. Each of the slave nodes computes half of the columns of the similarity matrix (vertical partition). Then the first vertical partition is used along with the first half of the class vector to generate the partial upper approximation vector at the first slave node. At the same time, the second vertical partition of the similarity matrix is used along with the second half of the class vector to generate a partial upper approximation vector at the second slave node. Finally, the two partial upper approximation vectors are combined to get the final upper approximation vector. Similar steps are taken to compute the lower approximation vector.

The approach is divided into two stages – data preparation and data processing – which we describe in more details in the next two sections. In the data preparation stage, data is made available and ready to be processed by the slave nodes. This includes the data set, class vectors and ranges of every attribute in the data set. In the data processing stage, upper and lower approximations are computed using the prepared data from the previous stage.

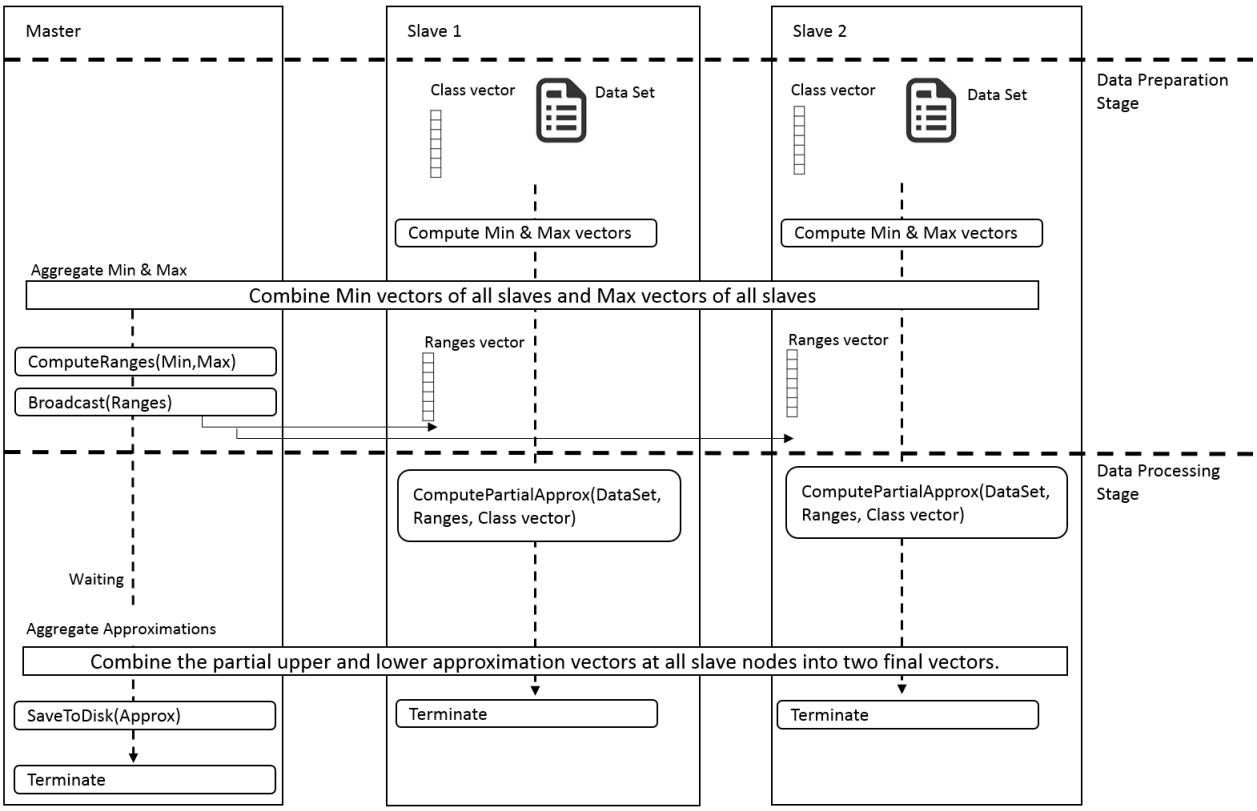

Figure 3.2: This figure summarizes the overall flow of our scalable approach to computing fuzzy lower and upper approximations. The approach has two stages, namely data preparation and data processing. The data preparation ensures that the data set partitions are loaded into memory. In addition, each slave node computes the minimum and maximum values of each attribute for a subset of the rows, and then these values get aggregated at the master node into two final vectors. Then the master computes the attribute ranges given the minimum and maximum vectors and finally broadcasts the ranges vector. This concludes the data preparation stage which gets the data ready to be processed. At the start of the data processing stage, each slave node has the data set, the ranges vector and the class vector as input and uses these to produce the partial upper and lower approximation vectors. Finally, the partial vectors get aggregated into the final upper and lower approximation vectors at the master node. At this point, all slave nodes terminate while the master saves the final approximation vectors to disk and then terminates.

3.3 Implementation on Spark

In this section, we describe our implementation of the approach in the previous section using the Apache Spark framework. We specifically describe how the data preparation and data

processing phases are implemented. The flow of this implementation is shown in Figure 3.3. The implementation steps are also shown in Algorithms 1, 2, 3, 4 and 5. In this section, we assume that the data set is stored on HDFS and that each node has access to it.

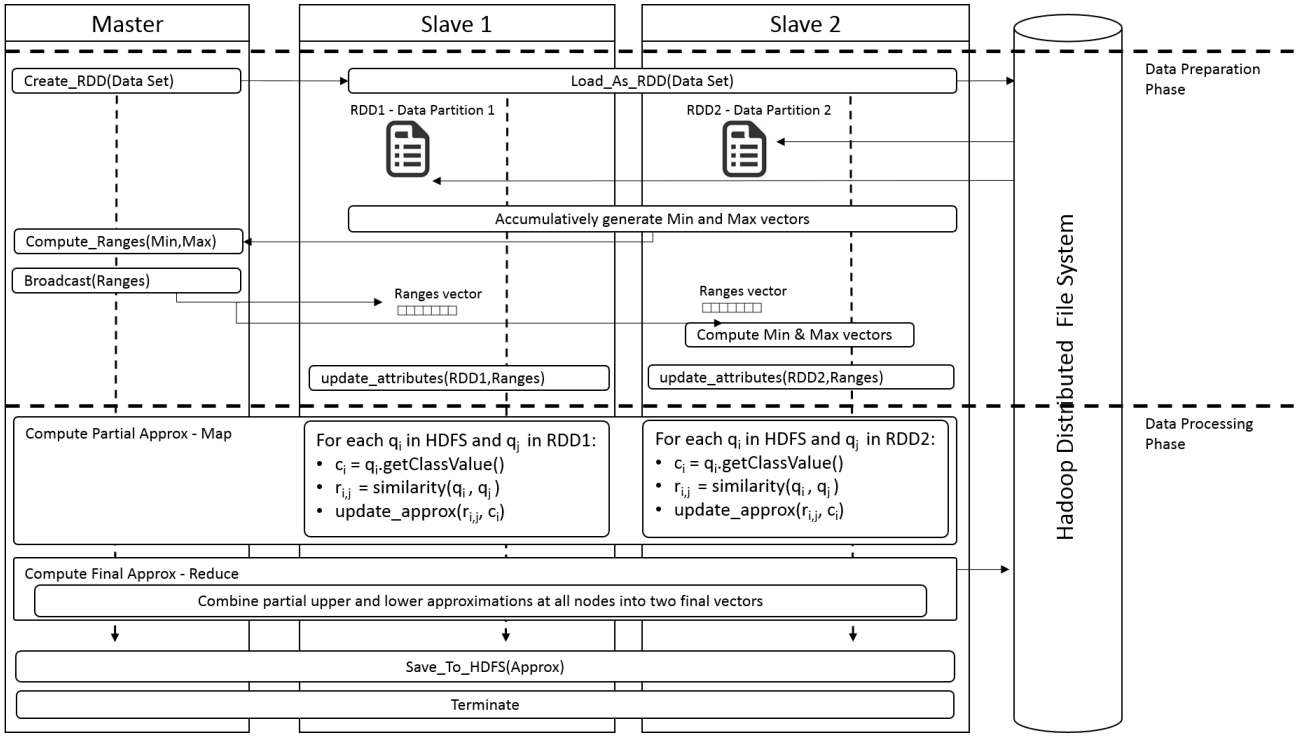

Figure 3.3: This figure summarizes the overall flow of the Spark implementation to computing fuzzy lower and upper approximations using one master node and two slave nodes. The approach has two stages, namely data preparation and data processing. The data preparation stage starts with loading the data set into memory as an RDD. The RDD splits the data set into K partitions (two partitions in this case). Then, the min and max vectors are generated and returned to the master node through the use of Spark accumulators. After that, the ranges vector gets computed using the min and max vectors and then the master node broadcasts the ranges vector. Finally, each attribute of every row gets divided by its range value. In the data processing stage, the partial upper and lower approximation vectors get computed using a Map call. Then, a Reduce call is used to combine them into two final upper and lower approximation vectors. Finally, these two vectors are saved back to HDFS.

3.3.1 Data Preparation

The objective of this phase is to get the data loaded into memory and have it ready for the data processing stage. This includes computing the range of every attribute – which is needed in the denominator of Equation 2.7. In the beginning, the data set gets loaded from HDFS into an RDD of rows. At this point, each row in the RDD is a comma separated text represented by a String object and has a row identifier number. This RDD divides the data set into K partitions, where each partition P_k is of size $\lceil \frac{n}{K} \rceil$, and distributes it over the K slave nodes. Since the default Spark partitioner doesn't always build RDD partitions of equal size, we created a custom data partitioner, RowPartition object, to ensure the data is split into K partitions and each is of size $\lceil \frac{n}{K} \rceil$. It is important to have the partitions to be, roughly, of equal size so that all worker nodes can complete processing their data partitions in relatively the same amount of time. On the contrary, inequal partition sizes could cause some worker nodes to finish their tasks and stay idle while others take a longer time to complete their tasks. In order to make sure that all of the K partitions have $\lceil \frac{n}{K} \rceil$ rows, the RowPartition partitioner assigns a partition number to each row equal to the row number mod K. For example, if the number of partitions is 8 then the 5^{th} row will belong to partition 5 ($5 \ mod \ 8 = 5$) and the 50^{th} row will belong to partition 2 ($50 \ mod \ 8 = 2$). After the RDD is created, a tranformation map call is applied on it so that each row string gets parsed into an ApproxRow object which stores row id, attributes, class vector values and fuzzy rough set upper and lower approximation values which are initialized to zeros and ones respectively.

After that, the range of every attribute gets computed in a parallel way. To do that, two vectors min and max each of size m get created where m is the number of attributes in the information system. The objective of these two vectors is to store the minimum value and maximum value of each attribute in the data set. The values in min are initially set to $+\infty$ and all values in max are initially set to $-\infty$. Two custom Spark accumulators, MaxAccumulator and MinAccumulator have been created to accumulatively populate the min and max vectors. Then, a foreach() method is called on the RDD to iterate over every row $q_i \in \{1, \ldots, n\}$. Then the attributes of q_i get passed to the add() method of the

MaxAccumulator and MinAccumulator to capture maximum and minimum values of every attribute $q_{i,j}$ with $j \in \{1, \ldots, m\}$ such that

$$min_j \leftarrow \min(q_{i,j}, min_j) \tag{3.3}$$

$$max_j \leftarrow \max(q_{i,j}, max_j) \tag{3.4}$$

Once the accumulation parallel loop is finished, the accumulators return min and max vectors to the master node which uses them to compute the ranges of every attribute and stores the result in the $ranges$ vector so that

$$ranges_j = max_j - min_j \tag{3.5}$$

Finally the driver node broadcasts the $ranges$ vector to all slave nodes. The steps to compute the $ranges$ are summarized in lines 4 through 12 of Algorithm 1. Recall that we need to divide by the ranges in Equation 2.8. This division operation is actually done before computing Equation 2.8. We do this by parallelly iterating over every ApproxRow object in the RDD and divide its attributes by the ranges as shown in Equation 3.6 and lines 13 trough 18 of Algorithm 1. In addition, we do this in order to prevent repetitive division operations in the data processing stage as we will clarify in the next section. Note also that the reason why we need to broadcast the $ranges$ vector is because in the data processing stage, the data set is going to be loaded from HDFS again and every attribute of every loaded row needs to be divided by its range. Having this change, Equation 2.8 can now be expressed as shown in Equation 3.7.

$$q'_{i,j} = \frac{q_{i,j}}{ranges_j} \tag{3.6}$$

$$f(q_{i,y}, q_{j,y}) = 1 - |q'_{i,y} - q'_{j,y}| \tag{3.7}$$

At this point, the data is partitioned, loaded in memory and ready to be processed.

3.3.2 Data Processing

The objective of this phase is to compute the upper and lower approximation vectors U and L respectively with their values u_i and l_i as defined in Equations 2.11 and 2.12. Recall that

a value in the upper approximation can be expressed as $u_i = \max\{(r_{i,1}\tilde{\wedge}c_1), (r_{i,2}\tilde{\wedge}c_2), \ldots, (r_{i,n}\tilde{\wedge}c_n)\}$ and a value in the lower approximation can be expressed as $l_i = \min\{(r_{i,1}\tilde{\rightarrow}c_1), (r_{i,2}\tilde{\rightarrow}c_2), \ldots, (r_{i,n}\tilde{\rightarrow}c_n)\}$. Recall also that in our approach, each slave node k computes a partial upper approximation vector U^k, with each value u_i^k equal to the maximum of $\lceil \frac{n}{K} \rceil$ of the n $(r_{i,j}\tilde{\wedge}c_j)$ terms, and a partial lower approximation vector L^k, with each value l_i^k equals to the minimum of $\lceil \frac{n}{K} \rceil$ of the n $(r_{i,j}\tilde{\rightarrow}c_j)$ terms.

The first step in the data processing phase is to compute these partial upper and lower approximation vectors U^k and L^k. This is done through a Map call. This Map call returns an RDD of key-value pairs where the key is a row number and the value is an ApproxRow object which has access to the partial upper and lower approximation values at that key. For example, if the key is equal to i then the value is an ApproxRow that has access to u_i^k and l_i^k. The second step is to aggregate the partial approximation values u_i^k and l_i^k from every slave node and compute the final upper and lower approximation values u_i and l_i. This is done through a Reduce call. Algorithms 2 and 4 provide the steps involved in both the Map and Reduce calls.

a) *Map*

In this Map call, each slave node k computes $\lceil \frac{n}{K} \rceil$ columns of the similarity matrix R and uses them along with P_k (which includes the class vector) and the ranges vector to compute the partial upper and lower approximation values u_i^k and l_i^k $\forall\ i \in \{1, \ldots, n\}$. To do that, each slave node directly accesses the data set in HDFS and runs a loop over the rows in it, refered to as the "outer loop", and then the following is done for every row q_i in HDFS:

1. The row q_i gets converted into an ApproxRow object.

2. Every attribute $q_{i,j}$ in q_i gets divided by $ranges_j$.

3. Another loop, refered to as the "inner loop", is run over every row q_j in the RDD partition P_k assigned to slave node k. In this loop, the following is done:

   (a) Compute the similarity value $r_{i,j}$, which is defined in Equations 2.7, as shown in Algorithm 5. Note that the variable allocated for an $r_{i,j}$ value gets overwritten

each time another $r_{i,j}$ value gets computed. This way, we prevent the problem associated with storing all of $n \times n$ values in the similarity matrix r.

(b) Compute the implicator $(r_{i,j} \tilde{\rightarrow} c_j)$ in Equation 2.11 as shown in line 12 in Algorithm 2.

(c) Compute the t-norm $(r_{j,i} \tilde{\wedge} c_i)$ in Equation 2.12 as shown in line 13 in Algorithm 2.

(d) Compute the partial lower and upper approximation values l_i and u_i in Equations 2.11 and 2.12 as shown in lines 14 and 15 in Algorithm 2. Note that the division in Equation 2.8 happens before this step and also Equation 2.8 is replaced by 3.7 in order to prevent repetitive division operations. This way, we perform one division for every attribute in the data set, resulting in a total of $n \times m$ division operations. If we don't do this, then we will end up doing around $\frac{n^2 \times m}{K}$ division operations inside the inner loop.

At this point, every slave node ends up with n ApproxRow objects that represent the rows in the data set and each of these objects has partial upper and lower approximation values. Each of these objects gets paired with their row number as a key and then the key-value pairs get returned. The steps in this Map call are summarized in Algorithm 2.

b) *Reduce*

In this step, a Spark reduceByKey operation is applied on the RDD from the Map step. The reduceByKey combines all ApproxRow objects that have the same key. Then the final upper and lower approximation values get computed as shown in Equation 3.2 and 3.1. Finally the upper and lower approximation vectors U and L get stored back to HDFS. The reduce steps are shown in Algorithm 4.

3.3.3 Performance Optimizations

Multithreading optimizations have been added to the Map call by creating a pool of multiple threads of size t. Step 4 in Algorithm 2 has been modified so that t lines are read from

HDFS at a time instead of one line. Then for each line in the t lines, a thread from the thread pool gets assigned to handle it.

We have also included another optimization that applies to data sets that have only numeric attributes. Our objective is to reduce the overhead caused by Java method calls inside the inner loop in the Map call (Algorithm 2), especially when computing the similarity values $r_{i,j}$. We do this by replacing method calls with array operations. With this optimization that we describe in more details below, Algorithm 2 is replaced with Algorithm 3.

Recall that each node has P_k in its memory which contains $\lceil \frac{n}{K} \rceil$ instances and each of these instances is represented by an ApproxRow object that encapsulates the m attributes of the instance. So, we replaced the ApproxRow objects with a one dimensional array of size $m \times \lceil \frac{n}{K} \rceil$ which represents all of the attributes in P_k such that the first m values in this array represent the first instance in P_k and the second m values represent the second instance in P_k and so on. This way, we only access a one dimensional array whenever we want to access attributes of an instance in P_k rather than calling methods of an ApproxRow object. We also replaced the Similarity method call at line 11 in Algorithm 2 with lines 14 through 19 in Algorithm 3. Since the similarity method internally calls methods of ApproxRow objects and since it also gets called $m \times \lceil \frac{n}{K} \rceil$ times in Algorithm 2, replacing it with lines 14 through 19 in Algorithm 3 results in replacing all of the $m \times \lceil \frac{n}{K} \rceil$ methods calls and all internal calls to methods of ApproxRow objects with array operations.

The experimental results in section 3.5.2 were taken for the optimized version of the implementation.

Algorithm 1 Spark Program

1: INPUT: $pathToDataset$
2: DECLARE: $MaxAccumulator, MinAccumulator$, ranges, min, max, $m, r1$
3: $rdd[String] \leftarrow createRDDFromHDFS(pathToDataset)$
4: $rdd[ApproxRow] \leftarrow transform(rdd)$
5: $r1 \leftarrow getFirstRow(rdd)$
6: $m \leftarrow r1.getAttributesCount()$
7: **RddForEach**$(row, MaxAccumulator, MinAccumulator)$
8: $\quad attributes \leftarrow row.getAttributes()$
9: $\quad MaxAccumulator.add(attributes)$
10: $\quad MinAccumulator.add(attributes)$
11: **EndRddForEach**
12: $min \leftarrow MinAccumulator.values()$
13: $max \leftarrow MaxAccumulator.values()$
14: **for** i from 1 to m **do**
15: $\quad ranges_i \leftarrow max_i - min_i$
16: **end for**
17: $broadcast(ranges)$
18: **RddForEach**$(row, ranges)$
19: $\quad attributes \leftarrow row.getAttributes()$
20: $\quad$ **for** i from 1 to m **do**
21: $\quad\quad attributes_i \leftarrow attributes_i/\text{ranges}_i$
22: $\quad$ **end for**
23: **EndRddForEach**
24: $partialRDD[Integer, ApproxRow] \leftarrow Map(rdd, pathToDataSet, ranges)$
25: $approxRDD[Integer, ApproxRow] \leftarrow ReduceByKey(partialRDD)$
26: $saveApproximationsToHDFS(approxRDD)$
27: Terminate

Algorithm 2 Spark Program - Map

1: INPUT: $RddPartition, pathToDataset, ranges$
2: DECLARE: $rowsList, attributes, P, m$
3: $P \leftarrow RddPartition.size$
4: **for** $line_i \leftarrow readFromHDFS()$ **do**
5: $r_i \leftarrow$ create $ApproxRow(line_i, ranges)$
6: $u_i \leftarrow r_i.getUpperApprox()$
7: $l_i \leftarrow r_i.getLowerApprox()$
8: **for** j from 1 to P **do**
9: $r_j \leftarrow RddPartition.getRow(j)$
10: $c_j \leftarrow r_j.getClassValue()$
11: $simVal \leftarrow Similarity(r_i, r_j)$
12: $implicator \leftarrow max(1 - simVal, c_j)$
13: $tnorm \leftarrow min(simVal, c_j)$
14: $l_i \leftarrow min(implicator, l_i)$
15: $u_i \leftarrow max(tnorm, u_i)$
16: **end for**
17: $rowsList.add(r_i)$
18: **end for**
19: return $rowsList$

Algorithm 3 Spark Program - MapN (An optimized version of Algorithm 2 that works for data sets with only numeric attributes)

INPUT: $RddPartition, pathToDataset, ranges$
DECLARE: $rowsList, attributes, P, m$
$P \leftarrow RddPartition.getSize()$
$m \leftarrow RddPartition.getNumberOfAttributes()$
$attributes \leftarrow convertToVector(RddPartition)$
$c \leftarrow getClassVector(RddPartition)$
for $line_i \leftarrow readFromHDFS()$ **do**
    $r_i \leftarrow$ create $ApproxRow(line_i, ranges)$
    $u_i \leftarrow r_i.getUpperApprox()$
    $l_i \leftarrow r_i.getLowerApprox()$
    for j from 1 to P **do**
        $simVal \leftarrow 0$
        $attrStartIndex \leftarrow j \times m$
        for a from 1 to m **do**
            $attr1 \leftarrow attributes[attrStartInd + a]$
            attr2 $\leftarrow r_j$[a]
            $simVal \leftarrow simVal + |attr1 - attr2|$
        end for
        $simVal \leftarrow (m - simVal)/m$
        $implicator \leftarrow max(1 - simVal, c_j)$
        $tnorm \leftarrow min(simVal, c_j)$
        $l_i \leftarrow min(implicator, l_i)$
        $u_i \leftarrow max(tnorm, u_i)$
    end for
    rowsList.add(r_i)
end for
return rowsList

Algorithm 4 Spark Program - ReduceByKey

1: INPUT: $row1, row2$

2: DECLARE: $row3$

3: $row3 \leftarrow createApproxRow()$

4: $u1 \leftarrow row1.getUpperApprox()$

5: $l1 \leftarrow row1.getLowerApprox()$

6: $u2 \leftarrow row2.getUpperApprox()$

7: $l2 \leftarrow row2.getLowerApprox()$

8: $u3 \leftarrow row3.getUpperApprox()$

9: $l3 \leftarrow row3.getLowerApprox()$

10: **for** i from 1 to m **do**

11: $\quad u3_i \leftarrow max(u1_i, u2_i)$

12: $\quad l3_i \leftarrow min(l1_i, l2_i)$

13: **end for**

14: return $row3$

Algorithm 5 Spark Program - Similarity

1: INPUT: $row1, row2$
2: DECLARE: $simVal, attrN1, attrN2, attrS1, attrS2, m$
3: $attrN1 \leftarrow row1.getNumericAttributes()$
4: $attrS1 \leftarrow row1.getStringAttributes()$
5: $attrN2 \leftarrow row2.getNumericAttributes()$
6: $attrS2 \leftarrow row2.getStringAttributes()$
7: $m \leftarrow row1.getNumberOfAttributes()$
8: $simVal \leftarrow 0$
9: **for** i from 1 to $row1.NumericAttrCount$ **do**
10: $\quad simVal \leftarrow simVal + |attrN1_i - attrN2_i|$
11: **end for**
12: **for** i from 1 to $row1.StringAttrCount$ **do**
13: $\quad$ **if** $attrS1[i] == attrS2[i]$ **then**
14: $\quad\quad simVal \leftarrow simVal + 1$
15: $\quad$ **end if**
16: **end for**
17: $simVal \leftarrow (m - simVal)/m$
18: return $simVal$

3.4 Implementation on MPI

The overall idea of the MPI implementation is similar to the Spark implementation and follows the general approach that we described earlier. Like the Spark implementation, the MPI implementation has a data preparation stage and a data processing stage. However, it has some differences related to where the data is stored, how the ranges get computed and also how the data is processed. Below are key differences between the two implementations:

1. Spark relies on HDFS to partitioning the data set and data is distributed over the nodes through an RDD object. In our MPI implementation, however, every node has access to the whole data set locally and is assigned a virtual partition for which it does computations.

