LARGE-SCALE NON-LINEAR REGRESSION WITHIN THE MAPREDUCE FRAMEWORK

by

Ahmed Khademzadeh

A thesis submitted to the College of Engineering at
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
July 2013

We the undersigned committee
hereby approve the attached thesis

LARGE-SCALE NON-LINEAR REGRESSION WITHIN
THE MAPREDUCE FRAMEWORK

by
Ahmed Khademzadeh

Philip Chan, Ph.D.
Associate Professor
Computer Sciences
Principal Adviser

Marius Silaghi, Ph.D.
Assistant Professor
Computer Sciences

Georgios C. Anagnostopoulos, Ph.D.
Associate Professor
Electrical & Computer Engineering

William D. Shoaff, Ph.D.
Associate Professor and Department Head
Computer Sciences

Abstract

Large-scale Non-linear Regression within the MapReduce Framework
By: Ahmed Khademzadeh
Thesis Advisor: Philip Chan, Ph.D.

Regression models have many applications in real world problems such as *finance*, *epidemiology*, *environmental science*, etc.. Big datasets are everywhere these days, and bigger datasets would help us to construct better models from the data. The issue with big datasets is that they would need a long time to be processed or even to be read on a single machine. This research employs MapReduce to model large-scale non-linear regression problems in a parallel fashion. MRRT (MapReduce Regression Tree) algorithm divides the feature space into overlapping subspaces and then shuffles each of the subspace's data items to a node in the cluster. Each node in the cluster then constructs a regression tree for the subspace of the data it has received. Different versions of algorithm (overlapping/non-overlapping subspaces and weighted/unweighted prediction using neighboring models) are proposed and compared with the regression tree (RT) algorithm implemented in Matlab libraries.

Experiments on synthetic and real datasets show that MRRT algorithm that is devised to be fast and scalable for MapReduce framework not only has a close to linear speedup, and close to optimum scalability, but also outperforms the RT algorithm in terms of accuracy (in most cases) and improves the prediction time by more than 80%. Although MRRT is designed for MapReduce framework, it could be used on a single machine, and in that case it improves the learning time by 60% (in most cases) comparing to RT algorithm, and shows to be of close to linear scalability (comparing to RT algorithm which is roughly of quadratic scalability).

Contents

List of Tables

List of Figures

List of Algorithms

Preface

Acknowledgments

Chapter 1

Introduction

1.1 Motivation

The goal is approximating a non-linear regression model using piecewise linear models for large-scale datasets. Regression models have many applications in real world among which we can name *trend line*, *finance*, *epidemiology*, *environmental science*, etc. Big datasets are everywhere these days, and bigger datasets would help us to find better models from the data. The issue with big datasets is that they also would need a long time to be processed on a single machine. When the dataset is very large (terabyte scale) even reading the content of the dataset would take a very long time (a high-end machine with four I/O channels each having a throughput of 100 MB/sec will require three hours to read a 4 TB data set! [12]). For this reason we need to use parallel and distributed methods to process big datasets.

There are many options for parallel data processing. We have decided to use MapReduce programming model as the distributed data processing framework. MapReduce is a programming model introduced by Google in 2004, for processing large datasets [3]. We have chosen MapReduce as the distributed data processing framework to use for the following reasons:

- MapReduce handles many of the issues with large-scale distributed data processing such as distributed file system. Google File System (GFS) is the original file system it uses. GFS makes all the data transfer and distribution on different cluster nodes transparent to the programmer. The user simply copies a file on the cluster, and GFS decides how the file to be distributed among cluster nodes, keeps track of the chunks of the file, and also manages replication of chunks of file on different nodes for fault tolerance purpose.
- Fault tolerance is another thing that MapReduce takes care of. The programmer does

not need to be worried about resolving the problem of nodes' failure. If a node fails, MapReduce itself manages the problem and assigns its tasks to other cluster nodes.

- Code and data migration is also managed by MapReduce. All the mapper nodes in the cluster run same map code on data. MapReduce takes care of delivering the code to all mappers and running the code on nodes. The result of map round needs to be shuffled among cluster nodes (delivered to reducers), and MapReduce takes care of this data shuffling too. Reduce phase and code migration in this phase is also managed by MapReduce framework.
- MapReduce simplifies solving a distributed data processing problem by introducing a high level programming model for distributed data processing. It helps programmers to concentrate on program logic and all the details and issues related to distributed nature of the solution is managed by MapReduce. Although MapReduce restricts us and reduces the flexibility in some ways, but it helps us to have a standard way of describing distributed data processing algorithms.
- MapReduce is one of the common ways of solving distributed data processing problems in industry these days.

Details about how MapReduce works is explained in section 2.2.

1.2 Problem Statement

We are handling a large-scale non-linear regression problem. Regression is a supervised learning technique in which the algorithm tries to find a model from a dataset to generate a numerical prediction for future data items. We will call the numerical dependent variable (target variable) y, and try to approximate its value as a function of other numerical values x. Here x is a vector consisting of n numerical values $x_1, x_2, \ldots, x_n$, where n is number of features (attributes) of each data item of the dataset.

$$y = f(x) + \epsilon \tag{1.1}$$

In above equation ϵ is the difference between actual and predicted value of target value. The predicted value for y is $f(x)$ and is indicated by $\hat{y}$ symbol. There are different ways to handle non-linear regression problem.

We intend to find a solution for large-scale datasets. Handling large-scale datasets could be very slow if parallel and distributed data processing techniques and frameworks are not used. Because of the reason mentioned in section 1.1 the programming model we have employed to handle large-scale datasets is MapReduce. When using MapReduce, the method that is employed to solve the problem sequentially needs to be coupled and translated into MapReduce programming model. Some details about how MapReduce works is explained in section 2.2.

Designing an algorithm for MapReduce framework (map and reduce phases) entails issues such as deciding about what process needs to be done by cluster nodes on their local piece of data and what information they need to extract in order to cover the issue of not having a global view of the data on each node of the cluster. The other challenge when designing a MapReduce based algorithm is how the final result is aggregated. Generally one problem could be handled by MapReduce in several different ways and choosing the best way to make use of MapReduce capabilities is the main challenge. Since there is no communication between different nodes during Map and Reduce phases, and the results only could be communicated when the Map phase is done, choosing an effective strategy on extracting useful information from partial views of different mappers from the partial data they have in hand, and making use of this data in Reduce phase (or next MapReduce rounds) is a problem that needs to be addressed.

1.3 Overview of Approach

In this work two different distributed algorithms for approximating non-linear regression model of a dataset using piecewise regression is suggested. Both algorithms are suggested for MapReduce framework.

The first algorithm is called MapReduce Regression Tree (MRRT) algorithm. This algorithm is dividing the feature space into equal-size partitions (equal-size in terms of volume and not number of data points in the partition). To form the partitions, the feature space is divided into partitions along one dimension of the feature space. This dimension is selected randomly or using a pre-processing method that is working on a sample of dataset. Data items belonging to different subspaces are then sent to different reducers, and all reducers construct regression tree models (in parallel) for the partition they have received. Although a reducer technically needs only one partition of the feature space to generate the model, we send left and right partitions of each partition to the reducer too (overlapping subspaces). This way each reducer would receive three partitions instead of one partition (leftmost and rightmost partitions of the dataset have only one neighbor and the corresponding reducer would receive two partitions instead of three). Since we are sending extra information to each reducer, the data that needs to be transferred in the network and processed in each machine would increase. This redundancy has a good side-effect which is increase in accuracy of the final model by decreasing prediction error of the data items that are located near the borderlines. This algorithm uses a weighted prediction mechanism in order to increase accuracy. Details of this algorithm is explained in section 3.1.

Second algorithm is called Slope-changing algorithm. In this algorithm dataset is distributed among cluster nodes in a random fashion. Every mapper has part of the dataset at hand and finds a set of candidate split points in that part of dataset. Split points are points that will be used to split the feature space into smaller subspaces. These candidate split points include points with local maximum and minimum target value. Points that the model's slope changes sharply in those points are also selected by mappers as candidate split points. All the candidate split points found by mappers are sent to a single reducer in order to select the split point set from this set of candidate points. Two different methods are suggested to make the selection of final split points from candidate split points. One method is using Parzen Window Classifier, and the other method is fitness proportional selection. After selecting the split point set, this set is sent to all mappers in the cluster. All mappers use these split points to partition the data based on the subspaces formed by these split points. All mappers then send the data points pertaining to a certain subspace to a certain reducer. This way each reducer would receive all the data points of a certain subspace from all mappers, and can construct a linear model for that subspace. This way a piecewise linear model for all subspaces of the feature space is constructed. This piecewise linear model will be used to predict the target value for future test items based on the subspace in which the test item is located. Details of this algorithm is explained in section 3.2.

Both MRRT and Slope-changing algorithms divide the feature space into subspaces and find models for each subspace, but they are dividing the feature spaces differently. MRRT divides the space in equal-size subspaces, but size and number of subspaces in Slope-changing algorithm might be different and is determined by split points chosen by the algorithm. Another difference is that MRRT constructs regression tree models for subspaces, but Slope-changing algorithm constructs linear models for subspaces. MRRT also uses overlapping subspaces, but Slope-changing algorithm does not use overlapping subspaces.

1.4 Overview of Contributions

Because of the limitation of Slope-changing algorithm (that is discussed in section 4.5.1), it is not applicable to high-dimensional datasets. For this reason we only list contributions of

MRRT algorithm:

- Overlapping subspaces (coupled with weighted prediction) not only solves the data distributed-ness problem, but also helps to improve accuracy over the baseline (regression tree) algorithm. If the *preProcess* method is employed to choose the dimension to split, MRRT improves the accuracy for 8 out of 10 synthetic datasets from 1.1% to 32.86% and for all three real datasets for 4.66%, 13.24%, and 22.73% respectively.
- MRRT algorithm shows to have close to linear speedup (for two out of four datasets experimented) and near to optimum scalability for all datasets.
- Although MRRT's prediction is done sequentially and not on a MapReduce framework, it improves the prediction time by more than 80% comparing to regression tree algorithm.
- MRRT could be used on a single machine, and in that case it improves the learning time by 60% (in most cases) comparing to regression tree algorithm.
- MRRT needs to choose a dimension to split along. *preProcess* method we have proposed for MRRT (to choose the dimension to splitp) increases accuracy of model for 11 out of 13 datasets comparing to model constructed by regression tree algorithm.

1.5 Overview of Chapters

In chapter 2 we will review the literature related to regression, piecewise regression and tree regression. We also talk about MapReduce and also some large-scale problem that is solved using MapReduce. Limitations of MapReduce and arguments about this limitations are also discussed in this chapter. Chapter 3 presents details of algorithms we proposed for solving the piecewise approximation of non-linear model within MapReduce. Next chapter presents the empirical evaluation of the algorithms and compares them with the baseline (regression tree) algorithm. Chapter 5 summarizes findings and presents the concluding remarks.

Chapter 2

Literature Review

In this thesis two distributed MapReduce-based algorithms for approximating large-scale non-linear regression using piecewise regression are proposed. Two major parts of the problem are approximating non-linear regression using piecewise regression, and MapReduce framework. We will review the literature related to these two major subproblems in following sections.

2.1 Approximating Non-linear Regression Using Piecewise Regression

2.1.1 Linear Regression

Linear regression could be used when there is a linear (or roughly linear) dependency between x and y (x and y are introduced in section 1.2). In this case the learning algorithm tries to model y as a linear function of x:

$$y = \beta_0 + \beta_1 x + \epsilon \tag{2.1}$$

In above equation size of x and β_1 vectors are equal to number of dimensions in the feature space, and ϵ is the difference between actual and predicted value of target variable (error term).We use $\hat{y}$ symbol to indicate the predicted value of target variable by the model and we have $\hat{y} = \beta_0 + \beta_1 x$. The learning algorithms tries to learn β_0 and β_1 values (called weights) from the training items in the dataset. When learning weights, the objective is to minimize difference of actual and predicted values for all data items (as an example this difference could be measured by minimizing sum of square of difference between actual and predicted target values):

$$\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 = \sum_{i=1}^{n}(y^{(i)} - (\beta_0 + \beta_1.x^{(i)}))^2 \qquad (2.2)$$

We use $< x^{(k)}, y^{(k)} >$ to indicate k^{th} data item in the dataset.

2.1.2 Non-linear Regression via Piecewise Linear Regression

One of the advantages of linear regression is its simplicity, and one of its disadvantages is its globality. When the relation between x and y is complex and non-linear, even the best possible linear model would have a high average prediction error value. Partitioning the feature space into smaller subspaces and constructing a model for each subspace of the feature space might be helpful in finding a better model and reducing the error. Piecewise methods are using this idea and find constant or linear models for each subspace of the feature space instead of one global linear model.

A constant model for a subspace containing a set of data items like $s_1 = \{< x^{(1)}, y^{(1)} >, < x^{(2)}, y^{(2)} >, \ldots, < x^{(n)}, y^{(n)} >\}$ would be calculated as following:

$$\hat{y}(s_1) = \frac{1}{size(s_1)} \sum_{k \in s_1} y^{(k)} \qquad (2.3)$$

and the prediction for any new data item that lies in this subspace would be $\hat{y}(s_1)$.

In most cases it is better to find a linear model for each subspace of the feature space. In this case, equation 2.2 that is given in previous section is used by linear regression learning algorithm to find a linear model for each subspace.

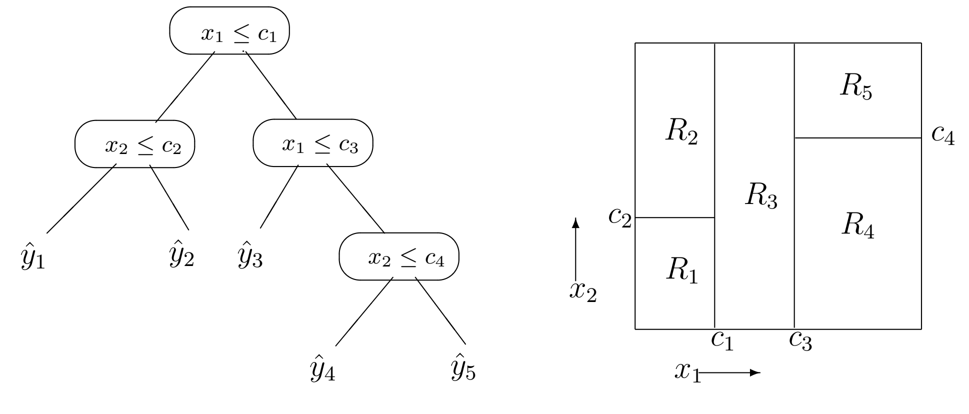

Figure 2.1: A regression tree (left), and the corresponding 2-dimensional feature space (right). Each of tree nodes are corresponding to a subspace in feature space [19]

Dividing the feature space into subspaces can be done in different ways. A simple way of dividing the feature space into smaller subspaces is using equal-size subspaces. It also is possible to let the algorithm decide on borderline of the subspaces. Regression tree that is presented in next section, uses a recursive method to divide the feature space into subspaces. Figure 2.1 depicts a regression tree and also shows how the feature space is divided into smaller subspaces based on this regression tree. Leaves of the regression tree are models for each subspace of the feature space ($\hat{y}_i$ is a model for R_i).

2.1.3 Piecewise Regression with Regression Trees

Regression tree is a piecewise method that recursively partitions the feature space into smaller subspaces. The tree itself consists of nodes and edges. Every node contains a simple condition, e.g. **if** $x_i < 10$ (i.e. if its i^{th} feature's value is smaller or bigger than 10), and one of the branches is chosen based on the answer of current data item to this question. To find the prediction for a new data item, tree is traversed starting from the root until we reach a leaf. Leaves of regression tree contain a model like linear model or constant model.

Constructing regression tree is an iterative task. In each iteration a feature and a corresponding threshold value needs to be chosen by the algorithm. We call a pair of $< Feature, Value >$ as a split point. Selecting split points could be a critical task when constructing piecewise models. When selecting a split point pair among different candidate split point pairs, a metric is used to evaluate different trees corresponding to different split point pairs. The tree and corresponding split point that performs better based on the metric is chosen to be used in next iteration. Basic regression tree algorithm can use Sum of Squared Errors (SSE) to evaluate a tree T [8]:

$$S = \sum_{c \in leaves(T)} \sum_{i \in c} (y^{(i)} - \hat{y}^{(c)})^2 \tag{2.4}$$

where

$$\hat{y}^{(c)} = \frac{1}{size(c)} \sum_{i \in c} y^{(i)} \tag{2.5}$$

is the predicted value for all data items landing in that leaf.

Algorithm 1 lists the basic algorithm for constructing regression tree. In this algorithm, first all the data items of dataset are assigned to the root node (line 2). The $\hat{y}^{(c)}$ and SSE values are then calculated for root node (lines 3-4). Afterward a repetitive task (lines 6-32) is applied on each leaf of the tree and each leaf is populated with two children until a certain condition is hold (lines 26-30). For each leaf of the tree all possible split pairs

$< Feature, Value >$ are examined and the pair that reduces SSE of the leaf most is chosen (lines 12-25). If the chosen pair reduces the SSE more than a threshold δ, then the node is populated with two children, otherwise that leaf will be kept untouched (lines 26-31). If number of data items of a node is less than a threshold q, that node also will be kept untouched (lines 8-10).

Algorithm 1 Basic Regression Tree Construction Algorithm

```
1: procedure ConstructRegTree($dataset$)
2:     $root.dataItems = dataset$
3:     $root.\hat{y}^{(c)} = \frac{1}{size(dataset)} \sum_{i \in dataset} y^{(i)}$
4:     $root.sse = \sum_{i \in dataset} (y^{(i)} - root.\hat{y}^{(c)})^2$
5:     $queue.add(root)$
6:     while !queue.isEmpty do
7:         $node = queue.remove$
8:         if $size(node.dataItems) < q$ then
9:             continue
10:        end if
11:        $bestSplitPair.sse = \infty$
12:        for $splitPair \in allSplitPairs$ do
13:            $left.dataItems = splitDataItems(node.dataItems, splitPair, left)$
14:            $right.dataItems = splitDataItems(node.dataItems, splitPair, right)$
15:            $left.\hat{y}^{(c)} = \frac{1}{size(left.dataItems)} \sum_{i \in left.dataItems} y^{(i)}$
16:            $left.sse = \sum_{i \in left.dataItems} (y^{(i)} - left.\hat{y}^{(c)})^2$
17:            $right.\hat{y}^{(c)} = \frac{1}{size(right.dataItems)} \sum_{i \in right.dataItems} y^{(i)}$
18:            $right.sse = \sum_{i \in right.dataItems} (y^{(i)} - right.\hat{y}^{(c)})^2$
19:            if $bestSplitPair.sse > left.sse + right.sse$ then
20:                $bestSplitPair.splitPair = splitPair$
21:                $bestSplitPair.sse = left.sse + right.sse$
22:                $bestSplitPair.left = left$
23:                $bestSplitPair.right = right$
24:            end if
25:        end for
26:        if $node.sse - bestSplitPair.sse > \delta$ then
27:            $node.left = left$
28:            $node.right = right$
29:            $queue.add(left)$
30:            $queue.add(right)$
31:        end if
32:    end while
33: end procedure
```

One of the issues with basic algorithm for regression tree is using a greedy method to select the feature and value to split. There are two problems with a greedy method to select the split points. First, since greedy methods make their decision based on a locally optimal choice, their final model might be a suboptimal model in terms of accuracy. Second, when number of dimensions and size of dataset is large, finding these split points (even greedily) would have a very high runtime. We need to find methods to increase accuracy and decrease the runtime.

Regression tree and piecewise linear regression are proposed when the dataset is not distributed. In the case when dataset is large, algorithms to generate regression model could be very slow (splitting all data items of all leaf nodes into two subsets for all different pairs of $< Feature, Value >$ is an expensive task for a high-volume high-dimensional dataset). Thus for large-scale datasets, new technologies, techniques and algorithms needs to be used to perform the task more efficiently. Section 2.2 discusses about MapReduce that is the framework we have used for distributed data processing.

2.1.4 Piecewise Linear Approximation of Time Series

Piecewise linear representation (PLR) is generally used to approximate time series with straight lines (hyper planes). Piecewise linear representation is more efficient than other modeling techniques in terms of storage, transmission and computation and has several applications in clustering, classification, similarity search, etc. [10].

Piecewise linear representation are also called Segmentation Algorithms (SA). Three different specification has been defined for SAs. For a time series T, find the best representation that

- Includes only K segments,
- The error for each segment does not exceed a threshold, and
- The total error does not exceed a threshold.

A PLR can be either online or batch [10].

PLR algorithms can be divided to 3 different categories: bottom-up, top-down, and sliding-windows. Bottom-up approach finds the approximation of small pieces of time series and find the final solution by merging them. Top-down approach recursively divides the time series until satisfaction of a stopping criterion [10, 13]. Sliding-windows grows a segment until the error exceeds a threshold. Sliding windows starts from the first point of T and adds points to it while sum of error is less than a threshold. At that point a segment is generated and process continues to generate a new segment form the next point. Several optimization are proposed for this algorithm: 1) adding a bigger value than 1 in each iteration of the process of finding one segment, 2) since the error is monotonically non-decreasing, methods such as binary search can be used [10].

Top-down methods find good split points and split T into two segments. An approximate linear model for each part is calculated and the error is calculated for each part. If error is less than a threshold, model for that part is finalized, otherwise the algorithm recursively repeats the process. Bottom-up methods start from smallest possible segments (totally n/2 segments). They find the cost of merging each pair of adjacent segments and merge the adjacent pair that has the lowest cost. This process is repeated until the minimum cost of merging is smaller than a threshold [10, 13].

Keogh et al. propose a new online algorithm called SWAB (Sliding-Window And Bottom-up). SWAB uses a sliding buffer of size close to 6 segments and uses bottom-up on that frame. After segmentation the leftmost segment is reported and the corresponding data is removed from the frame and more data are read into the frame [10].

D. Lemire suggests that instead of having linear models for each interval of a time series, we could have models of different degrees for different intervals [13]. Some intervals may have constant models, some linear, etc.. The method is called adaptive because degree of model in an interval is decided adaptively. The reason why adaptive method is suggested is that piecewise linear models might locally over-fit the data by trying to find a linear model for the data, while a constant model would fit the data better. Since time series datasets could be very large, efficiency of algorithm is very important. The adaptive method proposed in this paper tries to improve the quality of the model while keeping the cost of the model construction same as top-down[11] method.

Different algorithms with different advantages and dis-advantages could be used for approximating time series [13]. Optimal adaptive segmentation uses dynamic programming to find the best segmentation and thus is of high complexity ($\Omega(n^2)$). Top-down method on the other hand selects the worst segment and divides it to two smaller segments iteratively until the complexity of model reaches the maximum allowed complexity. Adaptive top-down algorithm first applies top-down algorithm on time series, and then replace linear model segments with two constant model segments if the error can be reduced with this replacement. Another version of adaptive top-down first constructs a top-down constant model and then merges constant models in order to have linear models. The optimal algorithm is not practical because it takes a very long time (weeks) to generate results for a time series with one million data points. The adaptive top-down is slightly slower than top-down algorithm, but generating results of higher quality.

