Using a Randomized Regression Approach to Estimate Hospital Admissions to Reduce Emergency Department Holding

by
Kleber Andres Garcia

Bachelor of Science in Computer Science
Florida Institute of Technology, Melbourne Fl
2009

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

Master of Science
In
Computer Science

Melbourne, Florida
May, 2011

Abstract

"Using a Randomized Regression Approach
to Estimate Hospital Admissions
to Reduce Emergency Department Holding"
By Kleber Andres Garcia
Thesis Advisor: Philip Chan, Ph.D.

Serious patients from the Emergency Department need to be admitted into the hospital for more specialized care. However, beds might not be available, which results in the patients being held in the Emergency Department. One reason is difficulty in estimating the number of patients accurately, which leads to the challenge of scheduling staff appropriately. This study proposes KGERS (K Gaussian Elimination on Randomized Subsets), a randomized algorithm for linear regression. KGERS is a good fit for Regression Trees that perform piecewise linear approximation (for nonlinear regression) because the data at the leaf nodes presents a roughly linear pattern by design. Empirical evidence shows that KGERS is able to perform regression on hospital data faster and more precise (error of 16%) than other traditional algorithms such as Neural Networks. Additionally, an analysis on synthetic data sets shows that KGERS is an efficient algorithm that degrades less under regular quantities of noise found in real life data sets.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The applications of machine learning have provided valuable contributions to our society. From video games to actual industrial applications, they all aim to push the power of computing to another level; a level of cognition and true understanding of data. This research aims to address an existing problem with hospitals over the nation and use modern machine learning techniques to solve it. The problems that hospital faces are based on estimation of future admissions that could cause bottlenecks in the workflow process, adding holding time for patients. This study aims to show that our proposed randomized regression algorithms are better suited for estimation of these admissions than some of the existing algorithms. Additionally, we discuss the strengths and weaknesses of our algorithms on synthetic data containing properties commonly found in the real world.

1.1. Motivation

Hospitals can be very crowded places, especially in the emergency room. They tend to overcrowd, especially when the hospital hasn't allocated enough resources for a particular "spike" day. The Emergency Department (ED) first accepts people coming from unexpected accidents or simply very risky sign of sicknesses that need immediate attention. If the patients need more care, they are admitted to the hospital. However, beds might not be available in the main hospital. Consequently they are being held in the ED. This could reduce the capacity of ED to treat new patients. Also the patients are not getting specialized care in the main hospital. In this study we investigate admissions to the hospital from the ED.

One reason for patient holding in the ED is staffing. In order to maintain a bed for a patient inside the hospital, nurses and doctors must be assigned to it. This requires calling ahead of time whichever staff member needs to be there. If the admission rate (coming from the ED) is unknown, the hospital has to guess on how and where to allocate the staff. A consequence of this problem is not having enough staff, and start adding people back to the ED placement. The next consequence is that now the ED is crowded with patients that are not supposed to be there, because of understaffing from the admissions side. The vicious cycle continues and hence the bottleneck and long waiting lines.

It is imperative for hospitals around the nation to have some sort of computational models that will help dealing with this problem, by estimating future admission rates. Estimating admission values will help with staffing, and hopefully reduce holding time for patients waiting to be admitted.

1.2. Problem Statement

1. *Estimation of future hospital Admissions from ED data:*
   Given data from the past, the first problem will be to build a model of this data and try to estimate future admissions to the hospital. The data comes as patient admissions from ED department, PCU, ICU and Floor unit admissions (more details in Chapter 4).
2. *Faster Non- Linear Regression:*
   Piecewise linear regression will be one of the methods used to do these admission estimations. As a result, a fast and reliable algorithm for linear regression is required in order to accomplish this goal.
3. *Analysis of algorithms in different environments:*
   Amongst the algorithms used, applicability to other data sets will be measured. The main question of this sub problem is: how likely are these algorithms to fail under noisy environments?

1.3. Overall Approach

To solve the first problem in the last section a set of existent algorithms will be used. On top of this, the data will be treated as a time series and several features will be derived. The reason why it is treated as a time series is because this is perhaps the simplest way to do an initial study on this data while not considering individual patient details. Features in the form of multidimensional vectors will be derived out of these time series admissions data sets.

The second problem will require a new algorithm that will satisfy the requirement of fast computation. We proposed a randomized algorithm called KGERS (K Gaussian Elimination from Randomized Subsets).

Lastly, for the third problem we will generate data with noise and irrelevant features. We then analyze the behavior of the different algorithms.

1.4. Overview of contributions

1. We introduced randomized algorithm (KGERS) for efficient linear regression and incorporate it into Regression Tree algorithms
2. For admissions from ED, our proposed algorithm (Regression Tree KGERS) is faster and more accurate than Neural Networks.

3. On a trial basis, the hospital using models proposed on this study to estimate future admissions.
4. Estimation of unit types (PCU, ICU and Floor) presents good accuracy in terms of the hospital, which spans around 16% of error.
5. Data degradation does not cause a high increase in errors in the Regression Tree KGERS (errors manage to stay around 26%).
6. KGERS time complexity is independent of N (size of input). KGERS time complexity is $O(M^3)$ where M is the number of dimensions.
7. Because the leaves in Regression Trees (using a linear splitting strategy) are mostly linear by design, KGERS is a proper efficient algorithm for linear regression at the leaves.

1.5. Chapters Overview

This study is divided into five main chapters. Chapter 2 will be an overall literature review and study on this topic. This chapter contains information on the previous machine learning algorithms mentioned. The two most important discussed will be Regression Trees and Feed Forward Neural Networks trained using back propagation. There is also information regarding previous studies done on the subject. These studies include semantic analysis on the data (which differs from treating the data as a time series problem) and modeling time series using ARMA models.

Chapter 3 will explain the linear regression algorithm proposed which is based on randomized search. This algorithm is KGERS. The purpose of this algorithm are explained and later linked to the concept of generating regression trees that perform piecewise linear regression. Other algorithms for regression trees are proposed specifically for splitting and stopping criteria.

Chapter 4 will be the experimental results on the hospital data. Contributions, weaknesses and results will be discussed here. An overview of the experimental process will be also given.

Chapter 5 will test the performance of the algorithms (algorithms proposed and cited in chapter 2) on synthetic data. This synthetic data meant to be an easy target for regression, and is used to test the effects of degradation and noise over the respective algorithms (noise will be added to this synthetic data).

Finally, Chapter 6 will give an overall overview of the findings, conclusions and further proposal and improvements of this work.

Chapter 2

Related Work

In this chapter we will show the resources available on the subject of study. There exist a vast number of papers that deal with hospital admission data prediction. This chapter is divided into four separate sections, each one containing a brief literature review on the subtopic studied. The first section talks about general existent approaches to forecasting. Several approaches are discussed here, such as simulation and some of the general features used. The second section discusses treating the forecasting problem itself as a time series. Several methods are explored, such as ARMA models and Neural Networks. The fourth section introduces Neural Networks and existent algorithms related to this topic. Finally, the fourth section introduces Regression Trees, its algorithms and applications to forecasting problems.

2.1. General Forecasting Approaches

Many existing forecasting approaches for hospital admission data involve the usage of experimental models. These experimental models are then configured under several variable constrains and executed. The result of such simulation is what is thought to be the actual forecasting which is believed to contain a manageable error range. This study's approach takes a different line, and treats the problem as a machine learning challenge.