2. Spark requires that each row has an identifier number so that it can be used in the reduce stage. In MPI, however, this is not needed because every node is required to allocate an array of the same size to perform a reduce operation such that elements that share the same index get aggregated together from all nodes.

3. Spark uses its Accumulator concept to compute the minimum and maximum values for every attributes that are needed for computing the ranges. The MPI implementation, however, iterates over the rows in its virtual partition to compute the minimum and maximum values and then applies a reduce operation on them.

Next, we provide more details of the MPI implementation. The flow of the MPI implementation is shown in Figure 3.4.

3.4.1 Data Preparation

a) *Data broadcasting*

At the very beginning of the data preparation stage, the master node has the instance-attribute value matrix Q as well as the class or concept vector C, each in a separate file. Before the MPI program starts, a Linux script is run at the master node to distribute the matrix Q over the slave nodes so that each slave node has a full copy of Q. This requires

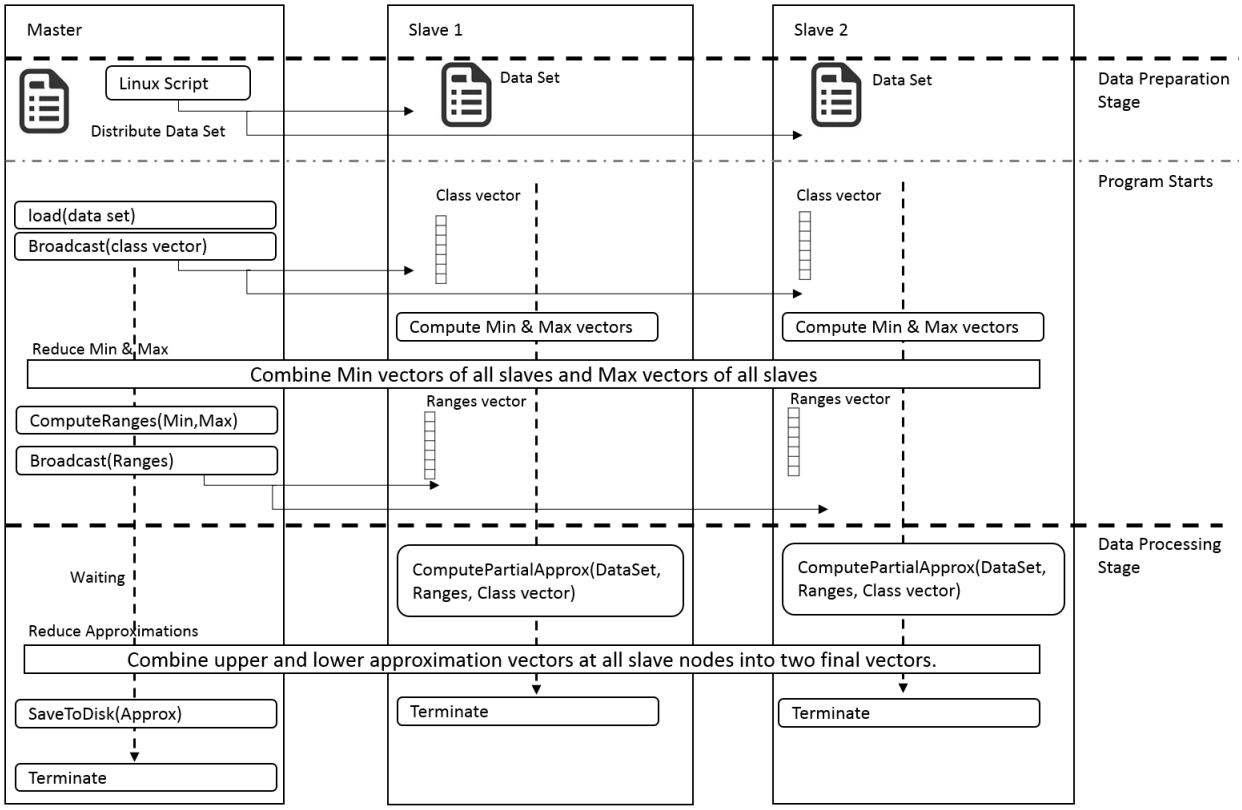

Figure 3.4: This figure summarizes the overall flow of our distributed approach in MPI to computing fuzzy lower and upper approximations. The approach has two stages, namely data preparation and data processing. The data preparation stage starts with a Linux script that distributes the data set, i.e. the instance-attribute matrix, to all slave nodes. Then the master node loads and broadcasts the concept or class vector. After that, each slave node computes the minimum and maximum values of each attribute for a subset of the rows, and then the master node reduces the minimum and maximum vectors created by the slaves into two final vectors. Then the master computes the attribute ranges given the minimum and maximum vectors and finally broadcasts the ranges vector. This concludes the data preparation stage which gets the data ready to be processed. At the start of the data processing stage, each slave node has the data set, the ranges vector and the class vector as input and uses them to produce the partial upper and lower approximation vectors. Finally the partial vectors get reduced at the master producing the final upper and lower approximation vectors. At this point, all slave nodes terminate while the master saves the final approximation vectors to disk and then terminates.

that each slave node has enough disk space to store the whole data set file. By doing this, we are emulating a parallel file system. As will become clear below, even though every slave k ($k \in \{1, \ldots, K\}$) has and needs a full copy of Q, it is only responsible for the computations on its assigned partition P_k of $\lceil \frac{n}{K} \rceil$ rows of Q.

After the matrix Q is sent to all slave nodes, the Linux script triggers the MPI program to run on all slave nodes in addition to the master node. Once started, the master node loads the class vector C from the disk and broadcasts it to all slave nodes.

b) ***Attribute range computation***

After that, the range of every attribute – which is needed in the denominator of Equation 2.8 – gets computed in a parallel way. To this end, each slave node k ($k \in \{1, \ldots, K\}$) creates two vectors $\min^k$ and $\max^k$, each of size m, with m the number of attributes in the information system. The objective of these two vectors is to keep track of the minimum values and maximum values of each attribute in the data set seen so far. Recall that each slave node k is responsible for a partition P_k of instances. Below, we denote the index set of these instances (rows) by I_k. Initially, within each slave node k, all values in $\min^k$ are set to $+\infty$ and all values in $\max^k$ are set to $-\infty$. Then, the slave node k iteratively goes through all instances q_i in its partition P_k and updates the minimum and maximum values for all attributes $q_{i,j}$ ($i \in I_k$ and $j \in \{1, \ldots, m\}$) so that

$$\min_j^k \leftarrow \min(q_{i,j}, \min_j^k) \tag{3.8}$$

$$\max_j^k \leftarrow \max(q_{i,j}, \max_j^k) \tag{3.9}$$

After each slave k is done populating its $\min^k$ and $\max^k$, they get reduced to two vectors $\min^f$ and $\max^f$ at the master node so that, for $j \in \{1, \ldots, m\}$,

$$\min_j^f = \min_{k \in \{1,\ldots,K\}} \min_j^k \tag{3.10}$$

$$\max_j^f = \max_{k \in \{1,\ldots,K\}} \max_j^k \tag{3.11}$$

Then, the master node computes the range of each attribute and stores the result in the ranges vector so that, for $j \in \{1, \ldots, m\}$,

$$\text{range}_j = \max_j^f - \min_j^f \tag{3.12}$$

Finally the master node broadcasts the ranges vector to all slave nodes.

3.4.2 Data Processing

In this stage, each slave node k ($k \in \{1, \ldots, K\}$) has the matrix Q, the class vector C and a vector with the range of each attribute. Recall that each slave k is responsible for a horizontal partition P_k of Q, and that we denote the index set of these instances by I_k. The goal of our algorithm is a lower approximation vector L and an upper approximation vector U, with their values l_i and u_i as defined in Equations (2.11) and (2.12), for $i \in I_k$. Note that (2.11) and (2.12) can be rewritten as

$$l_j = \min_{k \in \{1,\ldots,K\}} \min_{j \in \{1,\ldots,n\}} (r_{i,j} \tilde{\rightarrow} c_j) \tag{3.13}$$

$$u_i = \max_{k \in \{1,\ldots,K\}} \max_{j \in \{1,\ldots,n\}} (r_{i,j} \tilde{\wedge} c_j). \tag{3.14}$$

The computation of

$$l_i^k = \min_{j \in \{1,\ldots,n\}} (r_{i,j} \tilde{\rightarrow} c_j) \tag{3.15}$$

$$u_i^k = \max_{j \in \{1,\ldots,n\}} (r_{i,j} \tilde{\wedge} c_j) \tag{3.16}$$

for $k \in \{1, \ldots, K\}$, can be done fully in parallel on each of the K slave nodes. Since we need these values for all $i \in I_k$ and $j \in \{1, \ldots, n\}$, this involves each slave node k constructing the $\lceil \frac{n}{K} \rceil$ columns from the similarity matrix that correspond to the index set I_k. However, during calculation of (3.15) and (3.16), each value $r_{i,j}$ of the similarity matrix is immediately combined with c_j, eliminating the need to store (entire columns of) the similarity matrix.

At the beginning of the data processing stage, the master node is waiting for each slave node to complete its computations and produce l_i^k and u_i^k, for all $i \in I_k$. In order for a slave node to do that, it loads its assigned partition P_k and keeps it in memory throughout the execution of the program. In addition, for each loaded row q_i in P_k ($i \in I_k$), each value $q_{i,t}$ ($t \in \{1, \ldots, m\}$) in that row gets divided by ranges_t, i.e. by the previously computed range of the attribute $q_{i,j}$. Then, the slave node loops over each of the rows in the matrix Q that it has on disk and does the following (we can refer to this loop as the outer loop; it ranges over $i \in \{1, \ldots, n\}$):

1. Read row q_i from the data set file, and divide each value $q_{i,t}$ ($t \in \{1, \ldots, m\}$) in that row by ranges_t, i.e. by the previously computed range of the attribute $q_{i,t}$.

2. For each row q_j in P_k ($j \in I_k$), the following steps are carried out (we can refer to this as the inner loop):

   (a) Given row q_i from the data set file and row q_j from P_k, compute $r_{i,j}$ as defined by equation 2.7. Note that we do not need to divide by the range here because this has been taken care of at an earlier step.

   (b) Compute the values $r_{i,j} \tilde{\rightarrow} c_j$ and $r_{i,j} \tilde{\wedge} c_j$.

   (c) Update the values l_i^k and u_i^k:

$$l_i^k \leftarrow \min(l_i^k, r_{i,j} \tilde{\rightarrow} c_j)$$
$$u_i^k \leftarrow \max(u_i^k, r_{i,j} \tilde{\wedge} c_j)$$

Once all the K slave nodes have completed the above steps, the master node aggregates all computed values into

$$l_i = \min_{k \in \{1,\ldots,K\}} l_i^k \tag{3.17}$$
$$u_i = \max_{k \in \{1,\ldots,K\}} u_i^k \tag{3.18}$$

for all $i \in \{1, \ldots, n\}$. These vectors L and U represent the final lower and upper approximations respectively. At this point, all slave nodes terminate while the master node is saving these two vectors to disk and then terminates.

Pseudocode for the master and slave programs is presented in Algorithm 6 and 7. The execution of the outer loop, i.e. lines 26-36 in Algorithm 7, can be optimized by running multiple iterations in parallel by creating a separate thread for each iteration of the loop. The number of concurrent iterations is determined by a parameter passed to the program.

Algorithm 6 Master Program

1: INPUT: $pathToClassvector$, m, n, K
2: DECLARE: $ranges, min, max, l, u$
3: $classvector \leftarrow loadFromFile(pathToClassvector)$
4: $broadcast(classvector)$
5: **ReduceFromSlaves**$(min^1, \ldots, min^K, max^1, \ldots, max^K)$
6: **for** t from 1 to m **do**
7: $max_t^f \leftarrow max_{k\in\{1,\ldots,K\}} max_t^k$
8: $min_t^f \leftarrow min_{k\in\{1,\ldots,K\}} min_t^k$
9: **end for**
10: **EndReduce**
11: **for** t from 1 to m **do**
12: $range_t \leftarrow max_t^f - min_t^f$
13: **end for**
14: $broadcast(range)$
15: wait for slaves to finish
16: **ReduceFromSlaves**$(l^1, l^2, \ldots, l^K, u^1, u^2, \ldots, u^K)$
17: **for** j from 1 to n **do**
18: $l_j \leftarrow min_{k\in\{1,\ldots,K\}} l_j^k$
19: $u_j \leftarrow max_{k\in\{1,\ldots,K\}} u_j^k$
20: **end for**
21: **EndReduce**
22: save L and U to disk
23: Terminate

Algorithm 7 Slave Program

INPUT: $dataset$, m, n, K
$partitionSize \leftarrow 1 + (n/K)$
DECLARE: min^k, max^k, l^k, u^k
$initialize(min^k, +\infty)$
$initialize(max^k, -\infty)$
$initialize(l^k, 1)$
$initialize(u^k, 0)$
$startRow \leftarrow (k-1) \cdot \text{partitionSize}$
$endRow \leftarrow startRow + partitionSize$
receive broadcasted $classvector$
for i from $startRow$ to $endRow$ **do**
    $q_{i,.} \leftarrow getRowFromDataSet(dataset, i)$
    for t from 1 to m **do**
        $max_t^k \leftarrow max(max_t^k, q_{i,t})$
        $min_t^k \leftarrow min(min_t^k, q_{i,t})$
    end for
end for
reduce max^k and max^k to the master node
receive broadcasted $ranges$
for i from $startRow$ to $endRow$ **do**
    $q_{i,.} \leftarrow getRowFromDataSet(dataset, i)$
    for t from 1 to m **do**
        $q_{i,t} \leftarrow q_{i,t}/range_t$
    end for
end for
for i from 1 to n **do**
    $q_{j,.} \leftarrow getRowFromDataSet(dataset, i)$
    for j from 1 to m **do**
        $q_{j,t} \leftarrow q_{j,t}/range_t$
    end for
    for i from $startRow$ to $endRow$ **do**
        simValue $\leftarrow similarity(q_{j,.}, q_{i,.})$
        $l_j^k \leftarrow min(l_j^k, simValue \tilde{\rightarrow} c_i)$
        $u_j^k \leftarrow max(u_j^k, simValue \tilde{\wedge} c_i)$
    end for
end for
reduce u^k and l^k to the master node
Terminate

3.5 Experimental Results

3.5.1 Experimental setup

We tested the scalability of our distributed approach of computing the fuzzy rough set approximations on 13 synthetically generated data sets, with the number of instances (rows) varying from 10,000 to 10 million, and the number of attributes (columns) varying from 10 to 50. All attribute values are randomly generated numbers between 0 and 1000. The class vectors are generated in the same way, but with randomly generated values between 0 and 1. As an implicator and t-norm we used respectively the Kleene-Dienes implicator and the minimum t-norm (see Section 2.1). Table 3.1 provides a summary of the data set files.

File	Instances	Attributes
1	10000	10
2	100000	10
3	1000000	10
4	10000000	10
5	10000	50
6	100000	50
7	1000000	50
8	10000000	50
9	1000000	20
10	1000000	30
11	1000000	40

Table 3.1: Artificial data sets used in the experiment.

a) ***Hardware***

We ran the experiments on an Intel cluster of 8 slave nodes with a total of 384 cores and 496GB of RAM. The CPUs used in the cluster are 64-bit Intel(R) Xeon(R) CPUs. In addition, the cluster includes two additional nodes, one is used as a master node for the

Node	Cores/Node	Speed/Core (GHz)	Memory (GB)
MPI Master	16	2.67	16
Spark Master	8	2.27	16
Slave Nodes 1 to 8	48	2.70	62

Table 3.2: Hardware specifications of the cluster

MPI implementation and the other is used as a master node for the Spark implementation. Table 3.2 provides the specifications of each node.

b) *Software*

We compiled and ran the MPI code on Intel MPI 4.3. For the Spark code, we used Apache Spark 1.1.0, Hadoop 1.2 and Java 1.7. For the non-distributed R version, we used the 3.1 release of R and we ran the code on a single slave node in the cluster.

3.5.2 Results

We first used the implementation that comes with the "RoughSets" package in R [45], and as mentioned before we tested its limitations on one node. We could not find a distributed R version for fuzzy rough sets. This R implementation was only able to handle a data set with up to 30000 rows and 10 attributes likely because of inefficient memory usage, i.e. the fact that the algorithm keeps the entire indiscernibility matrix in memory at once. Table 3.3 compares the execution time of our distributed MPI and Spark implementations with the non-distributed implementation in R on a small data set with 10,000 rows. Execution times of the MPI and Spark implementations for data sets with 100,000, 1 million and 10 million rows are presented further in this section; no corresponding execution times for R are presented over these larger data sets because, as explained above, the R package ran out of memory.

We conducted four sets of experiments to observe factors that affect the scalability of our approach. These experiments focus on data set size (number of instances), number of

rows	attributes	Spark (8 nodes/46 threads)	MPI (8 nodes/46 threads)	R (1 node)
10000	10	10s	1s	105s
10000	50	15s	1s	365s

Table 3.3: Comparison of execution time (in sec) between our distributed Spark and MPI based algorithm and the non-distributed algorithms from the "RoughSets" package in R [45].

attributes, number of compute nodes, and number of threads per compute node.

The first set of experiments investigates the effect of an increase in the number of threads per compute node. Recall that our implementation is multithreaded for the main outer loop of the Map call of the Spark implementation and the outer loop of the slave MPI program, i.e. lines 4-18 in Algorithm 2 and lines 26-36 in Algorithm 7. Since each iteration of this loop can be executed in parallel and independently of the others, we expect to see the overall runtime decrease when more threads are used. Figure 3.5 validates that the execution time decreases almost linearly with an increase in the number of threads.

Figure 3.5 shows that a change in the number of attributes from 10 to 50 does not change the behaviour of the curve, but only shifts the values. When we focus on execution times with the number of threads approximately equal to the number of cores available per compute node (48 - a thread per core), we notice that the best performance is achieved for 46 threads (see Figure 3.6) where one of the 48 threads is assigned to the controller program (main method in Spark and main function in MPI) while another thread is assigned to system tasks. Hence, we fixed the number of threads to 46 for all other experiments. Moving from the MPI curves to the Spark curves, we notice that the curves are very much similar but an upward shift is introduced in the Spark curves.

The second set of experiments examines the effect of the number of available compute nodes, while keeping everything else constant. Figure 3.7 shows clearly that the overall execution time decreases as the computation is distributed over more nodes. Even though the magnitude of the decline becomes less as the number of compute nodes grows, Figures 3.5 and 3.7 both show that the computation of fuzzy lower and upper approximations achieve

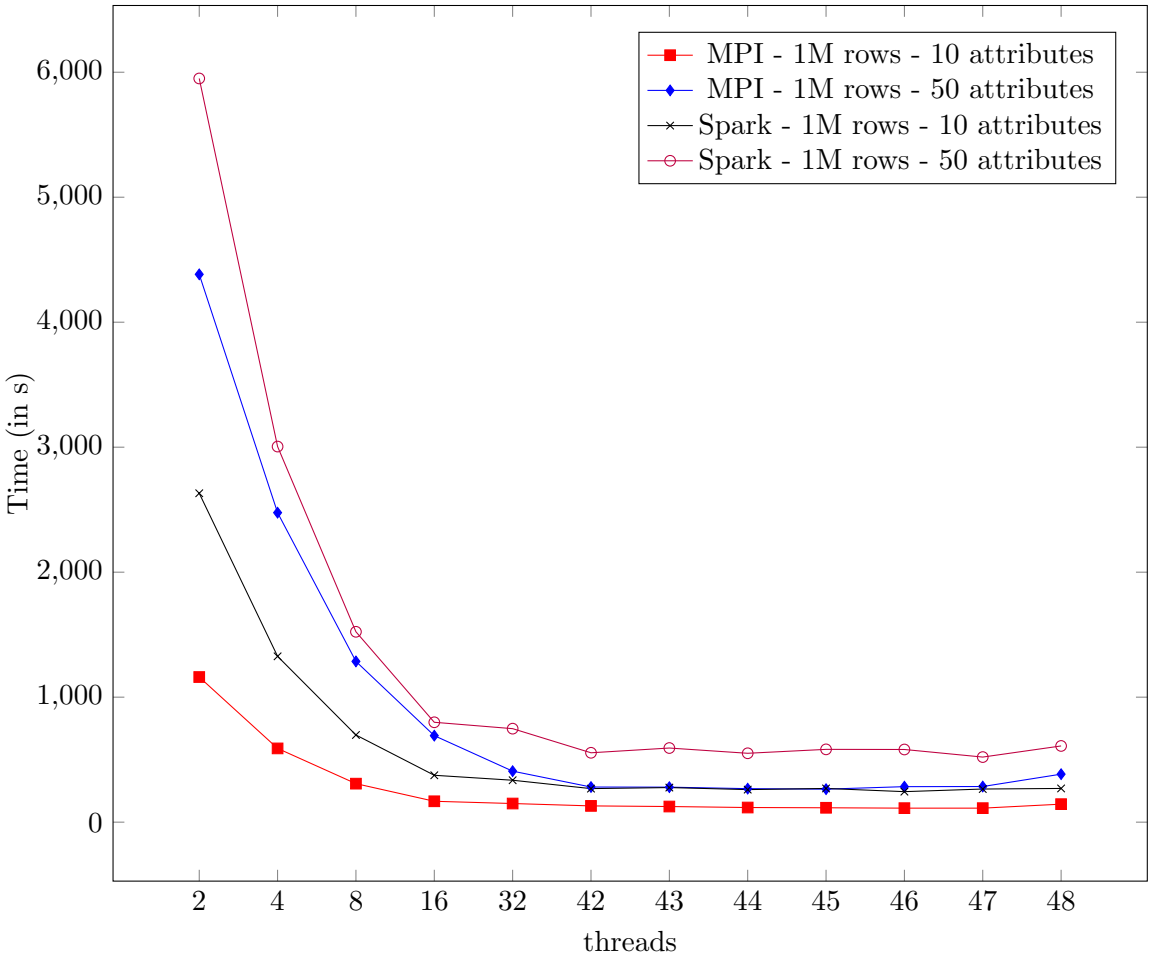

Figure 3.5: Execution time (in sec) for a varying number of threads per compute node. The experiments are performed on a cluster with 1 master node and 8 slave nodes, and on two different data sets (i.e. a data set with 1 million rows and 10 attributes, and a data set with 1 million rows and 50 attributes). There is a linear decrease in execution time of both the Spark program and the MPI program as the number of threads per node increases.