2.1.5 Online Approximation of Non-linear Models

XCSF and LWPR are the two algorithms for online linear approximation of an unknown function. These methods cluster the input space into small subspaces and find a linear model for each subspace and use a weighted sum to find the final model. For this we need to first structure the feature space into small subspaces in order to exploit the linearity of the target function in each subspace, and then we need to find the linear models in each patch. There are several solutions for the second step, but the first step is not straightforward. XCSF is an evolutionary-based algorithm that uses GA[22], and LWPR (Locally Weighted Projection Regression) is a statistics-based algorithm for function approximation for online approximation of non-linear multi-dimensional functions incrementally [20].

Receptive Fields (RFs) is the notion used by LWPR for the ellipsoidal subspaces. XCSF refers to subspaces as classifiers (another term for RF) [17] . Both algorithms has an empty population of RFs at the beginning, and add new members to this population when a new uncovered data item is received. An n-Dimensional ellipsoid that is not necessarily axes-aligned can be represented by a positive semi-definite and symmetric matrix (D). Then the squared distance of a data item (x) from the center (c) of this space can be defined as:

$$d^2 = (x - c)^T.D.(x - c) \tag{2.6}$$

If this distance is zero, then the data item is placed on the surface of the RF. This way the subspaces are found in both methods. A linear model for each subspace can be expressed as:

$$p(x) = \sum_{k=1} nb_k.x_k + b_0 \tag{2.7}$$

One data item can be covered by several subspaces and in that case a weight combination of linear models of those subspaces is considered and the model prediction for the input data item [17, 22, 20].

LWPR assigns a gaussian activity weight to each subspace based on its distance to the data item, and ignores those weights that are smaller than a threshold for the sake of performance. This way closer subspace has more significant effect on final prediction comparing to farther subspaces. XCSF, on the other hand, only assigns weight to subspaces with‘istance of less than 1. Weights are proportional to inverse value of prediction error in XCSF [17, 22, 20].

Finding a linear model for each subspace is straightforward and can be done using least squares methods. XCSF uses RLS (Recursive Least Squares), and LWPR uses incremental partial least squares (incremental version of PLS) to find the linear model in a subspace [17, 22, 20].

Learning the locality (which is the shape and location of receptive fields) is done by a steady state genetic algorithm in XCSF, and by a stochastic gradient descent in LWPR [17, 22, 20].

In XCSF, each RF has an approximate value for its current prediction error which is used to calculate its fitness for the GA. Fitness is shared among the RFs that cover same inputs. Tournament selection is used for selection task of GA, and crossover, and mutation operators are applied on the location of center, stretch and rotation which is defined by a matrix (D). When the population reaches a maximum value, some RFs are deleted from crowded regions of input space using a proportionate selection probability. During this process, RFs are tried to be generalized by making their coverage area larger while keeping their accuracy sufficiently large [17, 22, 20].

Center of subspaces are not changed in LWPR and the D matrix is changed (the size and direction of the ellipsoids). This optimization is done by an incremental gradient descent based on stochastic leave-one-out cross-validation. For this purpose first D is decomposed to a triangular matrix and then is updated. The cost function is activity weighted error plus a penalty term that is preventing the subspaces to shrink over iterations [17, 22, 20].

XCSF and LWPR are compared in [17], and for comparison purpose LWPR is tuned to hit a low target error (by decreasing size of RFs, changing learning rate, and penalty value) that is the target error hit by XCSF. Then XCSF's max population size is changed to be roughly equal to LWPR's number of RFs [17, 22, 20].

2.2 MapReduce

MapReduce is a programming model for processing large datasets. Programs written based on this programming model run on a cluster of nodes called MapReduce cluster. There are two kind of nodes in such a cluster: mappers and reducers. Mappers run part of the program called map procedure, and reducers run another part of the code called reduce procedure. All mappers and reducers run same code on different data. Mappers (map procedure) read the input data from hard disk of machine they are running on, and process the data to generate intermediate result. The data that is received by mapper is assumed to be as pairs of <key, value> (for example <filename, file content>, or <line number, content of the line>). Each mapper processes its part of data and generates the result as pair of <key, value>. The input <key, value> pair and output <key, value> pair might not have anything to do with each other. For example when input <key, value> is <filename, file content> the output <key, value> pair could be <word occurred most in the file, number of times the word occurred>.

One mapper might generate many pairs of <key, value> with different keys and values.

Paris of <key, value> generated by mappers then are sent to reducers for the next phase of process. <key, value> pairs are not sent randomly to reducers, and instead they are partitioned among reducers based on the key value of the pairs. For example from all <key, value> pairs that are generated by all mappers, those with key equal to key_1 is sent to a certain reducer.

Each reducer receives a group of <key, value> pairs generated by mappers and process them in order to generate the final result. Since map and reduce phases are run in parallel by all mappers, a large dataset that is distributed among cluster nodes is processed by the MapReduce framework much faster than it is possible to process it on a single machine.

2.2.1 Why <key, value> Pairs?

When data is processed by mappers, we need a way to aggregate result generated by different mappers. For example if the ultimate task is counting number of words starting with a, b, c, and d in a huge set of text files, each mapper could generate the result for the part of data it has locally, and we need a way to aggregate the result from all mappers. Having <key, value> pairs helps us to request all mappers to send count of all words starting with a certain character to a certain reducer in order to enable that reducer to have all partial results and calculate the final result. For this purpose all mappers would generate a result like <**a**, count of words starting with **a** in the text files a certain mapper has processed >, and all the results having **a** as their key would be sent to a certain reducer [3].

The key concept is that the programmer is aware of the way mappers need to generate result (pairs of <key, value>), and also aware of the way data is shuffled from mappers to reducers, and he needs to decide how to use this programming model to solve the problem he has at hand.

2.2.2 Is That All MapReduce Does?

So far we have talked about how MapReduce programming model helps us to solve data processing problem in a parallel fashion. But it is not all a MapReduce implementation offers us (Different MapReduce implementations are available among which we can named Hadoop [12] which is an open source implementation). When you write a MapReduce code you are done, and Hadoop (or any other MapReduce implementation) takes care of the rest of problems. The framework sends the mapper procedure to all mappers, and reducer procedure to all reducers. Then it asks the mappers to run the code on their local data and generate the result based on what is specified in the code. After result generation, the framework takes care of the shuffling the data among reducers. After reducers receive the data, it asks them to run the reducer procedure on the received <key, value> pairs.

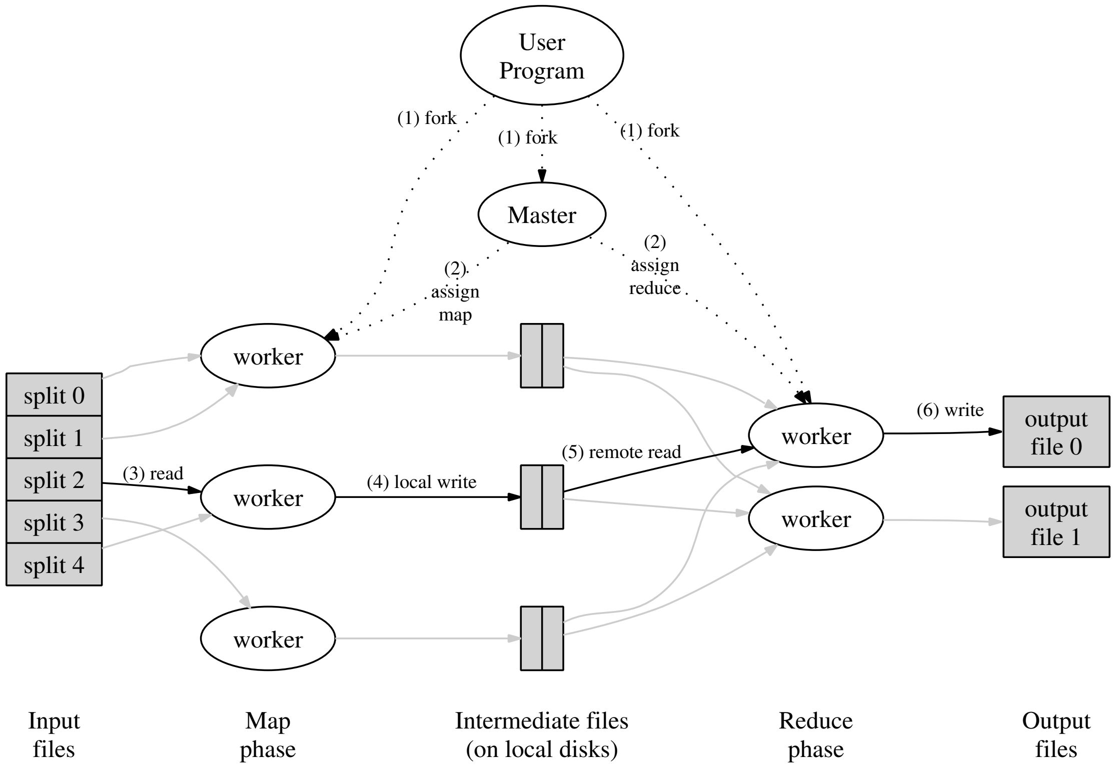

Figure 2.2: MapReduce Execution Overview [3].

A question here is how a programmer decides and copy the file on cluster nodes in order to be processed by MapReduce framework? Programmer does not need to do such a task. MapReduce framework has a distributed file system (Google File System or GFS and Hadoop Distribute File System or HDFS in Hadoop implementation of MapReduce) that facilitates this task. All you need is running the distributed file system and issuing a command like: copy bigFile.txt on the cluster. Rest of the work is done by the framework. Another question here is what if certain mapper fails in the middle of running? The answer is that MapReduce framework takes care of the issue. When distributed file system copies the data on the cluster, it replicates different chunks of data on different mappers (based on replication factor indicated in configuration file by user) and when a certain mapper fails, its task would be assigned to other mappers. MapReduce framework also takes care of other lower level tasks such as network communication. There are nodes in a MapReduce framework that their task is bookkeeping. They keep track of cluster nodes, mappers, reducers, data replication, etc.

Figure 2.2 illustrates execution of a MapReduce task on a MapReduce cluster. User program is distributed by master among worker nodes. Some of the worker nodes would work as mappers and some as reducers. Data is read by mappers and then they run the mapper procedure on the data. Intermediate data is generated, and then they are sent to reducer nodes. Reducer nodes process the <key, value> pairs they have received and generate the final result [3].

2.2.3 MapReduce for Clustering

One of the large-scale data processing that is data clustering. Several algorithms has presented recently for different clustering algorithms on MapReduce framework. In this section we review three clustering algorithm to see how they are using MapReduce power in order to cluster data.

Zhao et al. are arguing that all previous researches on parallel k-means so far are suffering from two problems [24]. First, they assume that all the data are in main memory, and second, they are using a restricted programming model. For these two reasons, those works are not applicable on peta-scale datasets. Since distance calculation (calculated n*k times in each iteration where n is number of data points and k is number of clusters) is the most expensive step of the algorithm, they try to exploit the parallelization of MapReduce to decrease this cost. Map function assigns each data point to its closest center, and Reduce function updates the centroids. There is one more function called Combine that aggregates the intermediate results of Map functions. A global variable called centers includes list of all centers and is used by all map tasks. Map tasks generate pairs of <ID of closest center, Data point>. Combine method, aggregates the results of the same map task. It calculates the partial sum of the data points assigned the same cluster. Output of this method is pairs of <Cluster index, Sum of the data points in the cluster and their count>. The Reduce function aggregates sum of all partial sums for each cluster, and calculates the new centroids. The output of Reduce is pair of <Cluster ID, Vector containing coordinates of centroid>. The speedup they have achieved for 4 machines is around 3 for the biggest dataset (8GB) which is a good speedup. The speedup for bigger datasets is bigger too which is a good indication. The authors are not talking about iterative nature of the algorithm and about how this issue is handled. They also do not talk about accuracy of the method and only talk about speed-up, scale-up, and size-up [24].

Ferreira Cordeiro et al. present an algorithm for very large multi-dimensional dataset clustering with MapReduce [7]. Since such a dataset doesn't fit in one or several disks, parallel processing is the only solution. In that case I/O and network cost are the two things that needs to be balanced. Best of both World (BoW) is the solutions that the authors are suggesting in this paper. They have worked on the largest real dataset ever in the database subspace clustering (Twitter crawl $>$ 12TB, and Yahoo! Operational data: 5 Petabytes - only reading 1TB from a modern 1TB disk takes around 3h). The contribution of the paper is combining sequential clustering algorithm with a parallelization method in an efficient way. Sequential subspace clustering algorithms can be plugged to this solution and the system would balance the I/O and network cost. The sequential algorithm that is plugged into the parallel algorithm finds the beta-clusters in a hyper-rectangle shape in the multi-dimensional space. Sequential subspace algorithm can be density-based or k-means-based [7].

I/O optimal version of the algorithm (ParC) reads the dataset one time and reduces the I/O. Another algorithm SnI (sample and ignore) improves the network cost but reads the data two times. Depending on number of reducers each of the two can be the winner. The BoW is a combined algorithm that decides to use which of those algorithm based on number of reducers, and keeps the cost as min(ParC, SnI) for any number of reducers. ParC partitions (using one of these methods: random, address space, or arrival order) the dataset across the cluster (mappers), finds beta-clusters in each partition (reducers), and finally merges the clusters (a single machine). SnI, on the other hand, first samples the dataset (exploits the skewed distribution of the data), and then clusters the sample using ParC ignoring the un-sampled data items. This way SnI avoids processing of many of the data items that belong to big clusters that are already sampled. SnI reads the data two times. In first read it samples the data, and in second read it only maps the sampled data items and avoids other points. The network cost will be reduced in a great amount by this technique. In sample step of the algorithm mappers map each point by probability of S_r to a single reducer. That reducer then clusters the data using the plugin clustering algorithm and passes the clusters description to next phase. In ignore phase each mapper reads its partition again and ignores the data points that fit into the clustering found in sample phase and send other data items to r reducers. Those reducers cluster the data points using the plugin clustering algorithm and pass the clustering description to one machine. That machine merges all the clusterings found in 2nd phase to the clustering found in phase 1 [7].

Both ParC and SnI have their own benefits. ParC optimizes I/O by reading the data file once, and SnI optimizes the network cost by reducing number of data points that needs to be transferred over the network in cost of reading the data file two times. To take advantage of the benefits of both of these methods we need a combined method that selects one of these based on the cost. A cost-based optimization method is used to select the better algorithm adaptively. The cost formula uses file size, network speed, disk speed, startup cost, and plugin cost to calculate the total cost for each algorithm. BoW algorithm first calculates both costParC and costSnI and select the better one based on the parameters and calls it. Experiments has been done to check the accuracy, scalability and performance of the cost-based method. The authors have shown that the quality of the clustering matches the quality of sequential clustering while its speed-up is close to linear. The cost-based method also has been shown to be the best of both world in all cases [7].

Ene A. et al. have designed the first approximate version of metric k-center and k-median algorithms for MapReduce [6]. They assumes that a set V of n data points and their corresponding distance is given and try to cluster the similar points into same clusters. The output of the algorithm is k data points that is considered to be the center points of the k clusters.The algorithms first sample the data (in a way that the sample represents all the data well) in order to decrease the dataset size. The sampling method incrementally add new points to the final sample set only if they are not already represented well by the final set. Sampling is different for k-median and k-center due to their different nature. Sampling for k-median needs more effort because it needs to consider each points distance from its cluster center. A version of algorithm is presented in the paper that can be run on MapReduce [6].

The MapReduce version of sampling is an iterative algorithm in each iteration of which we have three MapReduce operations. The first MapReduce operation partitions data arbitrarily among machines (mappers), and then each reducer construct two sets (S:final set, and H from which a pivot is selected). In next MapReduce step all the mappers pass the H and S sets to a single reducer and that reducer finds the pivot point. In the last MapReduce step mappers send pivot, S, one partition of R (remained data items that are not sampled yet), and the distance matrix to the reducers and reducers get rid of the well represented points. This steps are iterated until number of remaining points in the R falls under a certain threshold. K-center tries to minimize the maximum distance of the cluster center and the points in that cluster, while k-median tries to minimize sum of the distance of all the points in a cluster from the cluster center (both problems are know to be NP-hard). K-center uses the sampling produced by the sampling algorithm and mappers map all the points in the sampling along with their pairwise distance to a reducer, and the reducer runs a simple local clustering algorithm. K-median needs more information and its sample should have information about all the nodes that are to be clustered. For each un-sampled point the closest sample point is selected and its weight is increased by 1. In k-median first the sampling is done and then partitions of the original dataset along with the sample and part of the distance graph is sent to reducers. Each reducer finds weight of sample points partially. Then, in another MapReduce round the partial weights are summed up. Last step is a simple clustering on the sample considering weight of each sample point [6].

2.2.4 MapReduce and Iterative Tasks

Many machine learning and data mining algorithms are working iteratively on data but MapReduce is not well-suited for tasks with cyclic data flow. There are frameworks such as Twister [5], Spark [23], and HaLoop [1] that are iterative. Dave et al. present a cloud-based pattern for large-scale iterative data processing problems [2]. They have implemented CloudClustering, as a case study, that tries to show how iterative data processing problems can be handled on the cloud.

CloudClustering is the distributed version of k-means clustering algorithm, implemented on Microsoft's Windows Azure platform. They introduce a way to balance the performance-fault tolerance trade-off (that is the main trade-off when solving iterative problems on the cloud) using data affinity and buddy system. Some methods are using a central pool of state-less tasks in order to handle the fault tolerance issue, but this could lead to low performance because a cluster node might need to receive different parts of the data in different iteration (i.e. there is no affinity between data and workers) [2].

Windows Azure handles fault-tolerance by means of reliable queues. When a worker takes a task from the queue, the message becomes invisible and if it is not deleted after a timeout, it will reappear in the queue. This way if a worker fails, the task will be done by another worker node. One of the issues with the iterative tasks on the cloud is the stopping criterion. It can be handled in two different ways in this problem. If no data point is changed among clusters from one iteration to the next, we are done. This method needs to keep track of previous cluster of each data point. The other method checks the maximum amount of centroid movement and stops if it falls below a certain threshold. This method works on an read-only memory, but can't guaranty the convergence [2].

The proposed architecture is using the Windows Azure's queuing system and includes one master and a pool of worker nodes. Input dataset is stored centrally and is partitioned by the master. The workers download a task containing the address to the corresponding part of partition and the centroid list and perform the task. This method is working best in terms of fault-tolerance but since data affinity is not considered the performance is not good in this system. The other extreme is having one queue per worker that will solve the problem of data affinity (master will assign same partition of data to same workers in each iteration), but suffers from fault tolerance problem (there is not other worker to take over the current task in case the worker fails). Buddy system is grouping workers in buddy groups and a queue is shared among all members of each buddy group. Now size of the buddy group defines a balance between fault tolerance and performance [2].

2.2.5 Arguments about Using or not Using MapReduce

Schwarzkopf et al. have listed seven different assumptions and simplifications employed by researchers in the cloud research that threatens the practical applicability and scientific integrity of those researches [16].

One of the issues they have pointed out in their paper is unnecessary distributed parallelism. Very large datasets and frameworks such as MapReduce have made researchers to employ distributed parallelism more and more. Since the new high performance computing frameworks offer a fascinating simplicity and handle complicated issues like communication, synchronization, and data motion, a lot of people are willing to use these frameworks without considering whether these frameworks are useful for the problem at hand or not. Frameworks such as MapReduce reduce the engineering time needed to design a solution for a distributed version of an algorithm, but they mostly increase the runtime. For this matter the speedup of a program must be measured to show that the distributed solution outperforms the sequential solution. Furthermore, we need to make sure that we need to distributed the data over several machine even if we are sure that a parallel solution would be beneficial for the problem at hand. They also point out that as Rowstron et al. have shown, with nowadays multicore processors and huge amount of RAM we might not need to use a distributed solution for many problems [15], and we would be able to make use of fast communication mechanisms such as shared memory and also avoid data motion [16].

Another issue they have mentioned in their paper is forcing the abstraction. MapReduce is designed to alleviated the I/O bottleneck of big data by distribution of data over several hard disks. Time needed to process a job on a single machine is also assumed to be long. Some solutions are iterating and generating many short-time MapReduce jobs while it is better to have least number of jobs that are iteratively running on each system. Domain-specific systems (for stream processing, iterative processing and graph processing) have also emerged that seems to be a lot more justified that using the MapReduce for any problem [16].

Since many of Machine Learning and Data Mining algorithms are iterative, and MapReduce is not inherently an iterative programming model, and some other algorithms does not fit to this model for other reasons, many alternatives and extensions of MapReduce is provided by different research/industrial groups in recent years. Some theoretical studies have been done to show that Hadoop (an open source implementation of MapReduce) has limitations. Empirical studies also have been done and frameworks such as HaLoop [1] and Twister [5] are presenting a class of algorithms that Hadoop is not a good fit for, and try to extend the

Hadoop and solve the problem more efficient than Hadoop, and off course they outperform Hadoop at least when running that special algorithm. Jimmy Lin provides reasons why we need to either revise current algorithms to be run on MapReduce or devise new algorithms that follow MapReduce programming model. He suggests that since MapReduce is currently the widely used solution for large scale data processing problems, we can get rid of the iterative solution and try to use (or devise) alternative solutions that will fit MapReduce instead of devising new frameworks for algorithms that MapReduce is "good-enough". He discusses three classes of of problems to justify his claim: iterative graph algorithms (e.g., PageRank), gradient descent (e.g., for training logistic regression classifier), and EM (e.g., for k-means, and HMM training) [16].

Jimmy Line argues that extensions of Hadoop that support iterative constructs and thus alleviate some problems, but the problem with all these frameworks is that they are not Hadoop! It costs a lot for an organization to have another framework (other than Hadoop) for only graph and iterative algorithms. A better solution would be trying to solve the four above-mentioned problem by changing the algorithm in order to be runnable on Hadoop. If MapReduce is performing better than an alternative that is used to solve that problem. That does not mean that MapReduce needs to beat all the alternatives. For example MapReduce performing a lot better than GIZA++ for word-alignment algorithm, and also is considered an advance when used for k-means clustering [16]. The Hadoop stack is the standard and widely used platform for large-scale data analysis. Any large-scale data analysis needs to be able to process different types of structured and unstructured data and run different types of algorithms (graph, text, relational data, ML, etc.). No single programming model or framework can meet all the needs and be the best in terms of all the aspects such as performance, fault tolerance, expressivity, simplicity, abstracting low level features such as synchronization, etc.. No the question is: Dose adopting and deploying a new framework to solve a problem worth (in terms of cost, time, generality of framework, having mastered HR to use the framework, etc.) [14]?

Chapter 3

Approach

In this chapter we introduce two different piecewise regression algorithms. First algorithm is called MapReduce Regression Tree (MRRT), and second one is called Slope-changing algorithm. Both algorithms are trying to find a piecewise regression model for a dataset within the MapReduce framework. MapReduce Regression Tree algorithm is a Regression Tree based algorithm that can be used within the MapReduce framework. Slope-changing algorithm on the other hand is trying to introduce a non-greedy method to find good candidate split points and use this candidate set in order to find the final set of split points. Performance of these two algorithms is analyzed and compared in chapter 4.