2.1.1. Hospital admissions features

There has been several attempts in the past, such as that one by Leegon Jeffrey et al. (2006) involving the usage of neural networks. In his paper he attempts to predict the admissions of a hospital using a neural network.

The most interesting aspect of the paper is the features that they decide to implement. Most of this involves semantics of the data itself, as in very specific patient features. Data is feed from previous patients into the neural network. The neural network builds a model of a patient likely to be admitted into the hospital from the symptoms he presents to the ER. The new patients arrive, and the neural network tries to identify those patients that are likely to be admitted. This information is used to allocate the respective resources and staffing that the hospital requires. Additionally to this, the claim states that the hospital is also able to predict workload by just using this patient information.

A problem with this technique is the lack of flexibility on dates we want to predict. If the hospital wants to know the likely demand of patients in a week, they will have to wait several days before the actual day of the prediction. The reason is that each patient has to go through the learning process of the neural network in a real time fashion. Jiexun Li et al. (2009) also proposes a similar technique. This time they use a pre processed Chief Complain. Chief Complains (CC) are records that explain the reason for the patient to be hospitalized. There isn't a standard for CCs so several special strategies have to be applied in order to extract features out of them. The inputs are patient information and the output is hospital admission. The tweak here is that patient information is preprocessed (using the Raw's CC) and new features are generated by applying a CC standardization algorithm. Jiexun uses several off the shelf machine learning algorithms, such as support vector machines and decision trees. The standardization algorithm is explained in section 3.1. of Jiexun Li et al. (2009). In order to extract features out of these chief complains, the following strategies are applied:

- *Using semantic-enhanced features vectors*: Here the CC is still a variable, but instead of treating each CC as a dummy variable, a value is assigned to a CC feature by considering not only its occurrence but its related CCs. This process is explained in detail in section 3.2.1 of the Jiexun's study.

- *Performing data transformation*: In a high level, this process performs a semantic analysis on the chief complains. They also call this *Semantic Kernel Learning* (Jiexun Li et al, 2009). What this does is that several kernel functions are defined (for example comparing how similar are two patients or two CC's). At the end a linear combination is applied for all patients and the value is normalized, ready to be used by the learning algorithm as a feature vector.

Results show that by incorporating semantics an improvement in admission prediction was achieved (section 4.1 Jiexun et al. 2009).

2.1.2. Machine learning algorithms

What these methods have in common is the usage of data for some sort of operations prediction. Matsunaga and Fortes (2010) have devised a new approach, based on an algorithm called PQR (Predicting Query Time). This is a classification tree algorithm that at the end allows a regression problem to be fit at the leaf nodes. The new algorithm that they have implemented (PQR2) has better results on their target data set. The data set trying to model is execution time, memory and disk consumption of two applications.

These applications will also be running under different scenarios, and the goal is to model all these as features and apply machine learning algorithms yielding to low error.

PQR algorithm, just like any regression tree algorithm, consists of three main operations. These are: splitting criteria, stopping criteria and regression at leaf nodes. For splitting criteria, Matsunaga et al have chosen very simple maximization of normalized ranges. They will pick that feature with the biggest range in split. This will become a potential split point for the new tree. For stopping criteria, they use maximum number of examples filtered at leaf nodes. Whenever the regression tree reaches a maximum number of leaf nodes it will stop growing and apply the third operation which is regression. During regression the PQR tree does an average of the outputs of these remaining training values. A variation of the PQR method will be explored later, for now this is perhaps the simplest off the shelf regression tree. What PQR2 does is that it tries to fit the best regression model at leafs using a validation set. This of course reduces the number of training examples, but gives a hint of the best predictor found under the particular leaf. Finally, the best predictor is chosen and incorporated into the tree. The steps of PQR2 can be summarized in Figure 2.1.

```
PQR2
-inputs:
X(inputs), Y(outputs), S (max size of examples)
-outputs: PQR2 Tree
1: Begin
2: If (size(X) < S)
3:   Split X into X’ and Xvalidation using Y’s
     distribution
4:   For each regression candidate R choose the best
     one trained on X’ and evaluation on Xvalidation
5:   return Leaf node R
6: Else
7:   Choose best split for X
8:   let pqr.left = PQR2(Xleft,Yleft,S)
9:   let pqr.right= PQR2(Xright,Yright,S)
10:  return pqr
11:  END
```

Figure 2.1 PQR2 algorithm

While providing good results on the data sets, the algorithm is not tested for over fitting. The original source of the PQR tree comes from Chetan et al. (2008). The explanation of this original PQR tree is a bit more in-depth than that one shown by Matsunaga and Fortes (2010). The real algorithm produces a tree with two patterns: leaf node and internal node. The internal node is a range and a classifier function. The range represents the possible values this tree can predict well, and the classifier function (k means, or any other classification algorithm in general). The classifier will tell the current set of attributes to predict on which 2 child nodes to go. This happens recursively until we reach the leaf nodes, the leaf nodes are simply ranges of the possible values for the current class. Very similar from regression tree, but differs in the sense that the samples are split by distribution rather than a single class attribute. Another interesting thing is the introduction of several optimization and post pruning processes involved. Clearly these ones are not included in that paper by Matsunaga and Fortes (2010). At the end, the big difference here is the flexibility of PQR2, which is able to integrate different models into the leaf nodes of the tree.

So far a couple of machine learning algorithms and features have been exposed that tackle problems very similar to that one of hospital admissions. All these algorithms and features want to extend the idea of generating a model that reduces error. Perlich et al. (2006) propose the idea of using a high quartile model instead of an error reduction model. The problem statement they are trying to tackle here is to predict the amount of money a customer is willing to pay for a product in the future. This will give good insight of how much to produce and how the customer will be allocating resources for the product.

The way this relates to our current problem is that we are also trying to predict "how much" customers (patients) are willing to be admitted into the hospital, thus both problems describe resource allocation. Perlich et al. (2006) approach this by realizing the maximum q quantile as their probability that a price is within a range. Normally, this is calculated using the formula P (Y|x) where X is the current information and Y the around the customer is willing to spend.

By using quartiles the goal changes, the new goal now is to do regression on c(x), where c(x) is the function that gives the 0.9 high quartile such that (1) satisfies.

$$P(Y \leq c(x)|x) = 0 \qquad (1)$$

The most interesting technique for this prediction is using a regression tree, which is similar to a decision tree but offers more specialization towards continuous values. While this approach is unused on the current study, it is worth the time to explore in the future. It provides a potentially useful new point of view that the hospital might be interested in, just like predicting the size of “wallets”.

2.2. Forecasting as a time series problem

So far literature on semantic analysis of the data has been explored. This means that specific features such as patient type or type of consumer are explored. The problem could be analyzed from a different perspective in the way which features are derived, and instead treat the problem as a time series. A time series is a sequence of events that follow a specific order. Each event can be though as the y axis in a sample. The x axis can be though as the time. A simple example would be the number of admissions from the hospital in a particular day.