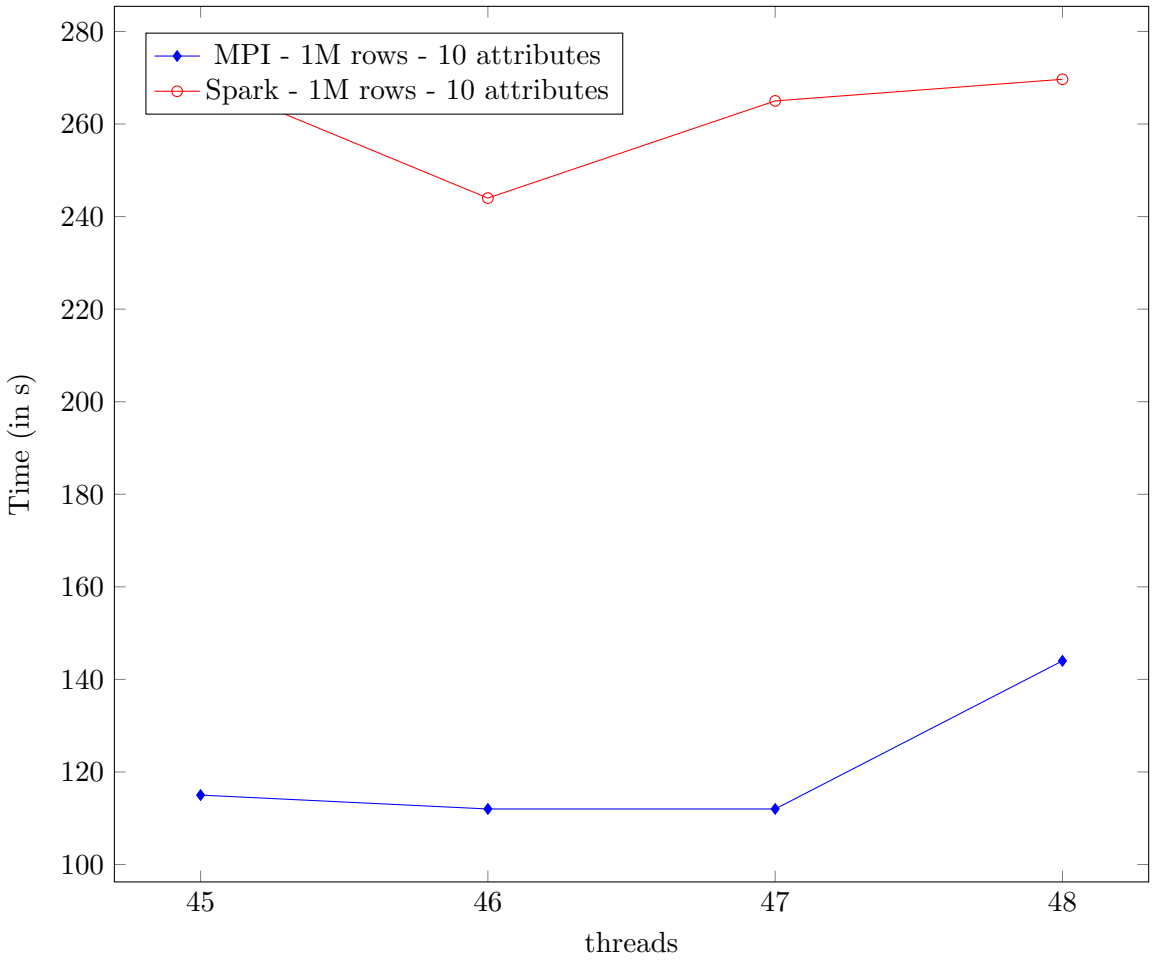

Figure 3.6: Execution time (in sec) for a varying number of threads per compute node, with the number of threads approximately equal to the number of available cores. The experiments are performed on a cluster with 1 master node and 8 slave nodes, and on a data set with 1 million rows and 10 attributes. The best performance is achieved with 46 threads.

strong scaling. The use of more hardware (nodes, threads) improves the runtime. Also note in Figure 3.7 that even with 1 slave node our algorithm is able to efficiently process a data set with 1 million rows, unlike the "RoughSets" package in R mentioned above [45] which would not allow us to get beyond 30,000 rows.

The final two sets of experiments examine the effect of the size of the data set on the execution time. Figure 3.8 shows that the execution time increases approximately quadratically with the number of rows. In addition, as expected, a higher number of attributes causes both the Spark and MPI implementations to run slower but it does not affect the general shape of the curve. Finally, Figure 3.9 shows that, as expected, the execution time of both implementations grows approximately linearly with the number of attributes.

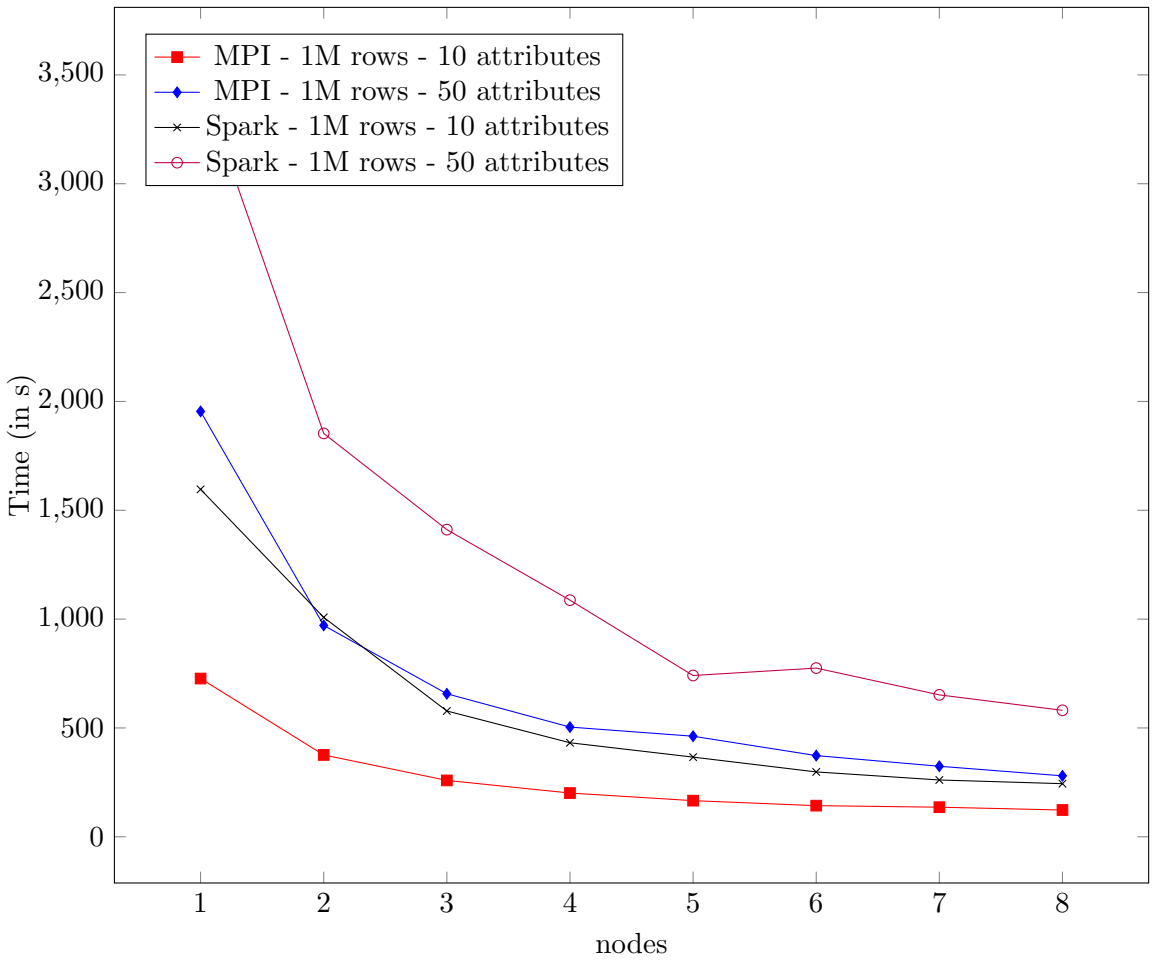

Figure 3.7: Execution time (in sec) for a varying number of nodes. The experiments are performed on a cluster with 1 master node and 1 to 8 slave nodes, and on two different data sets (i.e. a data set with 1 million rows and 10 attributes, and another data set with 1 million rows and 50 attributes). The number of threads is kept constant at 46. The runtime decreases with the increase in number of compute nodes. Even when only 1 slave node is available, our implementation efficiently computes lower and upper approximations of a concept vector with 1 million entries.

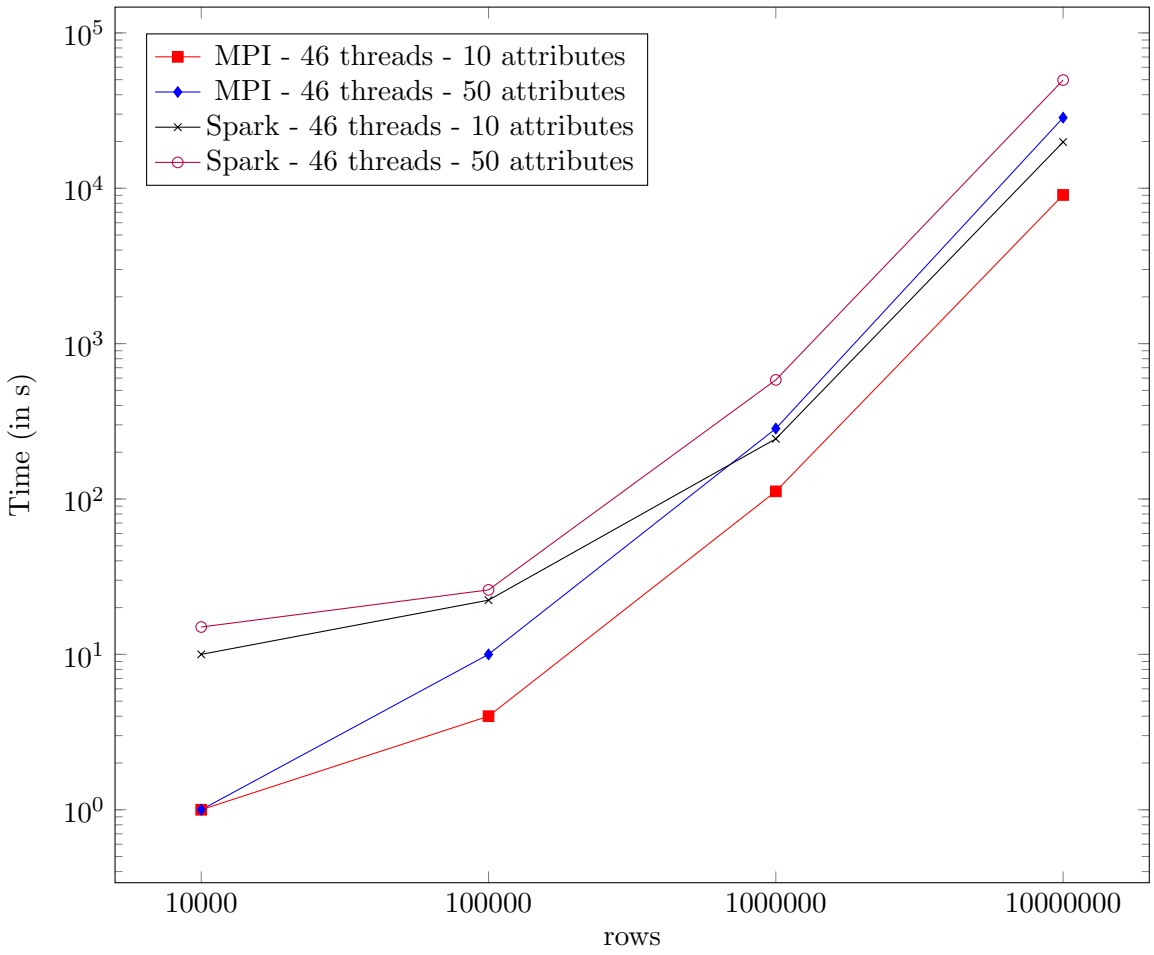

Figure 3.8: Execution time (in sec) for a varying number of rows in the data set. The experiments are performed on a cluster with 1 master node and 8 slave nodes, and on data sets with 10,000, 100,000, 1 million and 10 million instances and 10 or 50 attributes. The number of threads is kept constant at 46. The execution time of both the Spark program and the MPI program grows approximately quadratically in terms of the number of instances in the data set.

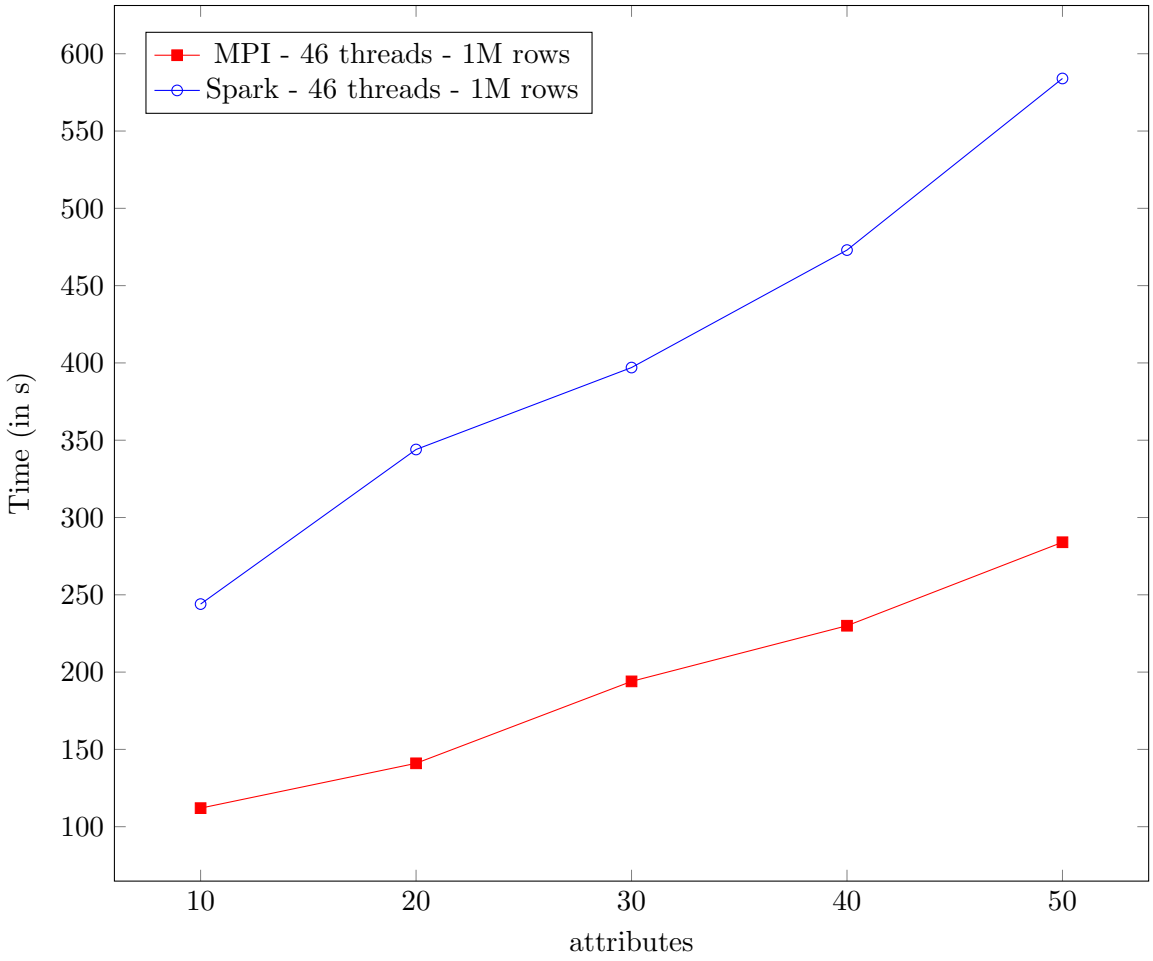

Figure 3.9: Execution time (in sec) for a varying number of attributes in the data set. The experiments are performed on a cluster with 1 master node and 8 slave nodes, and on data sets with 1 million rows and respectively 10, 20, 30, 40 and 50 attributes. The number of threads is kept constant at 46. The execution time of both the Spark program and the MPI program grows approximately linear in terms of the number of attributes in the data set.

3.6 Spark vs MPI

Spark and MPI are two frameworks that can be used to solve compute intensive problems. However, they have several differences that affect which one to choose to solve the problem at hand. The common implementations of MPI are written in C while those in Spark are written in the Java-based Scala programming language. Writing an MPI program means that the developer will have to deal with low level issues such as memory management and also might need to implement functionalities that come as out-of-the-box features in Java or Scala like string manipulation. Although such features can enable the developer to quickly build a solution to the problem at hand, they could reduce the amount of control over the behaviour of the code, which in turn lowers the amount of potential for low level optimizations. In addition, the fact that Java-based languages run on top of a virtual machine could cause the overall runtime of a solution to be lower than that of its MPI counterpart. In all of the experiments in this chapter, our MPI implementation performed faster. So, it is a trade-off between development friendliness and runtime performance.

Chapter 4

DISTRIBUTED WEIGHTED K NEAREST NEIGHBORS

In the previous chapter, we described fuzzy rough set theory and provided implementations for both fuzzy lower and upper approximations computations on Spark and MPI. Next, we will describe an application that uses some fuzzy rough set concepts. This application is a prototype selection algorithm that comes as a pre-processing step to the well-known k nearest neighbors algorithm (k-NN). So, we will first describe the k-NN related concepts in this chapter and then we will describe the prototype selection related concepts in the next chapter.

In this chapter, we specifically provide a distributed implementation of the weighted k nearest neighbors algorithm, an extension to the k nearest neighbors algorithm, on Apache Spark. We start this chapter by describing the k-NN algorithm, weighted k-NN and concepts related to them. Then, we describe some of the distributed k-NN implementations that have been done previously and how our implementation is different. After that, we present details of our distributed implementation of weighted k-NN on Spark. Finally, we provide experimental results for our implementation.

4.1 K Nearest Neighbors

In this section, we provide background information about concepts needed for the k-NN problem. These concepts include data classification, regression and 10-fold cross validation and the min-max heap data structure. After that, we describe the k-NN and the weighted k-NN problems.

4.1.1 Classification And Regression

In machine learning, classification [34] is the problem of predicting the category to which an instance of an unknown category belongs. The classification process involves training a

classification model using a training data set (e.g. historical data). In this problem, the set of possible categories is discrete. There are many classifiers that address this problem such as the Naive Bayes Classifier [16] and the Decission Tree Classifier [7]. In addition, the k-NN algorithm that we describe below can also be used to solve classification problems.

The decision system in Figure 2.4 can be considered as a training set that can be passed to a classifier in order to train a prediction model. The set of possible categories in this case is $\{Yes, No\}$ for the Flu decision attribute. If we have a patient P whose category is unknown, then we can use the prediction model to predict the category of that patient.

As opposed to classification, regression [34] is the process of approximating a real-valued function (a function that returns a real number). Unlike classification, regression is used to predict a number (also known as outcome) out of a range of real numbers instead of predicting a value out of a discrete set of values. Regression also involves the use of a training set to train a regression model and use it to predict a real-valued attribute. Linear Regression [34] is a very well known regression model that can be used to predict (or approximate) a real-valued attribute. In addition, the k-NN algorithm that we describe below can also be used to solve regression problems.

4.1.2 Cross-validation

Cross-validation is the process of measuring the accuracy of a prediction model. This process involves multiple iterations in which each receives a test set and a training set and applies the prediction model to predict the outcome of every instance in the test set based on the instances in the training set and finally computes the prediction average error rate. In each iteration, a new test set and a new training set are used. Both of these sets are usually subsets of one data set.

A common cross-validation technique is the k-fold cross-validation technique. In this method, the training set is split into k parts and then, k iterations (folds) are executed in which one part is used as a test set and the remaining parts are used as a training set. In each of these iterations, the error rate is computed and once all the k iterations are complete, the average of the error rates of each fold is the final measure of accuracy of the

prediction model. Figure 4.1 summarizes the k-fold cross validation process with k equal to 10.

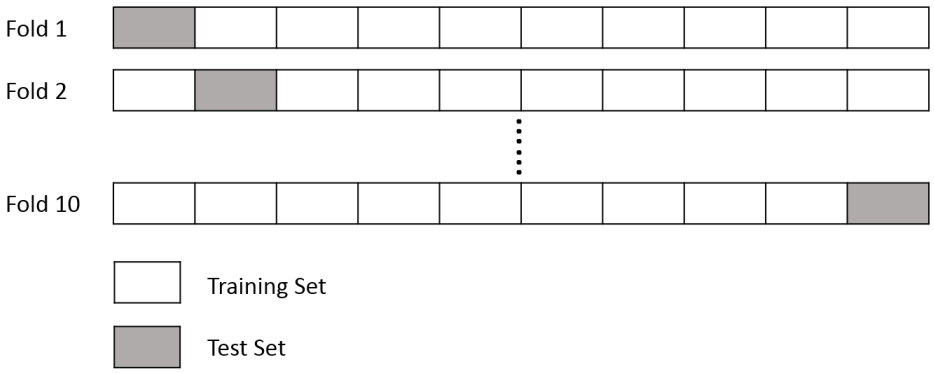

Figure 4.1: This figure summarizes the 10-fold cross validation process. The original data set is split into 10 parts. At each iteration (fold), one part is used as a test set and the rest are used as a training set.