3.1 MapReduce Regression Tree

Algorithm 2 lists the pseudocode for MapReduce Regression Tree algorithm. This algorithm partitions the feature space to smaller subspaces, but constructs a regression tree model (instead of a linear model in Slope-changing algorithm) for each subspace. The generated regression tree models are used to predict the target value of new data items.

Unlike Slope-changing algorithm that selects the split points based on the logic that maximum, minimum and slope-changing points are good candidates, this algorithm is not choosing the split points based on any heuristic, and the feature space is not divided into subspaces along different dimensions. The feature space is divided to subspaces of equal size (in terms of volume of the subspace and not number of data items in each subspace), and it is divided into smaller subspaces along one dimension of the feature space. This dimension is chosen randomly or using the *preProcess* method. The *preProcess* method is retrieving a sample of the dataset randomly (in our experiments we used 10% of each dataset) and runs the piecewise

Algorithm 2 MapReduce Regression Tree Algorithm - Main Method

```
1: function MR-REGRESSION-TREE-LEARN
2:     $dimToSplit = preProcess(dataset)$
3:     $rangeValues = Map1(dimToSplit)$        ▷ All mappers find min and max value of the
4:                                                    ▷ dimension that is being split
5:     $splitPoints = Reduce1(rangeValues, dimToSplit, nMappers)$        ▷ Split points are
   specified
6:                                 ▷ based on dimension size and number of mappers
7:     $Map2(splitPoints, dimToSplit)$                   ▷ Data is shuffled among reducers
8:     $models = Reduce2()$            ▷ Each reducer finds the model for the received data
9: end function
```

Regression Tree algorithm on all different dimensions of the dataset. One piecewise regression tree model is generated for each dimension of the dataset. The model is then tested against a validation set and the dimension corresponding to the dimension that has the least RMSE on validation set is chosen as the dimension to split the dataset in MapReduce Regression Tree algorithm along.

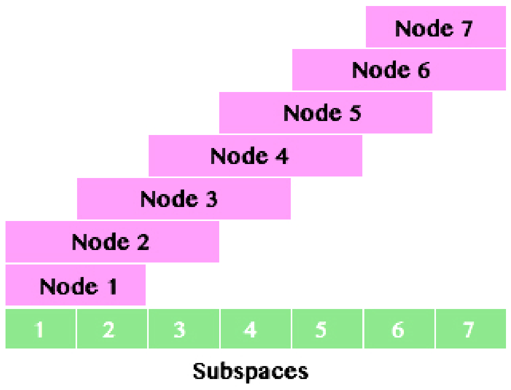

Figure 3.1: Dataset distribution among cluster nodes with overlap to decrease borderline data points prediction error

When dividing a feature space to subspaces, models constructed for two subspaces that are next to each other might have different predictions for a data point that is located on the borderline. It is same for data points that are located in one subspace, but are close to the borderline. For these data points, the neighbor model might have a better prediction than the actual model that the data point is located in. For this reason, smoothening methods try decrease the problem by using a weighted average of predictions of neighboring models for data points close to the borderline. Number of these models are two in a two dimensional feature space, and might be more in a n dimensional feature space based on the location of the data item (the data item might be close to a borderline in one dimension, and not in another dimension).

Since we are using a distributed method to solve the regression problem, we have more resources at hand and we might be able to afford a little bit of redundant calculation in order to increase the accuracy of the model. Based on this logic and the problem explained about borderline data points prediction, we decided to have overlapping subspaces and let each mapper have more data than what it needs in order to construct the model. Figure 3.1 depicts how 7 partitions of dataset that are partitioned along x axis are assigned to 7 mappers. All cluster nodes except first and last one receives three partitions of the dataset. All cluster nodes receive left and right partitions of the partition they are trying to construct the model for (we call it *main partition*, and call the left and right partition *neighbor partitions*). The reason that first and last nodes receive only two partitions instead of three is that their corresponding main partition has just one neighboring partition. Distributing dataset this way would let the system to construct the model based on 3 partitions but only predict the target value for the test items that are located in their main partition. This way we will not have any borderline data item and thus we will not need to use prediction smoothening methods that is used in Slope-changing algorithm.

Algorithm 3 MapReduce Regression Tree Algorithm - Map Phase of First MapReduce Round

```
1:  function MAP1($dimToSplit$)
2:      for all Mappers do
3:                  ▷ dataset is local part of dataset on the node that this mapper is running
4:          $minValue = +\infty$
5:          $maxValue = -\infty$
6:          for all $dataPoint \in dataset$ do
7:              $minValue = min(dataPoint[dimToSplit], minValue)$
8:              $maxValue = max(dataPoint[dimToSplit], maxValue)$
9:          end for
10:     end for
11:     send $< 1, < minValue, MaxValue >>$
12:                 ▷ By indicating key as 1 all information is sent to one reducer
13: end function
```

Figure 3.2 depicts different overlap factors when overlapping subspaces on cluster nodes. When overlap factor is 1, each node would receive its own subspace in addition to two neighboring subspaces that are as big as its own subspace. When overlapping factor is 0.5, size of

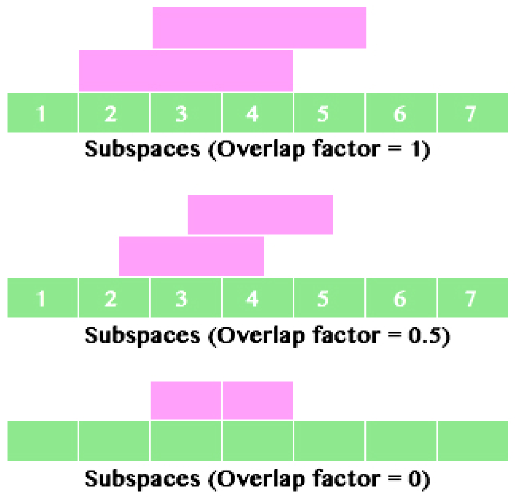
1
2
3
4
5
6
7
Subspaces (Overlap factor = 1)
1
2
3
4
5
6
7
Subspaces (Overlap factor = 0.5)
Subspaces (Overlap factor = 0)

Figure 3.2: Different overlap factors of subspaces on cluster nodes

neighboring subspaces that each node receives, is half of size of its own subspace. In case of overlapping factor of 0, no overlapping exists.

Now that we have discussed the concepts behind how MapReduce Regression Tree algorithm works, let us explain each line of algorithm 2 briefly. The $preProcess$ method selects what dimension to select to partition the dataset along. Then first round of MapReduce is started. In map phase of first MapReduce round, each mapper finds range of the data items (max, min) in the portion of dataset that mapper owns. These information are sent to one reducer. Reducer would receive $rangeValues$, and $nMappers$ (number of mappers), and would decide what portions of dataset should be sent to each mapper. In map phase of second round of MapReduce, all mappers receive the $splitPoints$ information and would send the data items they have to two or three mappers (we know that each partition of the dataset would be sent to several mappers due to overlapping). In reduce phase of second MapReduce round, all the reducers would have two or three partitions of the dataset. They construct a regression tree for the portion of dataset they have received. Each of this phases is explained in following sections.

3.1.1 $Map1$: Finding the Min and Max of Dimension that Is Being Split

Algorithm 3 lists the steps of this phase of the first MapReduce round. All mappers process the portion of the dataset they own and find the minimum and maximum value of the dimension

Algorithm 4 MapReduce Regression Tree Algorithm - Reduce Phase of First MapReduce Round

```
function REDUCE1($rangeValues$, $dimToSplit$, $nMappers$)
    $minValue = min(rangeValues.minValues)$    ▷ min value of $dimToSplit$ dimension
    $maxValue = max(rangeValues.maxValues)$    ▷ max value of $dimToSplit$ dimension
    $stepSize = (maxValue - minValue)/nMappers$
                        ▷ $stepSize$ in $dimToSplit$ dimension when partitioning the dataset
    $splitPoints[1].start = minValue$    ▷ $start$ and $end$ of the partition for
    $splitPoints[1].end = minValue + 2 * stepSize$    ▷ first mapper is calculated differntly
    for $i = 2, nMappers - 1$ do
        $splitPoints[i].start = minValue + (i - 2) * stepSize$
        $splitPoints[i].end = minValue + (i + 1) * stepSize$
    end for
    $splitPoints[nMappers].start = minValue + (nMappers - 2) * stepSize$
    $splitPoints[nMappers].end = minValue + nMappers * stepSize$
                        ▷ $start$ and $end$ of the partition for last mapper is calculated differntly
    send $splitPoints$ to all mappers    ▷ To be used by mappers of next round
end function
```

that is supposed to be split. All the mappers then send this minimum and maximum values to the same reducer. This is the reason the key value for the emitted $< key, value >$ pair is 1 for all mappers.

Algorithm 5 MapReduce Regression Tree Algorithm - Map Phase of Second MapReduce Round

```
function MAP2($splitPoints$, $dimToSplit$, $nMappers$)
    for all Mappers do
        for $dataPoint \in dataset$ do
            for $i = 1, nMappers$ do
                if $splitPoints[i].start < dataPoint[dimToSplit] < splitPoints[i].end$ then
                    send $< i, dataPoint >$ to corresponding reducer
                end if
            end for
        end for
    end for
end function
```

3.1.2 *Reduce*1: Finding Split Points Along the Dimension that Is Being Split

Algorithm 4 lists the reduce phase of first MapReduce round. In this algorithm all the maximum and minimum values sent by all mappers used to find the maximum and minimum value of the dimension that is being split. Using these two values, range of the dimension is found and *stepSize* of split points along that dimension is found by dividing range to number of mappers. Now *start* and *end* values for each mapper on dimension that is being split is found and stored in *splitPoints* array. All the mappers would have a partitions as big as triple of size of *stepSize*, except the partitions whose main partition is first or last partition of the dataset. Those two partitions would only have two partitions.

Algorithm 6 MapReduce Regression Tree Algorithm - Reduce Phase of Second MapReduce Round

```
1: function REDUCE2(dataPoints)
2:     Input: $dataPoints$: data items sent to this reducer
3:     for all Reducers do
4:         $models[i] = treeRegressionModel(dataset)$
5:     end for
6: end function
```

3.1.3 *Map*2: Shuffling the Data Among Cluster Nodes

The split points found in *Reduce*1 phase are used in this phase to shuffle the data. Algorithm 5 lists how the data is shuffled among cluster nodes in this phase. Each mapper sends each data item in its local portion of dataset to 2 or 3 mappers. The data points that are located in first or last partition of the feature space would be sent to two reducers, and all other data points would be sent to three reducers. This would cause to have more redundancy in the amount of the data that is transferred in the network, but would solve the problem of borderline data item's target value prediction, and would increase the accuracy of the model also.

Algorithm 7 MapReduce Regression Tree Algorithm - Prediction

```
1: function MR-REGRESSION-TREE-TEST($models$, $dataPoint$)
2:     $mainModel = findModel(dataPoint, models)$
3:     $leftModel = findLeft(mainModel, models)$
4:     $rightModel = findRight(mainModel, models)$
5:     $\hat{y} = \frac{1}{2}.predict(mainModel, dataPoint)$
6:     $\hat{y} = \hat{y} + \frac{1}{4}.predict(leftModel, dataPoint) + \frac{1}{4}.predict(rightModel, dataPoint)$
7: end function
```

3.1.4 *Reduce*2: Constructing the Tree Regression Models for Each Subspace

In this phase each mapper constructs a tree regression model for the data it has received from the mappers. Although this tree is constructed based on the data from three partitions, it only is used for prediction the target value for only the middle partition. In the simulation of the MapReduce that we have done in Matlab, *RegressionTree* class of Matlab library is used for modeling. fit method of this class constructs a regression tree with binary splits. *predict* method of the same class is used for prediction.

3.1.5 Using the MRRT Model to Predict

To predict the target value for a new data point using MapReduce Regression Tree algorithm, we use algorithm 7. This algorithm first finds the main model that should be used to predict the target value for data point. It then finds the neighbor models of the main model and uses all three models to predict the target value of the new data point. The weight that are used for weighted prediction are $\frac{1}{2}$ for the main model and $\frac{1}{4}$ for the neighbor models.

3.2 Slope-changing Algorithm

Slope-changing algorithm is a distribute algorithm that finds split points in a non-greedy way by help of many mappers. These split points are used to split the feature space into smaller subspaces. Data points in each subspace of the feature space is then sent to a different cluster node and each node generates a linear model for that subspace. When predicting the target value for a new data item using the constructed piecewise linear model, the corresponding subspace for that test item is first found, and then the corresponding linear model is used to predict the target value of the test item.

3.2.1 Choosing Good Split Points

This algorithm introduces a non-greedy approach for selecting split points. The reason behind trying to choose split points in a non-greedy way is that, greedy methods are just looking at what locally is the best choice and try to generate a globally optimum solution. We know that this might not lead to the optimum solution. As another reason, since the data is distributed among different nodes of a cluster, no single machine has access to all the data in the dataset, thus we need a method that makes each machine to extract some information from their partial subset of dataset, and use the extracted information to generate a set of good split points. It is worth mentioning that MapReduce and its corresponding distributed file system does not guaranty that data is distributed among cluster nodes in any special order, thus we might assume that data is distributed randomly among cluster nodes.

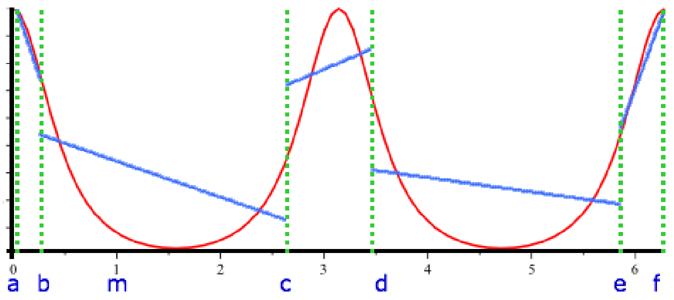

Figure 3.3: Bad split points causes bad piecewise linear models and higher prediction error

Figures 3.3 and 3.4 illustrate the difference between choosing good and bad split points. In both figures a, b, c, d, e, f are the split points that among each pair or points a linear model is fit. The curve line is the actual model, the vertical dashed lines are split points, and straight dashed lines are the piecewise linear models. As it can be seen, figure 3.3 has high prediction error in most points due to not choosing good split points. The point with highest error is m which has a very high prediction error. On the other hand, piecewise linear approximation in figure 3.4 is closer to real model because of choosing better split points. The point with highest prediction error in this model is n, and the point with highest prediction error on the left half of this model is m. The reason we have a better piecewise approximated model in figure 3.4 comparing to figure 3.3 is that the maximum and minimum points in the model is considered and they are chosen as split points. If we look at the left and right half of figure 3.4 more carefully we notice that left half has a lower average prediction error although right half is using the maximum and minimum points as its split points but left half is not using the minimum point as a split point. We can conclude that there are points other than maximum and minimum points that are also important. Those points are points on which we have higher changes in the slope of the model. **Slope-changing points** generaly include maximum and minimum points, but points other than maximum and minimum (in which the slope is changing sharply befor and after the point) also could be slope-changing points. Left half of figure 3.4 uses b and c as split points which are points with high change in slope of left and right part of the model at those points.

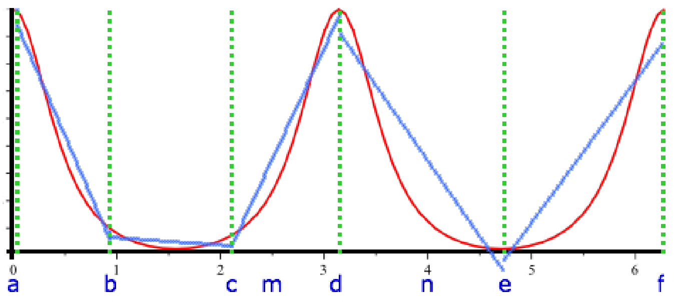

Figure 3.4: Good split points helps to have better piecewise linear models and lower prediction error.

3.2.2 Overview of the Algorithm

The dataset is assumed to be very large and also assumed to be distributed in a random way among cluster nodes. Two rounds of MapReduce is needed to generate the piecewise linear model. In the first MapReduce round split points are extracted and these split points are used in second MapReduce round to redistribute the dataset among cluster nodes. Afterward each cluster node would find a linear model for the portion of data it receives in the second MapReduce round.

Algorithm 8 Slope-changing Algorithm - Main Method

```
1: function Slope-changing-Learn
2:     $Initialize(dataset, m, r, p)$
3:     $candidates = Map1$                         ▷ candidate split points are extracted by each mapper
4:     $splitPoints = Reduce1(candidates)$         ▷ split points are chosen among candidate split points
5:     $Map2(splitPoints)$                         ▷ data is shuffled among mappers based on split points
6:     $models = Reduce2()$                        ▷ each reducer finds the model for the received data
7: end function
```

Algorithm 8 shows the pseudocode for Slope-changing algorithm. The Map phase ($Map1$) of first MapReduce round is done in parallel by all cluster mappers in order to find candidate split points. These candidate split points found by each mapper are all sent to one reducer in the shuffle phase of the first MapReduce round. Next the Reduce phase of first MapReduce round ($Reduce1$) is run and generates the final split points set using the candidate split points received from all mappers. In map phase of second MapReduce round ($Map2$), all the nodes map each data item to a key based on the split points. This way all the points in a grid-cell of the grid (specified by the split points) will be sent to the same reducer. In reduce phase of this MapReduce round ($Reduce2$), each reducer would have all the data points pertaining to a grid-cell of the grid, and would be able to simply generate the linear model for that grid-cell. Details of each of these four phases are explained in following sections.

Algorithm 9 Slope-changing Algorithm - Initialization

```
1: function INITIALIZE($dataset$, $m$, $r$, $p$)
2:     $nDim = dataset.numOfDims$                          ▷ Database dimension
3:     $nMappers = m$                                      ▷ Number of mappers
4:     $nReducers = r$                                     ▷ Number of reducers
5:     $minPointsPerCell = nDim + 3$  ▷ Minimum number of data points in each grid-cell
6: end function
```

3.2.3 *Map*1: Finding Candidate Split Points

As it is explained in section 3.2.1, maximum, minimum and slope-changing points in a model are good candidate split points. If all the data is available on a single machine, we could find all candidate split points by searching the whole dataset and feature space exhaustively. This might work, but is not an efficient way to do the task. As a more efficient way to find this set of points, we can use a randomized way and use hill-climbing method to find most of these points. In this case we might not find all the candidate split points, but we will end up having a candidate set using a more efficient algorithm.

Algorithm 10 Slope-changing Algorithm - Map Phase of First MapReduce Round

```
1: function MAP1
2:     for all Mappers do
3:             ▷ dataset is local part of dataset on the node that this mapper is running
4:         $grid = Gridify(dataset)$
5:         $candidates.maxPoints = findMaxPoints(grid, dataset)$
6:         $candidates.minPoints = findMinPoints(grid, dataset)$
7:         $candidates.slopeChangings = findslopeChanging(grid, dataset)$
8:     end for
9:     send $< 1, candidates >$
10:            ▷ By indicating key as 1 all candidate split points are sent to one reducer
11: end function
```

One way to find a data point with local maximum of target value in the dataset is to choose a random data point, and then find k of its nearest neighbors. Then use these k nearest neighbors and find the one with highest target value. Repeating the same procedure for the new maximum point would end up a case that the maximum point would not change anymore. In that case we have found a data point with local maximum of target value. In the extreme case when k is very large, we would end up finding global maximum. When k is small, we would end up finding local maximum points. If $k = 1$ then we will end up the very local maxima that is the randomly selected point. There are two problems with this method. First finding k nearest neighbor would need additional data structures such as K-Dimensional Tree (KD tree), and also time to construct and search this data structure. Second problem with this method is finding a proper value for k, and we know that same value of k would not work best for every dataset.

To find the points with maximum and minimum target value in a more efficient way, we can bucket the data points in cells of a grid. Then we can start from a random data point, and find the cell containing that data point. Afterward we find the neighbor cells of that cell and find the data point that has the highest target value among all data points in all neighboring cells. We repeat this step for the last maximum data point until the maximum data point is not changing anymore.

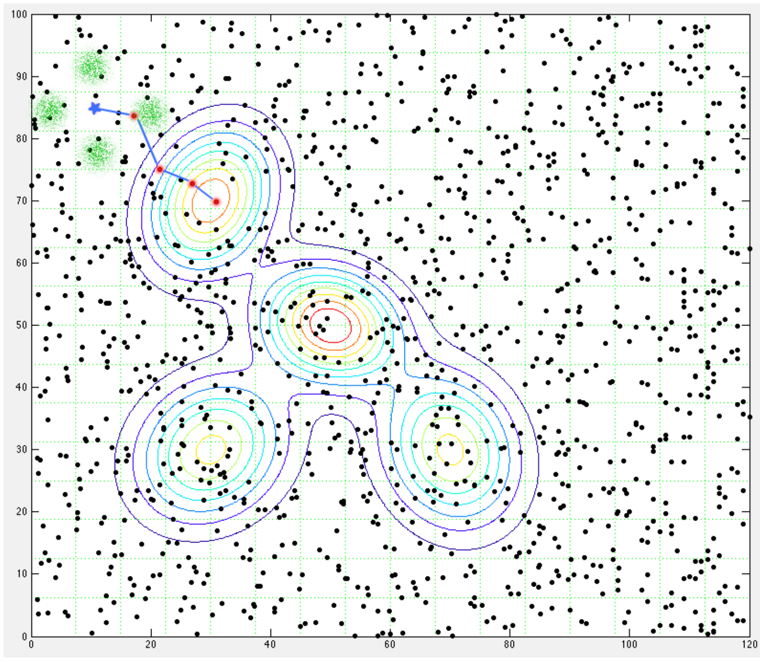

Figure 3.5: Finding the data points with maximum target value by gridifying data points and using a initial random seed

Gridifying the Feature Space

When we gridify the data points we need to decide about how big the grid-cells need to be. For this purpose we first need to decide on how many data points we would like to have in each grid-cell in average. In our experiments we have chosen to have *number of dimensions of dataset + 3* data points in each grid-cell. The reason for this decision is that we need at least *number of dimensions + 1* data points in each grid-cell in order to be able to find a linear regression model for the data points in that grid-cell. But we know that it is the least possible number of data points and having more data points would help to have a model of higher quality. Based on this logic we have used following equations in order to find the grid split points in each mapper:

$$x = \sqrt[nDim]{\frac{size(dataset)}{2 * nDim}} \tag{3.1}$$

$$stepSize = \frac{maxPerDim(dim) - minPerDim(dim)}{x} \tag{3.2}$$

In above equations $nDim$ is number of dimensions in the dataset, $size(dataset)$ is number of rows in the portion of dataset a certain mapper owns. x is total number of grid-cells in the grid. $maxPerDim(dim)$ and $minPerDim(dim)$ are calculated for each dimension of the dataset and are the maximum and minimum value for that feature (dimension) of the dataset. If we start from $minPerDim(dim)$ and add multiples of $stepSize$ to it until we reach to $maxPerDim(dim)$, we will find all the borderlines of the grid-cells for that dimension.