2.2.1. ARMA models

It has been introduced in the literature many times as the basis for mathematical analysis of time series. These are ARMA models (Auto Regressive Moving Average). Montgomery and Johnson (1976) describe this model as the composition of two sub models. Both sub-models can be observed in equation (2).

$$X_t = AR(X) + MA(X) \qquad (2)$$

The auto regressive model is expressed in equation (3). The element *c* is a constant.

$$AR(X_t) + MA(X_t) = c + \varepsilon_t + \sum_{i=1}^{p} \alpha_i X_{t-i} + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i} \qquad (3)$$

The ε found in the equation represents the random shocks that each time element in the time series has. The moving average segment is represented by the linear combination of the α_i with respect to the X values. It is called moving average because it is assumed that the time series is a stationary stochastic process, whose joint probability distribution is not changing. This also means that the underlying model's mean and variance should stay constant.

The next element in an ARMA model is the Auto Regressive part. This segment uses a new set of parameters for ε. These are also called random shocks or errors as mentioned before. They should come from a normal distribution with zero mean. Details on how to derive these random shocks from the data is out of the scope of this study, but what is important is that the error is modeled in some sense from these ARMA models.

Several problems can be inferred of this. The first one being that the time series studied must be stationary. Generally, stationary time series do not exist in practice, and many of the real life problems vary vastly from them. However, there are techniques proposed by Montgomery (1976) that will convert or estimate a stationary time series from a non stationary. The conversion must be done at the end to get back to the original time series. To test for stationary and non stationary time series, several measures have been explored, such as autocorrelation and autocovariance. The result of the graph generated by these measurements will surely tell the nature of the time series being studied. Equation (4) shows a transformation proposed by many sources.

$$\hat{X}_t = X_t - X_{t-1} \tag{4}$$

This transformation involves converting the time series to a new one. The new time series would be a time series of derivatives of the original. Every time a derivative is taken, the time series becomes simpler, and closer to a stationary one.

Other transformation steps are available on the literature. Bagnall and Janacek (2004) propose a technique based on fitting a model and then applying a clustering method. The first step is that the time series is transformed into a binary series. This decision is done based on an equation similar to (5).

$$C(t) = \begin{cases} 1, Y(t) > \mu \\ 0, Y(t) \leq \mu \end{cases} \tag{5}$$

This method is called 'clipping'. The next step tries to recognize hidden time series within the data by using clustering techniques.

The way this is done is by fitting an ARMA model to each series then cluster based on similarity of the fitted parameters α and θ shown in equation (3). The reason to use clipped data is that it helps in calculation of several heuristics such as auto-correlation. It also helps with a more compact representation as bit vectors, speeding up the model fitting process. They use several methods to cluster, including Euclidean distance and Cosine similarity of parameter vectors.

These adaptive techniques by Bagnall and Janacek (2004) show good results, and are a great example of taking the mathematical background and applying it to computer science algorithms such as clustering. For evaluation, Bagnall uses a similar technique used in this paper: a sensitivity analysis of the number of clusters used for each ARMA model. Within this analysis, it can be learned that clipped data's response keeps a higher accuracy than unclipped data. One of their main arguments is the loss of information, but not structure of the time series. When clipping the data, a lot of the noise is blurred away and integrated into the model. This decreases the possibility of over fitting and likelihood of bad results.

2.2.2. Machine learning on Time series

As seen previously, there exist many off the shelf machine learning algorithms, each one with particular weaknesses and strengths. Many of these algorithms can also be mixed with the concept of time series modeling. One example of this is Pissarenko's (2002) research work on neural networks and time series analysis. The problem they tackle is very similar in nature of errors. The techniques used have to be noise tolerant and be able to extract unknown patterns present in such. Financial time series analysis is a subject of study amongst data miners due to the quantity of data in its nature. In this study, there is a discussion about the implication of ARMA models with machine learning, and empirical evidence of its limitations. Some of the limitations mentioned by Pissarenko (2002) are the following:

- Neural Networks trained in finance require vast number of training cases.
- There is not known best Neural Network topology.
- The more complex, the more unreliable a neural network turns out to be.
- Requirement for statistical relevance on results.
- A good architecture of features.

In these studies many possibilities for features are used. Some of them include average of time windows mixed with speculated error. Others included partial derivatives of certain time windows. It was shown that predictability on unstable and non stationary time series is a big problem. Pissarenko's (2002) study shows a lot of literature exposed on the area of neural networks with time series, the impact and direction that some of them carry on.

Neural networks are not the possible set of algorithms to pick from when modeling a time series. Clustering is another possibility, as proposed by Goldin et al. (2006). The main algorithm bases prediction on average of similarity amongs subsequences within a time series. The steps of the original Goldin's algorithm can be summarized in the following listing:

- Using time series X, divide it into smaller chunks of length L
- Use kmeans to cluster these chunks
- Base predictions on clusters:
- Identify cluster which prediction best fits into
- Output prediction

The main sub problem here is figuring out some sort of distance measure that could be used for the subsequence. Euclidean distance is used for this, by using each element in the subsequence as a member of a vector. Euclidean distance is summarized in equation (6).

$$D(X,Z)=\sqrt{\sum_{i=1}^{n}(X_i-Z_i)^2} \tag{6}$$

This distance is very linear and evidence in Goldin et al.'s (2006) shows that it is not effective measuring similarity of subsequences. They introduce the notion of Cluster Shapes. These shapes use Euclidean distances as kernel functions and try to draw (using its centroid) a sorted sequence. The sequence composed is normalized against all the possible deltas within the kernels. The distance itself is a set of features derived from the Euclidean distance. To match the distances they have a special matching algorithm composed of the following steps:

- Store the resulting N constellations of clustering into a master table.
- Compute the shape of each entry that will be matched.
- Use Euclidean distance on the features described by the kernel functions.

Perhaps one of the simplest methods involving a neural network for time series forecasting is the one proposed by Nikolov (2010). In his research work, he proposes the usage of a neural network as a pure regression method for time series prediction. He describes a set of optimizations for clustering oriented towards time series learning. Given a time series for training, a set of vectors is extracted by sliding the window up to the end. Each element in the window will represent the "ith" dimension of these vectors. Clustering is performed on these vectors. Once clustering is done, for each cluster a Neural Network is trained (by splitting each cluster into training and validation sets respectively). When a new vector arrives and is queried for a prediction, it is matched with a cluster and then sent to its respective neural network. This research work also proposes some optimizations on the clustering side.

A very interesting strategy also proposed by Nicolov is the notion of reusing the neural network to generate predictions in which data is not available. This could fall under the category of simulation, but it stills uses previously defined data to draw models. The process is described in these steps:

- Use the time windows to train a neural network.
- Predict the unknown time X_{t+1}
- Incorporate X_{t+1} into the training set, and shift the window further
- Predict X_{t+2} with this new model, and repeat.

The previous steps can also be appreciated in Figure 2.2.