The smaller the average error rate, the more accurate the prediction model is. A well known error metric is the Root-Mean-Squared-Error RMSE. For a test set with n instances, RMSE is computed by first taking the squared difference (error) of the actual and predicted values for a test instance and then the square root of the mean of the squared errors of all test instances is returned as the RMSE. These steps are shown in Equation 4.1. In cross-validation, RMSE is computed for every fold and then the average of the k RMSEs is returned as the accuracy measure for the prediction model.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(predicted_i - actual_i)^2}{n}} \tag{4.1}$$

In this thesis, we use both 10-fold and RMSE to measure the accuracy of our implementation of k-NN prediction.

4.1.3 Min-Max Heap

A min-max heap [14] is a complete binary tree [4] data structure whose levels are alternating between two types of levels: min levels and max levels. A node in a min level (even level)

has a value less than or equal to the values of its children while a node in a max level (odd level) has a value greater than or equal to the values of its children. The root is located at level zero and has the smallest value while the largest value is at a child node of the root. Figure 4.2 shows an example min-max heap with alternating levels. The minimum value 3 is at the root level while the maximum value 60 is a child of the root node 3.

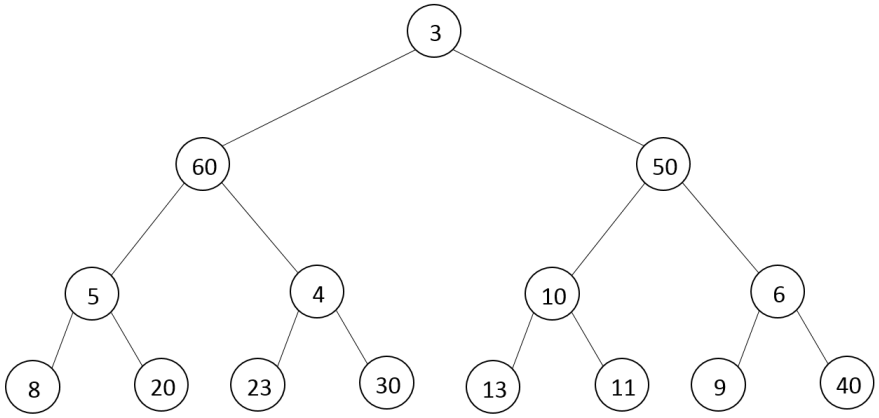

Figure 4.2: A min-max heap with the smallest value 3 at its root node and the largest value 60 at a child node of the root.

Finding the minimum or maximum values in a min-max heap take constant time while insertion and deletion operations are in $O(\log n)$.

Later in this chapter, we use a min-max heap in our k-NN distributed implementation to store the k nearest neighbors of an instance. Since we don't use the deletion operation, we will only focus on the insertion operation. To insert a new element in a min-max heap, the new element gets inserted at the first (left-most) available leaf position and then it propagates upward (bubbles up) until it reaches a correct position. The propagation depends on the type of level (max or min) at which the node is initially positioned. To clarify this, let us consider adding a node to the heap in Figure 4.2. Since the leaf nodes of this heap are at a max level, then the new node gets positioned at a min level. So, the new element becomes a child of node 8. To propagate this node, we do the following:

1. If the value of the new element is greater than the value of its parent node, then the

new node gets swapped with its parent. After that, it propagates upward by swapping it with its grandparent nodes (if any) until a larger grandparent node is encountered. For example, if the value of the new element is 100, then it gets swapped with its parent node 8 and then propagates by getting swapped with node 60. At this point, the propagation stops because node 60 has no grandparent node. Figure 4.3 shows how the value 100 is added to the heap in Figure 4.2.

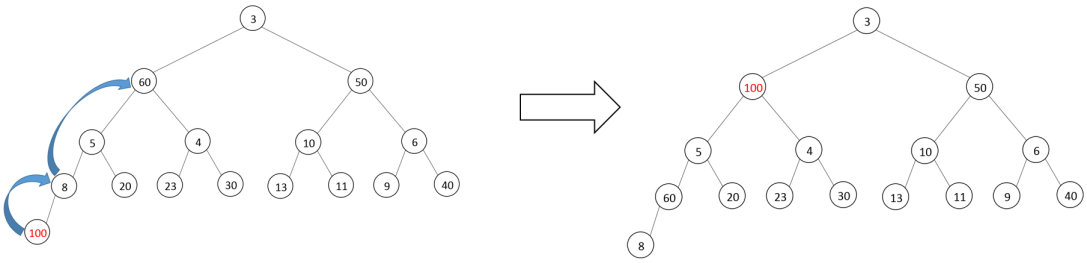

Figure 4.3: This example demonstrates adding a new value to a min-max heap with an even number of levels. Note that since the new value 100 is greater than its initial parent node 8, it gets swapped with it and then propagates by getting swapped with node 60.

2. If the value of the new element is smaller than the value of its parent node, then the new node does not get swapped with its parent. Rather, it propagates upward by getting swapped with its grandparent nodes (if any) until a smaller grandparent node is encountered. For example, if the value of the new element is 1 then it propagates by getting swapped with node 5 and node 3, the root node. Figure 4.4 shows how the value 1 is added to the heap in Figure 4.2. Note that because node 1 is the smallest, it becomes the new root node.

The above steps apply to a min-max heap with even number of levels (leaf nodes are at a max level. However, if the number of levels of the heap is odd then the process of swapping the new node with its parent is reversed. Specifically, swapping the new node with its direct parent node occurs only when the value of the new node is smaller than the value of its parent node. For example, if we want to add a new node with the value 3 to the

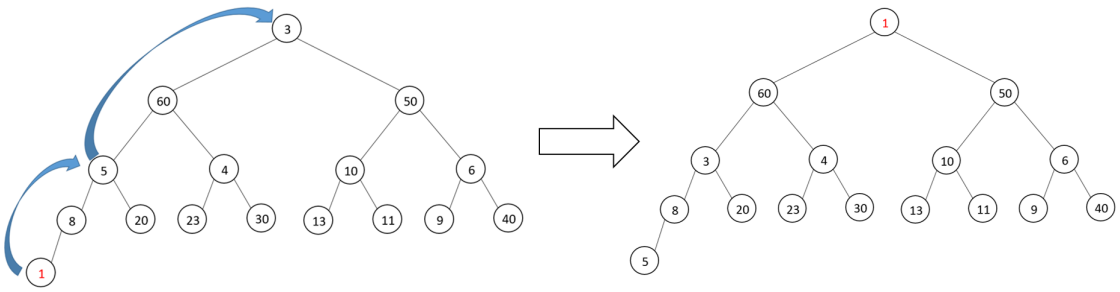

Figure 4.4: This example demonstrates adding a new value to a min-max heap with an even number of levels. Note that since the new value 1 is smaller than its initial parent node 8, it does not get swapped with its parent. Rather, it propagates by getting swapped with node 5 and then node 3.

heap in Figure 4.5 then this new element gets swapped with node 5 and then it propagates by getting swapped with node 4 as shown in Figure 4.6.

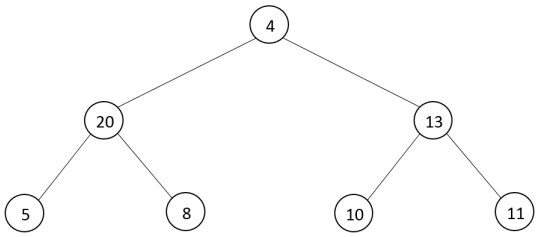

Figure 4.5: A min-max heap that has an odd number of levels with the smallest value 4 at its root node and the largest value 20 at a child node of the root.

If we add a node with the value 100 instead, then this node does not get swapped with 5 but rather propagates by getting swapped with node 20 as shown in Figure 4.7.

When inserting into a min-max heap with an even number of levels, the worst case insertion scenario occurs when inserting a new maximum node (with a value larger than any value in the heap) or a new minimum node (with a value smaller than any value in the heap) which triggers the highest number of upward swaps. The best case scenario happens when when no swaps are required. This occurs when the value of the new node is smaller than its parent and greater than the values of all nodes at min levels on the new node's path to the root. It also occurs when the value of the new node is greater than the value of

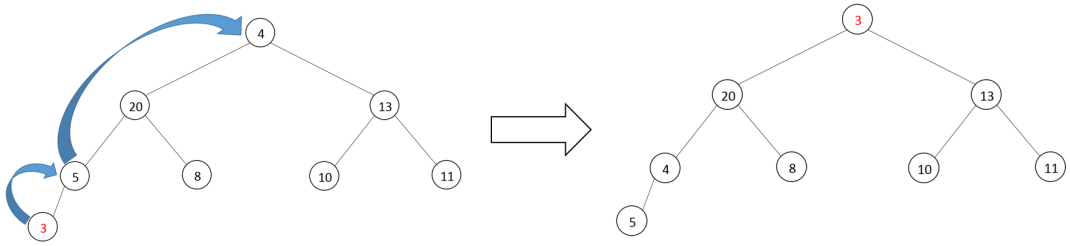

Figure 4.6: This example demonstrates adding a new value to a min-max heap with an odd number of levels. Note that since the new value 3 is smaller than its initial parent node 5, it gets swapped with it and then propagates by getting swapped with node 4.

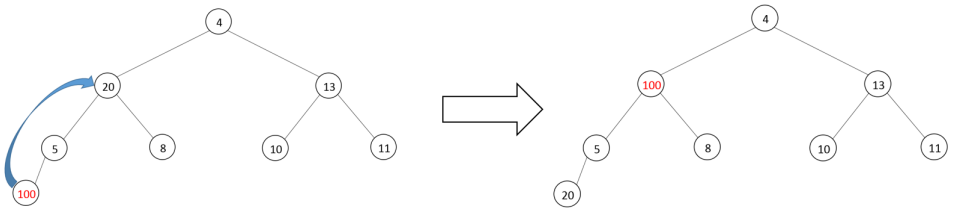

Figure 4.7: This example demonstrates adding a new value to a min-max heap with an odd number of levels. Note that since the new value 100 is greater than its initial parent node 5, it does not get swapped with its parent. Rather, it propagates by getting swapped with node 20.

its parent but smaller than the values of all nodes at max levels on the new node's path to the root.

4.1.4 The k-NN And Weighted k-NN Methods

The k-Nearest Neighbors algorithm relies on finding k closest points to a particular point in an m-dimensional space. A distance function such as the Euclidean distance and the Manhattan distance can be used to measure how far two points are from each other and consequently determine how close two points are to each other. An instance in a data set can be considered as a point in an m-dimensional space with m equal to the number of attributes in the information system. Computing k-NN of an instance q_i in a test set T with t instances and m attributes using a training set Q with n instances and m attributes involves computing the distance $d_{i,j}$ between q_i and every other instance q_j in Q where i

$\in \{1, \ldots, t\}$ and $j \in \{1, \ldots, n\}$. These distances form the distance matrix D. To compute the distance $d_{i,j}$, the distance is computed for each of the m attributes and then the average of these computed distances is returned as the final distance between two instances as shown in Equation 4.2. The summation $\sum_{y \in \{1,\ldots,m\}} f^{'}(q_{i,y}, q_{j,y})$ is also described in Algorithm 8.

$$f_{i,j} = \frac{1}{m} \sum_{y \in \{1,\ldots,m\}} f^{'}(q_{i,y}, q_{j,y}) \tag{4.2}$$

Depending on the attribute type, we use three distance functions. One for numeric attributes, one for string attributes and one for boolean attributes (with true or false values). We use the Manhattan distance in Equation 4.3 for numeric attributes as defined below

$$f^{'}(q_{i,y}, q_{j,y}) = |q_{i,y} - q_{j,y}| \quad y \in \{1, \ldots, m\} \tag{4.3}$$

For string attributes, we use Equation 4.4 which is defined as

$$f^{'}(q_{i,y}, q_{j,y}) = \begin{cases} 1 & \text{if } q_{i,y} = q_{j,y} \\ 0 & \text{Otherwise} \end{cases} \quad y \in \{1, \ldots, m\} \tag{4.4}$$

For boolean attributes, we convert every 64 boolean attributes into one binary string attribute which is a string that contains only zeros and ones. The ones represent the true values and the zeros represent the false values. For example if we have four boolean attributes with values {true, true, false, true} then we can replace them with one binary string attribute equal to 1101. The reason for this conversion is to reduce the number of boolean attributes into a fewer number of binary string attributes which makes them faster to process. So, we can replace 1000 boolean attributes with just 16 binary string attributes. Each of the first 15 binary string attributes represents 64 boolean attributes while the last binary string attribute represents the remaining 40 boolean attributes. After converting boolean attributes into binary string attributes, we compute the distance for a binary string attribute using a distance function that is based on Jaccard similarity [6]. Every binary string attribute can be viewed as a set. If we have four attributes A, B, C and D then the binary string attribute 1101 can be viewed as the set $\{A, B, D\}$. Our Jaccard

based distance function for two binary string attributes a and b is defined in Equation 4.5. Algorithm 9 provides the steps involved in Equation 4.5.

$$f^{'}(a,b) = 1 - \frac{|a \cap b|}{|a \cup b|} \tag{4.5}$$

The k-NN method has been used in both regression and classification [5] problems based on the idea that an instance of an unknown category should be assigned the class of its closest neighbors. In case the closest k neighbors to the instance belong to different categories, then an aggregation function is applied on all of these neighbors to predict the category of the instance. A typical aggregation function for classification problems would be to compute the number of occurrences of each category among the k nearest neighbors and then return the most frequent category (the majority vote). A typical aggregation function for regression problems would be to average the outcomes of the k nearest neighbors. An extension to the k-NN method is weighted k-NN [34] in which each neighbor is associated with a weight that is based on how close a neighbor is to the instance. The closer a neighbor to the instance, the more significant the weight is. Let q_i be the instance whose outcome value v_i we want to predict, d_{max} be the distance of the farthest neighbor to q_i and d_{min} is the distance of the closest neighbor to q_i. Then, we use Equation 4.6 to compute the weight w_h of a nearest neighbor to q_i that is d_h far from q_i with h in {1,. . . ,k}. As we focus on regression problems, we use weighted average in Equation 4.7 as the aggregation function to be applied on the k nearest neighbors to predict the value v_i for q_i.

$$w_h = \frac{d_{max} - d_h}{d_{max} - d_{min}} \tag{4.6}$$

$$v_i = \frac{\sum_{h=1}^{k} w_h \times v_h}{\sum_{h=1}^{k} w_h} \tag{4.7}$$

Algorithm 8 k-NN Program - Distance

1: INPUT: row1, row2
2: DECLARE: distance, bDistance, attr1, attr2, attrN1, attrS1, attrB1, attrN2, attrS2, attrB2
3: attrN1 ← row1.getNumericAttributes()
4: attrS1 ← row1.getStringAttributes()
5: attrB1 ← row1.getBinaryStringAttributes()
6: attrN2 ← row2.getNumericAttributes()
7: attrS2 ← row2.getStringAttributes()
8: attrB2 ← row2.getBinaryStringAttributes()
9: $distance \leftarrow 0$
10: **for** i from 1 to $row1.NumericAttrCount$ **do**
11: $\quad distance \leftarrow distance + |attrN1_i - attrN2_i|$
12: **end for**
13: **for** i from 1 to $row1.StringAttrCount$ **do**
14: $\quad$ **if** $attrS1[i] == attrS2[i]$ **then**
15: $\quad\quad distance \leftarrow distance + 1$
16: $\quad$ **end if**
17: **end for**
18: $bSim \leftarrow 0$
19: **for** i from 1 to $row1.BinaryAttrCount$ **do**
20: $\quad bSim \leftarrow bSim + JaccardSim(attrB1[i], attrB2[i])$
21: **end for**
22: $bDistance \leftarrow row1.BinaryAttrCount - bSim$
23: $distance \leftarrow distance + bDistance$
24: return $distance$

Algorithm 9 k-NN Program - JaccardSim

```
1: INPUT: set1, set2
2: DECLARE: intersection, union, jaccardSim
3: intersection ← set1  AND  set2
4: union ← set1  OR  set2
5: if union == 0 then
6:     return 1
7: end if
8: jaccardSim ← bitCount(intersection)/bitCount(union)
9: return jaccardSim
```

4.2 Related Work

In this section, we present different existing distributed implementations of k-NN and discuss how our implementation is different. The work in [44] has several components in common with our implementation. A k-NN graph is created for the data set through the use of a similarity matrix. However, the similarity matrix is approximate and thus the k-NN graph is also approximate while we will show that our implementation computes an exact similarity matrix and exact k-NN values. In addition, a block in the approximate similarity matrix in [44] is computed by a node and then its columns get sorted and finally the top k indices are returned. This means that memory will need to be allocated for the whole block. On the other hand, our implementation allocates memory for only one value in the similarity matrix because that value is computed and immediately consumed in a subsequent step so that there is no need to keep it stored. In a multi-threaded version of our implementation, memory will need to be allocated for a similarity value in every thread in a node which is still much less than allocating memory space for a similarity block.

The works in [18] and [23] are different from our work in the fact that a preprocessing step needs to be done on the data set. The data set needs to be indexed using a tree-based data structure and then that structure is used to compute k-NN. In our implementation, such preprocessing step is not needed because we perform a distributed full scan of the data

set. This implies that our approach fits best for a frequently changing training set.

In [35], a k-NN implementation on Apache Spark is provided. However it uses RDD cartesian product on the training set and test set to generate the distance matrix. The RDD cartesian product operation could overflow the memory for large data sets because it keeps a number of values in memory equal to ($training\ set\ size \times test\ set\ size$). In addition, it is slow because of the overhead of creating key-value pairs for each pair of instances in the test and training sets.

The work in [13] provides a Hadoop-based k-NN implementation that is close to our approach. It starts by applying a map stage which divides the test set S into $|S|/N$ blocks and the training set R into $|R|/N$ blocks and then groups every possible pair of S and R blocks into a bucket so that we end up with N^2 buckets. Then a reducer is executed for every bucket which applies a join operation on S and R blocks to compute the distances. In our approach, we only partition the training set while having the test set on HDFS as we will show later. Then in the map phase, each slave node accesses the test set to perform the distance computations. This way, we prevent the overhead accompanied by the join operation.

Our approach is very similar to the brute-force approaches described in [9]. All of these approaches, except [28], apply sorting to the maintained k-NN list. However, our approach and [28] use a fixed-size heap data structure to maintain a k-NN list for every instance in the test set. Our approach differs in the fact that it is implemented on a shared-nothing big data architecture while [28] is implemented on a shared-memory multi-GPUs architecture.

4.3 Distributed Implementation

In this section, we present our distributed implementation of the weighted k-NN for regression algorithm on Spark. The input to the program are two HDFS files which represent a training set and a test set. The test set contains the instances for which we want to find k nearest neighbors and compute predictions. The training set contains instances from which we select the k nearest neighbors.

The general idea is that the training set gets distributed over e multiple nodes such that each node is assigned a partition of the training set. Each node then computes k nearest

neighbors to every instance in the test set using the training set partition assigned to it. This way, $e \times k$ nearest neighbors are computed for every instance across all the nodes. Next, all of the $e \times k$ nearest neighbors that are computed for a test instance are grouped together, sorted and the top k nearest neighbors are returned as the final k-NN to that test instance. The same happens for the rest of the test instances. Recall that the computation of k-NN involves computing a distance matrix D. In this case, each node computes a subset of the columns of the distance matrix D without storing all of D at a time. This is similar to the approach that we used in the previous chapter to compute a similarity matrix.

Figure 4.8 and Algorithm 10 show the flow of this distributed implementation. The implementation consists of three stages which are the Map Stage 1, Grouping Stage and the Map Stage 2 (lines 5, 6 and 7 in Algorithm 10). First, an RDD is created for the training set to partition it into e partitions, distribute it over e compute nodes and keep it in memory. Then, the Map Stage 1 starts, which computes k-NN for every instance in the test set from the local partition of the training set at each node. Since each node has a partition of the training set at its disposal, it finds k nearest neighbors to a target within that partition. And since we have e compute nodes, the number of nearest neighbors computed for a test instance is equal to $e \times k$ nearest neighbors. The Grouping Stage combines all of the $e \times k$ nearest neighbors of a test instance, sorts them and finally returns the top k as the final k nearest neighbors of the test instance. In the Map Stage 2, the final stage, the weighted average of the k nearest neighbors is computed for each test instance and then these weighted averages are stored back to HDFS as the final result. In the rest of this section, we will describe each of the three stages in more details. Before these three stages start, each element in the RDD is converted into a $KnnRow$ object. A $KnnRow$ object represents a row that can have numeric, string and boolean attributes and also has an initially empty list of k nearest neighbors. We use a min-max heap that always maintains the k smallest elements of all elements that are added to it. The $KnnRow$ object can also apply a weighted average aggregation function on its list of nearest neighbors.

a) ***Map Stage 1***

The objective of this Map stage is to find k nearest neighbors in each training set partition

Algorithm 10 k-NN Program

1: INPUT: trainingSetPath,testSetPath

2: DECLARE: $MaxAccumulator$, $MinAccumulator$, ranges, min, max, m

3: rdd[String] ← create RDD From HDFS(trainingSetPath)

4: rdd[KnnRow] ← transform(rdd)

5: KnnRDD[Integer,KnnRow] ← map(rdd)

6: finalKnnRDD[Integer,KnnRow] ← groupByKey(KnnRDD)

7: predictedValueRDD[Integer,PredictedValue] ← map(finalKnnRDD)

8: saveToHDFS(predictedValueRDD)

9: Terminate

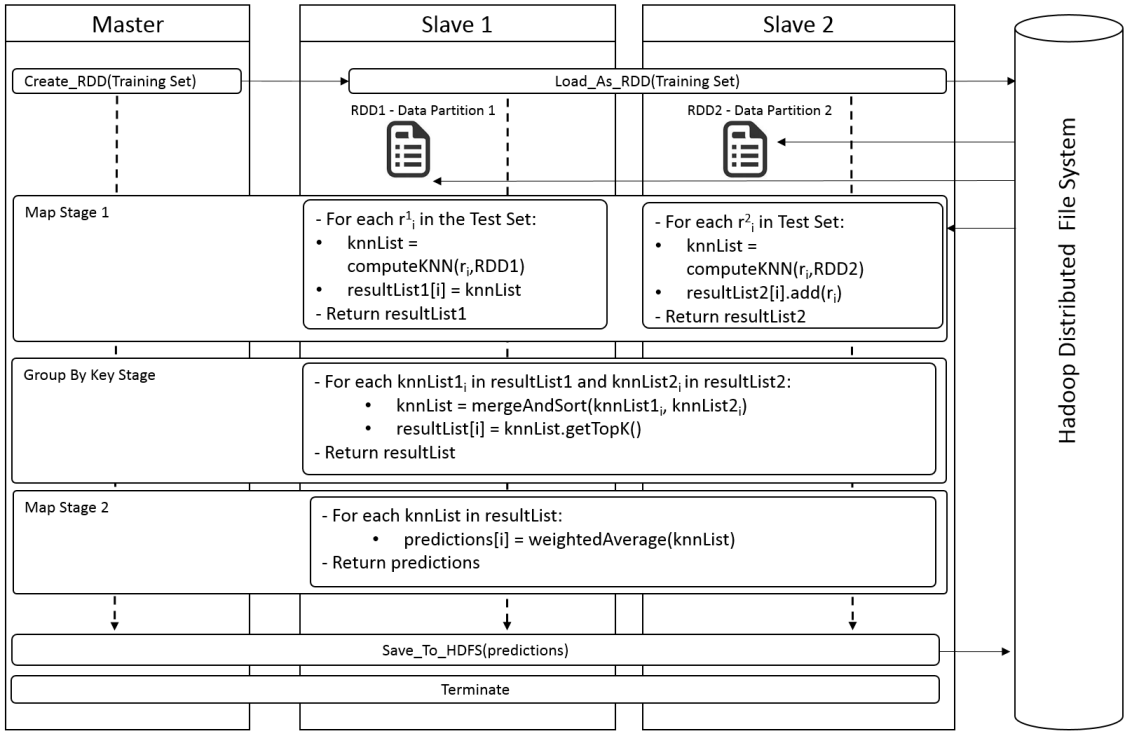

Figure 4.8: This figure summarizes the overall flow of the weighted k-NN implementation on Spark. The approach has three stages, namely Map Stage 1, Grouping Stage and Map Stage 2. This figure shows how the weighted k-NN is computed using two slave nodes. In reality, however, there can be more slave nodes.

to each test instance in the test set. This stage is summarized in Algorithm 11.