Finding Maximum Minimum Candidate Split Points

Figure 3.5 illustrates the randomized method for finding these points. A gaussian mixture model (indicated by contour map) is the actual model in this figure. Black dots in the figure are the data points among which we are trying to find data points with local maximum target value using a randomized method explained above. The data is first gridified (dashed green lines are the borderlines for grid-cells) based on the explained logic, and then a random data point is chosen. The initial random data point is indicated by a blue star and is located in cell $(10, 85)$. Next, algorithm finds all neighboring grid-cells of the grid-cell containing the initial data point. In case of two dimensional feature space, every cell has four neighboring cells. Four neighbors of this cell are highlighted by green color. Target value of all the data points in these four cells are compared with the target value of current maximum data point (currently the initial random data point) and the data point with highest target value is chosen. This data point is located in right cell of the current cell in figure 3.5, and the data point is highlighted by red color and is connected to the original data point by a blue line. This process is repeated for the new data point. Other blue lines connecting red data points are showing the next maximum data points and the final maximum data point is located in cell $(35, 73)$. When the current maximum data point is this data point, finding the four neighboring cells and comparing the current maximum with the data points in those cells will not change the maximum data point anymore. This way finding one local maximum target value is done and we can repeat the same process to find more local maximum data points.

Finding the minimum candidate split points is done in a similar way as maximum candidate split points, except we choose the data points with lowest target value among all data points in the neighboring cells. Method calls $findMaxPoints(grid, dataset)$ and $findMinPoints(grid, dataset)$ in algorithm 10 are calling methods that are finding maximum and minimum data points in the same way explained above.

Finding Slope-changing Candidate Split Points

Finding the slope-changing points in $findslopeChangings(grid, dataset)$ method call of algorithm 10 is also done using a randomize method. We first explain the method for a $2-dimensional$ feature space and then would extend it to an $n-dimensional$ feature space. We first randomly choose if we want to find a high slope-changing point in x or y direction. Suppose x direction is chosen. Now we randomly choose one of the rows of the grid-cell depicted in figure 3.5 (assume row r is chosen randomly). We start from the first cell in that row and calculate the following formula for all grid-cells of the row in an array, say $slope$:

$$slope_i = \frac{\bar{y}_{i,r} - \bar{y}_{i-1,r}}{\bar{x}_{i,r} - \bar{x}_{i-1,r}} \tag{3.3}$$

In this equation $\bar{y}_{i,r}$ is the average target value of all the data points in grid-cell with coordinates (i, r). This way we find the slope of a line connecting the data points in two neighboring grid-cells using the average values of target value and x. Now we can use the calculated values in $slope$ array in order to find out how the slope of the model before and after a cell is changing. We cannot simply calculate the difference of the slope value found for the left and right of a cell, because in high degrees (for example $> 88°$) the degree difference of $1°$ would cause to have a very big slope change, while the same change in degree for small degrees (for example $< 45°$) would cause a very smaller change in slope. For this reason we need to find the corresponding angle to a slope value and find the change in angle before and after a cell in order to decide if the change is significant or not. Here you see that we are not using hill-climbing as we did for maximum and minimum. Hill-climbing would not work for slope-changing points, because having a change in slope on the right of a cell would not tell us anything about future changes in slope in that direction. We keep list of all the grid-cells that the slope change of their right and left cell is bigger than a threshold (the threshold we have used in our experiment is $15°$) and pass the average value of all the data points of those grid-cells as candidate split points to the reducer. We also have assigned a quality to each candidate slope-changing point. A slope-changing point that the difference of slope on its right and left is higher, would have a higher quality.

The explained method above would be repeated several times in each mapper and each time we would select a row or a column (in the $2 - dimensional$ feature space) and find all the candidate split points in that row or column. In $n - dimensional$ feature space instead of choosing a row or a column, a dimension is chosen randomly and random numbers are generated for all other dimensions. Those values for other dimensions are kept constant and the only value that is changing is the value for the chosen dimension. Same method as explained above is used to find the slope for that dimension and the candidate data points are chosen accordingly.

Using Several Mappers to Find Candidate Split Points

In a MapReduce based algorithm, data is not all on one machine and different machines has different parts of data at hand. In this case we can find a set of candidates for each mapper using the same method explained above. After finding this set of candidate split points in each mapper, all the candidate data points are sent to one reducer. The process to find a set of split points from this candidate set, would be described in next section.

We need to point out here that although the randomized method of candidate selection would not be able to find as many as candidate data points as exhaustive method, but in this case that we use a lot of mapper to find candidate split points in parallel, it is very likely that some mappers find some candidate data points from parts of the feature space that other mappers do not find candidate points from that part of the feature space because of the nature of the randomize method used. This way the union set of all candidate split point sets of all mappers would be more likely to include all possible candidate split points. It also is possible that the union set includes many candidate split points for a small portion of the feature space, because many mappers have found candidate split points for that area. In this case the next phase ($Reduce1$) would solve the problem.

Algorithm 10 lists the process explained in this section. Each mapper would work on its local part of the dataset and gridify it in a grid first. Next it finds maximum, minimum and slope-changing candidate points. Finally each mapper sends the generated set of points to the reducer. Since all the candidate split points needs to be sent to same reducer the key for the key-value pair of the result of mapper is indicated as 1 for all mappers.

3.2.4 $Reduce1$: Generating a Split Point Set from Candidate Set

Now we have all the candidate split points and would like to choose a subset of these candidate points. Since different mappers might have found maximum, minimum, and slope-changing points that belong to same maximum, minimum and slope-changing points in the real model, we need to remove some of the candidate split points that are redundant and different mappers have found them only because of having a subset of data points in the dataset. Since the population of the candidate split points in some parts of the feature space might be high, we need to use a method to merges candidate split points in highly populated areas, and also try to select split points from those areas. We assume that if several mappers have found candidate split points in a small subarea of the feature space, those small subareas are important areas and potentially good places to have split points. Based on this logic, we have used two different methods for selecting split points that are explained in two following sections.

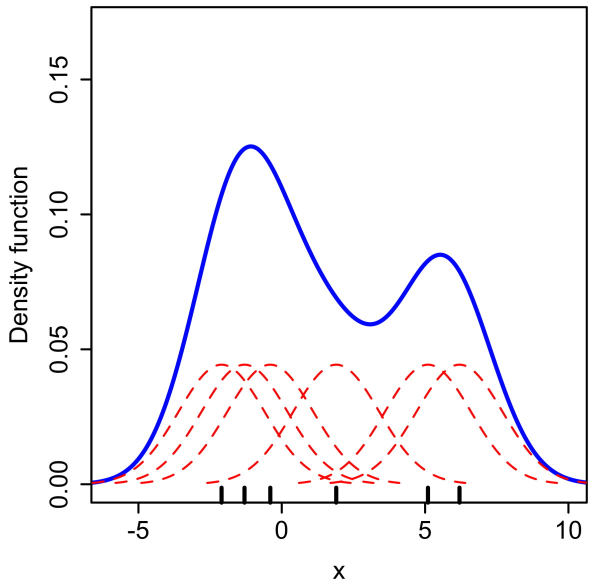

Figure 3.6: Using Parzen Window Classifier to find areas with many candidate split points [18]

3.2.4.1 Selecting Split Points Using Density Estimation

To take the highly populated areas into account, we can use a method similar to Parzen Window Classifier [4] to find the areas with high density of candidate split points, and chose points from those areas as the split points. We use a gaussian kernel with a kernel spread of two times of the distance between closest candidate split points. After calculating the density estimation at all candidate split points, all the points with an estimated value which is a local maximum in its neighborhood are chosen as a split point. Figure 3.6 shows how Parzen

Window Classifier works. In this image the dashed gaussian curves are centered at each data point. Data points themselves are shown by a small line on the x axis. The solid curve is sum of all dashed curve and shows us which points on x axis is more populated. This density estimation technique is applied separately on images of all candidate split points on different axes. The reason is that performing the density estimation on different axes separately is more efficient and the outcome would be same if we perform the density estimation on all axes, and then map the outcome on each axes.

Algorithm 11 Slope-changing Algorithm - Reduce Phase of First MapReduce Round (Parzen Window Classifier Version)

```
function REDUCE1($candidSplitPoints$)
    for each dim of the $dataset$ do                    ▷ for each dimension of the dataset
        $splitPoints[dim] = [\ ]$
        $candidSplitPts = candidSplitPoints[dim]$
        for $i = 1, size(candidSplitPts)$ do
            $candidSplitPts[i].density = pwcEstimate(candidSplitPts[i], candidSplitPts)$
        end for
        for $i = 1, size(candidSplitPts)$ do
            if $candidSplitPts[i].density > candidSplitPts[i-1].density$ then
                if $candidSplitPts[i].density > candidSplitPts[i+1].density$ then
                    $splitPoints[dim].add(candidSplitPts[i])$
                end if
            end if
        end for
    end for
    send $splitPoints$ to all mappers                  ▷ To be used by mappers of next round
end function
```

Algorithm 11 list the code to apply Parzen Window Classifier to extract the split set from candidate split points set. The first **for** loop is calculating the density estimation of all the data points in the candidate set on each of the candidate split points. Next we choose the candidate split points that are bigger than their neighbors (are local maximum in their neighborhood), and add them to list of split points. This list split points is used in next round of MapReduce in order to partition the data among mappers and find the linear model for each subspace of the feature space constructed by this split points.

3.2.4.2 Selecting Split Points Using Fitness Proportional Selection

As explained in section 3.2.3 different slope-changing points are assigned different quality values based on the amount of slope change we have in the left and right side of the point. Algorithm 11 only uses the density to select the split points. In another word, algorithm 11 only considers quantity of the candidate split points to choose the split points and is not considering the quality of split points. This algorithm intends to consider both quality and quantity when choosing split points from the candidate set. We assume that the slope-changing points with higher quality are better candidate as split points.

Algorithm 12 Slope-changing Algorithm - Reduce Phase of First MapReduce Round (Fitness Proportional Selection Version)

```
 1: function REDUCE1($candidSplitPoints$, $n$)    ▷ $n$ is number of split points we would like
    to select for each dimension
 2:     for each dimension of the $dataset$ do
 3:         $splitPoints[dim] = [\,]$
 4:         for $i = 1, n$ do
 5:             $point = randFitnessProportional(candidSplitPoints, candidSplitPoints.fitness)$
 6:             $splitPoints[dim].add(point)$
 7:         end for
 8:     end for
 9:     send $splitPoints$ to all mappers             ▷ To be used by mappers of next round
10: end function
```

Fitness proportional selection algorithm is a randomized way of selecting split points that considers both quality and quantity. It gives the points with higher fitness value (quality) higher chance to be selected, and moreover highly populated areas would have more chance of having candidates to be selected in the final set of split points. In case of this version, we need to decide about number of split points that we would like to be chosen for each axis. These numbers are found using same equation as equation 3.1. The only difference is that $size(dataset)$ would be size of the actual dataset, and not size of local dataset of a certain mapper. Algorithm 12 lists the pseudocode for this version of split point selection, and again this algorithm needs to be run for each dimension separately. In this algorithm $randFitnessProportional$ receives the candidate split points and their corresponding fitness value, and returns a point randomly proportional to fitness values.

Algorithm 13 Slope-changing Algorithm - Map Phase of Second MapReduce Round

```
1: function MAP2($splitPoints$)
2:     for all Mappers do
3:         for $dataPoint \in dataset$ do
4:             $subSpace = findSubspaceID(dataPoint, splitPoints)$
5:             send $< subSpace, dataPoint >$ to corresponding reducer
6:         end for
7:     end for
8: end function
```

3.2.5 $Map2$: Shuffling the Data Points Based on Split Points

After finding the split points in first round of MapReduce, map phase of second round of MapReduce is run by all mappers. All mappers read the local part of the dataset they have and find the proper subspace each data item belongs to. All the data items belong to a certain subspace should be send to a certain mapper. These data items might be distributed among all mappers (we explained before that MapReduce distributed file system does not guaranty distribution of a file in any order), and all mappers would send the data items pertaining to a certain subspace to the same reducer. Algorithm 13 lists the pseudocode for this operation that needs to be done by all mappers.

Algorithm 14 Slope-changing Algorithm - Reduce Phase of Second MapReduce Round

```
1: function REDUCE2(dataPoints)
2:     for all Reducers do
3:         $dataset$ = data items sent to this reducer and are in subspace $i$
4:         $models[i] = linearRegressionModel(dataset)$
5:     end for
6: end function
```

3.2.6 $Reduce2$: Finding the Linear Model for Each Subspace

Each of the reducers or second round of MapReduce receives the data items pertaining to a certain subspace of the feature space. Since the subspaces are limited by split points found in first MapReduce phase, we assume that each subspace contains data items that could be modeled by a linear model. The data points are used to find the linear regression model for each subspace. The algorithm for finding the linear regression model of these data points in each subspace could be any linear regression algorithm. In our MapReduce simulation that is done using Matlab, we have used *regress* library method to find linear regression model. If number of subspaces is more than number of reducers, some reducers need to compute the linear regression model for more than one subspace. Because of data sparsity in the dataset, it also is possible that some subspaces of the feature space do not have enough data points to construct the linear regression model. In this case we do not construct a model for that cell and use the neighboring models to predict the target value of a data item that sits in that cell. Algorithm 14 lists the pseudocode for this phase.

Algorithm 15 Slope-changing Algorithm - Prediction

```
function SLOPE-CHANGING-TEST($models$, $dataPoint$)
    $model = findModel(dataPoint, models)$
    if $model \neq null$ then
        $\hat{y} = predict(model, dataPoint)$
    else
        $models = findNeighboringModelsFor(dataPoint, models)$
        if $models \neq null$ then
            $\hat{y} = predict(models, dataPoint)$
        else
            $\hat{y} = globalConstantValue$
        end if
    end if
end function
```

3.2.7 Using the Slope-changing Model to Predict

Algorithm 15 lists the test method of the model generated by Slope-changing algorithm. Prediction is performed for a new data point. The data item is used to extract the corresponding model. We know that based on feature values of a data item, it might land in a certain subspace of the feature space. Finding that subspace means that we have found the corresponding model to predict the target value of the test item. It is probable that the corresponding model of that subspace is not generated by the Slope-changing algorithm due to not having enough data items in that subspace when generating the model (it might be cause because of the data scarcity in that part of the feature space, or having a very small subspace of the feature space because of the closeness of the split points). In this case the neighboring models of that subspace is extracted and used to generate a prediction for that data items. It is unlikely, but probable, that all of those neighboring subspaces also do not have model due to same problems mentioned before. In this case a global constant value (average value of all the data items in the dataset) is used as the prediction for the test item.

3.2.7.1 Smoothening the prediction

When creating a model for one cell of the grid, the prediction for data points on the borders of the cells could be done with each of the models that share the border. When the data point is close to the border, it also is possible that the neighbor model has a better prediction then the model of the grid-cell containing the data point. For this reason we use a smoothening method for the data points near the border of the cells. If a data point is in the $\frac{1}{4}$ distance of the border ($\frac{1}{4}$ of the length of the grid-cell in that dimension), we use the neighbor model in that dimension to smoothen the prediction. In this case the prediction would be the weighted average of main model (by $\frac{2}{3}$ weight), and neighboring model (by $\frac{1}{3}$ weight).

Chapter 4

Empirical Evaluation

In this chapter we will evaluate performance of proposed algorithms we discussed in chapter 3. We first define the evaluation criteria we will use to compare these algorithms in section 4.1 and afterward we will describe the synthetic and real datasets on which the experiments are performed in section 4.2. Results of the experiments are provided and analyzed in section 4.4 and 4.5.

4.1 Evaluation Criteria

To measure the performance of algorithms, we try to answer to the following questions:

- What is the accuracy of distributed algorithms and what is the effect of distributed processing on model accuracy?
- What is the speedup of the distributed algorithms comparing to when they are run on a single machine?
- How does distributed algorithms perform when size of the dataset is increasing? Are they more scalable than sequential algorithms?

We use three different measurements to measure the results when trying to answer above questions.

4.1.1 Accuracy

We use RMSE to measure accuracy of an algorithm. RMSE is calculated using the following equation [21]:

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

In above equation ϵ_i is the difference between the actual and predicted value of target value (it is introduced in section 2.1.1) for i^{th} data item in the test. To measure the RMSE of an algorithm on a test set, we first construct the regression tree or piecewise regression model by training the corresponding algorithm on training set. Afterward we use a test set (containing data items that has not been used in training phase) to measure the RMSE of the algorithm. As it is obvious in equation 4.1, RMSE is measured by calculating the squared root of average of sum of square of difference of the predicted target value by model and the actual target value of data item in the test set.

4.1.2 Speedup

In this case we aim to find out how faster an algorithm running on a MapReduce cluster with 64 mappers and n reducers constructs a regression model for a dataset compared to when same algorithm is run on a single machine. This metric is measured as following:

$$speedup_n = \frac{t_{seq}}{t_{mr}} \tag{4.2}$$

In above equation t_{mr} is the time needed to construct the regression model on a MapReduce cluster and t_{seq} is the time needed to construct the regression model on a single machine. Same algorithm that is used to generate the MapReduce model is also used to generate the model on a single machine. In our simulation, we run the simulated MapReduce version of the distributed algorithm on a single machine too (all map and reduce phases are run sequentially on a single machine). To measure the runtime of map (reduce) phase we measure the maximum time of all the mappers (reducers) in the map (reduce) phase. Time overhead in data communication between map and reduce phases is not estimated in our simulation. Therefore, our speedup results are overestimates and can be considered the upper bound of speedup we can achieve in practice. For both single machine and MapReduce run of the program we measured and use wall clock time to calculate the speedup.

4.1.3 Scalability

It is one of the most important metrics to measure quality of distributed algorithms. In our experiments scalability tells us how our algorithms perform when size of a dataset that is used as the training set increases. When measuring scalability we need to fix number of nodes in the cluster and change size of the training set. This way by looking at changes in time needed to construct the model by two different algorithms we would be compare their scalability.

4.2 Overview of Datasets

4.2.1 Real Datasets

The real datasets are borrowed from UCI Machine Learning repository. The borrowed dataset that our datasets are generated from is called IHEPC (Individual Household Electric Power Consumption) [9]. This dataset contains 2,075,259 data items and 9 features. We have chosen randomly 4,111 data items (out of all data items in the dataset) as test set, and rest of data items are considered as training set.

This dataset has some missing values (nearly 1.25% of the rows) for which we have used average value of all data items from beginning of the dataset to the missing value, as its value (average value of same feature in other data items is used to populate a missing feature).

Table 4.1: Summary of Real Datasets

Dataset	Model Type	Axes	Predicted Feature	Training size	Test size
$IHEPC_1$ [9]	Real Dataset	9	*sub_metering_3*	2,071,148	4,111
$IHEPC_2$ [9]			*sub_metering_2*		
$IHEPC_3$ [9]			*sub_metering_1*		

Although the original dataset has 9 features (including target feature), we have dropped the first feature (*date* feature) due to its irrelevance to the target feature. Thus the dataset includes 8 features from which one is target feature. Second feature of the dataset (*time* feature) which is time of the day in $hh : mm : ss$ is also converted to an integer which is equal to number of minutes passed from midnight. For example value $17 : 24 : 05$ is converted to 1044 (only hour and minute is considered and second is dropped).

Three features of this dataset called *sub_metering_1*, *sub_metering_2*, and *sub_metering_3* could be used as the target value of the dataset to predict. Thus we decided to use this dataset to generate three datasets called $IHEPC_1$, $IHEPC_2$, and $IHEPC_3$, by target features *sub_metering_3*, *sub_metering_2*, and *sub_metering_1* correspondingly. Information about the real datasets are summarized in table 4.1. We will use dataset name (first column of the table) to refer to each of these datasets in results and analysis sections.

Table 4.2: Summary of Synthetic Datasets

Dataset	Model Type	Axes	Training size	Test size
ptoy10d1	Polynomial	10	100,000	1,000
ptoy10d2	Polynomial	10		
ptoy20d1	Polynomial	20		
ptoy20d2	Polynomial	20		
ttoy10d1	Trigonometry	10		
ttoy10d2	Trigonometry	10		
ttoy20d1	Trigonometry	20		
ttoy20d2	Trigonometry	20		
gtoy10d1	Gaussian Mixture	10		
gtoy10d2	Gaussian Mixture	10		
ttoy10d3	Trigonometry	20	1,000,000	1,000

4.2.2 Synthetic Datasets

We have generated 11 different synthetic datasets. We used three different class of models (Polynomial, Trigonometry, and Gaussian Mixture) to generate 10 and 20 dimensional datasets. Each of the generated datasets has 100,000 training items and 1,000 test items. One of the 20-dimensional trigonometry datasets has 1,000,000 training items, but still has 1000 test items in its test set. This dataset used same model as ttoy10d2, but only has more data items in its training set. Summary of the information about these datasets is presented in table 4.2. First column of this table includes the datasets' names that will be used to refer them in results and analysis sections. The functions used to generate these synthetic datasets is listed in appendix A.

4.3 Overview of Experiments

All experiments are performed on a MacBook Air machine with Intel Core i5 processor and 4GB of DDR3 Ram and SSD hard drive. The machine runs Mac OS X 10.7.5. Three algorithms are analyzed and compared in this chapter:

- **Regression Tree algorithm** that is running sequentially on a single machine referred as **baseline** algorithm in experiments. We compare the accuracy results of Slope-changing and MRRT algorithm with this algorithm. It also is used in scalability, speedup and prediction speed analysis. We also compare the MRRR(*S*) algorithm with the baseline algorithm. MRRT(*S*) is explained in the following paragraph (here * is a wildcard character).

- **MapReduce Regression Tree algorithm (MRRT)** simulated in Matlab is the other algorithm. In case of this algorithm the code for mappers and reducers are run sequentially and the highest time among mappers and reducers is considered as the map and reduce time (mappers are run in parallel when running on a real MapReduce framework). As described in section 2.2, in real implementations of MapReduce framework (such as Hadoop), data are shuffled among machines between map and reduce phases. This shuffling needs communication time that we are unable to measure in a simulated version of algorithms. Thus we will use t_{com} to indicate communication time needed for shuffling when we present and discuss the results.

  We experiment different versions of MRRT algorithm. Each version could be a combination of *Weighted*, *Overlapped*, and *Sequential* features. These features of the algorithm are described in section 3.1. The weighted version uses prediction of neighboring models in addition to prediction of the main model in order to predict the target value. In overlapped version, subspaces of the feature space are overlapped in order to decrease the borderline data points' prediction. Sequential version is when we run the algorithm on a single machine instead of a MapReduce cluster. These different features could be combined in an experiment. For example MRRT(WO) means a version of MRRT that uses overlapped subspaces of the feature space to construct the model, and also generates a prediction that is a weighted average of prediction of the model for the subspace that data points is located in, and prediction of the neighboring models of that model.