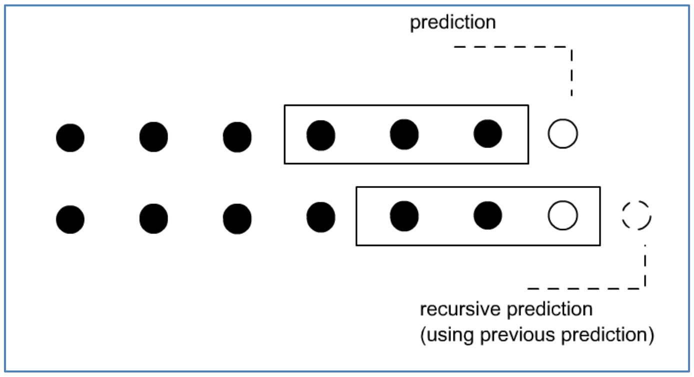


Figure 2.2 Recursive prediction (Figure 1 in Nikolov's research 2010)

What is not good about this research work is the lack of evidence supporting that this kind of unsupervised learning generates good results. While the claim that the process mentioned before is a good approach might be valid, it is still to be considered as an actual practical solution. It is very hard for a neural network to draw and predict a function more than its actual boundaries. For this to happen, the neural network would need an outstanding number of hidden layer neurons and a viable training set. Also the assumption here is that the change in seasonality of the time series remains constant. This means that the time series has to be close to stationary.

Many strategies for regression itself on the time series are also proposed by Nikilov (2010). These include self organizing maps, which will be studied in later sections of this chapter.

While contrasting these works on time series forecasting it can be noticed that all of them show good grounds of empirical evidence supporting their claims.

2.2.3. Linear Regression

Linear regression is the process of estimating the weights that would describe a hyperplane whose error with respect to a space of points is minimized. The reason why we study linear regression is because is very tied to the definition of a time series. As we saw in ARMA models, it is all about estimating the weights in a linear combination. Linear regression is also useful to understand non linear regression algorithms, such as piece wise linear approximation done by a regression tree. These will be explored in later sections, but before it will be explained here two of the most common of “the shelve” methods. These methods follow the definitions and algebra of Mitchell et al. (1997).

2.2.3.1. Gradient Descent

The main goal in gradient descent is to minimize the error squared, defined as:

$$E(\vec{w}) = \frac{1}{2}\sum_{i=0}^{n}[y_i - \hat{y}(x_i)]^2 \quad (7)$$

We will try to minimize this by setting the derivative of the error with respect to the weights (associated with ŷ). Starting we get:

$$\frac{\partial E}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_{i=0}^{n}[y_i - \hat{y}(x_i)]^2 \quad (8)$$

Solving the derivative of (8), we get a final partial derivative in terms of the weights:

$$\frac{\partial E}{\partial w_j} = \sum_{i=0}^{n}[y_i - \hat{y}(x_i)][x_{ij}] \quad (9)$$

Finally, this derivative can be written as a single weight update:

$$\Delta w_j = \eta\sum_{i=0}^{n}[y_i - \hat{y}(x_i)][x_{ij}] \quad (10)$$

The η term represents the *learning rate*. This term dictates how fast the updates for gradient descent should be. Experimental data shows that a stochastic update is better for the weights; this means that updating the weight using one data point at a time is better than aggregating the entire sum.

Finally the algorithm proposed by Mitchel et al (1997) can be written as show in Figure 2.3.

```
Gradient Descent
-inputs: X, Y, $\eta$, T
-outputs: W
1: Begin
2: Initialize wj to small values
3: For t := 0 to T
4:   For i := 0 to n
5:     For j := 0 to M
6:       If (j equals 0)
7:         Wj = Wj + $\eta$*( Yi - eval(W,Xi))
8:         Else
9:         Wj = Wj + $\eta$* (Yi -eval(W,Xi))*(Xij)
10:Return W
11:END
```

Figure 2.3. Gradient Descent Pseudocode

The *eval* function will do a linear combination of the X_j vector with the current weight vectors. At the end, error will be likely minimized, after several iterations that is. A problem with this algorithm is its greedy approach, and the fact that it might converge into a local minima (not the optimal set of weights). To summarize the steps, step 2 in Figure 2.3 will be the loop that performs updates to the weights until the algorithm converges and step 4 will go through each data point in the training data. Finally step 5 will perform the update through each weight in a stochastic manner.

2.2.3.2. Linear Least Squares

This linear approach starts with the fact that we can arrange each feature X_{ij} into a matrix:

$$X=\begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1m} \\ X_{21} & X_{22} & \cdots & X_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ X_{n1} & X_{n2} & \cdots & X_{nm} \end{bmatrix} \quad (11)$$

Each row represents a data point, and each column a dimension element. With this matrix in had we can finally model our predictions (into the ŷ vector) as follows:

$$XW = \hat{Y} \quad (12)$$

It can also be written in matrix form, the error squared quantity as show in equation (13).

$$E = ||(Y - XW)^2|| \quad (13)$$

And also the error vector:

$$R = (Y - XW) \quad (14)$$

What we want at the end is to minimize this error E. In order to do this, we can use the gradient derivative, and write it as follows:

$$\frac{\partial E}{\partial W_j} = \frac{\partial}{\partial W_j} \sum_{i=1}^{n} R_i^{\,2} \quad (15)$$

The derivative of R is explained in equation (16).

$$\frac{\partial R_i}{\partial W_j} = -X_{ij} \tag{16}$$

Replacement and rearrangement of the elements gives the normal equations.

$$(X^T X)W = X^T Y \tag{17}$$

The final task is to solve this system for W. This is summarized in Figure 2.4.

```
Linear Least Squares
-inputs: X, Y
-outputs: W
1: Begin
2: Initialize matrix A := XTX
3: Initialize vector B := XTY
4: Initialize W to be length M
5: Solve system AW:=B for W
6: Return W
7: END
```

Figure 2.4 Linear Least Squares pseudocode

Unfortunately this algorithm will not always find the answer. If the matrix X^TX is not invertible, that means that the system cannot be solved; therefore there is no global optimal. Because our chosen method to find the inverse is singular value decomposition, whenever there is no global optimal, the algorithm gives a pseudo inverse, which can be thought of a sub optimal solution.

2.3. Neural Networks Regression

So far there has been a lot of talk about neural networks. But what are these data structures? Neural networks are data structures capable of approximate any function, non-linear continuous. The principle lies in that the function must be continuous and differentiable. Neural networks try to represent and break up the problem into smaller activation segments, where each segment gets activated as a feature triggers the next neuron. This is very similar on how a real neural network works in the animal kingdom. The problem is disseminated within the network, and individual neurons process the information accordingly.

It has been shown that neural networks are a widely used technique to model both approaches, time series and semantics features. The following sections will show a couple of techniques used specifically to train neural networks, and some of their architecture.

2.3.1. Self organizing maps

Self organizing maps are a variation of neural networks. Teuvo Kohonen (2008) in his research work shows the basic steps for the algorithm to work. The main goal and purpose of this structures is to potentially reduce the dimensionality of a problem. In addition, it also creates topological relationships in the form of networks helping to categorize or even do regression. To explain how these networks work, the first step is to look and understand their structure. Figure 2.5 shows the structure of a self organizing map.