Algorithm 11 k-NN Program - Map Stage 1

```
1: INPUT: RddPartition,testSetPath
2: DECLARE: rowsList
3: P ← RddPartition.size
4: for line_i ← readFromHDFS(testSetPath) do
5:     t_i ← create KnnRow(line_i)
6:     t_i.knnList ← create MinMaxHeap
7:     for j from 1 to P do
8:         t_j ← RddPartition.getRow(j)
9:         distance ← distance(t_i,t_j)
10:        neighbor ← create NearestNeighbor(j,distance)
11:        t_i.knnList.add(neighbour)
12:    end for
13:    rowsList.add(t_i)
14: end for
15: return rowsList
```

The input to this stage is the test set (of size T) and the training set RDD partitions (each of size $\lceil \frac{n}{e} \rceil$). It begins by reading the test set from HDFS and then iterates over each test instance t_i and performs the following (this loop is refered to as the *outerloop*):

1. A KnnRow object is created for t_i.

2. The k-NN of t_i in the test set get retrieved by iterating over every training instance t_j in the RDD partition and performing the following steps (this loop is refered to as the *innerloop*):

   (a) Fetch the KnnRow object for t_j.

   (b) Compute the distance between t_i and t_j as shown in Algorithms 8 and 9.

(c) Create a NearestNeighbor object which links between the distance computed in the previous step and the training instance t_j. This object represents the distance between the test instance t_i and the training instance t_j.

(d) The created NearestNeighbor object gets added to the min-max heap of t_i. Note that the min-max heap determines the location of the NearestNeighbor object within the heap based on the computed distance.

3. Add t_i to the result list. Note that t_i represents a test instance along with all of its k nearest neighbors.

After completing the above steps, each compute node returns its result list which has the test instance KnnRow objects. Each of these objects encapsulates k nearest neighbors. In addition, each compute node returns k nearest neighbors for every test instance t_i and this makes the total number of returned nearest neighbors to be $e \times k$ with e is the number of compute nodes.

To maximize hardware utilization, we have added a multi-threading optimization to Algorithm 11 by modifying line 4 to read multiple lines (equal to the number of available threads) at a time from the test data set in HDFS and then equally distribute these lines over the threads so that each line is completely and independently processed by one single thread. This means each thread finds the k nearest neighbors from the node's training set partition to one test instance in the data set. At some points, there could be more threads than test instances. For example, we might want to find k-NN for only two test instances while we have four threads. In this case, we further partition the node's training set partition into sub-partitions equal to the number of available threads and then we assign one training set sub-partition to each thread. After that, we iterate over the test instances one by one so that for each test instance t_i, each thread finds k nearest neighbors from the thread's assigned sub-partition to t_i. Finally, all of the k-NN elements found by the threads are combined into one list and then the top k nearest neighbors in this list are returned as the k-NN to t_i in the node's training set partition.

b) ***Grouping Stage***

The objective of this stage is to find the top k nearest neighbors out of the $e \times k$ nearest neighbors found in the previous stage for every test instance t_i. We use the groupByKey Spark function instead of reduceByKey because we are grouping objects (k-NN lists) rather than aggregating primitive values. Algorithm 12 shows the steps involved in this stage. All of the $e \times k$ nearest neighbors for t_i are merged together, sorted and then the top k nearest neighbors are returned.

Algorithm 12 k-NN Program - Group By Key Stage

```
1: INPUT: knnList1,knnList2
2: DECLARE: finalKnnList
3: finalKnnList ← merge(knnList1,knnList2)
4: sort(finalKnnList)
5: return top k from finalKnnList
```

c) ***Map Stage 2***

The objective of this stage is to predict the value of the response attribute of every test instance t_i by aggregating the k nearest neighbors found in the previous stage. The aggregation function applied here is the one defined in Equations 4.6 and 4.7. Algorithm 13 provides the steps involved in this stage. First, the distance range is computed using the minimum and maximum distances of the k nearest neighbors of a test instance t_i. Then the weighted average is computed as shown in lines 10 through 16. Note that when the range is equal to 0, every neighbor is assigned a weight equal to 1.

Algorithm 13 k-NN Program - Map Stage 2

1: INPUT: test instances t_i
2: DECLARE: $predictedValue$, $knnList$, totalWeight, d_{max}, d_{min}, $distanceRange$,sum
3: $knnList \leftarrow t_i$.knnList
4: $d_{max} \leftarrow$ knnList.maxDistance
5: $d_{min} \leftarrow$ knnList.minDistance
6: $distanceRange \leftarrow d_{max} - d_{min}$
7: $totalWeight \leftarrow 0$
8: **if** $distanceRange == 0$ **then**
9: $predictedValue \leftarrow$ average(knnList)
10: **else**
11: $sum \leftarrow 0$
12: **for** j from 1 to k **do**
13: $c_j \leftarrow$ knnList[j].responseAttribute
14: $d_j \leftarrow$ knnList[j].distance
15: $w_j \leftarrow (d_{max} - d_j)/distanceRange$
16: $totalWeight \leftarrow totalWeight + w_j$
17: $sum \leftarrow$ sum $+ w_j \times c_j$
18: **end for**
19: predictedValue $\leftarrow sum/totalWeight$
20: **end if**
21: return predictedValue

4.4 Experimental Results

We tested the scalability of our distributed implementation of k-NN on the data sets in Table 3.1 and using the Intel cluster nodes described in Table 3.2. We conducted five sets of experiments to observe factors that affect the scalability of our implementation. These experiments focus on the data set size (number of instances), the number of attributes, the number of compute nodes, the number of threads per compute node and the number of nearest neighbors (k). The first set of experiments investigates the effect of the increase in the number of threads per compute node. Recall that the first map stage (Algorithm 11) is multithreaded such that each test instance has a dedicated thread to compute its k-NN. This way, we expect to see the overall runtime decrease when more threads are used. Figure 4.9 validates that the execution time decreases almost linearly with an increase in the number of threads.

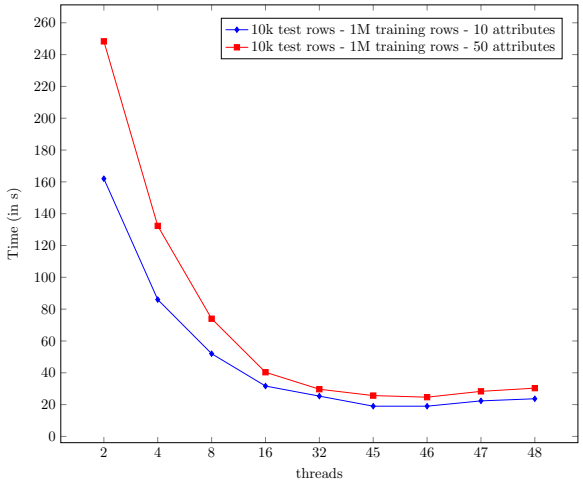

Figure 4.9: Execution time (in sec) for our k-NN implementation with a varying number of threads per compute node, with the number of threads approximately equal to the number of available cores. The experiments are performed on a cluster with 1 master node and 8 slave nodes and on two different training sets (i.e. a training set with 1 million rows and 10 attributes, and another training set with 1 million rows and 50 attributes) as well as a test set with 10000 rows. The best performance is achieved with 46 threads. See Figure 4.10 for a more detailed view.

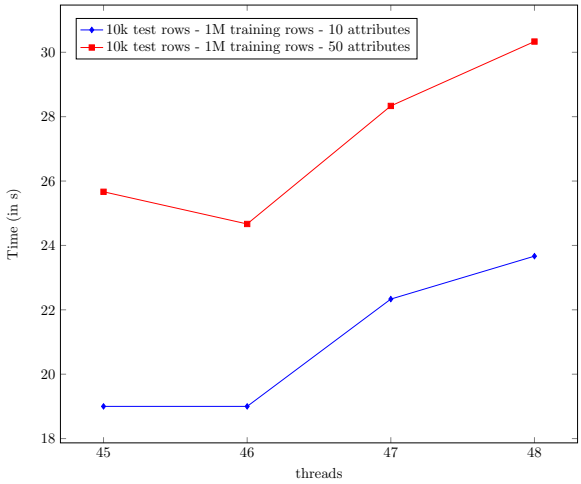

Figure 4.10: This is a zoomed-in version of Figure 4.9 showing that the best performance is achieved with 46 threads.

The second set of experiments examines the effect of changing the number of available compute nodes, while keeping everything else constant. Figure 4.11 shows that the overall execution time decreases as the computation is distributed over more nodes.

The third and fourth sets of the experiments examine the effect of changing the size of the data set on the execution time. Figure 4.12 shows that the execution time increases approximately quadratically with the increase in the number of rows which is an expected behavior of the two loops in the Map Stage 1 (lines 4 and 7 in Algorithm 11). In addition, the increase in the number of attributes has a linear runtime effect as shown in Figure 4.13. This behavior is also expected because computing the distance requires one pass over all the attributes (see Algorithm 8).

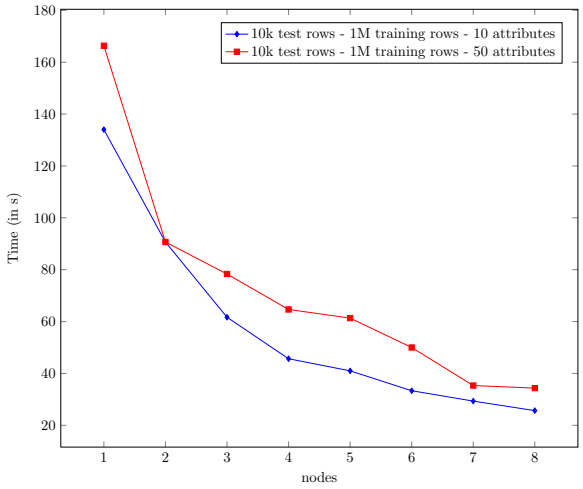

Figure 4.11: Execution time (in sec) for our k-NN implementation with a varying number of nodes. The experiments are performed on a cluster with 1 master node and 1 to 8 slave nodes, and on two different training sets (i.e. a training set with 1 million rows and 10 attributes, and another training set with 1 million rows and 50 attributes) as well as a test set with 10000 rows. The number of threads is kept constant at 46. The runtime generally decreases with the increase in the number of compute nodes.

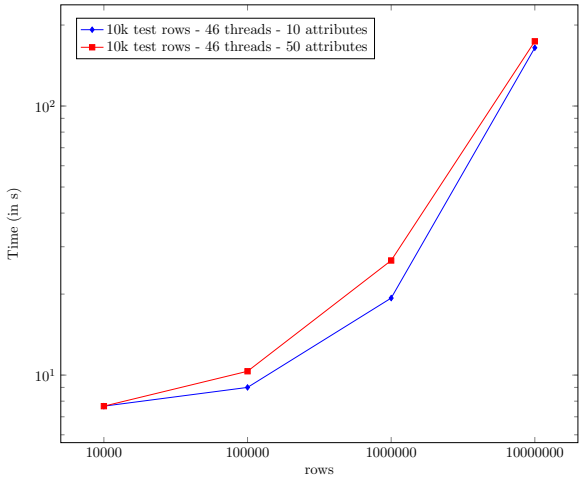

Figure 4.12: Execution time (in sec) for our k-NN implementation with a varying number of rows in the data set. The experiments are performed on a cluster with 1 master node and 8 slave nodes, and on training sets with 10,000, 100,000, 1 million and 10 million instances and 10 and 50 attributes in addition to a test set with 10000 rows. The number of threads is kept constant at 46. The execution time grows approximately quadratically in terms of the number of instances in the data set.

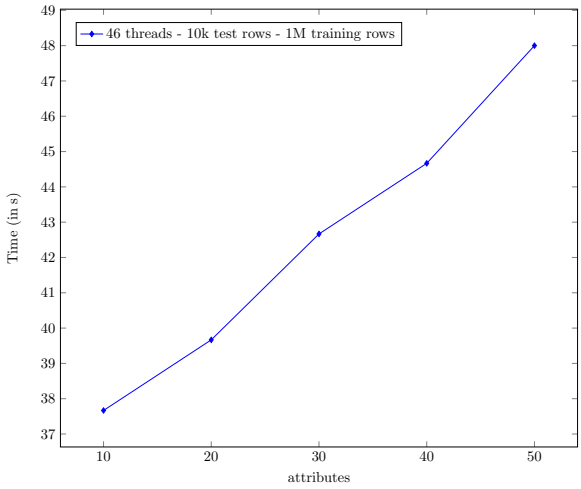

Figure 4.13: Execution time (in sec) for a varying number of attributes in the data set. The experiments are performed on a cluster with 1 master node and 8 slave nodes, and on training sets with 1 million rows and respectively 10, 20, 30, 40 and 50 attributes in addition to a test set with 10000 rows. The number of threads is kept constant at 46. The execution time grows approximately linearly in terms of the number of attributes in the data set.

Finally, we examined the effect of changing the value of the number of nearest neighbors k. We decided to choose small values of k as it is usually set to small values in practice. As shown in Figure 4.14, a variation between small values of k (up to 23) has only a small effect on the overall runtime. It basically kept the runtime fluctuating within a range of three seconds. The fluctuation occurs because of the runtime behavior of the min-max heap used to store the k nearest neighbors. Recall from section 4.1.3 that the run-time complexity of insertion into a min-max heap relies on the comparison criteria of its elements which is in our case the distance of neighbors. Since the distance depends on the values of the attributes in the data set, then these values affect the time it takes to insert into the min-max heap. So, a neighbor to be inserted into the min-max heap might form a best case, worst case or something in between depending on the distance values associated with it. To verify that the fluctuation is affected by the values of the attributes in the data set, we repeated the same experiment on similar data sets (with the same sizes) but with different values as shown in Figure 4.14. It is clear from the figure that the first data sets (in blue) follow different fluctuation patterns from the second data sets (in red). So, depending on the

attribute values in the data set, the computed distances will be different (the comparison criteria when inserting into the heap). This results in a heap insertion runtime cost that fluctuates within the best and worst case runtime range.

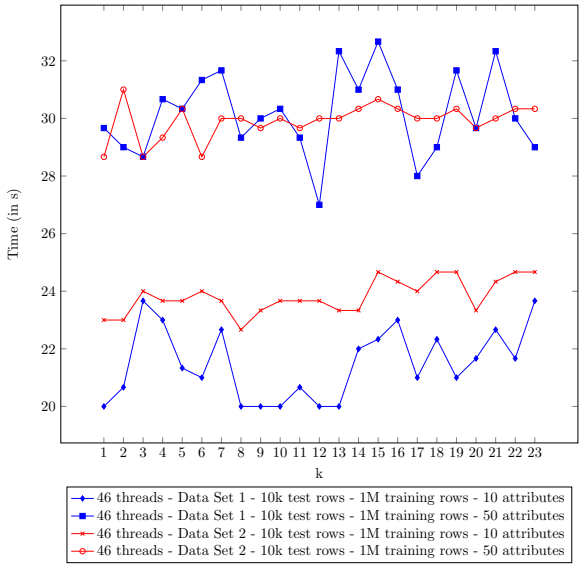

Figure 4.14: Execution time (in sec) for changing the value of k while keeping everything else constant. The experiments are performed on a cluster with 1 master node and 8 slave nodes, and on four training sets each with 1 million instances and two of them with 10 attributes while the other two have 50 attributes. A test set of 10000 rows is also used. The values in all four data sets are different. The number of threads is kept constant at 46. The values of k range between 1 and 23. Varying the size of k (from 1 to 23) does not impact the average running time of the implementation of the algorithm. The fluctuations that are visible when changing the value of k are very minor and appear to go hand in hand with the specific values in the data set.

In this chapter, we provided a distributed implementation of the k-NN algorithm on Apache Spark. We also showed that this implementation scales with the change in data set sizes. In the next chapter, we develop prototype selection approaches and observe their effect on the accuracy of this k-NN implementation.

Chapter 5

DISTRIBUTED FUZZY ROUGH PROTOTYPE SELECTION

In the previous chapter, we provided a description of the weighted k-NN algorithm and concepts related to it. We also provided a Spark based distributed implementation of the weighted k-NN algorithm. In this chapter, we talk about prototype selection (PS) and also provide two distributed approaches for applying prototype selection on a large data set.

Prototype selection is the process of selecting a subset of instances in a training set that best represents the training set. This subset, also called the prototype set, is then used to perform k-NN prediction. The objective of PS is to improve the prediction accuracy by removing the noisy instances from the training set. This reduces the training set size and thus improves the prediction run-time efficiency.

In this chapter, we present two distributed prototype selection approaches that use fuzzy rough set theory. These two approaches are composed of two steps. In the first step, a quality score is generated for every instance in the training set. In the second step, a subset of the training set is selected as the prototype set based on the quality scores generated in the first step. Both prototype selection approaches have the second step in common but differ in the first step.

This chapter is divided into five sections. In the first section, we discuss how the two prototype selection approaches are different from similar fuzzy rough set based prototype selection techniques that have been previously done. In the second and third sections, we describe the first step in the two prototype selection approaches while we describe the second step in the fourth section. In the fifth section, we provide scalability and accuracy experimental results.

5.1 Related Work

In [48], a fuzzy rough set prototype selection (FRPS) method is proposed that was shown to perform well for k-nearest neighbor classification. The method was evaluated in combination with 1NN and 3NN on 58 data sets from the KEEL repository and compared with 22 state-of-the-art prototype selection algorithms. FRPS came out as the best method of all w.r.t. prediction accuracy. FRPS has however only been developed and tested on small to medium sized data sets with less than 20,000 instances and less than 100 attributes. In addition, it is only intended for classification problems, not for regression problems. In this thesis, we propose a distributed fuzzy rough prototype selection approach (DFRPS), an extension of FRPS that can be used together with weighted k-NN to solve regression problems. We also evaluate its performance – both in terms of prediction accuracy and in terms of scalability – on very large information systems with millions of instances.

5.2 High-Mid-Low Approach

The general idea in this approach is based on the fact that every instance belongs to a fuzzy set to a certain degree of membership. We define three sets (buckets) for the number that we want to predict. These buckets are the high, medium and low (HML) buckets. We represent these three buckets with three class vectors and then we compute the lower approximation for each of these class vectors. Finally we aggregate the three resulting lower approximation vectors into one final vector that contains the quality score of each instance in the data set. We refer to this vector as the HML quality vector. A quality score threshold can be set to select the prototypes such that instances with quality value greater than or equal to the threshold would belong to the prototype set and instances with quality value below the threshold would be excluded from the prototype set. In the next subsection, we will describe the details of how this quality measure can be computed for an instance. In the second subsection, we provide details of a distributed implementation for computing the HML quality values.

5.2.1 Instance Quality Measure

Let d be the numeric decision value of a data set, d_{min} be the lowest decision value in the data set, d_{max} be the highest decision value in the data set, $d_{range} = d_{max} - d_{min}$ be the range of the decision values, $d_{25} = 0.25 \times d_{range}$ be 25% of d_{range}, $d_{50} = 0.50 \times d_{range}$ be 50% of d_{range} and $d_{75} = 0.75 \times d_{range}$ be 75% of d_{range}. The first step involved in computing the HML quality values is to generate the degree of membership of every instance to each of the three HML buckets. To do that, we use Figure 5.1 to derive the membership functions.

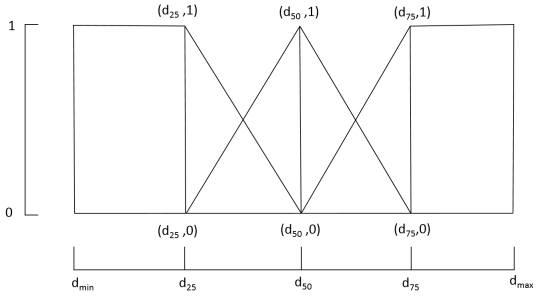

Figure 5.1: A figure used to derive the high, medium and low buckets.

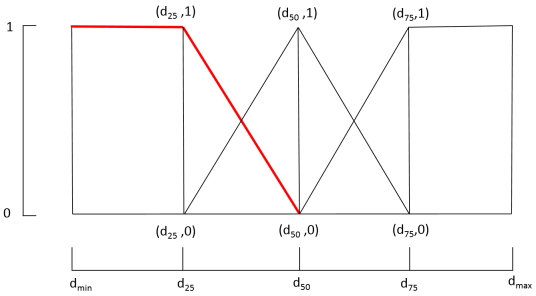

Figure 5.2: The low bucket values lie along the red line.

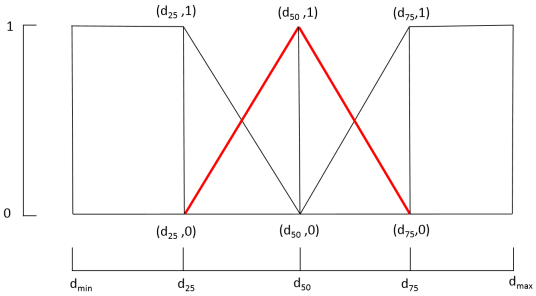

Figure 5.3: The medium bucket values lie along the red line.

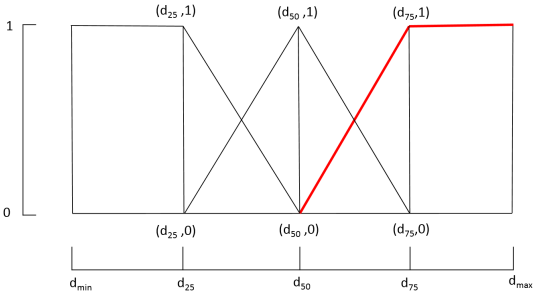

Figure 5.4: The high bucket values lie along the red line.