- **Slope-changing Algorithm** simulated in Matlab is the third algorithm we analyze in this chapter. This algorithm also need shuffling time that is not possible to measure in a simulated version of the algorithm, thus we use t_{com} to indicate communication time in this algorithm. The estimated time for this algorithm is also measured like MapReduce Regression Tree algorithm by considering highest map and highest reduce time in map and reduce phases. This algorithm is not currently scalable enough to work on datasets with dimensions more than 5, thus we do not compare this algorithm with the MRRT algorithm that works on high-dimensional datasets. This algorithm is compared to baseline algorithms.

We would perform different experiments in order to analyze accuracy, speedup, and scalability of different algorithms. We also run some experiments in order to compare different versions of MRRT algorithm.

4.4 MRRT Experiment Results

4.4.1 Number of Dimensions to Split Along

Dividing the feature space into subspaces can be done in different ways. We could divide it along one dimension, or divide it along different dimensions simultaneously. For example for a 2-dimensional feature space, we could divide the feature space into smaller rectangular subspaces (dividing along both dimensions of the feature space) or we could divide it only along one dimension and generate parallel strips of subspaces in the feature space (see figure 4.1). Since one model is constructed for each of feature space's subspaces, the way we divide the feature space would affect the quality of model. In this experiment we try to find out whether dividing along one dimension or dividing along several dimensions would generate a model of higher quality.

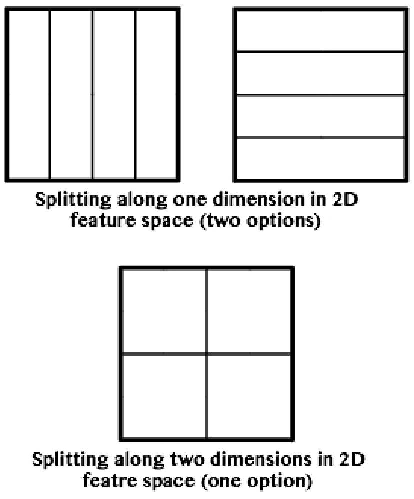
Splitting along one dimension in 2D feature space (two options)
Splitting along two dimensions in 2D featre space (one option)

Figure 4.1: Splitting the feature space to subspaces.

This experiment is done on all synthetic and real datasets. To perform the experiment first divided the feature space of each dataset along first dimension and measured learning time and also accuracy of generated model over the test set. Next we divided the feature space of each dataset along first and second dimensions (two dimensions) simultaneously. Number of cluster node is fixed to 64 for both division methods of this experiment. We used neither overlapped nor weighted version of the MRRT algorithm in both cases.

Table 4.3: Comparing accuracy and learning time of MRRT when dividing the feature space along one dimension versus two dimensions on synthetic datasets. As it can be seen, none of the methods for dividing the feature space is superceeding the other one and there is no obvious reason to prefer one over the other one based on this experiment. The learning time of algorithms in both methods is also similar.

	One Dimension		Two Dimensions	
Dataset	MRRT RMSE	Learning Time	MRRT RMSE	Learning Time
gtoy10d1	46511.14	0.28	**38366.02**	0.28
gtoy10d2	39614.44	0.29	**25655.23**	0.27
ptoy10d1	23.83	0.35	23.31	0.26
ptoy10d2	303.71	0.46	**283.01**	0.31
ttoy10d1	**345.23**	0.32	381.87	0.39
ttoy10d2	**118.50**	0.30	122.99	0.31
ptoy20d1	**13.69**	0.44	14.30	0.37
ptoy20d2	**147.48**	0.23	159.60	0.23
ttoy20d1	800.41	0.58	**780.07**	0.45
ttoy20d2	**576.87**	0.48	603.22	0.43

4.4.1.1 Result on Synthetic Datasets

Table 4.3 exhibits the result of this experiment on all synthetic datasets. As it can be seen each method wins in 5 out of 10 datasets. The learning time is also similar in both methods. And thus we could conclude that splitting the feature space over several dimensions might not have a major advantage over splitting along one dimension.

Table 4.4: Comparing accuracy and learning time of MRRT when dividing the feature space along one dimension versus two dimensions on real datasets. As it can be seen, Two Dimensions split wins in accuracy and One Dimension split wins in learing time. The accuracy difference is not a major difference, but the learning time difference is significant.

	One Dimension		Two Dimensions	
Dataset	MRRT RMSE	Learning Time	MRRT RMSE	Learning Time
$IHEPC_1$	3.83	**4.70**	**3.76**	49.43
$IHEPC_2$	2.89	**3.26**	**2.78**	26.13
$IHEPC_3$	3.19	**2.71**	**3.12**	7.63

4.4.1.2 Result on Real Datasets

Table 4.4 lists result of this experiment on real datasets. We expect the results be similar because these three datasets are copy of each other except the change in their target feature (see section 4.2). Considering this we see that splitting along first two dimensions wins in all three cases, but not by a significant difference. On the other hand splitting along one dimensions wins on learning time by a significant difference. We are thinking of two reasons for why we observe a significant difference in learning time.

Since the accuracy is higher and learning time is also higher, we can assume that the models constructed in two dimensions split version might be more complex than models constructed by one dimension split version. For this reason learning time is higher and consequently accuracy is higher.

The other reason for having a large learning rate might be the way data is shuffled among nodes of MapReduce cluster. Since we want to be able to split the feature space along as many as dimensions that we want simultaneously, we need to implement this feature recursively. If we implement it iteratively we need to have as many as number of dimensions we would like to split along nested loops. Since the code needs to work for datasets with different number of dimensions, either we need to generate the code dynamically on the fly at runtime when number of dimensions to split along is specified, or have a recursive code. We have used the latter choice, and thus we think it affects the learning time of the two dimensions split method. We need to point out that learning time is not significantly different in the same experiment we had for synthetic datasets. The reason is that all the synthetic datasets are of size 100,000 data items while real datasets include around 2,000,000 data items. Thus the problem shows itself in the real dataset experiment and not in synthetic dataset experiment.

4.4.1.3 Summary

Based on the result in two preceding subsections, we assume dividing along one dimensions is preferable to dividing along several dimensions at the same time for the following reasons, and we would split the feature space along only one dimension in other experiments we perform:

- When dividing the feature space along several dimensions we do not have full control over number of final subspaces, because final number of subspaces is multiplication of number of split points we have chosen along each dimension. On the other hand when we divide the feature space along one dimension, we can simply generate as many as subspace we need. The former method would make the load balancing a challenging task. For example assume we have n node for constructing the final model for each subspace, but number of subspaces is slightly more than number of nodes. In this case some nodes need to perform the model construction task two times and thus this will affect the total runtime of the MapReduce round which is equal to: max(map-time of all nodes) + communication time + max(reduce-time of all nodes). If one node runs two map or reduce tasks, then it will increase the MapReduce time not for a minor amount.
- Recursive implementation of the part of the MapReduce program in which we want to

split the data among nodes in order to deliver data items in each subspace of the feature space to a certain node (in order to construct the model for that subspace), might have a negative effect on learning time when dataset size is large.

- In either synthetic and real dataset experiment we have not observed a major difference in accuracy of the model.

4.4.2 Overlapping Subspaces and Neighbor-weighted Predictions

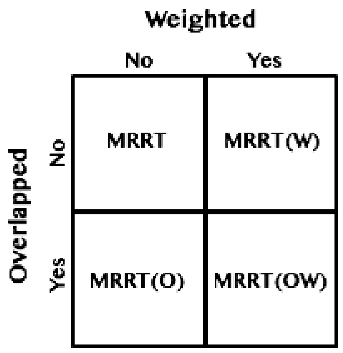

Figure 4.2: Summary of MRRT versions

We could have four different MRRT versions based on overlapping/non-overlapping subspaces, and weighted/un-weighted prediction. Figure 4.2 depicts these four different versions.

When presenting MapReduce Regression Tree in section 3.1, we talked about overlapping subspaces. If we divide the feature space with no overlap, no each data item would be given to two nodes of the cluster, but with overlapping we deliberately let the subspaces overlap and some data items to be given to more than one node. We intuitively know that overlapping would help to increase the accuracy of generated model, but at the same time would have a negative effect on learning time. We would run experiments to verify whether our intuition is correct or not. This way we would compare two different versions of MMRT which are MRRT and MRRT(O).

After the model is generated, predicting the target value for a new data item could be done simply by retrieving the corresponding model and making the prediction. The alternative is using using the neighboring models of the main model corresponding to the date item and use all models to predict the target value. A simple choice for using the main model and two neighboring model to predict the target value is calculating a weighted average by weights of

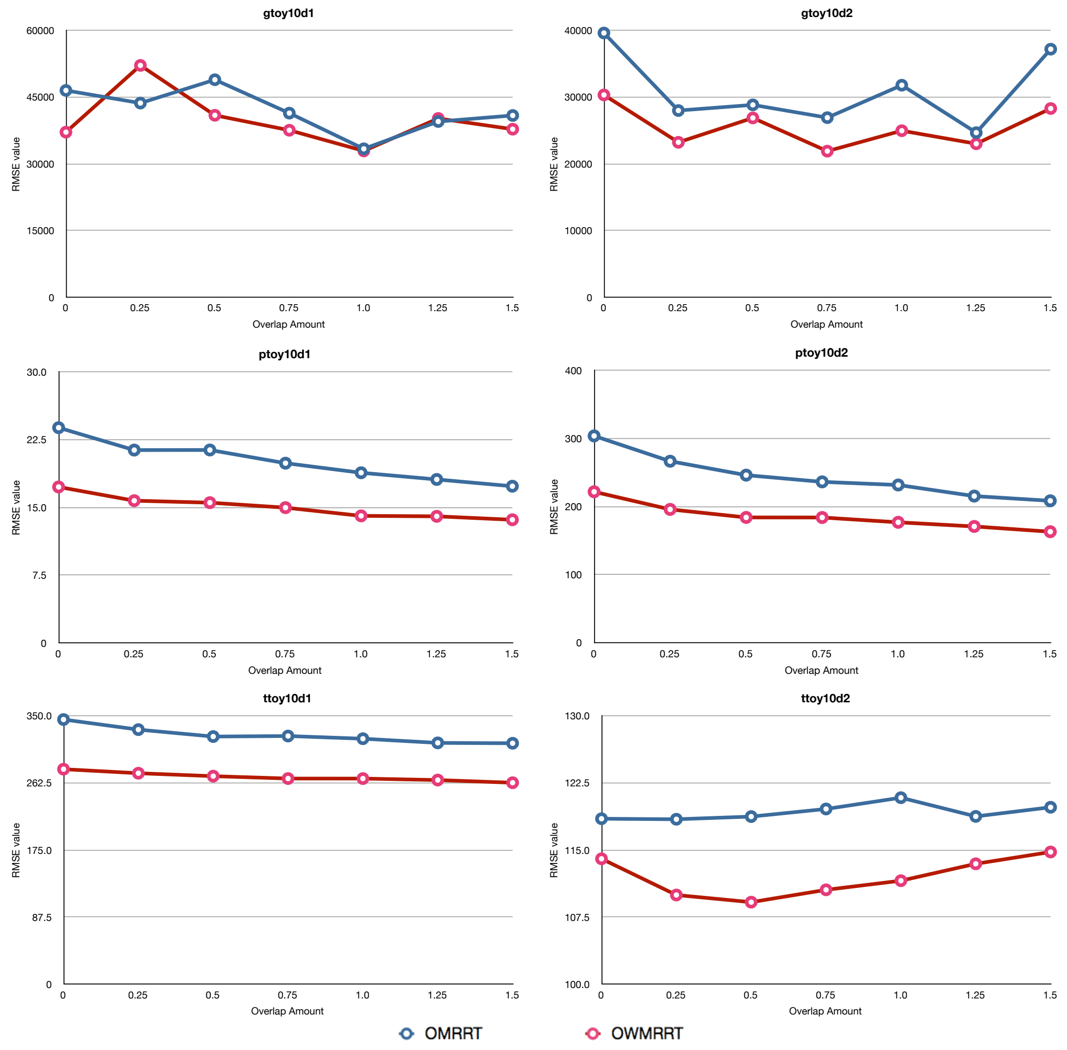

Figure 4.3: Analyzing accuracy of MRRT(O) and MRRT(WO) algorithms on 10-dimensional datasets (gtoy10d1, gtoy10d2, ptoy10d1, ptoy10d2, ttoy10d1 and ttoy10d2) datasets with different overlap values when dividing along first dimension.

2 for main model and 1 for each neighboring model (when dividing the feature space along one dimension only). Intuitively we could say that using weighted prediction at least would decrease the prediction error for data items that are close to the split point borderline, but we need to very this intuition by an experiment. By performing this experiment we would compare MRRT and MRRT(W) to each other.

There is another possiblity for learning and prediction which is having a combined version of algorithm that uses both subspace overlapping and weighted prediction. Since both of these methods are partially trying to help with data items that are close to the split point borderline, it is not obvious if combining these two features would help to improve accuracy or not. We need to run an experiment to find out the answer to this question. By performing this experiment we would have the result for all four different versions of MRRT (MMRT, MRRT(O), MRRT(W), MRRT(WO)), and would be able to decide and choose the best one among them.

We run the following experiment to answer all above questions. We change the overlapping factor from 0 to 1.5 by step size of 0.25, and would calculate the RMSE of MRRT(O), and MRRT(WO). This way we also have calculated RMSE of MRRT and MRRT(W) when overlap is 0, and we would be able to compare these four algorithm. To run the experiment we fixed number of nodes in the cluster to 64, and run the experiment on all synthetic and real datasets. Results of this experiment is illustrated in two following subsections.

4.4.2.1 Result on Synthetic Datasets

Figures 4.3 and 4.4 depicts result of experiment for all 10 dimensional and 20 dimensional synthetic datasets. As it can be seen in these two figures in 8 out of 10 datasets, the MRRT(WO) is below or equal to MRRT(O). This means that for all datasets except ptoy20d1 and ptoy20d2 MRRT(WO) has lower RMSE than MRRT(O) and thus weighted combination of WO has lower RMSE than O. If we look at the the RMSE on the y axis of the figures, where overlapping is 0, we observe again that weighted version has lower RMSE than un-weighted version for 8 datasets (Table 4.5 represents parts of the results depicted in figures 4.3 and 4.4 that is related to 0 overlapping in tabular format). This means that MRRT(W) is performing better than MRRT. No the only remaining question is if MRRT(WO) is performing better than or equal to MRRT(W). The answer to this question can be found by checking if there on points weighted line if diagrams that have RMSE lower or equal to the start point of same line. The answer is yes, and we see that at overlap around 0.75 and 1 we have RMSE lower than or equal to overlap 0 point of the weighted line in 8 out of 10 datasets. Thus we conclude that the MRRT(WO) has lowest RMSE among these four versions of MRRT algorithm.

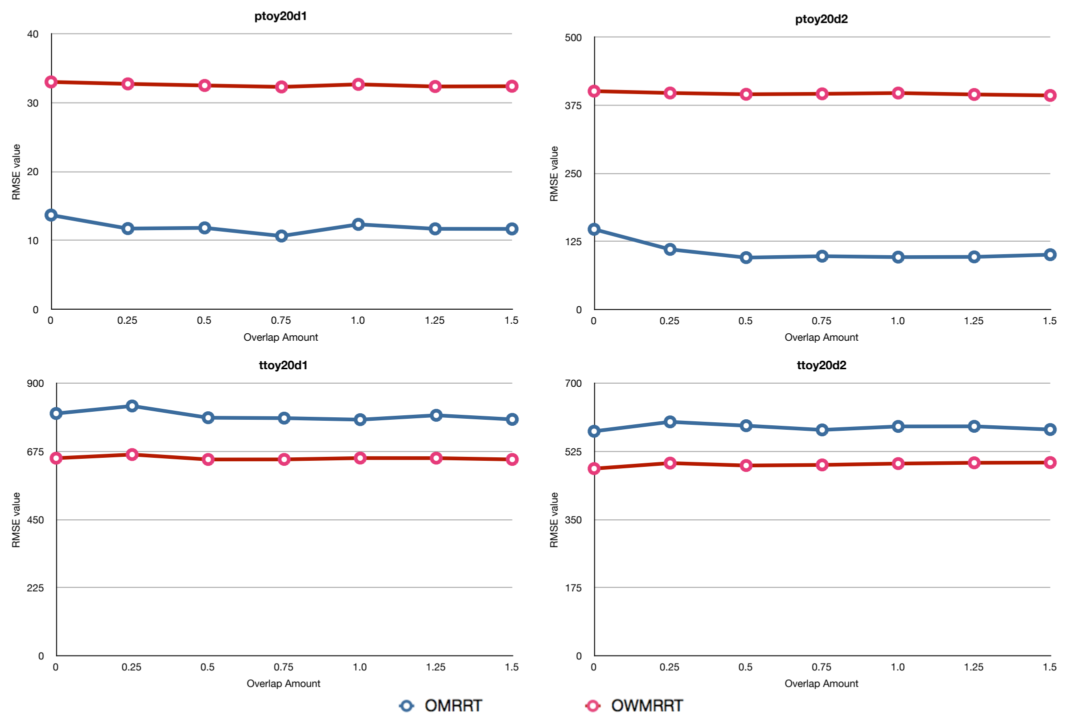

Figure 4.4: Analyzing accuracy of MRRT(O) and MRRT(WO) algorithms on 20-dimensional datasets (ptoy20d1, ptoy20d2, ttoy20d1 and ttoy20d2) datasets with different overlap values when dividing along first dimension.

Table 4.5: Comparing accuracy of MRRT(W) and MRRT both with no overlap. As it can be seen the MRRT(W) algorithms works bettern than MRRT on most datasets.

Dataset	MRRT RMSE	MRRT(W) RMSE
gtoy10d1	46511.14	**37135.21**
gtoy10d2	39614.44	**30333.90**
ptoy10d1	23.83	**17.27**
ptoy10d2	303.71	**221.72**
ttoy10d1	345.23	**280.44**
ttoy10d2	118.50	**114.02**
ptoy20d1	**13.69**	33.02
ptoy20d2	**147.48**	400.87
ttoy20d1	800.41	**652.79**
ttoy20d2	576.87	**480.99**

Using weighted version would increase the prediction time, and using overlapping would increase the learning time. By looking at the diagrams we notice that we will not be able to have better RMSE in non-weighted version comparing to weighted version. Thus we cannot ignore the the weighted feature of the algorithm, but it can be observed that weighted non-overlapped version could be a good choice for many cases.

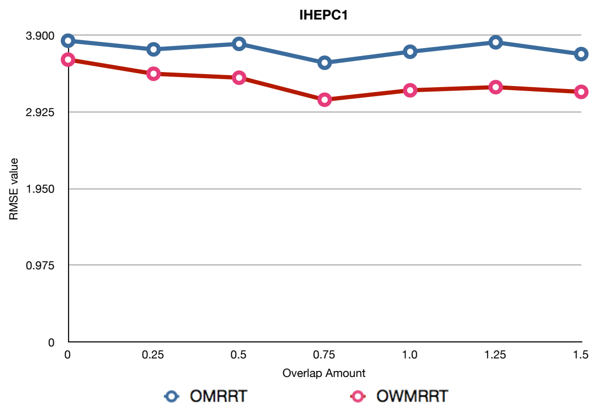

Figure 4.5: Analyzing accuracy of MRRT(O) and MRRT(WO) algorithms on $IHEPC_1$ real dataset with different overlap values when dividing along first dimension.

4.4.2.2 Result on Real Datasets

Figure 4.5 depicts result of overlapping subspaces and weighted prediction for $IHEPC_1$ real dataset. As it is obvious all the analysis for the synthetic datasets in previous subsection is valid for real dataset also. The overlapping value with lowest RMSE for this dataset is 0.75.

4.4.2.3 Summary

MRRT(WO) version of algorithm is the one with the lowest RMSE on 8 out of 10 synthetic datasets and also on $IHEPC_1$ real dataset. The overlapping value with lowest RMSE error is also around 0.75 and 1. We have chosen 0.75 as the overlapping value for our following experiments.

4.4.3 Comparing the Accuracy of MRRT and the Baseline Algorithm

Although main goal of implementing an algorithm for MapReduce framework is increasing speed for large datasets, but it would be nice if the devised algorithm achieves both high accuracy and high speed. The question here is how often MRRT algorithm will achieve higher accuracy than baseline algorithm on test set. To answer this question we run an experiment in which we fix number of nodes in the cluster to 64, and overlapping factor to 0.75. Then we run the MRRT(WO) algorithm on all synthetic datasets and also all real datasets. Then we calculated the highest possible improvement in RMSE and also number of dimensions that might help us to achieve a RMSE lower than baseline algorithm. We also calculated the expected value of MRRT(WO)'s RMSE to show what would be the expected value of RMSE in case a random dimension is chosen to split the dataset along. These numbers would help us to answer to the raised question.

Table 4.6: Comparing accuracy of Weighted Overlapping MapReduce Regression Tree and baseline algorithm on 10-dimensional synthetic datasets. Numbers in the table are RMSE values. MRRT(WO) algorithm always performs better than baseline algorithm, when splitting the feature space is done along one dimension and if the dimension to split is chosen properly.

Dataset:		gtoy10d1	gtoy10d2	ptoy10d1	ptoy10d2	ttoy10d1	ttoy10d2
Baseline RMSE:		37985.11	27691.92	10.61	112.82	298.8	119.3
	1	37566.29	21912.22	14.93	175.36	267.33	110.04
	2	40205.10	35617.12	14.80	172.23	332.59	108.07
	3	32009.29	20059.11	10.82	122.49	300.70	**106.75**
MRRT(WO)	4	40952.27	20766.81	14.49	171.50	334.25	109.10
RMSE when	5	**31812.41**	25576.34	9.32	**96.90**	338.11	107.12
split along	6	37887.38	27875.03	15.52	179.55	317.81	107.28
dimension:	7	45558.38	26696.26	15.17	161.38	**255.21**	108.36
	8	35921.19	27483.59	13.61	177.81	321.46	107.05
	9	43081.88	**18591.08**	9.45	102.84	294.35	110.34
	10	33530.82	25113.43	**9.17**	99.08	264.81	107.07
Max Improvement:		16.3%	32.9%	13.6%	14.1%	14.6%	10.5%
Avg Improvement:		0.3%	9.8%	-20.0%	-29.3%	-1.3%	9.4%
Count[2]:		6	8	3	3	4	10
Expected value[3]:		**37852.50**	**24969.10**	12.73	145.91	302.66	**108.12**

4.4.3.1 Result on Synthetic Datasets

Tables 4.7 and 4.7 list the RMSE of baseline algorithm on test set of different datasets, and also RMSE of models constructed by MRRT(WO) when feature space is divided to subspaces along each dimension of all 10 and 20 dimensional synthetic datasets on test set. It could be observed that it is possible to decrease RMSE achieved by baseline algorithm by a minimum of 10.2% (for ttoy10d2 dataset) and a maximum of 32.9% (for gtoy10d2 dataset).