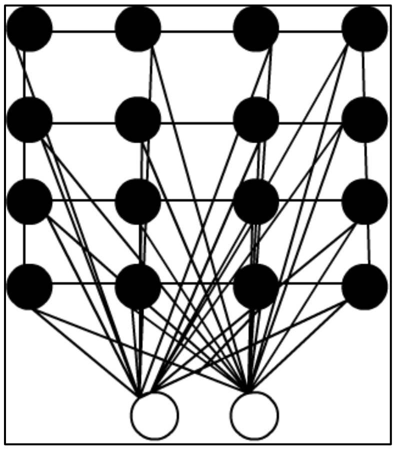


Figure 2.5 Self Organizing map architecture (Figure 4.3.a in Kohonen's book 2008)

Each internal node in the self organizing map shown in Figure 2.5 is connected to the input nodes. These input nodes are displayed in white in Figure 2.5. For simplicities' sake, we will only work with a two dimensional self organizing map, with a neuron 4x4 mesh. Each SOM contains an N dimensionality input vector (2 dimensions in the case of Figure 2.5). Each internal node is a set of weights of the same dimensionality. This means that if the input layer consists of X_i, for i = 1, 2 … N then each internal node can be described as shown in equation (18).

$$W_{x,y} = \{w_1^{x,y}, w_2^{x,y}, w_3^{x,y} ... w_n^{x,y}\} \qquad (18)$$

Each node has an x and y coordinate associated with it. In the case of Figure 2.5, the map is a 4x4 so it means that x = 1, 2... 4 and y = 1, 2… 4.

But at the end, what is the output of such data structures? The main purpose is to have a final output displaying clustering information that has a continuous nature. One famous example is clustering of colors in a two dimensional space. What we want evaluated at the end is the result of linear combination between the weights and inputs on each node in the two dimensional special grid.

Figure 2.6 shows the high level pseudo-code explaining the training process of a SOM.

```
SOM learning
-inputs:
X(inputs)
-outputs: SOM of DxD
1: Begin
2: Initialize weights of DxD map to be small
3: For each x in X
4:  Select Node whose weights matching x vector the
    best (known as the best matching unit BMU)
5:  Calculate radius of BMU's neighborhood
6:  Alter weights of BMUs radius so they look more
    like
    BMU's weights. The closer the bigger the update
7:  Reduce the BMU's radius rate
13:END
```

Figure 2.6 SOM learning algorithm

As displayed in Figure 2.6, step 4, a best matching unit must be picked. This BMU must be the set of weights that appear to be closer to the original input weights. This could be done by calculating a simple Euclidean distance measure, which is shown in equation (6). As we can see, both the input and the weights are vectors of the same dimensionality. Teuvo Kohonen (2008) also argues about other distances, that are proportional to the shape of the entire input space, but these go out of scope of this study. The main purpose is to keep it simple so it can be later extended and refined.

Step 5 of Figure 2.6 needs to calculate all the nodes that are within the radius of the BMU. These can be done by using a simple Pythagoras theorem for each node, and realizing if this is inside the radius of the BMU. These nodes should be selected and kept somewhere for the next step which is their update. But before going to these, it is important to know that we want to converge at some point. This can be done by dampening the radius as each example is examined. Equation (19) shows an exponential decay function, which can be used to damper the radius update as each training example is presented to the lattice.

$$\sigma(t) = \sigma_0 e^{-\left(\frac{t}{\lambda}\right)}, t = 1,2,3... \qquad (19)$$

The σ_0 element in the equation denotes the initial radius of the BMU at time t_0. As each "epoch" progresses this radius decays exponentially. The λ represents the time constant, which is a parameter of the algorithm telling it how fast to decay.

Finally, each weight should be updated, using the rule in equation (20).

$$w_i^{x,y} = w_i^{x,y} + \Theta(x,y)L(t)(X_i - w_i^{x,y}) \tag{20}$$

As seen before, $w_i^{x,y}$ represents the ith weight of the x and y coordinate node. This weight is updated by adding a delta. The delta itself contains the subtraction from the corresponding vector i dimension component X_i of the input node. But this delta must be multiplied times a learning rate composed of two functions. The first one is the *L(t)* function. This function, similarly to equation (19), decays with time. It is meant to help on convergence of the self organizing map after several iterations. The next section of the learning rate is the *Θ(x,y)* function. As seen before, this function is dependent on the coordinates of the original internal weight vector. The bigger the distance from the BMU, the smaller this learning rate should be.

With self organizing maps, clustering can be achieved. Not only this, but pseudo features can be derived out of a semantically input data. The truth is that these require a special training set that perhaps a time series or hospital data wouldn't match well. It is also seen that there is a lack of good literature pointing towards a good feature derivation tactic using self organizing maps. In any measure, Teuvo Kohonen has done an impressive job in deriving this interesting data structures.

2.3.2. General Regression Neural network

Clustering is proven to be a useful technique, and its application can also be integrated as part of a neural network's algorithm. Specht (1991) propose a new type of neural network, capable of maintaining the original input data. This neural network is based on the principle of memorization. We as human beings, tend to mix generalization and memorization. Both techniques try to be integrated here and prove some sort of success. The main architecture of a General Regression Neural network can be seen in Figure 2.7.

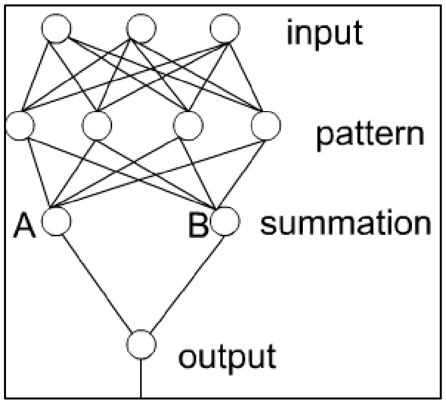


Figure 2.7 General Regression Neural Network (GRNN) architecture (Figure 1 in Specht's research 1991)

To properly understand regression and construction of this neural network, the first step is to know exactly the way regression occurs. Notice that there are 2 outer most layers here, the summation and output layer. The output layer can be explained in equation (21).

$$\hat{Y}(X) = \frac{\sum_{i=1}^{m} A^i e^{-\left(\frac{C_i}{\sigma}\right)}}{\sum_{i=1}^{m} B^i e^{-\left(\frac{C_i}{\sigma}\right)}} \tag{21}$$

The numerator represents the A neuron of the summation layer. The same applies for the B neuron, which is the denominator. At the end the operation is simply a division of these two layers. But what does each layer represent? As we can see, A has a set of weights A^i where i = 1, 2… m and B^i where i = 1, 2… m. Each weight in I is related to a pattern neuron. Pattern neurons can be thought as clusters of the original data. Depending on an analysis, and how well the data is memorized, the pattern neurons are realized by calculating the centroids of the original data. So for the algorithm, a preprocessing step is needed. The definition of these pattern neurons (C_i) is defined in equation (22).

$$C_i = \sum_{j=1}^{P} | X_j - X_j^i | \tag{22}$$

The distance in (22) is also known as the city block distance. This distance represents how far the training example from the cluster being observed is. In simple words is the sum of deltas from every element of a cluster minus the observed example.

The final step in the learning process is to realize the weight values that each of these clusters have with respect for the data. This can be represented using the A and B neurons. So this is the generalization part, where as the memorization part involves clustering. Equation (23) and (24) show the respective learning rules for A and B.

$$A^i(k) = A^i(k-1) + Y^j \quad (23)$$

$$B^i(k) = B^i(k-1) + 1 \quad (24)$$

These steps are repeated until k iterations have been met. There is a better derivation for these rules, which is more complex. More information can be found in Section C of Spechts (1991) research.