This figure basically defines the values of the three HML buckets with respect to the range d_{range} of the decision values. The x-axis represents the decision values while the y-axis represents the membership value of a bucket. The low bucket is represented by the emphasized line in Figure 5.2 which corresponds to Equation 5.1. For example, an instance with a decision value below d_{25} will have a degree of membership equal to 1. If

the decision value is more than d_{25} then the degree of membership to the low bucket will start to decrement along the emphasized line in Figure 5.2 until it becomes 0 at d_{50}. In a similar way, Equation 5.2 which corresponds to the emphasized line in Figure 5.3 defines the membership function of the medium bucket and Equation 5.3 which corresponds to the emphasized line in Figure 5.4 defines the membership function for the high bucket.

$$low(d) = \begin{cases} 1 & \text{if } d \leq d_{25} \\ \frac{d-d_{50}}{d_{25}-d_{50}} & \text{if } d_{25} < d < d_{50} \\ 0 & \text{if } d \geq d_{50} \end{cases} \tag{5.1}$$

$$mid(d) = \begin{cases} 0 & \text{if } d \leq d_{25} \quad or \quad d \geq d_{75} \\ \frac{d-d_{25}}{d_{50}-d_{25}} & \text{if } d_{25} < d \leq d_{50} \\ \frac{d-d_{75}}{d_{50}-d_{75}} & \text{if } d_{50} < d < d_{75} \end{cases} \tag{5.2}$$

$$high(d) = \begin{cases} 1 & \text{if } d \geq d_{75} \\ \frac{d-d_{50}}{d_{75}-d_{50}} & \text{if } d_{50} < d < d_{75} \\ 0 & \text{if } d \leq d_{50} \end{cases} \tag{5.3}$$

By applying these membership functions on every instance in the data set, we get three class vectors, one for each bucket. After having the class vectors ready, we compute the lower approximation vectors la^{low}, la^{mid} and la^{high} for the low, medium and high buckets respectively as described in chapter 3. Finally, we apply a fuzzy union to all of the three HML vectors to get the HML quality vector q so that $q_i = \max(la_i^{low}, la_i^{mid}, la_i^{high})$. The i^{th} value in the quality vector represents the quality of the i^{th} instance in the data set.

5.2.2 Distributed Implementation

The implementation of the HML approach is very similar to the fuzzy rough set lower approximation computation approach that we presented in Chapter 3. The main difference is that we first generate three class vectors representing the HML buckets. This is done in a map stage which goes over each instance and compute its low, mid and high values as defined in Equations 5.1, 5.2 and 5.3 respectively. Next, we compute a lower approximation vector for each of the three HML class vectors exactly as we have shown in Chapter 3.

Another difference is that we use a similarity function that considers not only numeric and string attributes, but also boolean attributes. The function that we use to compute the similarity is defined in Equation 5.4 below.

$$r_{i,j} = \frac{1}{|\mathcal{A}|} \sum_{y \in \{1,\ldots,|\mathcal{A}|\}} 1 - f^{'}(q_{i,y}, q_{j,y}) \tag{5.4}$$

where $r_{i,j}$ is the similarity value between the i^{th} and the j^{th} instances, $q_{i,y}$ and $q_{j,y}$ are the y^{th} attributes of the i^{th} and the j^{th} instances, $f^{'}_{i,j}$ is the distance between $q_{i,y}$ and $q_{j,y}$ and $|\mathcal{A}|$ is the number of attributes in the data set. We use the definitions in Equations 4.3, 4.4 and 4.5 to represent the distance function $f^{'}$ for numeric, string and boolean attributes. To reduce the number of subtraction operations in Equation 5.4, we replace it with Equation 5.5 below. The steps involved in this similarity computation are described in Algorithm 14[1].

$$r_{i,j} = \frac{|\mathcal{A}| - f^{'}(q_{i,y}, q_{j,y})}{|\mathcal{A}|} \tag{5.5}$$

At this point, the computed lower approximation vectors l^{low}, l^{mid} and l^{high} for low, medium and high respectively are stored in memory in a distributed way so that the i^{th} element of each of the vectors is located in the same compute node. Finally, we run a map call to apply a fuzzy union operation on all of the three lower approximation vectors and produce a final quality vector q with each value $q_i = \max(l_i^{low}, l_i^{mid}, l_i^{high})$.

[1]Note that Algorithm 14 is exactly the same as Algorithm 8 except for the last two steps which convert the computation from being a distance computation to become a similarity computation.

Algorithm 14 MultiTypeSimilarity

INPUT: $row1, row2$
DECLARE: $distance$, $bDistance$, $attr1$, $attr2$, $attrN1$, $attrS1$, $attrB1$,$attrN2$, $attrS2$, $attrB2$, $totalAttr$, $simVal$, $numCount$, $strCount$, $boolCount$
$attrN1 \leftarrow row1.getNumericAttributes()$
$attrS1 \leftarrow row1.getStringAttributes()$
$attrB1 \leftarrow row1.getBinaryAttributes()$
$attrN2 \leftarrow row2.getNumericAttributes()$
$attrS2 \leftarrow row2.getStringAttributes()$
$attrB2 \leftarrow row2.getBinaryAttributes()$
$strCount \leftarrow row1.getStringAttrCount$
$numCount \leftarrow row1.getNumericAttrCount$
$boolCount \leftarrow row1.getBooleanAttrCount$
$distance \leftarrow 0$
$bDistance \leftarrow 0$
for i from 1 to $numCount$ **do**
    $distance \leftarrow distance + |attrN1_i - attrN2_i|$
end for
for i from 1 to $strCount$ **do**
    if $attrS1[i] == attrS2[i]$ **then**
        $distance \leftarrow distance + 1$
    end if
end for
for i from 1 to $boolCount$ **do**
    $bDistance \leftarrow bDistance + JaccardSim(attrB1[i], attrB2[i])$
end for
$bDistance \leftarrow row1.BinaryAttrCount - bDistance$
$distance \leftarrow distance + bDistance$
$totalAttr \leftarrow numCount + strCount + boolCount$
$simVal \leftarrow (totalAttr - distance)/totalAttr$
return $simVal$

5.3 POWA Approach

In the previous section, we developed a quality measure for instances that uses the high, medium and low fuzzy sets as class vectors representing the outcome attribute in the data set and then we compute the fuzzy union of the lower approximation for each of these vectors to produce the quality vector. In this section, we develop a different quality measure that relies on the idea that a good predictor for an instance tends to have similar attributes and similar outcome. We also provide a distributed implementation that generates this quality measure for each instance in the data set.

The prototype selection approach that we present in this section is based on the fuzzy rough prototype selection approach FRPS in [48]. However, FRPS is designed for classification problems and it aims to predict a class while we are trying to solve regression problems and predict a number. In addition, we are designing our approach with big data and a distributed implementation in mind.

5.3.1 Instance Quality Measure

In this section, we generate quality scores for instances in the training set based on the idea that similar instances tend to have similar output values. For instance, similar people tend to have similar total yearly medical expenses. This implies that we are working with two similarity relation. The first similarity relation $R_{\mathcal{A}}$ defines how similar two instances are based on their attribute values. The second similarity relation R_d defines how similar two instances are based on their outcome values.

Consider a decision system $(X, \mathcal{A} \cup \{d\})$ that consists of a universe of instances $X = \{x_1, \ldots, x_n\}$, a set of attributes $\mathcal{A} = \{a_1, \ldots, a_m\}$ and a fixed decision (outcome) attribute $d \notin \mathcal{A}$. The value of an instance x for attribute a is denoted by $a(x)$. For simplicity, we assume that all continuous attributes are normalized between 0 and 1. The similarity relation $R_{\mathcal{A}}$ can be defined over X as (with x and y in X)

$$R_{\mathcal{A}}(x, y) = \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} R_a(x, y). \tag{5.6}$$

where $R_a(x, y) = 1 - |a(x) - a(y)|$ for a continuous attribute a, and $R_a(x, y) = 1$ if x and

y have the same values for a and 0 otherwise for a discrete attribute a. We also convert boolean attributes into binary string attributes as we described in Chapter 4 and then use the Jaccard similarity to compute the similarity between binary string attributes. The similarity relation R_d is defined in the same way, i.e. $R_d(x, y) = 1 - |d(x) - d(y)|$ when dealing with a regression task, and $R_d(x, y) = 1$ if x and y have the same values for d and 0 otherwise for a classification task. Because of our assumption that all continuous attributes are normalized between 0 and 1, we have that $R_{\mathcal{A}}(x, y) \in [0, 1]$ and $R_d(x, y) \in [0, 1]$. When d is discrete, i.e. when we are dealing with a classification task, $R_d(x, y) \in \{0, 1\}$.

We now develop a measure to express the predictive ability of an instance. An instance y is a good predictor when other similar instances with similar attributes (high $R_{\mathcal{A}}$) also have similar outcome (high R_d). Let us consider a particular instance z. The more z resembles y w.r.t. the input attributes, i.e. the higher $R_{\mathcal{A}}(z, y)$ is, the higher we expect $R_d(z, y)$ to be. This idea can be expressed using a residual[2] implicator from fuzzy logic, such as the Łukasiewicz implicator $\mathcal{I}$ which is defined as $\mathcal{I}(p, q) = \min(1 - p + q, 1)$ for all p and q in $[0, 1]$. Note that this operator satisfies $\mathcal{I}(0, 0) = \mathcal{I}(0, 1) = \mathcal{I}(1, 1) = 1$ and $\mathcal{I}(1, 0) = 0$, hence, interpreting 0 as *false* and 1 as *true*, it is a generalization of implication from boolean logic. The formula

$$P(z, y) = \mathcal{I}(R_{\mathcal{A}}(z, y), R_d(z, y)) \tag{5.7}$$

expresses the extent to which the similarity of z and y w.r.t. the input attributes from $\mathcal{A}$ implies the similarity of the outcomes of z and y. Note that $P(z, y) = 1$ iff $R_{\mathcal{A}}(z, y) \leq R_d(z, y)$.

The overall predictive quality of an instance y can be defined as the minimum of all the $P(z, y)$ scores taken over all instances z in X:

$$\begin{aligned} Q'(y) &= \min_{z \in X} \mathcal{I}(R_{\mathcal{A}}(z, y), R_d(z, y)) && (5.8) \\ &= \min_{z \in X} \min(1 - R_{\mathcal{A}}(z, y) + R_d(z, y), 1) && (5.9) \end{aligned}$$

This corresponds to the simplified definition[3] of the degree to which y belongs to the so-called

[2]Given a t-norm T, the residual implication derived from T is defined as $\mathcal{I}_T(x, y) = sup\{z \in [0, 1] \mid T(x, z) \leq y\}\ \forall\ x, y \in [0, 1]$

[3]This simplified definition requires a computation of $Q'(y)$ that is linear in the number of instances in

positive region (see [12]). Step (5.9) is justified by our choice of the Łukasiewicz implicator. Because $Q'(y) \leq \min(1 - R_{\mathcal{A}}(y,y) + R_d(y,y), 1) = 1$ and $(1 - R_{\mathcal{A}}(z,y) + R_d(z,y)) > 1$ whenever $R_{\mathcal{A}}(z,y) < R_d(z,y)$, (5.9) simplifies to:

$$Q'(y) = \min_{R_{\mathcal{A}}(z,y) > R_d(z,y)} (1 - R_{\mathcal{A}}(z,y) + R_d(z,y)) \tag{5.10}$$

$$= \min_{R_{\mathcal{A}}(z,y) > R_d(z,y)} (1 - (R_{\mathcal{A}}(z,y) - R_d(z,y))) \tag{5.11}$$

with the implicit assumption that $Q'(y) = 1$ in case there is no z for which $R_{\mathcal{A}}(z,y) > R_d(z,y)$.

5.3.2 Partitioned Ordered Weighted Average

In [48] it was experimentally shown for classification tasks that better prototype selection is achieved when the minimum in Equation (5.8) is replaced by an ordered weighted averaging operator (OWA) with inverse additive weights. We adopt the same approach.

Definition 1 (OWA operator with inverse additive weights) Assume that the values $V = \{v_1, \ldots, v_p\}$ need to be aggregated, and that a weight vector $W = \langle w_1, \ldots, w_p \rangle$ is provided for which

- $\sum_{i=1}^{p} w_i = 1$, and
- $w_i \in [0,1]$ for all $i \in \{1, \ldots, p\}$.

If for all $i \in \{1, \ldots, p\}, c_i$ is the i^{th} largest value in V, then the OWA_W aggregation of the values in V is given by:

$$\text{OWA}_W(V) = \sum_{i=1}^{p} (w_i \cdot c_i) \tag{5.12}$$

The OWA-aggregation of the values in V is in other words obtained by first ranking these values in decreasing order, and then taking their weighted average. An inverse additive weight vector of length p is defined as

$$W^p = \left\langle \frac{1}{p \sum_{i=1}^{p} \frac{1}{i}}, \frac{1}{(p-1) \sum_{i=1}^{p} \frac{1}{i}}, \ldots, \frac{1}{1 \sum_{i=1}^{p} \frac{1}{i}} \right\rangle \tag{5.13}$$

X, while the original definition requires a computation that is quadratic in the number of instances in X. Such a computational cost would be too high in large scale information systems [12].

Definition 2 (Quality of an instance for regression) Let $(X, \mathcal{A} \cup \{d\})$ be a decision system in which d defines a regression task, and let $y \in X$. The predictive quality of y is defined as

$$Q(y) = \text{OWA}_{W^{|B|}}(\{1 - (R_{\mathcal{A}}(z, y) - R_d(z, y)) \,|\, z \in B\}) \tag{5.14}$$

with $B = \{z \,|\, R_{\mathcal{A}}(z, y) > R_d(z, y)\}$.

When performing the OWA-aggregation, the values to be aggregated are ranked in decreasing order of $1 - (R_{\mathcal{A}}(z, y) - R_d(z, y))$ (Definition 2). The values at the tail of these lists get the highest weight when computing the weighted average, i.e. the higher $R_{\mathcal{A}}(z, y)$ (respectively $R_{\mathcal{A}}(z, y) - R_d(z, y)$) is, the more weight it carries. Since the set B can be very large, we therefore propose to approximate $Q(y)$ using the Partitioned OWA (POWA) approach. In POWA, we divide B into partitions (usually equal to the number of available compute nodes) and then let each compute node compute OWA on a partition. Then, we compute another OWA on the results from each node and this is the final approximated quality value. The POWA approximation works best with very large data sets and few number of compute nodes. Figure 5.5 shows the effect of data set size and number of compute nodes on the values approximated by POWA. Note that computing POWA on one node produces the same result as computing OWA. As we increase the number of nodes, POWA becomes an approximation to the OWA computation.

5.3.3 Distributed Implementation

The general flow of our distributed implementation of POWA is very similar to our implementation of the fuzzy rough set approximations computation described in Chapter 3. It has a similar data preparation stage (see Figure 3.3) in which the data set gets loaded from HDFS into an RDD. In Chapter 3, we divide the each attribute by its range in order to ensure that any computed similarity value always lies between 0 and 1. In this implementation, however, we normalize every attribute in the data set so that it becomes between 0 and 1. This is a slightly different way of ensuring that the similarity values always lie between 0 and 1. To normalize the attributes, we compute the minimum and maximum

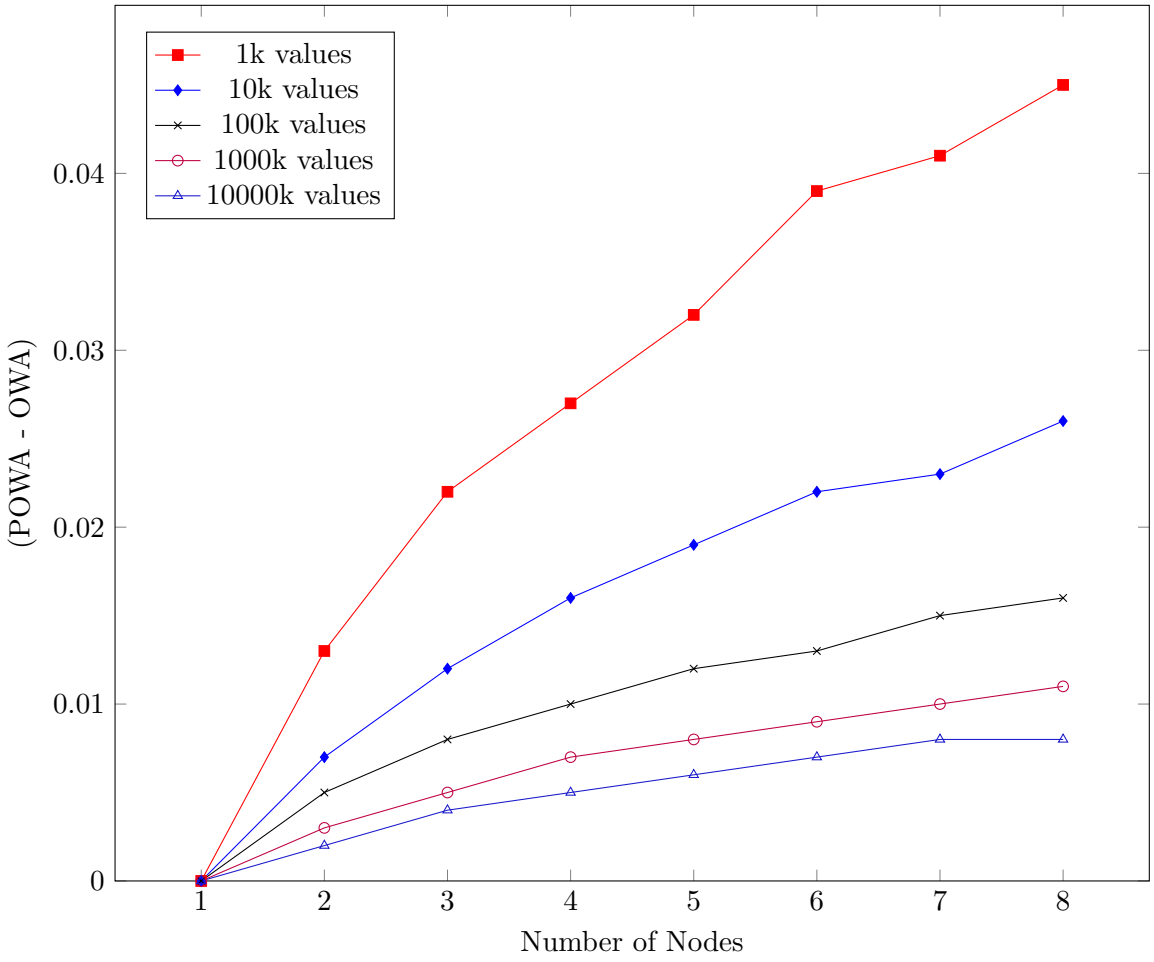

Figure 5.5: This figure shows how the OWA operator gets approximated by POWA as the number of nodes and vector size increases. The x-axis represents the number of nodes used while the y-axis represents the difference between the value that results from the POWA aggregation and the value that results from the OWA aggregation. Each vector is represented by a curve. The vectors are of sizes 1000 values, 10000 values, 100000 values, 1000000 values and 10000000 values. Each value in these vectors is between 0 and 1. With only one node, the POWA produces a value equal to the OWA produced value. As the number of nodes increase, the difference between the POWA and OWA values increases. In addition, the difference between the POWA and OWA values increases as the vector size increases. With eight nodes, for example, the difference between POWA and OWA values is around 0.05 when the vector size is 1000 while it is 0.01 when the vector size is 10 million. This means that in large data sets, the POWA aggregation produces very close value to the value produced by the OWA aggregation.

values of every attribute and store them in the min and max vectors respectively. Next, we normalize each attribute $q_{i,j}$ such that

$$q_{i,j} = \frac{q_{i,j} - min_j}{max_j - min_j} \tag{5.15}$$

Note that the instances with normalized attributes are, at this point, distributed across the nodes and that at a later step, each node will load the instances in the data set again. For this reason, we broadcast both the min and max vectors so that the instances that are loaded again get normalized as well without the need to compute the min and max values again.

In addition to the data preparation stage, this implementation has a data processing stage. In the processing stage, our aim is to compute the quality values defined in Equation 5.14 which involve computing the similarity matrices $R_{\mathcal{A}}$ and R_d. To distribute the computations of $R_{\mathcal{A}}$ and R_d, we follow a similar approach to what we did in Chapter 3 to compute the similarity matrix. That is, we set each node to compute a subset of the columns of $R_{\mathcal{A}}$ and a subset of the columns of R_d (see Figure 3.1) and use these computed columns to generate partial quality values, as we will show later. This part is done in one map stage (Map1) that produces, for each instance, one partial quality value from every compute node. These partial quality values then get grouped and aggregated together to produce final quality values. The grouping and aggregation is done with a Spark groupByKey call as well as another map stage (Map2). In the next subsections, we describe the Map1, GroupByKey and Map2 stages in more details.

a) ***Map1 Stage***

The objective of this stage is to compute a partial quality value for every instance and in every node. This involves computing a subset of the columns of both $R_{\mathcal{A}}$ and R_d similarity matrices. This stage corresponds to line 19 in Algorithm 15 and is also detailed in Algorithm 16. This stage receives an RDD partition (subset of the data set instances) of size P, an HDFS path to the data set and the broadcasted min and max vectors from the preparation stage. It declares an empty list of decimal numbers q which is used to store the values that will be computed by Equation 5.7 and also is used in an OWA operation at a later step.

It also declares another empty list of decimal numbers (*weights*) which is used to store the generated weights (line 18 in Algorithm 16) to be used later in the OWA operation.

This stage starts by reading the data set from HDFS in a row by row fashion as shown in line 4 of Algorithm 16. Then, the following is done to each row r_i:

1. The attributes in r_i get normalized using the min and max vectors as defined in Equation 5.15.

2. The following is done to each row r_j in the RDD partition:

   (a) Compute the per-attribute similarity $R_{\mathcal{A}}$ between rows r_i and r_j as described in Section 5.3.1 (line 8 in Algorithm 16).

   (b) Compute the outcome similarity R_d between rows r_i and r_j as described in Section 5.3.1 (line 11 in Algorithm 16).

   (c) Compute the implicator value $\mathcal{I}(R_{\mathcal{A}},R_d)$ and add it to q as shown in lines 13 and 14 in Algorithm 16. Recall that in the approaches described in Chapter 2 as well as the HML approach in section 5.2, we didn't need to store the implicator values because we needed to aggregate them using the (associative) minimum operator where the order does not matter. In the POWA approach, however, we need to store them so that we can order them before applying the weighted aggregation. This means that this approach requires enough amount of memory to store P decimal numbers per compute node.

3. Generate the weights values and store them on the *weights* list. Note that the number of values generated is equal to the number of elements in q at this point.

4. Both the q list and the *weights* list created earlier are passed to the OWA function, described in Algorithm 18. The result of the OWA function is a decimal number representing a partial quality for the instance represented by r_i. Note that each partial quality value is stored as a key-value pair where the key is the instance number (i) and the value is the partial quality of that instance (q_i).