[2]Number of Dimensions giving better RMSE than Regression Tree algorithm (out of 10)
[3]Expected value of RMSE for MRRT(WO)

Table 4.7: Comparing accuracy of Weighted Overlapping MapReduce Regression Tree and baseline algorithm on 20-dimensional synthetic datasets. Numbers in the table are RMSE values. MRRT(WO) algorithm always performs better than baseline algorithm, when splitting the feature space is done along one dimension and if the dimension to split is chosen properly.

Dataset:		ptoy20d1	ptoy20d2	ttoy20d1	ttoy20d2
Baseline RMSE:		10.20	106.29	733.20	582.96
MRRT(WO) RMSE when split along dimension:	1	9.60	**86.43**	649.71	490.82
	2	10.26	92.45	624.76	496.87
	3	9.86	91.43	687.27	524.25
	4	9.81	92.63	696.12	546.04
	5	10.04	90.26	678.56	509.32
	6	10.21	92.66	670.53	537.87
	7	9.74	86.99	678.85	554.68
	8	10.65	97.20	631.24	554.62
	9	10.96	115.77	679.03	562.4
	10	10.29	95.77	627.24	548.81
	11	10.25	94.89	653.55	550.88
	12	10.32	94.00	**595.96**	**486.74**
	13	9.81	96.47	682.69	546.26
	14	10.42	95.44	665.98	546.35
	15	9.75	91.15	681.24	525.63
	16	9.95	97.25	674.28	556.59
	17	10.51	89.61	676.80	558.21
	18	11.02	99.73	609.81	556.44
	19	10.51	93.55	644.86	512.29
	20	**8.99**	93.14	667.54	553.13
Max Improvement:		11.9%	18.7%	18.7%	16.5%
Avg Improvement:		0.5%	11.2%	10.1%	8.1%
Count[4]:		9	19	20	20
Expected value[5]:		**10.153**	**94.52**	**655.27**	**539.25**

There is no dataset for which the MRRT(WO) algorithm cannot improve the baseline algorithm's RMSE, if the dimension to split along is chosen properly. We will introduce a method for choosing the dimension along in section 4.4.4. Expected value of RMSE in 7 out of 10 datasets is less than baseline algorithm's RMSE. It means that if the dimension to split along is chosen randomly in 7 out of 10 cases the expected RMSE is less than baseline algorithm. For ttoy20d1 and ttoy20d2 20 dimensional datasets, MRRT(WO) achieves lower RMSE on test set than baseline algorithm if the feature space is divided on subspace on any dimension. That means that there is no way for baseline algorithm to achieve higher accuracy than MRRT(WO). For ptoy20d2 20 dimensional dataset this value is 19 out of 20 dimensions. MRRT(WO) would achieve higher accuracy for ttoy10d2 10 dimensional dataset also on any dimension.

Table 4.8: Comparing accuracy of MRRT(WO) and baseline algorithm on real datasets. Numbers in the table are RMSE values. MRRT(WO) algorithm always performs better than baseline algorithm, when splitting the feature space is done along one dimension and if the dimension to split is chosen properly.

Dataset:		$IHEPC_1$	$IHEPC_2$	$IHEPC_3$
Baseline RMSE:		3.55	3.08	2.79
MRRT(WO) RMSE when split along dimension:	1	**3.08**	2.43	2.46
	2	3.24	**2.38**	2.74
	3	3.29	2.67	2.66
	4	3.26	2.60	2.82
	5	3.32	2.47	**2.37**
	6	3.56	2.84	3.07
	7	3.55	2.87	2.90
Max Improvement:		13.24%	22.73%	15.05%
Avg Improvement:		6.2%	15.3%	2.6%
Count[4]:		6	7	4
Expected value[5]:		**3.33**	**2.61**	**2.72**

4.4.3.2 Result on Real Datasets

Table 4.8 lists the RMSE of baseline algorithm and also RMSE of MRRT(WO) on all models constructed by splitting along each dimension of $IHEPC_1$, $IHEPC_2$, and $IHEPC_2$ real datasets. It could be observed that it is possible to decrease RMSE achieved by baseline algorithm 13.24% (for $IHEPC_1$ dataset), 22.73% (for $IHEPC_2$ dataset), and 15.05% (for

[4]Number of Dimensions giving better RMSE than Regression Tree algorithm (out of 20)
[5]Expected value of RMSE for MRRT(WO)
[4]Number of Dimensions giving better RMSE than Regression Tree algorithm (out of 20)
[5]Expected value of RMSE for MRRT(WO)

$IHEPC_3$ dataset). There is no dataset on which the MRRT(WO) algorithm cannot improve the baseline RMSE if the dimensions to split along is chosen properly. Expected value of RMSE for all real datasets is also less than baseline algorithm's RMSE. MRRT(WO) would have a lower RMSE than baseline algorithm for dataset $IHEPC_2$ if any dimension is chosen to split along. Same thing happens for 6 out of 7 dimensions of $IHEPC_1$.

4.4.3.3 Summary

If we consider expected value of RMSE for MRRT(WO) algorithm for all synthetic and real datasets, we observe that in 7 out of 10 synthetic datasets and all real datasets the expected value of RMSE would be less than baseline algorithm's RMSE, and in case of one synthetic dataset (ttoy10d1) the expected value of RMSE is roughly equal to baseline algorithm's RMSE. It means that for 77% of datasets the expected value of RMSE by MRRT(WO) is less than baseline algorithm's RMSE, and in 84.5% of datasets it is less than or equal to baseline algorithm's RMSE.

It also worth mentioning that even in case of two synthetic datasets that MRRT(WO) does not have a lower expected RMSE, it is possible to achieve 13.6% and 14.1% improvement in RMSE if the dimension to split along is chosen properly. Next session we introduce a method for choosing the best dimensions to split along.

4.4.4 Choosing the Dimension to Split Along

When we divide the feature space along one dimension, MRRT algorithm's accuracy would depend on the dimension we choose to split along. Running the algorithm as many as number of dimensions in the dataset is not the preferable way of finding out which dimension is better to split along. As another solution we could select a small sample of dataset (for example 10% of data items in the dataset), and run the MRRT algorithm on the sample to see what dimension of the sample would lead to a model with highest accuracy. The same dimension might be the dimension with lowest RMSE to split along on the original dataset. This assumption seems reasonable, but we need to run experiments to verify if the assumption is valid.

In this experiment we inspect how accuracy of the model constructed using original dataset would change when dividing the feature space along different dimensions. Same procedure is applied on a sample of size 10% of the original dataset to see how model accuracy changes when dimension to split changes. This way we would be able to see if there is any relation among accuracy of models constructed based on original dataset and model constructed based on sample dataset. This experiment would show how helpful is the *preProcess* method

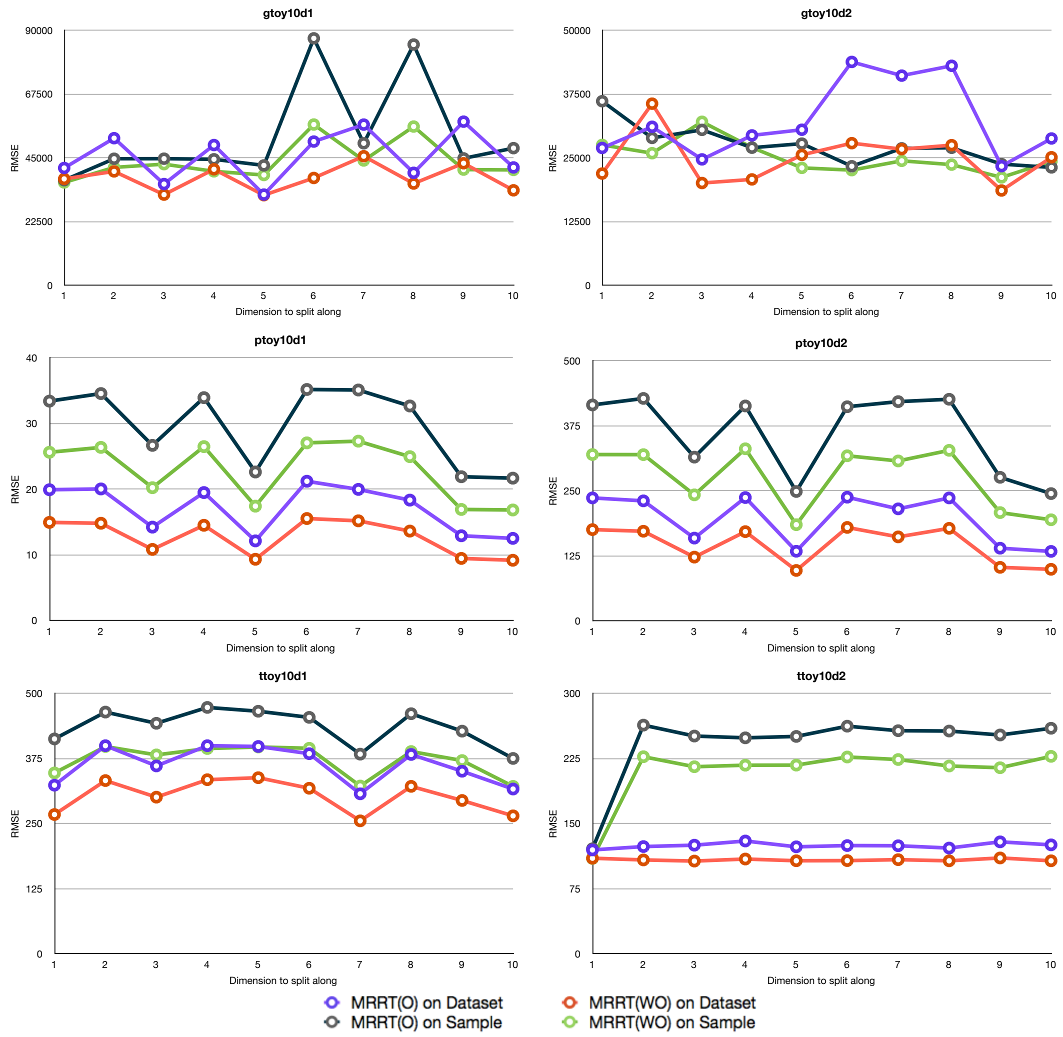

Figure 4.6: Analyzing accuracy of MRRT(O) and MRRT(WO) algorithms on 10 dimensional datasets with overlap = 0.75, when splitting along diffent dimensions.

introduced in section 3.1.

This experiment is performed on all synthetic datasets and the all real datasets. Number of cluster nodes is fixed to 64, and overlapping factor is set to 0.75 in all experiments. The experiment runs the MRRT algorithm on each dataset as many time as number of dimensions in each dataset.

4.4.4.1 Result on Synthetic Datasets

Figures 4.6 and 4.7 shows result of this experiment on 10 and 20 dimensional synthetic datasets. The purple and red lines are results of running MRRT(O) and MRRT(WO) on original datasets. Black and green lines are result of running MRRT(O) and MRRT(WO) on sample of size 10% of datasets. To see if the run on sample is helping to find the dimension with lowest RMSE to split, we need to see if the lowest RMSE value on sample dataset is matching the lowest RMSE value on dataset itself (for each algorithm; i.e purple and black lines need to be compared to each other and red and green lines to each other).

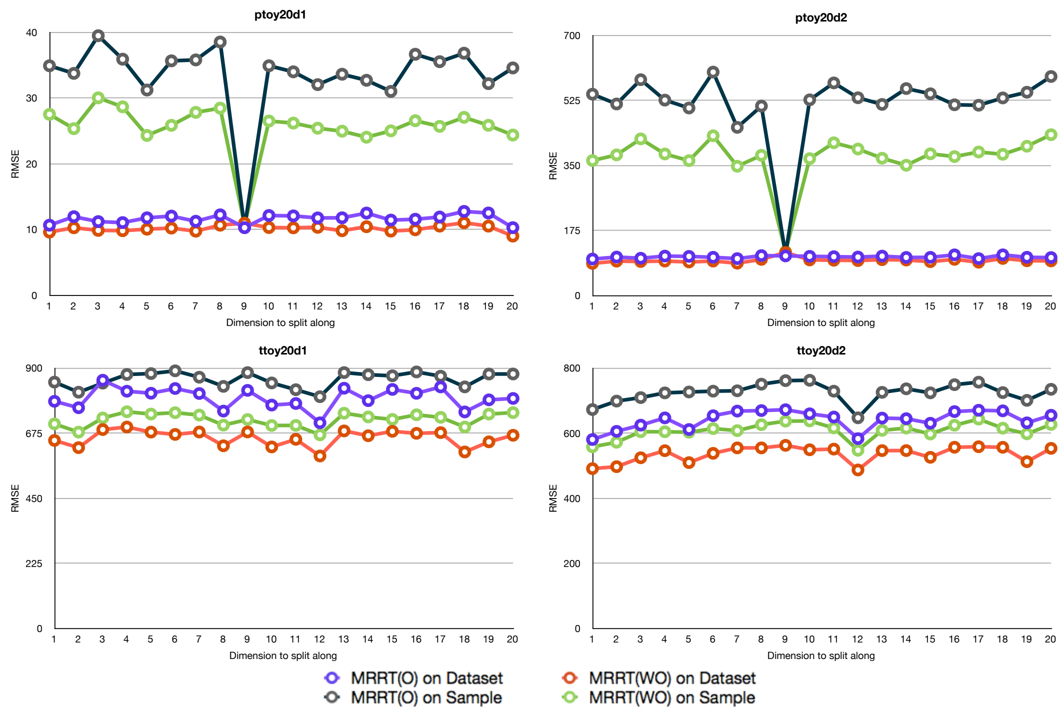

Figure 4.7: Analyzing accuracy of MRRT(O) and MRRT(WO) algorithms on 20 dimensional datasets with overlap = 0.75, when splitting along diffrent dimensions.

Except diagrams for gtoy10d1 and gtoy10d2 in figures 4.6 and 4.7 that could be read as no relation between sample RMSE and dataset RMSE, all other diagrams offer that if we choose a dimension with low sample RMSE to split the feature space of original dataset along, we would have a low RMSE. Although diagrams could give us a high level overview of this relation, but we need more precise information to be able to judge if sampling could help with choosing the best dimension to split along. Tables 4.9 and 4.10 give more information in this regard.

Table 4.9: Dimensions with lowest RMSE on synthetic datasets and rank of same dimension on samples using MRRT(O) and MRRT(WO) algorithms.

	MRRT(O)		MRRT(WO)	
Dataset	Dim with lowest RMSE on dataset	Rank of the Dim on sample	Dim with lowest RMSE on dataset	Rank of the Dim on sample
gtoy10d1	5	2	5	2
gtoy10d2	9	3	9	1
ptoy10d1	5	3	10	1
ptoy10d2	10	1	5	1
ttoy10d1	7	2	7	2
ttoy10d2	1	1	3	3
ptoy20d1	9,20	1,8	20	4
ptoy20d2	1	13	1	5
ttoy20d1	12	1	12	1
ttoy20d2	1	2	12	1

Table 4.9 answers the following question: If we have the result of running the algorithms on sample of datasets, how many dimensions we need to run the algorithm for on real dataset to make sure that we get the lowest possible RMSE. This table lists dimensions with lowest RMSE on synthetic datasets and rank of same dimension on samples using MRRT(O) and MRRT(WO) algorithms. For example for gtoy10d1 dataset dimension 5 would lead to lowest RMSE using MRRT(O) algorithm and same dimension would have the second lowest RMSE (among all dimensions) when MRRT(O) algorithm is performed on the sample of the dataset (with size 10% of size of original dataset). Ignoring ptoy20d2 dataset, this table suggests that the dimension that results in the lowest RMSE on original dataset is among 3 dimensions that results in lowest RMSE on sample for 10 dimensional datasets and among 5 dimensions that results in lowest RMSE on sample for 20 dimensional datasets.

What if we do not want to achieve the best possible RMSE, and just want to get an

[1]If the dimension with 3rd or 4th lowest value of RMSE on sample (dims 5, 20 respectively) is selected to split along on real dataset, RMSE on real dataset would be 10.04 and 8.99 respectively

[2]If the dimension with 2nd lowest value of RMSE on sample (dims7) is selected to split along on real dataset, RMSE on real dataset would be 86.99

Table 4.10: Dimensions with lowest RMSE on sample of synthetic datasets and RMSE of dataset when divided along same dimension using MRRT(WO) algorithm.

	MRRT(WO)			
Dataset	Dim with lowest RMSE on sample	RMSE of the Dim on dataset	Baseline RMSE	Improvement
gtoy10d1	1	**37566.29**	37985.11	1.10%
gtoy10d2	9	**18591.08**	27691.92	32.86%
ptoy10d1	10	**9.17**	10.61	13.57%
ptoy10d2	5	**96.90**	112.82	14.11%
ttoy10d1	10	**264.81**	298.8	11.38%
ttoy10d2	1	**110.04**	119.3	7.76%
ptoy20d1	9	**10.96**[1]	**10.20**	-7.45%
ptoy20d2	9	**115.77**[2]	**106.29**	-8.92%
ttoy20d1	12	**595.96**	733.20	18.72%
ttoy20d2	12	**486.74**	582.96	16.51%

RMSE near or lower that baseline RMSE? Table4.10 is answering another question. It lists the dimensions the achieves lowest RMSE on on test set along RMSE of same dimensions on original dataset. As the last column of this table exhibits, The RMSE of this dimension would be less than RMSE of baseline algorithm in 8 out of 10 datasets, and in two remaining datasets, the RMSE is acceptable, but if we are not satisfied with that result we only need to try 1 more dimension (2nd rank dimension on sample) for ptoy20d2 dataset to achieve 18.16% performance over the baseline algorithm, and 2 or 3 more dimensions (3rd and 4th rank dimensions on sample) for ptoy20d1 dataset to achieve 1.57% and 11.86% performance over the baseline algorithm respectively.

Two more observations could be made from diagrams in figures 4.6 and 4.7. First observation is that although a 10% sample is going to be more sparse comparing to original dataset, but the MRRT(WO) yet shows to have lower RMSE than MRRT(O). One might expect that since the data is more sparse, using the neighboring models to help in prediction might not lead to decreasing the RMSE, but these diagrams show that it might not be the case. Second observation is that for some datasets such as ttoy10d2, ptoy20d1, and ptoy20d2 model for all dimensions would lead to close RMSE values.

4.4.4.2 Result on Real Datasets

Figure 4.8 depicts result of this experiment on $IHEPC_1$, $IHEPC_2$, and $IHEPC_3$ real datasets. As it could be seen in diagrams, here samples again could suggest what dimensions would result in lower RMSEs.

Tables 4.11 and 4.12 also offer same information as tables 4.9 and 4.10 offer for synthetic

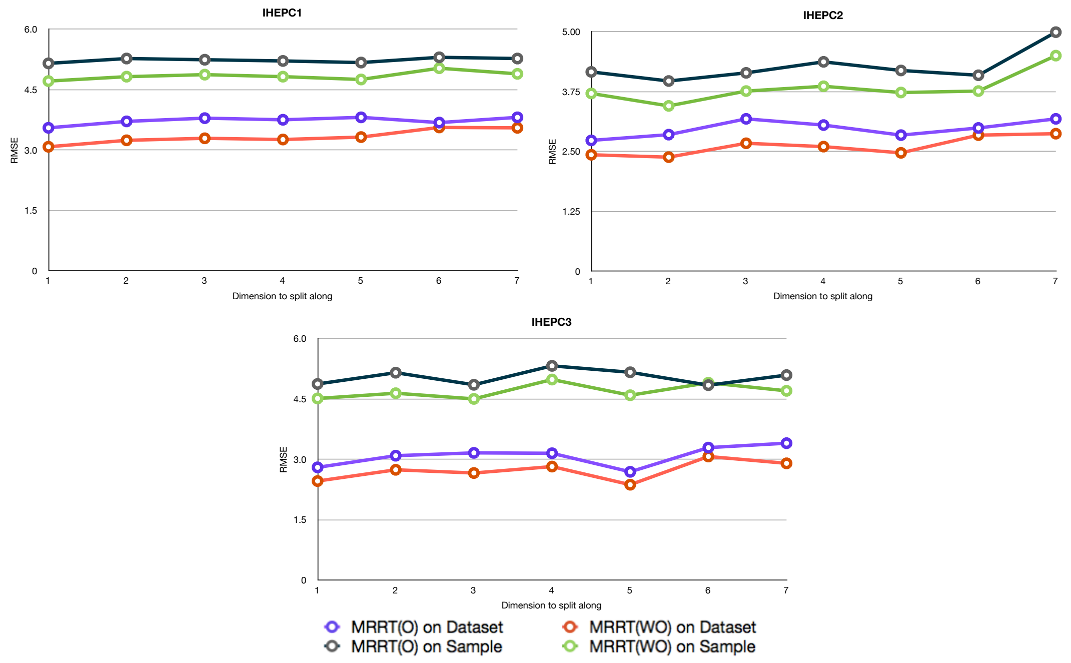

Figure 4.8: Analyzing accuracy of MRRT(O) and MRRT(WO) algorithms on $IHEPC_1$ real datasets with overlap = 0.75, when splitting along diffrent dimensions.

Table 4.11: Dimensions with lowest RMSE on real datasets and rank of same dimension on samples using MRRT(O) and MRRT(WO) algorithms.

	MRRT(O)		MRRT(WO)	
Dataset	Dim with lowest RMSE on dataset	Rank of the Dim on sample	Dim with lowest RMSE on dataset	Rank of the Dim on sample
$IHEPC_1$	1	1	1	1
$IHEPC_2$	1	4	2	1
$IHEPC_3$	5	6	5	3

datasets. For real datasets if we chose the dimension that has achieved lowest RMSE on sample, we always would achieve a lower RMSE comparing to baseline algorithm and improvement would be between 4.66% and 22.73%.

Table 4.12: Dimensions with lowest RMSE on sample of real datasets and RMSE of dataset when divided along same dimension using MRRT(WO) algorithm.

	MRRT(WO)			
Dataset	Dim with lowest RMSE on sample	RMSE of the Dim on dataset	Baseline RMSE	Improvement
$IHEPC_1$	1	**3.08**	3.55	13.24%
$IHEPC_2$	2	**2.38**	3.08	22.73%
$IHEPC_3$	3	**2.66**	2.79	4.66%

4.4.4.3 Summary

Based on experiments we had on synthetic and real datasets in preceding subsections, *preProcess* method that uses a sample of dataset to determine the dimension with lowest RMSE is matching the dimension with lowest RMSE in the original dataset in most cases, and could reduce the cost of dimension selection comparing to naive method of running the algorithm on all dimensions of the original dataset in order to chose the dimension with lowest RMSE.

4.4.5 Prediction Time

When having a single regression tree model for the whole dataset (baseline algorithm), the tree would be bigger than trees constructed by MRRT algorithm that construct regression tree model for subspaces of the dataset. For this reason traversing the tree and consequently the test time might be different in two algorithms. In this experiment we measure the time needed to predict the target value for all data items in the test set using models generated by each algorithm to see which model is predicting faster, and how faster it is.