What is interesting about the techniques used in this neural network is the fact that the usage of memorization is applied. This helps the system to keep exact track of the previous data set and bring a possibility of enhancing regression. As we can notice the constants A and B are conceptual nodes. These define how well the pattern nodes influence the outcome. This is very similar on how a human brain works, by memorizing and then generalizing with certain thresholds and mixture of concepts.

Back propagation and feed forward neural networks

The last neural network we saw falls under the category of feed forward neural networks. Feed forward networks are those that compromise multiple layers, each obfuscating the next one. Communication between layers can be only done directly, and not through random wiring. The feed forward neural networks and back propagation methods are explained in Mitchell’s book (1997). These algorithms will be explained in later chapters of this study, since they are an essential part of the experimental data.

2.3.3. Back propagation

The most common method to train a neural network is perhaps back propagation. Mitchel (1997) shows us the derivation and ways back propagation works. In order to understand it, it is also needed to understand the meaning of a Feed Forward Neural Network. A Feed Forward Neural Network comes from a biological inspired approach. This approach dictates that a problem can be explained by individual units called “neurons” connected together. Each neuron will process a particular part of the problem, and feed its results into the next layer of neurons. This is the basic principle of how the human brain works. Many scientists say that this could be an explanation of why human thought is heavily based on inspiration (1997). The neuron on this case is something we will call a perceptron. A perceptron graphical picture is show in Figure 2.8.

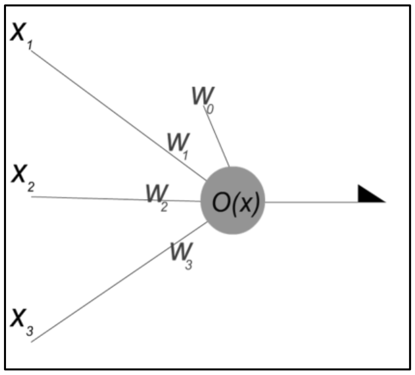


Figure 2.8 Unthresholded Perceptron

The perceptron shown in Figure 2.8 represents the linear equation $w_0 + x_1w_1 + x_2w_2 + x_3w_3$. Each element in X in this linear equation is the output from a previous perceptron. Notice that the perceptron is just the name for a hyperplane. A perceptron itself is linear, so a neural network of unthresholded perceptrons will still be linear. The non-linearity occurs at the end, by making the perceptron the function displayed at (25).

$$O(x) = \begin{cases} 1, & \sum_{i=1}^{n} x_i w_i > w_0 \\ -1, & otherwise \end{cases} \tag{25}$$

As seen before, a set of thresholded perceptrons will be able to represent any non linear function; given that the last perceptron is unthresholded (this is done so the output has a free domain). A neural network can be graphically represented as in Figure 2.9.

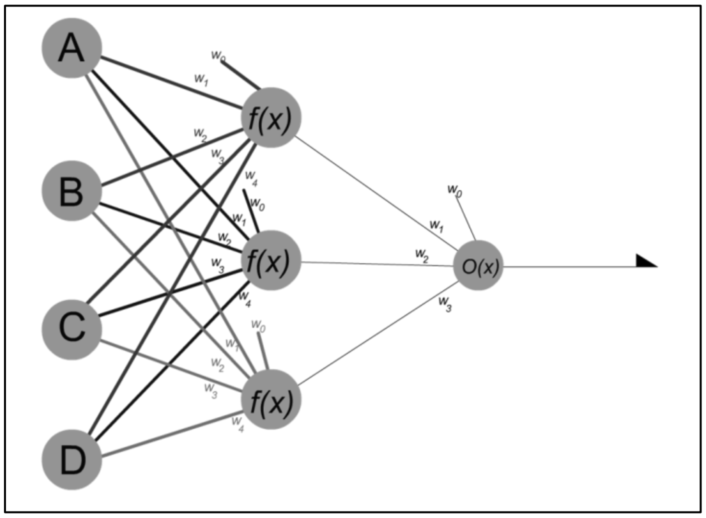


Figure 2.9 Neural network representation

Notice in Figure 2.9 the neural network is divided into three separate layers. The first layer is the input layer, this layer represents the original inputs (the vector features) that will feed into the neural network. These values get propagated to the hidden layer. In this layer, which is the first one with thresholded perceptrons, values get to the outer most, which is the final output layer. At the end the output should go from here. There are networks that have more than one hidden layer, thus requiring specific number of hidden neurons. The more hidden neurons, the better coverage of a function a neural network model can do. This also brings up the problem of learning the correct weights. If there are more neurons, there has to be more data to learn the proper weights. There are a lot of algorithms that help learning the weights in a neural network. Back propagation was chosen for this study, since it's the basic algorithm for feed forward networks (as the one in Figure 2.9). The basic principle in back propagation is very similar to that of gradient descent. Weights are updated gradually by a small derivative. This is done by first figuring out the error of the neural network (therefore input propagates to the front first and the total error is discovered). After this, error is sent back, and the weights are updated layer by layer using this rule. In order to have a smooth derivative approach, it is better if the threshold units use a smooth function instead of a non-continuous threshold. This is called a sigmoid unit. The sigmoid function is expressed in (26).

$$O(y) = \frac{1}{1+e^{-y}} \tag{26}$$

A very useful property of this sigmoid unit is that the derivative can be written in terms of itself, as show in (27).

$$\frac{\delta O(y)}{\delta y} = O(y)(1 - O(y)) \tag{27}$$

In order to define a proper set of operations to update these weights, a first glance to the target function is needed in (28) for the output units.

$$E(\vec{w}) = \frac{1}{2}\sum_{k\in outputs}(t_k - o_k)^2 \tag{28}$$

Where t is the training value for the correspondent k output unit and o is the output value of the corresponding output unit. At the end we want the gradient vector which describes derivatives. We will want to minimize the outcome of the error function by changing the derivatives. The gradient rule is shown in (29), using derivatives chain rule.

$$\frac{\delta E_d}{\delta w_{ji}} = \frac{\delta E_d}{\delta net_j}\frac{\delta net_j}{\delta w_{ji}} = \frac{\delta E_d}{\delta net_j}x_{ji} \quad (29)$$

The chain rule in (29) tries to extract the weight derivative from the error. Since the weights are under the *net* function (which is the linear combination of weight values and input values), its derivative with respect to the weights is the ith input value X of the jth unit. The next step involves solving the $\frac{\delta E_d}{\delta net_j}$ derivative. This can be thought of two separate cases. The first case involves updating the weights of output layers and the second involves the update of any internal node's weights. This entire section is based on Mitchell's work (1997). Once the update rules have been derived, it's all about iterating through the example set and updating the weights of the neural network for several iterations. Figure 2.10 displays the pseudo-code for back propagation. The main loop in gradient descent has to update the neural network iteratively. The updates are done in a stochastic fashion, as in one example at a time. For each one of these iterations and for each example update, each neuron's δ error is calculated. As shown in the derivation of error updates, each neuron gets its own δ measurement depending on the layer they are. A problem with this is the convergence to local minima.

```
BackPropagation
-inputs:
X(inputs), Y(outputs), η(learning rate), T(iterations)
-outputs: ANN
1: Begin
2:   FOR t = 1 to T
3:    FOR i=1 to N
4:      For each output Unit k
5:        $\delta$ k = (yik - Outputx)
6:      For each hidden Unit h
7:        $\delta$ h = Outputh(1 - Outputh)$\sum_{k \in Downstream(h)} \delta_k w_{kh}$
8:      Update every $w_{ji} = w_{ji} + \Delta w_{ji}$ $where$ $\Delta w_{ji} = \eta\ \delta_j x_{ji}$
5: RETURN ANN
7:   END
```

Figure 2.10 Back Propagation

The main loop starting in Step 2 will update the weights in a stochastic manner. Steps 5 and 7 of Figure 2.10 apply the weight update directly. Notice how the loop in Step 3 starts first from the outer most layers into the inner most hence called back propagation. This means that the algorithm might find a set of weights that is not necessarily the optimal. Several techniques are available to help the algorithm to fall into a local minimum set of weights. The technique used in the Admission data set uses a validation set and takes a snapshot of the Artificial Neural Network during each iteration.