5. Empty q and $weights$ so that they can be used in later iterations.

6. return all the partial quality value pairs that are computed for each instance in the form of an RDD and exit the map stage.

b) *GroupByKey Stage*

This stage receives the partial quality value pairs (one pair for every instance in the data set) from each compute node. In a cluster with e nodes, there will be e quality pairs for each instance (one from each compute node). The objective of this stage is to group all partial quality values that share the same key (belong to the same instance) in one vector such that instance i is paired with a vector containing e partial quality values with i as the key and the vector as the value. These pairs are then returned as an RDD. The reason why we need this vector of partial quality values is to be able to order these values and apply the OWA operation on them. In this stage, which corresponds to line 20 in Algorithm 16, we use the out-of-the-box groupByKey Spark function which can be directly applied on the RDD that results from the Map1 stage.

c) *Map2 Stage*

This stage, represented by Algorithm 17, simply goes over each pair, associated with an instance number i, in the RDD from the GroupByKey stage and applies the OWA operation (detailed in Algorithm 18) on the pair's partial quality values to produce a final quality value for that i^{th} instance. The result is an RDD that contains key-value pairs with the key equal to an instance number (i) and the value equal to its OWA computed quality value. This RDD gets saved back to HDFS as a text file with each line containing an instance number and its quality separated by a comma. This file can be used later to apply prototype selection on a data set as we describe in the next section.

Algorithm 15 POWA Program

1: INPUT: $pathToDataset$
2: DECLARE: $MaxAccumulator, MinAccumulator, min, max, m$
3: $rdd \leftarrow createRddFromHDFS(pathToDataset)$
4: **RddForEach**$(row, MaxAccumulator, MinAccumulator)$
5: $\quad attributes \leftarrow row.getAttributes()$
6: $\quad MaxAccumulator.add(attributes)$
7: $\quad MinAccumulator.add(attributes)$
8: **EndRddForEach**
9: $min \leftarrow MinAccumulator.values()$
10: $max \leftarrow MaxAccumulator.values()$
11: $broadcast(min)$
12: $broadcast(max)$
13: **RddForEach**(row, min, max)
14: $\quad attributes \leftarrow row.getAttributes()$
15: $\quad$ **for** i from 1 to m **do**
16: $\quad\quad attributes_i \leftarrow (attributes_i - min_i)/(max_i - min_i)$
17: $\quad$ **end for**
18: **EndRddForEach**
19: $partialOwaRDD \leftarrow Map1(rdd, pathToDataset)$
20: $powaValuesByKeyRDD \leftarrow GroupByKey(partialOwaRDD)$
21: $qualitiesRDD \leftarrow Map2(powaValuesByKeyRDD)$
22: $saveToHDFS(qualitiesRDD)$
23: Terminate

Algorithm 16 POWA - Map1

1: INPUT: $RddPartition, pathToDataset$
2: DECLARE: $partialQualities, q, quality, weights, wSize, R_{\mathcal{A}}, R_d, d1, d2$
3: $P \leftarrow RddPartition.size$
4: **for** $r_i \leftarrow getRowFromHDFS(pathToDataset)$ **do**
5: $normalizeAttributes(r_i, min, max)$
6: **for** j from 1 to P **do**
7: $r_j \leftarrow RddPartition.getRow(j)$
8: $R_{\mathcal{A}} \leftarrow multiTypeSimilarity(r_i, r_j)$
9: $d1 \leftarrow getOutcome(r_i)$
10: $d2 \leftarrow getOutcome(r_j)$
11: $R_d \leftarrow 1 - |d1 - d2|$
12: **if** $R_{\mathcal{A}} > R_d$ **then**
13: $implicator \leftarrow 1 - R_{\mathcal{A}} + R_d$
14: add $implicator$ to q
15: **end if**
16: **end for**
17: $wSize \leftarrow sizeOf(q)$
18: $weights \leftarrow generateWeights(wSize)$
19: $quality \leftarrow owa(q, weights)$
20: clean q and $weights$
21: add $quality$ to $partialQualities$
22: **end for**
23: return $partialQualities$

Algorithm 17 POWA - Map2

1: INPUT: $q[double]$
2: DECLARE: $quality, weights$
3: $weights \leftarrow generateWeights(q.size)$
4: $quality \leftarrow owa(q, weights)$
5: return $quality$

Algorithm 18 OWA

INPUT: $q[double], weights$
DECLARE: $quality$
sort q in decreasing order
for i from 1 to $q.size$ **do**
    $quality \leftarrow quality + q_i \times weights_i$
end for
return $quality$

Algorithm 19 generateWeights

INPUT: $size$
DECLARE: $d, j, weights$
$weights \leftarrow$ create array of $size$ elements
for i from 1 to $size$ **do**
    $d \leftarrow d + \frac{1}{i}$
end for
$j \leftarrow size$
for i from 1 to $size$ **do**
    $weights_i \leftarrow \frac{1}{d \times j}$
    $j \leftarrow j - 1$
end for
return $weights$

5.4 Selecting the Prototypes

The instance quality measure that we defined earlier in this chapter in both HML and OWA/POWA approaches can be used to filter out a training set and get a prototype set with instances that improve the k-NN prediction accuracy. A quality score threshold determines which instance gets included in the prototype set and which one gets excluded. If the quality score of an instance is greater than or equal to the threshold then it gets included in the prototype set. To determine a good quality threshold, we do the following:

1. From all the quality scores, select the highest quality score as the quality threshold and use it to filter out instances in the training set and produce a prototype set.

2. Apply the 10-fold cross-validation for k-NN on the prototype set that is produced in the previous step. This produces an RMSE associated with the current quality threshold.

3. Decrement the quality threshold by (highest quality score - lowest quality score)/10. Use the new quality threshold to filter out the training set and produce a new prototype set and then apply 10-fold cross-validation on it and produce a new RMSE.

4. Repeat the previous step until we have ten RMSEs computed for 10 quality thresholds.

5. Return the quality threshold with the lowest RMSE.

5.5 Experimental Results

In this section, we provide scalability experimental results, for both the HML and POWA approaches, that investigate the effect of change in the number of threads, the number of compute nodes, the number of attributes and the number of instances. In addition, we provide accuracy results for both HML and POWA prototype selection approaches and using the MEPS and SID health care data sets.

5.5.1 Scalability

We conducted five sets of experiments to observe factors that affect the scalability of both the HML and POWA implementations. These experiments focus on the data set size (number of instances), the number of attributes, the number of compute nodes, number of threads per compute node and how the choice of HML or POWA affects the time it takes to determine the best quality score threshold. The first set of experiments investigates the effect of an increase in the number of threads per compute node. Figure 5.6 shows the effect of changing the number of threads on the runtime of both HML and POWA. The runtime decreases for both approaches as the number of threads increases, as expected, and remain nearly constant between 45 and 48 threads. The best result is achieved with threads between 45 and 47 as shown in Figure 5.7. So, we decided to run the remaining experiments with 46 threads. Note that HML is noticably faster than POWA because of the fact that POWA represents instances as java objects while HML uses the implementation of Chapter 3 which is optimized to replace the object representation of instances with array representation (see section 3.3.3).

The second set of experiments, summarized in Figure 5.8, shows the effect of varying the number of available compute nodes, while keeping everything else constant. It clearly shows that the overall execution time decreases as the computation is distributed over more nodes. Figure 5.8 and Figure 5.6 show that the implementation of both the HML and the POWA approaches achieve strong scaling. The use of more hardware (nodes, threads) improves the runtime.

The third and fourth sets of experiments examine the effect of the size of the data set on the execution time of both the HML and the POWA implementations. Figure 5.9 shows that the execution time increases approximately quadratically with the number of rows. In addition, as expected, a higher number of attributes causes both implementations to run slower but it does not affect the general shape of the curve. Finally, Figure 5.10 shows that the execution time of both implementations grows approximately linearly with the number of attributes.

The final set of experiments, shown in Figure 5.11, examines whether the choice of HML

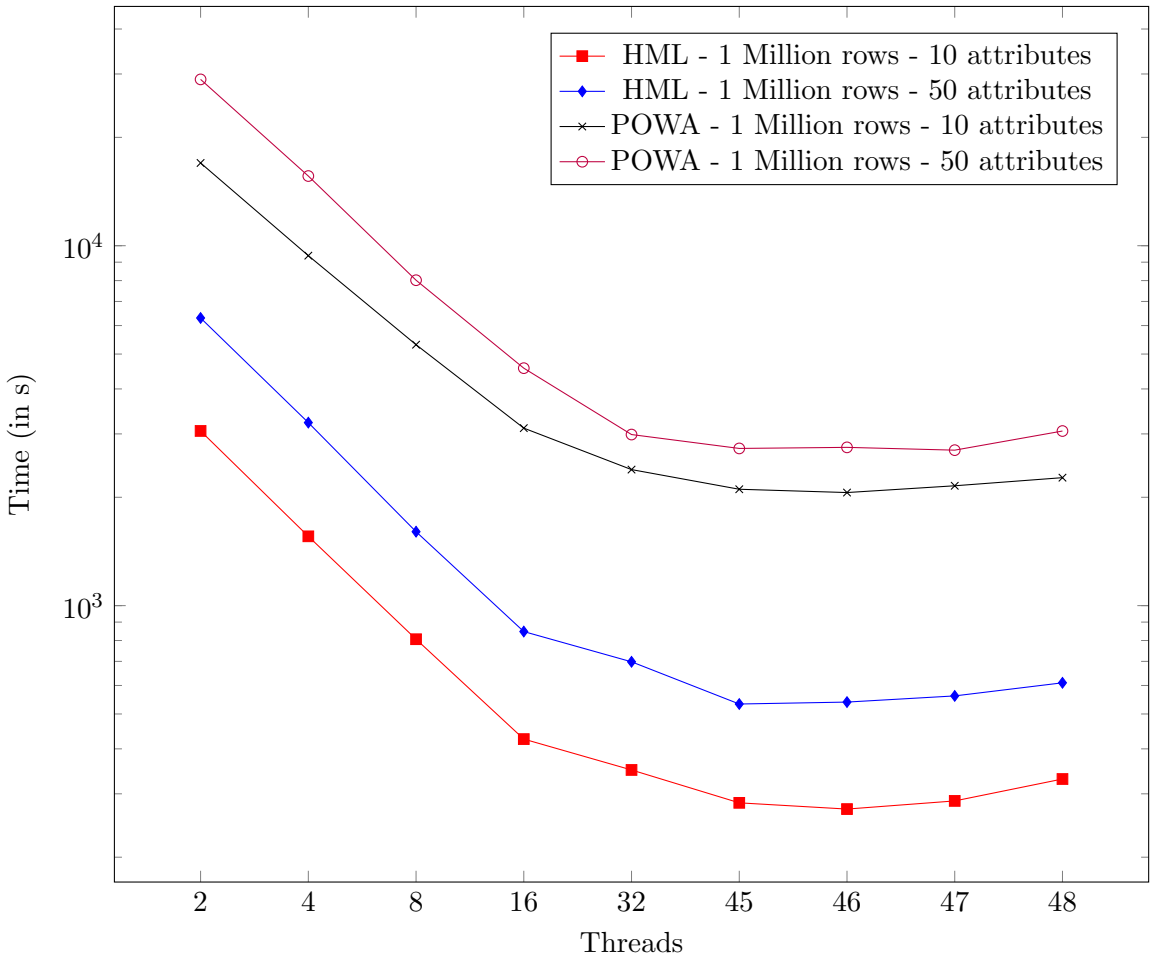

Figure 5.6: Execution time (in sec) for both HML and POWA at different number of threads per compute node. The experiments are performed on a cluster with 1 master node and 8 slave nodes, and on two different data sets (i.e. a data set with 1 million rows and 10 attributes, and a data set with 1 million rows and 50 attributes). There is a general decrease in execution time of both the HML and the POWA approaches as the number of threads per node increases.

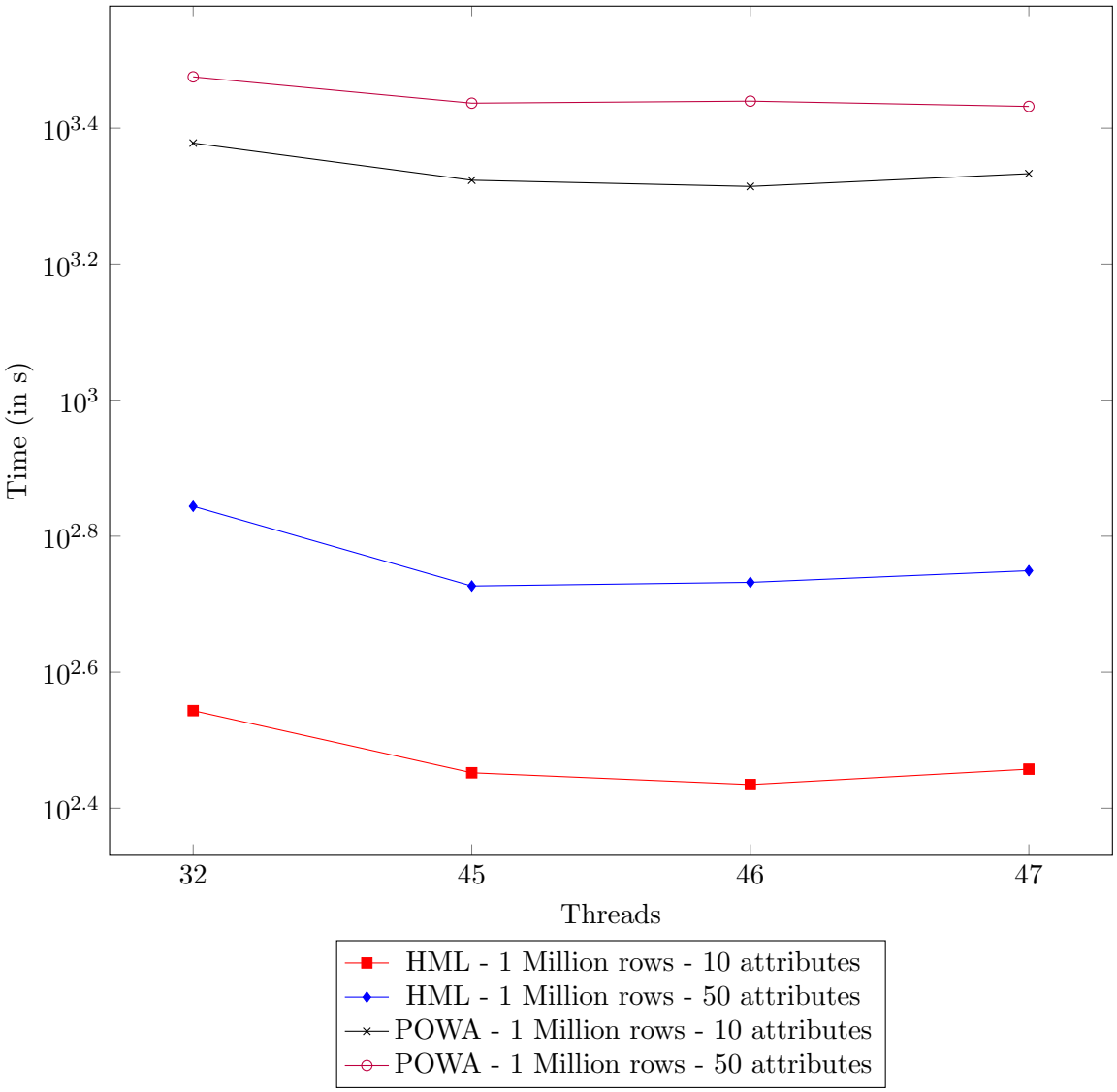

Figure 5.7: Execution time (in sec) for a varying number of threads per compute node, with the number of threads approximately equal to the number of available cores. The experiments are performed on a cluster with 1 master node and 8 slave nodes, and on a data set with 1 million rows and 10 attributes. The best performance is achieved between 45 and 47 threads.

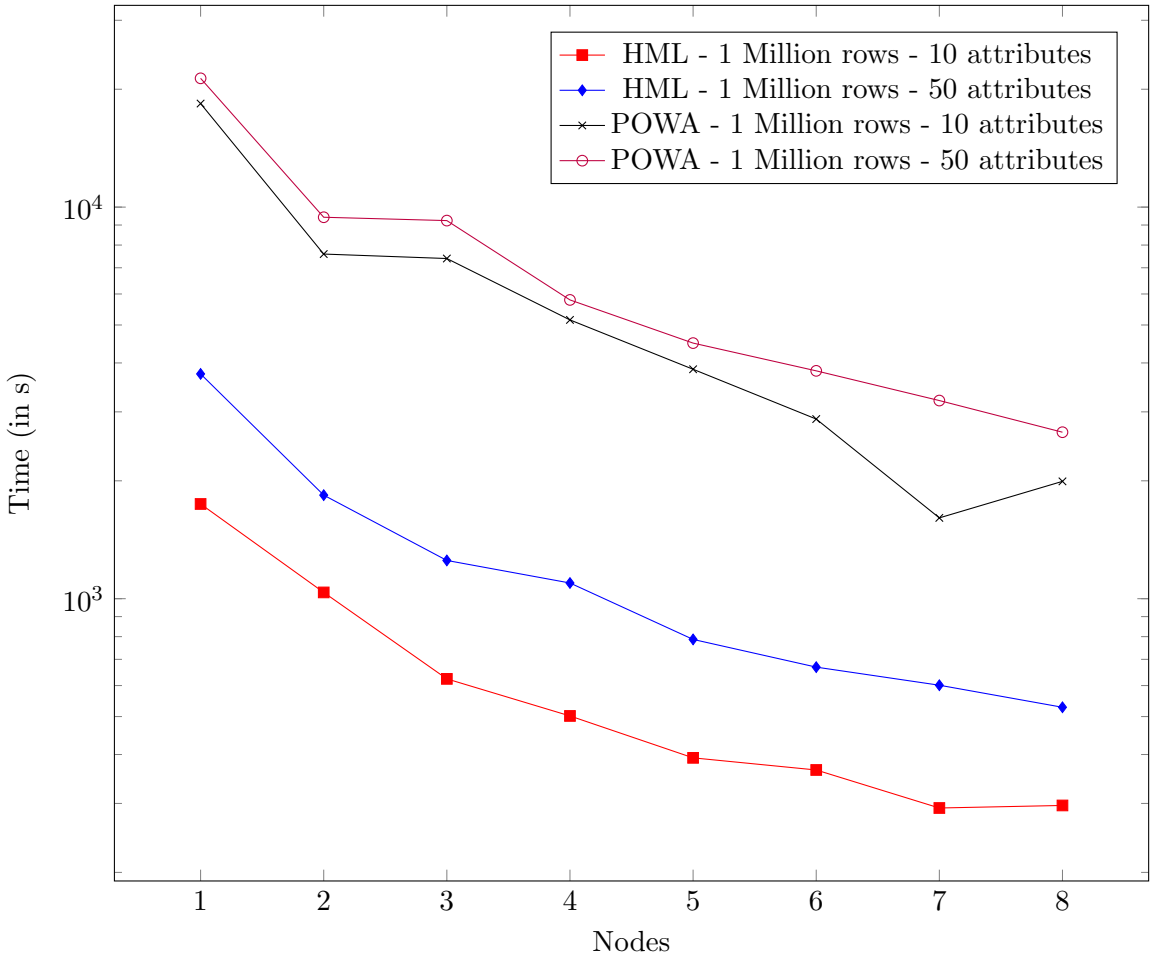

Figure 5.8: Execution time (in sec) at different number of nodes for both HML and POWA implementations. The experiments are performed on a cluster with 1 master node and 1 to 8 slave nodes, and on two different data sets (i.e. a data set with 1 million rows and 10 attributes, and another data set with 1 million rows and 50 attributes). The number of threads is kept constant at 46. The runtime decreases with the increase in number of compute nodes. Even when only 1 slave node is available, both implementations efficiently compute a quality vector for a data set with 1 million instances.

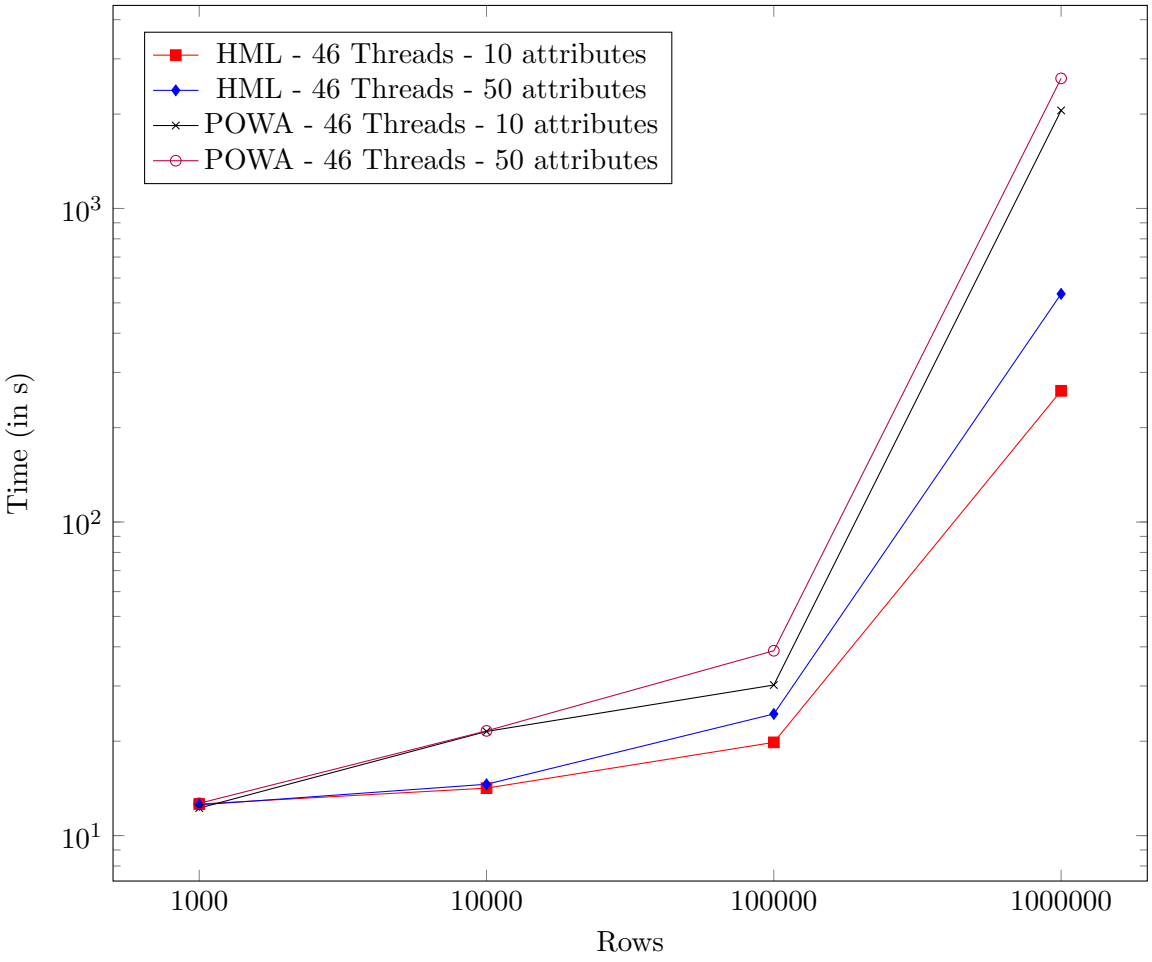

Figure 5.9: Execution time (in sec) of HML and POWA implementations for a varying number of rows in the data set.The experiments are performed on a cluster with 1 master node and 8 slave nodes, and on data sets with 1000, 10,000, 100,000 and 1 million instances and 10 or 50 attributes. The number of threads is kept constant at 46. The execution time for both implementations grows approximately quadratically in terms of the number of instances in the data set.