In this experiment we changed number of nodes in the cluster and evaluated the result for 32, 64, 96, 128, and 256 nodes in the cluster. Overlap factor is fixed to 0.75 (this parameter does not affect prediction time). The experiment is done on ttoy20d3 dataset that includes 1,000,000 data items in training set and 1,000 data items in test set. The reason that number of cluster nodes is changed is that different number of nodes would affect number of subspaces in the feature space, and size of each subspace consequently. Thus we would have more smaller regression trees instead less bigger trees (and in the extreme case one huge tree in baseline algorithm) when number of nodes increases. We expect to have lower prediction time for MRRT algorithms than baseline. Since MRRT(WO) uses 3 models to predict the target

Table 4.13: Comparing prediction time of MRRT(O), MRRT(WO) and baseline algorithm on 20-dimensional ttoy20d3 synthetic test set containing 1000 test items on different size of clusters. MRRT(WO) and MRRT(O) algorithms reduce prediction time by more than 80% comparing to baseline algorithm in all cases.

Baseline		MRRT(O)		MRRT(WO)	
Pred. Time	Number of nodes	Pred. Time	Improvement	Pred. Time	Improvement
194.79	32	16.94	91.30%	33.95	82.57%
	64	8.61	95.58%	16.58	91.49%
	96	6.73	96.54%	13.23	93.21%
	128	5.49	97.18%	10.53	94.59%
	160	5.22	97.32%	10.49	94.61%
	192	3.75	98.07%	6.92	96.45%
	224	3.15	98.38%	6.60	96.61%
	256	3	98.46%	5.79	97.03%

value, and MRRT(O) uses 1 model to predict the target value, we expect that learning time of MRRT(O) be less than MRRT(WO). It is worth mentioning that the prediction time for MRRT algorithms is not done in parallel and is done on a single machine.

4.4.5.1 Result on Synthetic Datasets

Table 4.13 lists result of this experiment for ttoy20d3 synthetic dataset. As expected prediction time of baseline algorithm for 1,000 test items is 194.79 seconds and maximum time for MRRT(WO) is 32.79 seconds and it is when the model is constructed using 32 nodes. The lowest prediction time of MRRT(WO) algorithm is 6.06 seconds for all 1,000 test items and it is when the model is constructed using 256 nodes.

4.4.5.2 Result on Real Datasets

Table 4.14 lists result of this experiment on all three real dataset. Prediction time of baseline algorithm is improved by more than 80% for all datasets and all cluster sizes. Again, improvement for MRRT(O) is more than MRRT(WO) due to less calculations during prediction. Prediction time for 256 node cluster size is improved more than 95% while it is improved for 32 node cluster size for more than 80%.

4.4.5.3 Summary

Although prediction time of MRRT algorithms is done sequentially, we observe a high improvement in prediction time comparing to baseline algorithm. This improvement is more than 80% for all real datasets and the experimented synthetic dataset. The improvement

Table 4.14: Comparing prediction time of MRRT(O), MRRT(WO) and baseline algorithm on real datasets' test sets containing 4111 test items on different size of clusters. MRRT(WO) and MRRT(O) algorithms reduce prediction time by more than 80% comparing to baseline algorithm in all cases.

	Baseline		MRRT(O)		MRRT(WO)	
Dataset	Time	Num of nodes	Time	Improvement	Time	Improvement
$IHEPC_1$	1266.40	32	123.81	90.22%	244.83	80.67%
		64	63.76	94.97%	126.56	90.01%
		96	40.64	96.79%	80.90	93.61%
		128	34.33	97.29%	68.67	94.58%
		160	26.00	97.95%	51.07	95.97%
		192	22.33	98.24%	44.44	96.49%
		224	19.76	98.44%	38.75	96.94%
		256	17.09	98.65%	33.46	97.36%
$IHEPC_3$	979.40	32	75.26	92.32%	148.85	84.80%
		64	39.53	95.96%	78.17	92.02%
		96	28.12	97.13%	54.70	94.41%
		128	22.36	97.72%	44.46	95.46%
		160	17.90	98.17%	36.31	96.29%
		192	17.20	98.24%	33.27	96.60%
		224	16.42	98.32%	31.43	96.79%
		256	14.84	98.48%	28.71	97.07%
$IHEPC_3$	351.28	32	29.45	91.62%	58.78	83.27%
		64	17.17	95.11%	32.40	90.78%
		96	13.70	96.10%	26.06	92.58%
		128	11.33	96.77%	22.10	93.71%
		160	10.08	97.13%	19.00	94.59%
		192	8.73	97.51%	16.40	95.33%
		224	8.35	97.62%	16.27	95.37%
		256	7.59	97.84%	15.71	95.53%

is more when cluster size is bigger, and less when cluster size is smaller. It is because the regression tree constructed by smaller clusters (due to bigger size of subspaces) are bigger.

4.4.6 Speedup of MRRT Algorithm

An ideal parallel program is a program that uses the resources most, and decrease the runtime to $\frac{1}{n}$ of runtime on a single machine (here n is number of nodes in the cluster). When the program is I/O intensive, runtime might decrease even to a value less than $\frac{1}{n}$ of runtime on a single machine. The question is how MRRT performs in this regard?

This experiment measures speedup of MRRT algorithm by fixing the data set size and changing number of nodes in the cluster. After measuring the learning time on each setting, we compare them with runtime of the same algorithm on a single machine. By dividing these values we would calculate the speedup the algorithm and see if speedup is as big is number of nodes in the cluster or not.

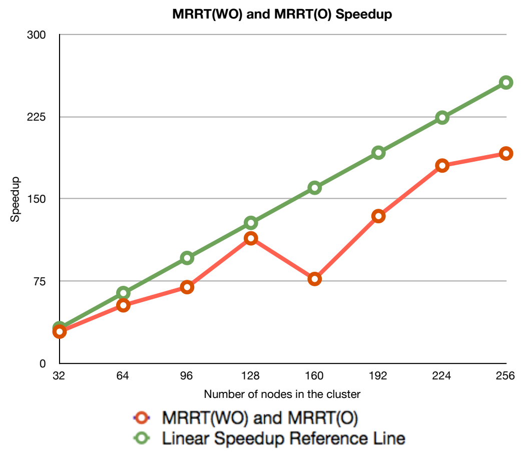

Figure 4.9: Speedup of MRRT(O) and MRRT(WO) algorithms in log scale and linear scale on ttoy20d3 dataset with overlap = 0.75 when splitting along the first dimensions. The runtime of same algorithms on a single machien is 2609.6 seconds.

4.4.6.1 Result on Synthetic Datasets

Learning time of MRRT(O) and MRRT(WO) is same. They both overlap the subspaces and their subspace size for same cluster size and dataset is same. Thus the speedup values presented in this section is valid for both algorithms. Figure 4.9 depicts speedup of MRRT algorithm when running on different cluster sizes. The green line is the linear speedup reference line, and the speedup of algorithm is roughly same as linear speedup. The left diagram depicts the speedup in log scale and the right diagram depicts it in linear scale. For this dataset with 1,000,000 data items and 20 dimensions the MRRT algorithm achieves roughly a linear speedup, but ee believe that the speedup could be higher if the dataset size was larger.

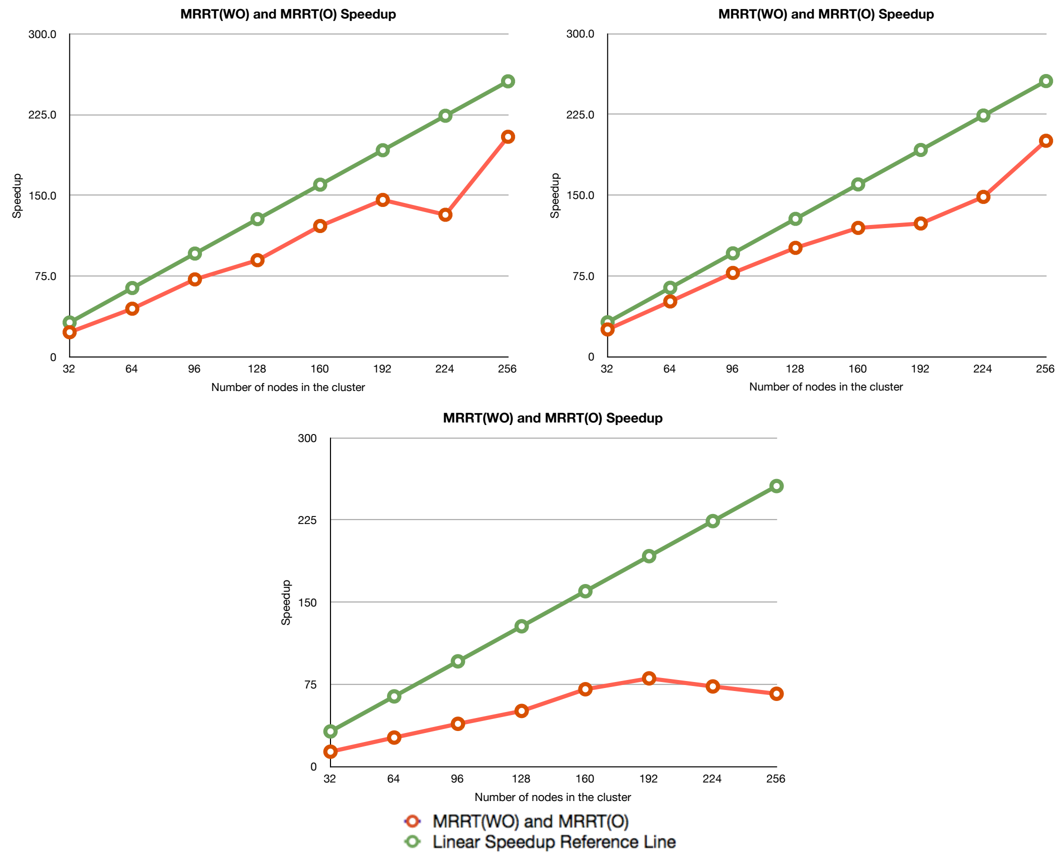

Figure 4.10: Speedup of MRRT(O) and MRRT(WO) algorithms in log scale and linear scale on all $IHEPC_1$, $IHEPC_2$, and $IHEPC_3$ real datasets respectively with overlap = 0.75 when splitting along the first dimensions.

4.4.6.2 Result on Real Datasets

Figure 4.10 depicts the log scale and linear scale diagrams for speedup on all three real datasets. As we pointed out before since the MRRT algorithm is I/O intensive, we observe a close to linear speedup for $IHEPC_1$ and $IHEPC_2$ algorithms. The speedup for $IHEPC_3$ is sub linear on the other hand.

4.4.6.3 Summary

MRRT algorithm achieves a close to linear speedup in 2 of 3 real datasets and also synthetic dataset. We believe that if the dataset size is larger, then the speedup of the algorithm would stand out better.

4.4.7 Scalability of MRRT Algorithm

In this experiment we are trying to find out which of the MRRT(WO), MRRT(WOS), and baseline algorithms scales better when dataset size is large. MRRT(WOS) is same is MRRT(WO) except we assume that it runs on a single machine and all the map and reduce phases of all subspaces is run by the same machine sequentially. The reason for measuring the scalability of this version of MRRT is that we would like to see if the MRRT is only useful for parallel setting or it also could be used when we need a sequential algorithm.

To run this experiment we keep number of nodes in the cluster constant and change number of data items in the datasets and observe how runtime of each algorithm changes (how scalable is the algorithm).

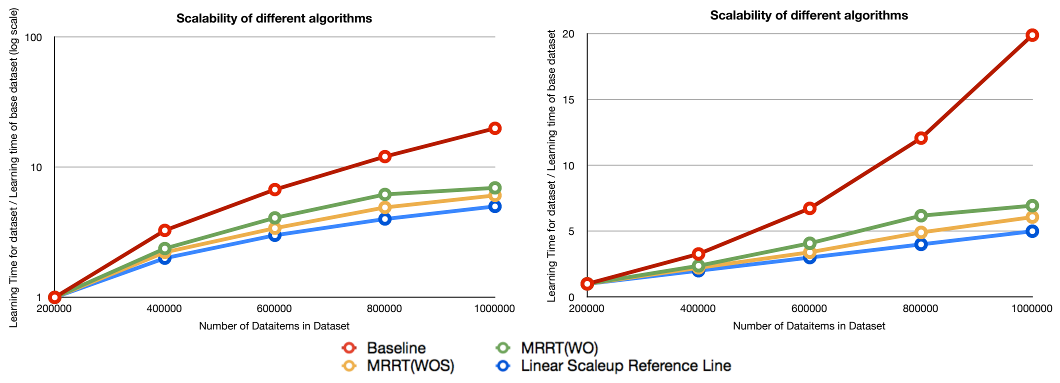

Figure 4.11: Analyzing scalability of baseline, MRRT(WO) and MRRT(WOS) algorithms on ttoy20d3 datasets with overlap = 0.75 when changing the dataset size from 50,000 items to 1,000,000 data itmes.

4.4.7.1 Result on Synthetic Datasets

Figure 4.11 depicts how learning time of baseline, MRRT(WO), and MRRT(WOS) algorithm changes when number of data items is changed from 50,000 to 100,000, 200,000, 400,000, and

1,000,000. The linear scale-up reference line is also depicted in this figure. This line shows how a reference algorithm that its runtime increases linearly with its input size would perform. The result is depicted in both log scale and linear scale formats to show the difference more clearly. As it can be seen in the figure MRRT(WO)'s scale up is close to linear, while baseline algorithm's scale up is far from linear.

Comparing MRRT(WOS) and baseline algorithm's scalability would help us to see if MRRT(WOS) is a good candidate to be used when we need a sequential algorithm to be run on a single machine. It can be observed in the figure that although runtime of MRRT(WOS) for small number of data items is slightly more than baseline algorithm, but it scales better than baseline algorithm, and its runtime would be less than baseline algorithm when dataset size is large, and we expect that the difference would increase when the dataset size increases for a large amount.

4.4.7.2 Result on Real Datasets

Figure 4.12 depicts same information as explained in previous subsection for all three real datasets. Here number of data items is changed from 103,557 to 207,115, 414,230, 828,459, and 2,071,148. Although number of data items in real dataset is more than number of data items in synthetic dataset (for same experiment in previous subsection), but dimensionality of dataset is 20 in synthetic dataset and is 7 here. Thus total size of datasets in each step is roughly same. The linear scale-up reference line is depicted in this figure too, and the result is depicted in both log scale and linear scale formats to show the difference more clearly.

As it can be observed in the figure MRRT(WO) and MRRT(WOS)'s scalability are close to optimum scalability in case of all three real datasets ($IHEPC_1$, $IHEPC_2$, and $IHEPC_3$), while baseline algorithm's scale up is $O(n^2)$.

As in case of synthetic experiment, runtime of MRRT(WOS) for small number of data items is slightly more than baseline algorithm, but it scales better than baseline algorithm, and its runtime would be less than baseline algorithm when dataset size is large, and we expect that the difference would increase when the dataset size increases for a large amount. We could conclude with this result that MRRT(WOS) is a good replacement to be used when we need a sequential algorithm to be run on a single machine instead of baseline algorithm.

4.4.7.3 Summary

Experiments on synthetic and real datasets suggest than MRRT's learning time is scaling roughly close to linear with dataset size. That means that the bigger the the dataset size, the better the performance of the algorithm. This is what we need for large scale datasets.

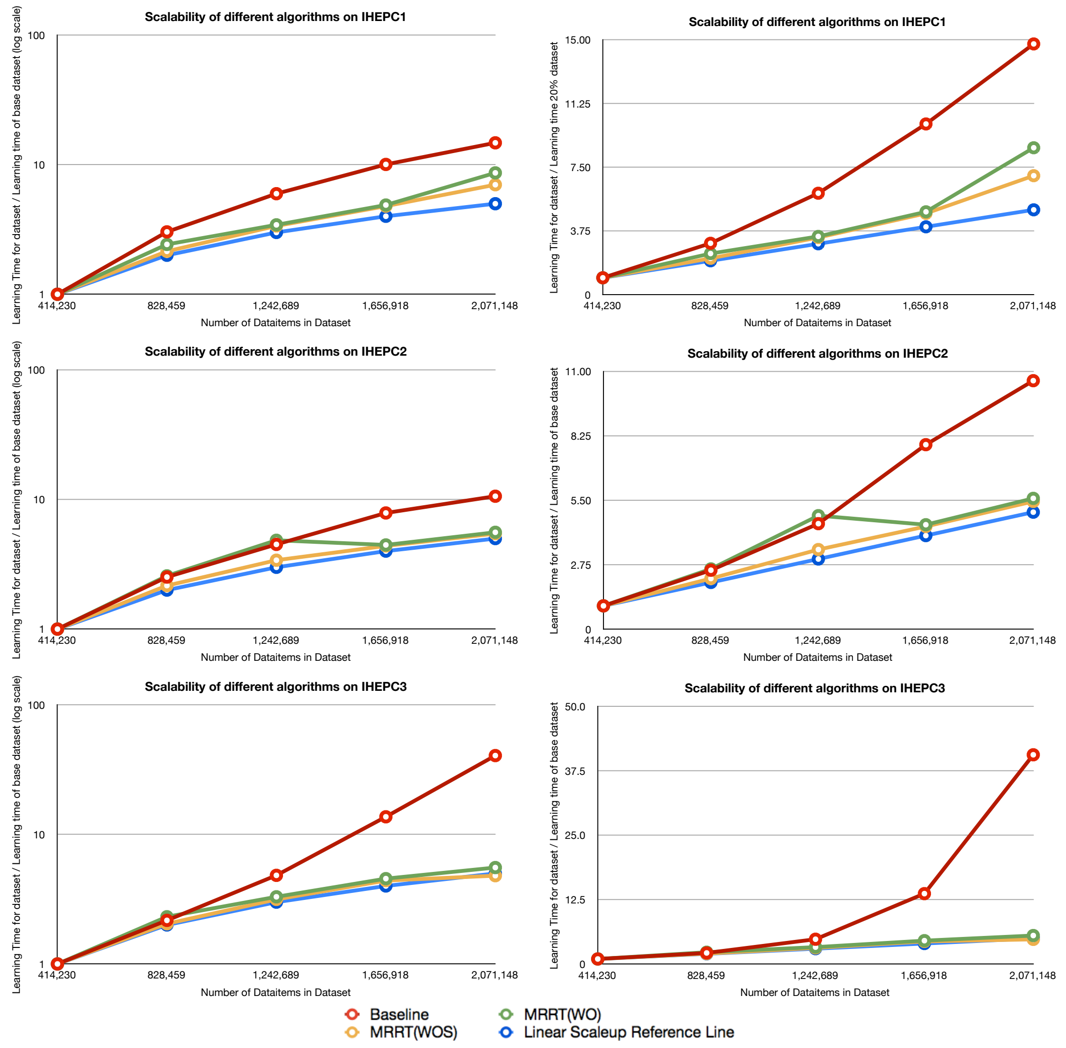

Figure 4.12: Analyzing scalability of baseline, MRRT(WO) and MRRT(WOS) algorithms on $IHEPC_1$, $IHEPC_2$, and $IHEPC_3$ real datasets with overlap = 0.75 when changing the dataset size from 103,557 items to 2,071,148 data itmes.

Comparing scalability of MRRT(WOS) and baseline algorithm also shows that MRRT(WOS)'s scalability is higher than baseline algorithm and thus MRRT(WOS) algorithm would be faster than baseline algorithm when the dataset is larger.

4.4.8 Could MRRT Be Used as a Sequential Algorithm?

We know that MRRT's accuracy results is valid for both sequential and parallel version of the algorithm (i.e. MRRT(WOS), and MRRT(WO) would have same RMSE values on same dataset). In section 4.4.3 we experimented if MRRT algorithm can achieve a RMSE lower than baseline algorithm, and results were promising. Moreover we presented a method (that could be used in sequential setting) by which the best dimensions to split along could be chosen in an effective way (see section 4.4.4). In section 4.4.5 also we showed that MRRT algorithm's prediction time is lower than baseline algorithm by more than 80% improvement. There we discussed that the prediction is assumed to be performed on a single machine for all data items in the dataset. In section 4.4.7 we also observed that MRRT(WOS) scales better than baseline algorithm when dataset size is lager.

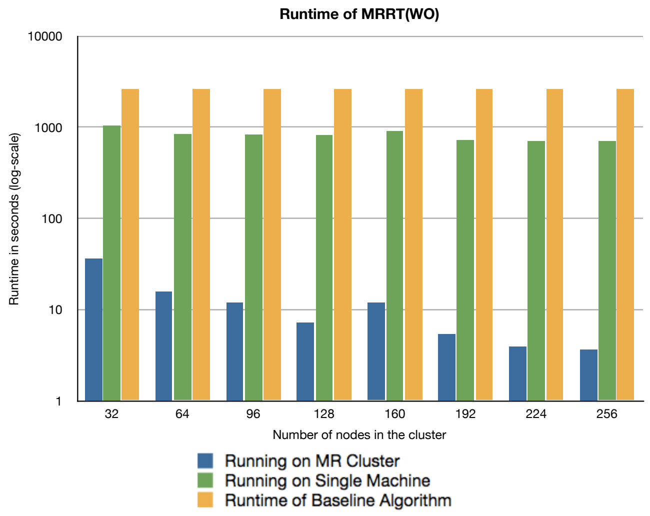

Figure 4.13: Comparing runtime of MRRT(WO), MRRT(WOS) and baseline algorithm on ttoy20d3 dataset with overlap = 0.75 when splitting along first dimensions.

In this section we run one more experiment to answer this question that if MRRT(WOS) could be used in as a sequential algorithm. The experiment in this section compare runtime of MRRT(WO), MRRT(WOS) and baseline algorithms when dataset size is fixed and number of nodes in the cluster changes. Although we know from results in section 4.4.7 that

MRRT(WOS) scales up is higher than baseline algorithm when dataset size is increasing, but we would like to show the runtime difference when dataset size is fixed and number of nodes is changing.

Table 4.15: Comparing learning time of MRRT(WOS) and baseline algorithm on 20-dimensional ttoy20d3 synthetic dataset on different number of subspaces. MRRT(WOS) always perform better than baseline algorithm although it also has better accuracy.

Baseline		MRRT(WOS)	
Learning Time	Number of nodes	Learning Time	Improvement
2609.57	32	1042.33	60.06%
	64	836.15	67.96%
	96	828.62	68.25%
	128	823.53	68.44%
	160	911.25	65.08%
	192	721.88	72.34%
	224	706.52	72.93%
	256	700.10	73.17%

4.4.8.1 Result on Synthetic Datasets

Figure 4.13 depicts runtime of MRRT(WO), MRRT(WOS) and baseline algorithm on ttoy20d3 dataset. Although this experiment is not for comparing MRRT(WO)'s runtime, but comparing its runtime with other two algorithms shows how faster this algorithm is than the other two algorithms. Comparing MRRT(WOS) and baseline algorithms shows that MRRT(WOS)' runtime is smaller than baseline algorithm in all cases. The other observation is that MRRT(WOS)'s runtime is decreasing when number of cluster nodes is increasing although size of dataset is not changing. Table 4.15 also compares learning time of MRRT(WOS) and baseline algorithms in a numerical way. The results are same as results shown in figure 4.13, but in a numeric format.