The ANN that has the least error with respect to the validation set gets picked at the end of the iterations. This tries to maximize the accuracy of the function the ANN is modeling.

2.4. Regression trees

Regression trees are a type of classification and clustering technique used. There are many ways of generating and representing a regression tree. Many fields use regression trees as a data structure capable of approximating non linear functions. Many of this techniques involve some sort of meta algorithm and fitting a the leaf nodes. In this section, a couple of the existent methodologies will be discussed.

2.4.1. Regression tree algorithms

Neural networks have a very powerful representation. Another approach is the divide and conquer technique. It would require taking the entire problem and dividing it in specific chunks that are solvable through a known technique. The regression tree does this, by creating a binary decision tree data structure. Each node in the tree will represent a feature of the input space in which a decision is made. The edges connecting the nodes represent the respective thresholds that should undergo when facing this decision. The leaf nodes are the ones that do the regression of the specific input space. Figure 2.11 shows an example of a Regression Tree.

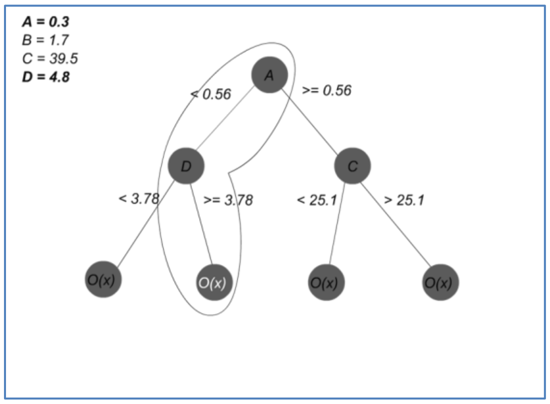


Figure 2.11 Regression tree

Figure 2.11 shows that each node internally represents a feature, with its respective thresholds. A path in the tree will be followed based on the features and a specific example picked to do regression on. Figure 2.12 shows the example of a continuous function with a regression tree on top. On this case, the regression tree leaf nodes consist of trained hyper planes. As we can see each hyper plane fits a segment of the function and tries to approximate it. The combination of these lines approximate to the function.

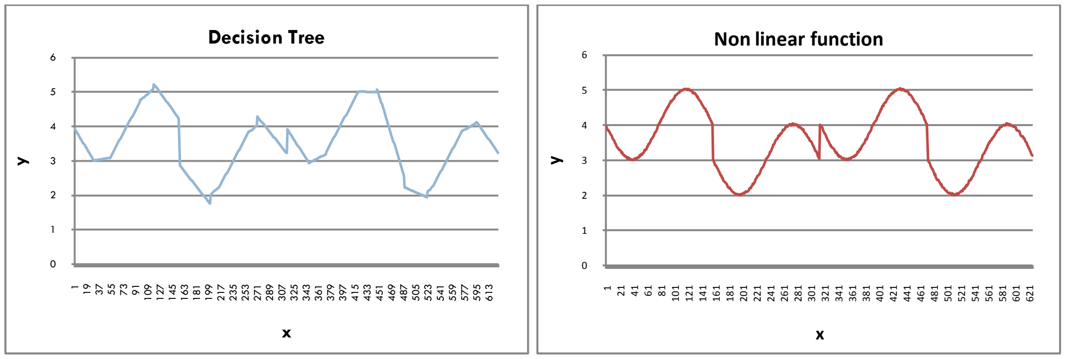


Figure 2.12 Left hand side regression tree, right hand side non linear function.

The representational power of a regression tree is more discrete than that of the neural network. This could mean that non continuous functions are potentially better represented with a regression tree, whereas in a neural network a differentiable function is needed. The problem with a regression tree lies in its building, which requires many sub algorithms depending on the function that is being learned. There are three main sub problems in a regression tree: splitting criteria, stopping criteria and regression leaf nodes. For now, the pseudo code will be shown independently in Figure 2.13, and these three sub problems will become three sub procedures.

```
RegressionTree
-inputs:
Features, X(inputs), Y(outputs)
-outputs: RegressionTree
1: Begin
2: if STOPPING CRITERIA is true
3:    perform REGRESSION in current X|Y
4: else
5:    perform SPLITTING CRITERIA and split X|Y into two
      subsets (left and right)
6:    Recursively call RegressionTree in left and right
      Nodes
7: End
```

Figure 2.13 Regression Tree

The algorithm has a recursive nature. There is one main test case, which tests for the stopping criteria. If the algorithm meets the stopping criteria, then immediately all the examples used become the new leaf node. This will call the *REGRESSION* sub procedure in Step 3. This sub-procedure will be in charge of using the current data to fit a model. If the stopping criteria decides otherwise, a serious of splitting steps occur. The first and foremost is the sub procedure that tries to find the best splitting point. There are many techniques which will be explained later under the splitting criteria sub procedure. This procedure might fail, as in not being able to find a better splitting point. This would mean that splitting more the regression tree will actually decrease its accuracy instead of helping it. If a splitting point is found, then this point is used to split the input and output sets respectively. Then the algorithm is called recursively on these children. The following sections explain with detail the type of algorithms needed for this.

Splitting by variance (CART):

Step 5 of Figure 2.13 calls the subprocedure SPLITTING CRITERIA. There are many algorithms and techniques present. We will first study the one implemented by CART, which deals with variance.

Variance split will check for every possible splitting point in the data and perform a splitting test. Out of these splitting point candidates it will try to discover that one that minimizes the impurity gain. The impurity is calculated using the variance of each subset and lastly getting an impurity measurement. When there is more variance there will be higher impurity, the least output similarity from the examples. Figure 2.14 shows the sub procedure variance version of PickBestSplit.

```
PickBestSplit (variance version)
-inputs:
Features(F), X(inputs), Y(outputs)
-outputs: (best_f,best_$f_v$)
1: Begin
2: BestGain = 0, (f_best, f_$best_v$) = null;
3: AllImpurity = CalcVariance (Y)
4: For each f in F
5:  For each split point $f_v$ in f
6:    Split X using (f,$f_v$) into $X_{left}$ and $X_{right}$
7:    Split Y using $X_{left}$ and $X_{right}$ into $Y_{left}$ and $Y_{right}$
8:    Gain = Allimpurity - (CalcVariance ($Y_{left}$) + CalcVariance ($Y_{right}$))
9:    If (Gain > BestGain)
10:     BestGain = Gain
11:     (f_best, f_$best_v$) = (f,$f_v$)
12: Return (best_f,best_$f_v$)
13: END
```

Figure 2.14 Pseudo code for PickBestSplit by variance

The main goal is to pick a split in which we have a “gain”. Gain means a possible improvement in data granularity. The gain is a measurement that compares the impurity before versus the impurity after. For this particular case, as mentioned before, the impurity is just the variance that the data has. It can be thought as how far the data is from average and how variable it is. If there is such a case of improvement then the split that maximizes this gain wins. Notice that Step 2 of Figure 2.14 initializes the best split point as null. That means that if there is no such split that maximizes gain, the regression tree algorithm should decide to stop and fit a regressive model by creating a leaf node.