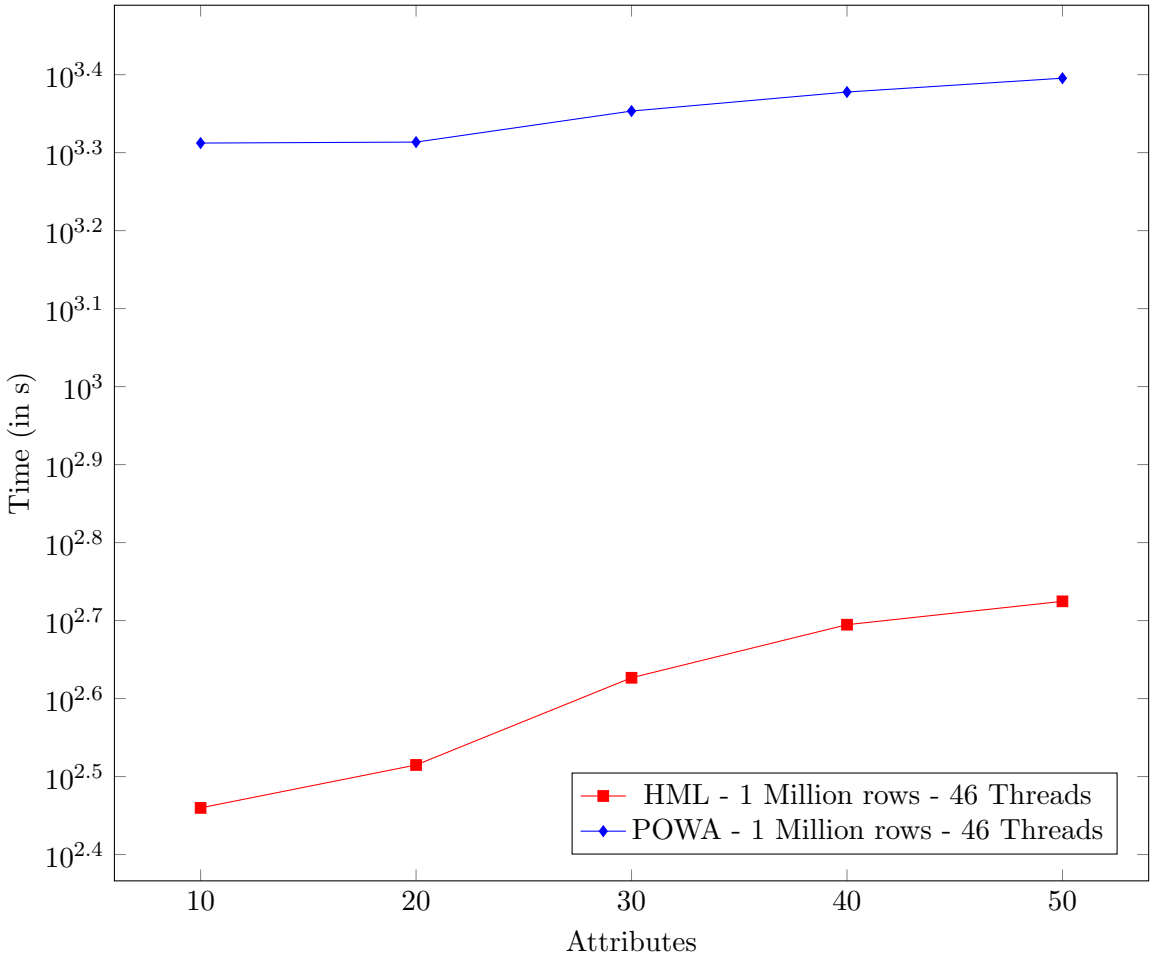

Figure 5.10: Execution time (in sec) of both HML and POWA implementations for a varying number of attributes in the data set. The experiments are performed on a cluster with 1 master node and 8 slave nodes, and on data sets with 1 million rows and respectively 10, 20, 30, 40 and 50 attributes. The number of threads is kept constant at 46. The execution time for both implementations grows approximately linear in terms of the number of attributes in the data set.

or POWA affects the time it takes to compute the best quality score threshold. Note that the process assumes that quality scores using HML and POWA are already computed. In other words, each point in the figure expresses the time it takes to compute RMSE for k-NN at ten different quality score thresholds. The choice between POWA and HML has a minor effect on the time it takes to determine the best quality score threshold.

5.5.2 Accuracy

To measure the accuracy for both the HML and POWA approaches, we use two healthcare data sets, namely the MEPS and SID data sets. Both of these data sets have been preprocessed in [46]. The preprocessed version of the MEPS data set that we use in this section contains only numeric attributes with each attribute, except the last, representing the number of visits related to a certain medical condition. The last attribute represents the total cost of a patient for the year 2011. The aim is to predict the total cost of patients in the year 2012. Since all the attributes in the MEPS data set are numeric, they don't need special handling. In this case, Equations 5.4 and 4.3 are used to compute similarity between instances in the case of both HML and POWA while Equations 4.2 and 4.3 are used to compute distances between instances when computing k-NN.

The SID data set, on the other hand, contains different attributes types. It contains age (numeric), gender (string), race (string), 773 diagnosis attributes (numeric) and 29 comorbidity attributes (boolean). We first convert the 29 comorbidity boolean attributes into a binary string attribute. Then, we pass the SID data set to HML or POWA to generate a quality vector for it as we described previously. Equation 5.5 is used to compute the similarity between two SID instances in the case of HML and POWA while Equation 4.2 is used to compute the distance between two SID instances when computing k-NN. Note that the numeric attributes are handled by Equation 4.3, the string attributes are handled by Equation 4.4 and boolean attributes are handled by Equation 4.5.

Figure 5.1 shows the accuracy of the HML and POWA approaches when applied to both the SID and the MEPS data sets. The POWA approach generates less error than the HML approach and improves the accuracy by 1.47% in the case of SID while it improves the

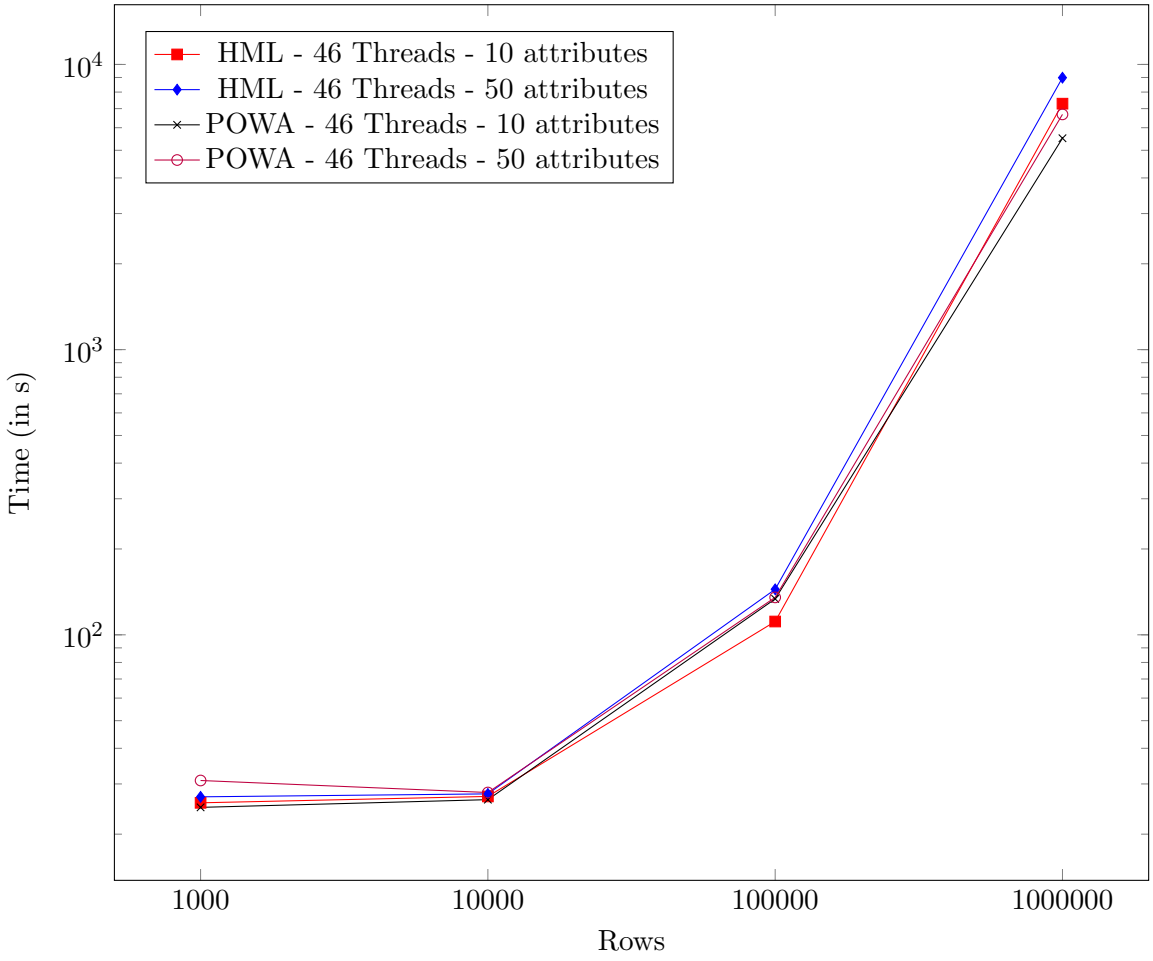

Figure 5.11: The time (in seconds) it takes to compute the best quality score threshold using a quality vector produced by HML or POWA at various numbers of rows. This part includes only computing k-NN 10-fold for 10 quality scores. The time it takes to compute the HML or the POWA quality vector is excluded.

accuracy by 2.44% in the case of MEPS.

Since HML and POWA improve accuracy by removing noisy instances in the data set, we decided to evaluate the performance of both methods in a larger presence of noise. So, we included five noisy versions of both the SID and MEPS data sets. We specifically add the noisy to the outcome attribute, namely the cost. This is done by selecting a predefined percentage of instances and modify their outcomes by adding a number taken from the standard normal Gaussian distribution, multiplied by the standard deviation of the outcome in the entire dataset. Figure 5.1 shows that both HML and POWA result in in accuracy improvement when noise is introduced.

Dataset	Noise	HML	POWA	No PS	HML Imprv.	POWA Imprv.
SID	0%	54485.51	53684.31	54485.51	0.00%	1.47%
SID	10%	57185	56722	57503	0.55%	1.36%
SID	20%	59966	59379	60447	0.80%	1.77%
SID	30%	62599	62145	63205	0.96%	1.68%
SID	40%	65757	64787	65785	0.04%	1.52%
SID	50%	67564	67455	68409	1.24%	1.39%
MEPS	0%	35433.88	34677.137	35545.53	0.31%	2.44%
MEPS	10%	37771.07	36922.01	37875.66	0.28%	2.52%
MEPS	20%	39607.32	38767.49	39707.62	0.25%	2.37%
MEPS	30%	41768.12	40983.49	41857.23	0.21%	2.09%
MEPS	40%	43308.94	42560.53	43384.63	0.17%	1.90%
MEPS	50%	44502.84	44267.25	45066.72	1.25%	1.77%

Table 5.1: This table shows the RMSE results obtained for 10-NN both when using HML and POWA prototype selection approaches and without prototype selection. It also shows the percentage accuracy improvement as a result of applying prototype selection. The data sets that are used are the SID data set with 807 attributes and 320395 instances as well as the MEPS data set with 570 attributes and 14039 instances.

In this chapter, we have presented two prototype selection approaches that are based on fuzzy rough set theory. These approaches are the HML and the POWA approaches. We

have also shown that our implementations of these two approaches scales with an increase in data set size. In addition, we have shown how both methods improve the accuracy when predicting healthcare cost using the SID and MEPS data sets.

Chapter 6

CONCLUSION

In this thesis, we explore the use of fuzzy rough set analysis in large scale information systems.

We started off in Chapter 3 by presenting a state-of-the-art distributed approach to compute fuzzy rough set upper and lower approximations for large data sets. We also presented an implementation of this approach on Spark and another implementation on MPI. In addition, we showed that both implementations scale with an increase in data set size and the runtime of both implementations improves as we add more hardware (nodes and threads). We provided a comparison between Spark and MPI in light of our implementations of computing fuzzy rough set approximations and also showed how a high level framework such as Spark can achieve runtime results close to a low level framework such as MPI. Even though we have showed that our implementations of fuzzy rough set approximations computation scale well, the symmetry feature of the similarity matrix, required for this computation, could have a potential for further runtime optimizations.

In Chapter 4, we tackled regression problems on large data sets using the weighted k-NN machine learning technique. We presented a distributed implementation on Spark that solves regression problems using the weighted k-NN approach. We also showed that this implementation scales with large data sets with even more than 1 million instances. The implementation that we presented in this chapter can be further optimized by incorporating a distributed indexing data structure that can reduce the k-NN search space while maintaining the power of parallelism and ability to scale.

In Chapter 5, we developed the HML and POWA fuzzy rough set based prototype selection approaches that aim to improve the prediction accuracy of the weighted k-NN prediction in large data sets. We showed how these two prototype selection approaches can be used hand-in-hand with our implementation of weighted k-NN to predict healthcare

cost using the SID and MEPS data sets. Even though our results showed that fuzzy rough prototype selection produce some improvement to healthcare cost prediction accuracy in large data sets, it is still worth exploring the effect of incorporating dimensionality reduction in the process.

BIBLIOGRAPHY

[1] Hadoop MapReduce Tutorial. `http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html`. Accessed Oct 9, 2014.

[2] Intel MPI Library. `https://software.intel.com/en-us/intel-mpi-library`. Accessed Oct 9, 2014.

[3] MapReduce demonstration figure of the word count example. `http://www.cs.uml.edu/~jlu1/doc/source/report/img/MapReduceExample.png`. Accessed Oct 9, 2014.

[4] Drozdek A. *Data Structures and Algorithms in Java*. Cengage Learning, 2004.

[5] Mucherino A., Papajorgji P. J., and Pardalos P. M. k-Nearest Neighbor Classification. In *Data Mining in Agriculture*, pages 83–106. Springer, 2009.

[6] Rajaraman A. and Ullman J. D. *Mining of massive datasets*. Cambridge University Press, 2011.

[7] Srivastava A., Han E. H., Kumar V., and Singh V. *Parallel formulations of decision-tree classification algorithms*. Springer, 2002.

[8] Zadeh L. A. Fuzzy Sets. *Information and Control*, 8(3):338–353, 1965.

[9] AYDIN B. Parallel Algorithms on Nearest Neighbor Search.

[10] Bede B. *Mathematics of fuzzy sets and fuzzy logic*. Springer, 2013.

[11] Cornelis C., De Cock M., and Radzikowska A. M. Fuzzy rough sets: from theory into practice. *Handbook of Granular Computing. Wiley, Chichester*, 2008.

[12] Cornelis C., Jensen R., Hurtado G., and le D. Attribute Selection with Fuzzy Decision Reducts. *Information Sciences*, 180(2):209–224, 2010.

[13] Zhang C., Li F., and Jestes J. Efficient parallel kNN joins for large data in MapReduce. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 38–49. ACM, 2012.

[14] Atkinson M. D., Sack J. R., Santoro N., and Strothotte T. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29(10):996–1000, 1986.

[15] Dubois D. and Prade H. Rough fuzzy sets and fuzzy rough sets. *International Journal of General System*, 17(2-3):191–209, 1990.

[16] Lewis D. D. Naive (Bayes) at forty: The independence assumption in information retrieval. In *Machine learning: ECML-98*, pages 4–15. Springer, 1998.

[17] Gabriel E., Fagg G. E., Bosilca G., Angskun T., Dongarra J. J., Squyres J. M., Woodall T. S., et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[18] Gieseke F., Heinermann J., Oancea C., and Igel C. Buffer kd trees: processing massive nearest neighbor queries on GPUs. In *Proceedings of The 31st International Conference on Machine Learning*, pages 172–180, 2014.

[19] Asfoor H., Srinivasan R., Vasudevan G., Verbiest N., Cornells C., Tolentino M., Teredesai A., and De Cock M. Computing Fuzzy Rough Approximations in Large Scale Information Systems. pages 9–16, 2014.

[20] Bisseling R. H. *Parallel Scientific Computation*. Oxford University Press, 2004.

[21] Buckley J. J. and Eslami E. *An introduction to fuzzy logic and fuzzy sets*, volume 13. Springer Science & Business Media, 2002.

[22] Dean J. and Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] Pan J. and Manocha D. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 211–220. ACM, 2011.

[24] Zhang J., Wong J. S., Li T., and Pan Y. A comparison of parallel large-scale knowledge acquisition using rough set theory on different MapReduce runtime systems. *International Journal of Approximate Reasoning*, 55:896–907, 2014.

[25] Zhang J., Li T., Ruan D., Gao Z., and Zhao C. A parallel method for computing rough set approximations. *Information Sciences*, 194:209–223, 2012.

[26] Zhang J., Li T., and Pan Y. Parallel Rough Set Based Knowledge Acquisition Using MapReduce from Big Data. In *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 20–27, 2012.

[27] Zhang J., Zhu Y., Pan Y., and Li T. A Parallel Implementation of Computing Composite Rough Set Approximations on GPUs. In *Proceedings of the Conference on Rough Sets and Knowledge Technology*, pages 240–250. 2013.

[28] Kato K. and Hosino T. Solving k-nearest neighbor problem on multiple graphics processors. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 769–773. IEEE Computer Society, 2010.

[29] Shvachko K., Kuang H., Radia S., and Chansler R. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[30] Polkowski L., Tsumoto S., and Lin T. Y. *Rough Set Methods and Applications: New Developments in Knowledge Discovery in Information Systems*, volume 56. Springer Science & Business Media, 2000.

[31] Prechelt L. An Empirical Comparison of C, C++, Java, Perl, Python, Rexx and Tcl. *IEEE Computer*, 33(10):23–29, 2000.

[32] Bull J. M., Smith L. A., Pottage L., and Freeman R. Benchmarking Java against C and Fortran for scientific applications. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, pages 97–105. ACM, 2001.

[33] De Cock M., Cornelis C., and Kerre E. E. Fuzzy rough sets: the forgotten step. *Fuzzy Systems, IEEE Transactions on*, 15(1):121–130, 2007.

[34] Mitchell T. M. Machine Learning, 1997.

[35] Parsian M. *Data Algorithms: Recipes for Scaling up with Hadoop and Spark*. O'Reilly Media, 2014.

[36] Radzikowska A. M. and Kerre E. E. A comparative study of fuzzy rough sets. *Fuzzy Sets and Systems*, 126(2):137–155, 2002.

[37] Zaharia M., Chowdhury M., Franklin M. J., Shenker S., and Stoica I. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, 2010.

[38] Zaharia M., Chowdhury M., Das T., Dave A., Ma J., McCauley M., Franklin M. J., Shenker S., and Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[39] Fan Y. N. and Chern C. C. An agent model for incremental rough set-based rule induction: a big data analysis in sales promotion. In *Proceedings of the 46th Hawaii International Conference on System Sciences*, pages 985–994, 2013.

[40] Verbiest N., Cornelis C., and Herrera F. FRPS: A fuzzy rough prototype selection method. *Pattern Recognition*, 46(10):2770–2782, 2013.

[41] Verbiest N., Cornelis C., and Herrera F. OWA-FRPS: A Prototype Selection Method Based on Ordered Weighted Average Fuzzy Rough Set Theory. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 180–190. Springer, 2013.

[42] Nikishkov G. P., Nikishkov Y. G., and Savchenko V. V. Comparison of C and Java performance in finite element computations. *Computers & Structures*, 81(24):2401–2408, 2003.

[43] Jensen R. and Shen Q. New approaches to fuzzy-rough feature selection. *Fuzzy Systems, IEEE Transactions on*, 17(4):824–838, 2009.

[44] Mall R., Jumutc V., Langone R., and Suykens J. A. Representative subsets for big data learning using k-NN graphs. pages 37–42, 2014.

[45] Riza L. S., Janusz A., Bergmeir C., Cornelis C., Herrera F., Ślezak D., and Benítez J. M. Implementing algorithms of rough set theory and fuzzy rough set theory in the R package roughsets. *Information Sciences*, 287:68–89, 2014.

[46] Sushmita S., Newman S., Marquardt J., Ram P., De Cock M., Teredesai A., and Prasad V. Population Cost Prediction on Public Healthcare Datasets. to appear in: Proceedings of ACM Digital Health 2015 (5th International Conference on Digital Health), 2015.

[47] Feifei X. U., Lai W. E. I., Zhongqin B. I., and Lin Z. H. U. Research on fuzzy rough parallel reduction based on mutual information. *Journal of Computational Information Systems*, 10(12):5391–5401, 2014.

[48] Verbiest N. and Cornelis C. and Herrera F. A prototype selection method based on ordered weighted average fuzzy rough set theory. In *Proceedings of the 14th International Conference on Rough Sets, Fuzzy Sets, Data Mining and Granular Computing*, pages 180–190, 2013.

[49] Gropp W. and Lusk E. Installation guide for MPICH, a portable implementation of MPI. Technical report, Technical Report ANL-96/5, Argonne National Laboratory, 1996.

[50] Yang Y., Chen Z., Liang Z., and Wang G. Attribute Reduction for Massive Data Based on Rough Set Theory and MapReduce. In *Proceedings of the Conference on Rough Sets and Knowledge Technology*, pages 672–678. 2010.

[51] Yang Y., Chen Z., Liang Z., and Wang G. Parallelized Computing of Attribute Core Based on Rough Set Theory and MapReduce. In *Proceedings of the Conference on Rough Sets and Knowledge Technology*, pages 155–160. 2012.

[52] Pawlak Z. Rough Sets. `http://bcpw.bg.pw.edu.pl/Content/2026/RoughSetsRep29.pdf`. Accessed Oct 9, 2014.

[53] Pawlak Z. Rough sets. *International Journal of Computer & Information Sciences*, 11(5):341–356, 1982.

[54] Pawlak Z. Rough set theory and its applications to data analysis. *Cybernetics & Systems*, 29(7):661–688, 1998.