Table 4.16 compares accuracy of MRRT(WOS) and baseline algorithms and shows that its accuracy is also not affected negatively by increasing number of nodes in the cluster same as its runtime.

4.4.8.2 Result on real datasets

Figure 4.14 depicts runtime of MRRT(WO), MRRT(WOS) and baseline algorithm on all three real datasets. Comparing MRRT(WOS) and baseline algorithms confirms the results on synthetic dataset and we see that MRRT(WOS)'s runtime is smaller than baseline algorithm in all cases except two case when number of cluster nodes is 32. The other observation about

Table 4.16: Comparing accuracy of MRRT(WOS) and baseline algorithm on 20-dimensional ttoy20d3 synthetic datasets on different number of subspaces when dataset is divided into supspaces along first dimension. MRRT(WOS) algorithm's RMSE is lower than baseline algorithm in all cases.

Baseline		MRRT(WOS)	
RMSE	Number of subspaces	RMSE	Improvement
541.87	32	457.56	15.56%
	64	443.68	18.12%
	96	449.71	17.01%
	128	448.71	17.19%
	160	449.69	17.01%
	192	468.19	13.60%
	224	471.16	13.05%
	256	469.49	13.36%

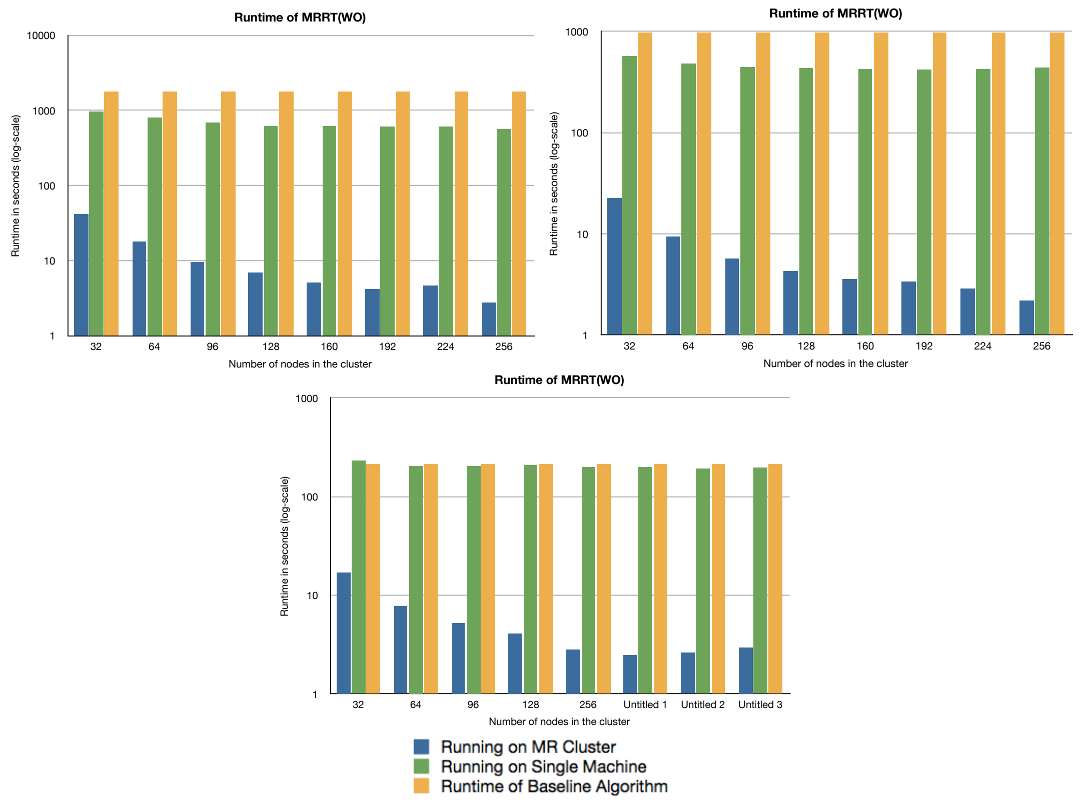

Figure 4.14: Comparing runtime of MRRT(WO), MRRT(WOS) and baseline algorithm on $IHEPC_1$, $IHEPC_2$, and $IHEPC_3$ real datasets with overlap $= 0.75$ when splitting along first dimensions.

synthetic dataset is also confirmed here and we see that MRRT(WOS)'s runtime is decreasing when number of cluster nodes is increasing (size of dataset is not changing). This decrease is significant in some cases. For example for $IHEPC_1$ dataset, the runtime of MRRT(WOS) is $\frac{2}{3}$ of runtime of baseline algorithm when number of cluster nodes is 32, but it reduces to around $\frac{1}{3}$ when number of nodes in the clusters is increased to 256 (with fixed dataset size). Table 4.17 also depicts same information as figure 4.14 but in numeric format and shows that runtime improvement percentage increases when number of nodes in the cluster increases.

Table 4.17: Comparing learning time of MRRT(WOS) and baseline algorithm on real datasets on different number of subspaces. MRRT(WOS) algorithm's learning time is always less than baseline algorithm except in one case when number of subspaces is 32.

	Baseline		MRRT(WOS)	
Dataset	Learning Time	Number of subspaces	Learning Time	Improvement
IHEPC1	1788.09	32	963.28	46.13%
		64	801.90	55.15%
		96	688.59	61.49%
		128	624.59	65.07%
		160	623.86	65.11%
		192	612.51	65.75%
		224	612.66	65.74%
		256	564.36	68.44%
IHEPC2	974.66	32	571.03	41.41%
		64	481.69	50.58%
		96	445.01	54.34%
		128	434.61	55.41%
		160	427.00	56.19%
		192	419.31	56.98%
		224	428.89	56.00%
		256	439.17	54.94%
IHEPC3	212.89	32	231.20	-8.60%
		64	205.13	3.65%
		96	204.55	3.92%
		128	208.41	2.10%
		160	199.57	6.26%
		192	198.70	6.67%
		224	192.18	9.73%
		256	196.52	7.69%

Table 4.18 confirms the result with synthetic dataset and shows that accuracy of MRRT(WOS) is also not affected negatively by increasing number of nodes in the cluster when experimenting real datasets.

Table 4.18: Comparing accuracy of MRRT(WOS) and baseline algorithm on three real datasets on different number of subspaces when dataset is divided into supspaces along first dimension. MRRT(WOS) algorithm's RMSE is lower than baseline algorithm in all cases, and it mostly decreases with increasing number of subspaces.

	Baseline		MRRT(WOS)	
Dataset	RMSE	Number of subspaces	RMSE	Improvement
$IHEPC_1$	3.55	32	3.40	4.23%
		64	3.13	11.83%
		96	3.29	7.32%
		128	3.21	9.58%
		160	3.31	6.76%
		192	3.29	7.32%
		224	3.34	5.92%
		256	3.28	7.61%
$IHEPC_2$	3.08	32	2.69	12.66%
		64	2.43	21.10%
		96	2.68	12.99%
		128	2.62	14.94%
		160	2.83	8.12%
		192	2.69	12.66%
		224	2.61	15.26%
		256	2.82	8.44%
$IHEPC_3$	2.79	32	2.71	2.87%
		64	2.46	11.83%
		96	2.63	5.73%
		128	2.86	-2.51%
		160	2.75	1.43%
		192	2.81	-0.72%
		224	2.85	-2.15%
		256	2.75	1.43%

4.4.8.3 Summary

In this subsection we overviewed parts of preceding experiments that could be related to using MRRT as a sequential algorithm. Accuracy, prediction time, learning time, sensitivity of accuracy and learning time to number of cluster nodes, scalability, and speedup are features that summarized and experimented in this section and confirmed that MRRT(WOS) is over-performing baseline algorithm in most cases. This suggests that MRRT(WOS) algorithm is not only could be used in a parallel fashion, but also could be used as a sequential algorithm on a single machine.

4.5 Slope-changing Experiments Results

4.5.1 Slope-changing Algorithm Limitation

Slope-changing algorithm finds split points on all dimensions of the dataset. That means that if we have d dimensions, and s split points on each dimension, we would have s^d subspaces. This is 531,441 when $d = 12$ and $s = 3$ which is a very big number for a 12 dimensional dataset. If we want to have even 5 data points in each subspace, we would need $2,657,205$ data items (n). That is $n \geq s^d * k$ data item is needed in order to have k data points in each cell. Changing number of dimensions to 15, would restrict number of data points to 71,744,535. As you have noticed it is even when we have considered a low number of split points for each dimensions. In a dimension with this number of split points finding Slope-changing points is not useful. Because of this limitation, Slope-changing algorithm is not working for high dimensional datasets and we need to work on it to see how we can improve this problem.

For this reason we are not running big dataset experiments on this algorithm and we would not be able to run experiment on speedup and scalability. We only compare its accuracy and runtime to baseline algorithm. The dataset w used to experiments in this section are four datasets listed in table 4.19. All of the datasets

Table 4.19: Summary of synthetic datasets

Dataset	Model Type	Axes	Training size	Test size
gtoy10d1	Gaussian Mixture	2	100,000	1,000
gtoy10d2	Gaussian Mixture	2		
ptoy20d1	Polynomial	2		
ttoy20d2	Trigonometry	2		

4.5.2 Comparing Accuracy of Slope-changing Algorithm to Baseline Algorithm

This experiment compares accuracy of Slope-changing algorithm and baseline algorithm. Both algorithms are run on four datasets and RMSE error on test set is calculated. Number of mappers for Slope-changing algorithm is set to 64, and number of reducers is set to 1. Due to randomness of one step of the Slope-changing algorithm, result might change from one run to another, thus the experiment is run 10 times for Slope-changing algorithms and the average values are used for comparison.

Results of this experiment is listed in table 4.20, and as it can be seen, baseline algorithm performs better than both Slope-changing algorithms, and Slope-changing(PWC) performs better than Slope-changing(FPS). The reason for low accuracy of Slope-changing algorithm is that some split points are selected very close to each other and some far from each other. For this reason some subspaces of the feature space is very large and some are very small. There is no problem with subspaces that their size is not very big or small. Small subspace would not have any data points and when predicting if a data point lands into that subspace there would be no model for that subspace and thus neighboring model would be used for prediction. In the worst case if neighboring subspaces are also not available, then the global average of the target value for the training set is used as the predicted value of test item. For this reason very small subspaces generally would lead to a high prediction error. Big subspaces also would have high prediction error. The reason is that when s subspace is big, a linear model with high accuracy would not fit in that subspace and thus prediction error would be high.

Table 4.20: Comparing accuracy of slope-changing algorithm (PWC and FPS versions) and baseline algorithm on four datasets.

	Baseline	Slope-changing Algorithm	
Dataset	RMSE	FPS RMSE	PWS RMSE
gtoy2d1	79.52	231.12	85.226
gtoy2d2	62.98	243.79	92.30
ptoy2d1	1.64	9.15	1.95
ttoy2d1	7.64	34.46	10.40

4.5.3 Comparing Runtime of Slope-changing Algorithm to Baseline Algorithm

This experiment compares learning time of Slope-changing algorithm and baseline algorithm. Both algorithms are run on four datasets and RMSE error on test set is calculated. Number of mappers for Slope-changing algorithm is set to 64, and number of reducers is set to 1. Due to randomness of one step of the Slope-changing algorithm, result might change from one run to another, thus the experiment is run 10 times for Slope-changing algorithms and the average values are used for comparison.

Results of this experiment is listed in table 4.20, and we expected, Slope-changing algorithms runs faster than baseline algorithm and Slope-changing(FPS) runs faster better than Slope-changing(PWC). The reason why FPS version runs faster than PWC is that, FPS use a randomized method to choose among split points that is of linear complexity in number of split points. On the other hand PWS needs that density estimation is calculated for every candidate split point. This needs to calculate share of each candidate split point in density of every other candidate split point which is of quadratic complexity in number of candidate split points.

Table 4.21: Comparing learning time of slope-changing algorithm (PWC and FPS versions) and baseline algorithm on four datasets.

	Baseline	Slope-changing Algorithm	
Dataset	Learning Time	FPS Learning Time	PWS Learning Time
gtoy2d1	7.26	0.05	0.48
gtoy2d2	14.85	0.04	0.42
ptoy2d1	14.73	0.05	0.83
ttoy2d1	17.23	0.04	0.53

Chapter 5

Concluding Remarks

5.1 Summary of Findings

- MRRT algorithm reduces the prediction time significantly (more than 80%) comparing to baseline algorithm.
- Proposed *preProcess* method is helping to reduce expected valud of MRRT's RMSE to less than baseline algorithm's RMSE for 10 out 13 datasets.
- Weighted prediction of MRRT algorithm helps to increase accuracy comparing to baseline algorithm for most datasets.
- Overlapping subspaces (coupled with weighted prediction) not only solves the data distributed-ness problem but also helps to improve accuracy comparing to baseline algorithm
- MRRT Improves the prediction time by more than 80%.
- MRRT could be used on a single machine, and in that case it improves the learning time by 60% (in most cases) comparing to baseline algorithm, and it also shows to be of close to linear scalability (comparing to baseline algorithm which is far from linear scalability).

5.2 Possible Improvements

- When splitting the feature space along one dimension, each subspace would have only two neighbors and a weighted average by weights of 2, 1, 1 (main model and two neighbors respectively) is returned as predicted value. There are possibilities that other weight combination works better for different datasets. Maybe a preprocessing step would help to determine a good combination for weights in order to increase the accuracy of algorithm. We also observed in experiments that accuracy of MRRT(W) is lower than unweighted MRRT for two datasets. This means that weighted method for prediction might sometimes have negative effect. Again a preprocess method might help with answering this question, and the algorithm would be able to decide if it should use weighted prediction or not.

- When talking about number of dimensions to split, we only compared two cases of splitting along one dimension or two dimensions. More experiments could be done in future to asses the effect of splitting on more dimensions on accuracy of generated model and also learning time. If it is found that splitting along several dimensions is better than splitting along one dimension, using weighted method of prediction would be more challenging, because we will have 2^d neighbors for each subspace where d is number of dimensions to split along.

- We have suggested the *preProcess* method to find the best dimension to split along. What could be other ways to find best dimension to split along? How about if we decide that splitting along several dimensions is better? How would we select those dimensions to split along in this case?

- More experiments needs to be done and more datasets needs to be used to find out why weighted prediction decreases accuracy of prediction for two datasets. We think that maybe a partially weighted prediction might be useful and increase the accuracy of MRRT algorithm for all datasets. By partially weighted we mean we could decide on what neighbors would increase prediction accuracy and what neighbors would decrease it. This way a preprocess method would help in increasing accuracy of MRRT.

- When using overlapped subspaces, we chose overlap of .75 to 1 based on the experiments. We believe that overlap amount is a dataset dependent feature, and finding a way to tune overlap amount for a dataset by a preprocess method could be a future work.

- We only tested the MRRT algorithm by three real datasets. Testing the algorithm with more real datasets, and dataset with higher number of dimensions (synthetic or real) would test the quality of algorithm more accurately.

- The algorithms presented in this work are compared only to Regression Tree algorithm implemented in Matlab library. It would be nice to compare it with other regression algorithms.

- We proposed a method for preprocessing and finding the best dimension to split. The experiments showed that this method works and is able to suggest the best dimension to split in most cases, but we did not run experiments about cost of this preprocess method and it could be done in future works.

Appendix A

Synthetic Datasets Details

- **ptoy10d1:**
  $y = (2x_1^2 + x_2^2 - x_3^3 - 3x_4^2 + 2x_5^3 - x_6 + 2x_7^2 - 5x_8^2 + 2x_9^3 + 2x_{10}^3)/10^2 + 20$

- **ptoy10d2:**
  $y = (2x_1^2 - x_2^2 - x_3^3 + 3x_4 + 2x_5^3 + 5x_6 + 22x_7^2 - 5x_8^2 + 2x_9^3 + 2x_{10}^3)/10^3 + 100$

- **ttoy10d1:**
  $y = 250.sin(x_1) + 40.cos(x_2) - 150.sin(x_3) + 100.cos(x_4) + sin(x_5) + 100.cos(x_6) - 500.sin(x_7) + 40.cos(x_8) - 200.sin(x_9) + 300.cos(x_{10}) + 20$

- **ttoy10d2:**
  $y = 250.sin(x_1) + 40.cos(x_2) - 15.cos(x_3) + 100.cos(x_4) + 25.sin(x_5) + cos(x_6) - 50.sin(x_7) + cos(x_8) + sin(x_9) + 30.cos(x_{10}) + 50$

- **ptoy20d1:**
  $y = (2x_1^2 + x_2^3 - x_3^3 - 3x_4^2 + 2x_5^3 + x_6 + 22x_7^2 - 5x_8^2 + 2x_9^7 + 2x_{10}^3 + 2x_{11}^2 + x_{12}^2 - 2x_{13} - 3x_{14}^4 + 21x_{15}^2 + x_{16}^3 + 2x_{17}^4 - 15x_{18}^3 + 21x_{19}^4 + 2x_{20}^5)/10^6$

- **ptoy20d2:**
  $y = (x_1^3 - 2x_2^4 - x_3^2 - 3x_4^3 + 2x_5^2 + 2x_6 + 22x_7^2 - 5x_8^2 + 2x_9^7 + 2x_{10}^3 + x_{11}^2 - x_{12}^2 - 2x_{13} +$

$3x_{14}^3 + 2x_{15}^2 + x_{16}^3 + x_{17}^3 - 15x_{18}^3 + 21x_{19}^3 + x_{20}^5)/10^6$

- **ttoy20d1:**
$y = 250.sin(x_1) + 400.cos(x_2) - 150.sin(x_3) + 10.sin(x_4) + 25.sin(x_5) + 100.cos(x_6) - 5.sin(x_7) + 400.cos(x_8) - 20.sin(x_9) + 301.cos(x_{10}) + 250.sin(x_{11}) + 400.cos(x_{12}) - 150.sin(x_{13}) + 10.sin(x_{14}) + 25.sin(x_{15}) + 100.cos(x_{16}) - 5.sin(x_{17}) + 400.cos(x_{18}) - 20.sin(x_{19}) + 301.cos(x_{20}) + 200$

- **ttoy20d2:**
$y = 350.sin(x_1) + 300.cos(x_2) - 250.cos(x_3) + 100.sin(x_4) + 215.sin(x_5) + 100.sin(x_6) - 52.sin(x_7) + 40.cos(x_8) - 23.cos(x_9) - 31.cos(x_{10}) + 123.sin(x_{11}) - 400.cos(x_{12}) - 150.sin(x_{13}) + 101.cos(x_{14}) + 251.sin(x_{15}) + 10.cos(x_{16}) - 51.sin(x_{17}) + 40.cos(x_{18}) + 200.sin(x_{19}) + 31.cos(x_{20}) + 200$

- **ttoy20d3:** Same equation as ttoy20d2, but number of data items is different.

- **gtoy10d1:** A mixture model of three gaussians with means:

$$\begin{aligned} m_1 &= [5, 8, 5, 9, 10, 7, 12, 11, 8, 8] \\ m_2 &= [9, 19, 10, 10, 13, 14, 14, 16, 11, 2] \\ m_3 &= [15, 3, 7, 8, 1, 4, 4, 6, 12, 12] \\ m_4 &= [5, 13, 7, 8, 11, 4, 14, 6, 2, 12] \end{aligned}$$

and variances:
$s_1 =$

$$\begin{pmatrix} 15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 12 \end{pmatrix}$$

$s_2 =$

$$\begin{pmatrix} 15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 19 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 12 \end{pmatrix}$$

$s_3 =$

$$\begin{pmatrix} 15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 12 \end{pmatrix}$$

$s_4 =$

$$\begin{pmatrix} 17 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 19 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 19 \end{pmatrix}$$

- **gtoy10d2:** A mixture model of four gaussians with means:

$$m_1 = [5, 8, 5, 3, 10, 4, 15, 5, 8, 8]$$
$$m_2 = [8, 9, 10, 10, 13, 14, 14, 16, 5, 2]$$
$$m_3 = [11, 5, 5, 5, 5, 5, 5, 5, 5, 5]$$
$$m_4 = [15, 15, 15, 15, 15, 15, 15, 15, 15, 15]$$

and variances:

$s_1 =$

$$\begin{pmatrix} 11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 12 \end{pmatrix}$$

$s_2 =$

$$\begin{pmatrix} 12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 12 \end{pmatrix}$$

$$s_3 = \begin{pmatrix} 13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 17 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 18 \end{pmatrix}$$

$$s_4 = \begin{pmatrix} 17 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 19 \end{pmatrix}$$

Bibliography

[1] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[2] Ankur Dave, Wei Lu, Jared Jackson, and Roger Barga. Cloudclustering: Toward an iterative data processing pattern on the cloud. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1132–1137. IEEE, 2011.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification.* Wiley-interscience, 2012.

[5] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[6] Alina Ene, Sungjin Im, and Benjamin Moseley. Fast clustering using mapreduce. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 681–689. ACM, 2011.

[7] Robson Leonardo Ferreira Cordeiro, Caetano Traina Junior, Agma Juci Machado Traina, Julio López, U Kang, and Christos Faloutsos. Clustering very large multi-dimensional datasets with mapreduce. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 690–698. ACM, 2011.

[8] David J Hand, Heikki Mannila, and Padhraic Smyth. *Principles of data mining.* MIT press, 2001.

[9] Georges Hebrail and Alice Berard. Individual household electric power consumption data set, UCI machine learning repository, 2012. "`http://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption`".

[10] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. An online algorithm for segmenting time series. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 289–296. IEEE, 2001.

[11] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. Segmenting time series: A survey and novel approach. *Data mining in time series databases*, 57:1–22, 2004.

[12] Chuck Lam. *Hadoop in action*. Manning Publications Co., 2010.

[13] Daniel Lemire. A better alternative to piecewise linear time series segmentation. *SIAM Data Mining*, 2007.

[14] Jimmy Lin. Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail! *Big Data*, 2012.

[15] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got fired for using hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, page 2. ACM, 2012.

[16] Malte Schwarzkopf, Derek G Murray, and Steven Hand. The seven deadly sins of cloud computing research. *HotCloud, June*, 2012.

[17] Patrick O Stalph, Jérémie Rubinsztajn, Olivier Sigaud, and Martin V Butz. A comparative study: Function approximation with lwpr and xcsf. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, pages 1863–1870. ACM, 2010.

[18] Wikipedia The Free Encyclopedia. Image, "`http://en.wikipedia.org/wiki/File:Comparison_of_1D_histogram_and_KDE.png`". [Online; accessed 28-March-2013].

[19] Dennis van Heijst, Rob Potharst, and Michiel van Wezel. A support system for predicting ebay end prices. *Decision Support Systems*, 44(4):970–982, 2008.

[20] Sethu Vijayakumar and Stefan Schaal. Locally weighted projection regression: An o (n) algorithm for incremental real time learning in high dimensional space. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000)*, volume 1, pages 288–293, 2000.

[21] Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate Research*, 30(1):79, 2005.

[22] Stewart W Wilson. Classifiers that approximate functions. *Natural Computing*, 1(2-3):211–234, 2002.

[23] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[24] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In *Cloud Computing*, pages 674–679. Springer, 2009.