There are some variations of these splitting criteria. Popular ones, as used by CART, involved a weighted variance. That means that step 8 of Figure 2.14 will use relative weights of each split point. It does this to be fair and influence splitting points to be proportional.

Stopping Criteria:

Step 2 of Figure 2.13 queries the *STOPPING CRITERIA* sub procedure. This sub procedure will decide if dividing the regression tree is worth it. If the sub procedure dictates to stop, then the regression algorithm must be called and generate a model on the data left. There are two kinds of stopping criteria, pre splitting and post splitting. The pre splitting stopping criteria analyses the pieces being split. The post splitting analyses just after the split point has been picked. The following sections will explain the 4 major stopping rules used during earlier experimentation with the admissions data set.

Number of Levels

This pre splitting stopping criteria will first look at the number of levels being traversed in the tree. If the current depth of the tree exceeds a threshold, then *STOPPING CRITERIA* in step 2 of Figure 2.13 should return true and force the algorithm to generate a leaf node. The problem with this method is that stopping the tree growth in a level might not generate a good fitting. The level forces the tree to be of a single height with no chance to have denser nodes than others. This can potentially limit the power of a regression tree. Also, intuitively, is very hard to find a correlation with data and the number of levels that a tree possesses.

Leaf Size

Stopping the tree growth whenever the number of examples falls under a threshold seems better than stopping by level. While still hard to correlate against the data's semantics it will intuitively generate better trees than those generated by the minimum levels stopping criteria. Trees that stop their growth with a leaf size threshold do not have to be of a single level, they can be of several levels and depths. This increases the power of the regression tree and its shape.

Regression at Leaf Nodes:

The final sub problem in regression tree formation is the actual step of regression. For this there are several techniques proposed, each one with certain advantages and disadvantages over data.

Simple Average

The simplest method of regression is an average of the output Y values clustered at the leaf nodes. A disadvantage with this is that the size of the tree will be the one in charge of deciding its power. The models at the leaf nodes are very naïve and inflexible and high amount of errors might be present especially if there are outliers in the training data. Splitting methods such as variance splitting and error splitting are appropriate for this situation.

Linear Regression

For linear regression there are many options. Gradient descent could be applied. The biggest problems with gradient descent are its greedy nature and time complexity. This time complexity might not be feasible and become impractical when the tree has several hundred leaf nodes. Another disadvantage of gradient descent is the number of parameters, such as the learning rate and iterations numbers. It was argued before that such parameters are very linked to the numerical distribution of the Y data. Higher averages require high learning rates, and conversely.

2.4.2. Summary of other methodologies

Wei-Yin Loh (2008) offer a variety of algorithms that could be used for regression tree generation.

The main purpose of a regression tree is to output a data structure as described in Figure 2.11. The regression serves as an extra clustering technique that categorizes and fits every vector of the input space into a particular leaf node. The simplest method described by Wei-Yin (2008) is CART. CART tries to divide the space by minimizing variance. Another technique used is QUEST. While CART does a brute force selection on the best possible split value, QUEST instead does a selective search. It uses several heuristics to categorize the best feature to perform a split.

After the feature is picked, it will use it to pick the best split point that would fit the model. QUEST is said to be an unbiased classification method for this reason, since it first selects the feature and then the split point.

Another interesting algorithm developed by Wei-Yin (2008) is GUIDE. The algorithm tries to stay away from a typical greedy tree methodology. It constructs a multiple linear and simple polynomial model on a target error function (Wei Yin, 2008). This error function could be least squares, quantile or Poisson error distribution. Just like QUEST, variable selection is unbiased. It also performs a post pruning step once the tree has been formed.

Wei-Yin in his study concludes that all these algorithms have their advantages and disadvantages. While CART is very simple, and QUEST very complex, CART can actually perform very good with noisy data under with Normal error distribution. QUEST on the other hand can over fit easily to data, but its representational power is greater, meaning that it is better fitting very complex functions. Another problem with QUEST is that is restricted by Univariate splits. Amongst these methods, the principal method studied and extended is a variation of CART. This variation uses piecewise linear approximation and in later chapters it will be explained and supported throughout.

D. Vogel et al (2007) suggests a technique of building a piece wise approximation regression tree, but using not so greedy techniques. The technique consists on construction linear models for splitting attempt. This will generate a tree that potentially can't overfit since is spending more computational time in the search of the best tree possibility. At a first glance, one would think that doing such thing is very computational expensive. For example, fitting linear models per each split possibility sounds very expensive. In fact, through several linear algebra simplifications (therefore using Gaussian elimination as its leaf node regressive algorithm), Vogel is able to recycle operations and minimize the computational expense. The problem with this technique was that he didn't try any other toy data sets. Only practical sets are used, and there is a lack of explanation about this data in his research. Maybe it could be biased, but it is always necessary to have an artificial data set that tries to contrast the best and worst qualities that a regressive algorithm could potentially engage into.

2.4.3. Regression tree applications

Regression trees have found many applications on the literature. One of the biggest reasons why regression trees are preferred is their easy of readability. In principle, the regression tree can be easily understood by humans. It can reveal patterns and expose them clearly to the audience. Sometimes regression trees don't have always show this behavior. When a tree grows too long in an unbalanced manner, it is very hard to distinguish the real patterns hidden behind the node thresholds. Buja and Lee (2001) proposes a methodology that is not meant to increase the tree's accuracy but its readability. In CART, the splitting criteria points always towards the split value that minimizes the variance impurity measure on a weighted average fashion. This impurity measure is simply the weighted variance of the two sets of the split data. Buja on the other hand forces the tree to grow into one side. This is done by minimizing the impurity measure described in equation (30).

$$impurity = \max(\mu_L, \mu_R) \qquad (30)$$

As shown in equation (36), the impurity is biased towards one side of the tree. This will generate one sided imbalanced trees which could be less complex. This can relatively help with readability and is a good approach for early data analysis. An example of a tree generated using this strategy can be summarized in Figure 2.15.

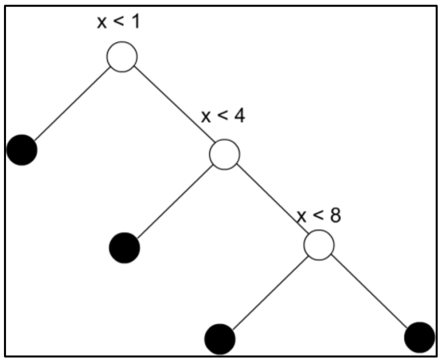


Figure 2.15 Regression tree optimized for readability (Figure 1 of Buja and Lee 2001)

As Buja and Lee (2001) explain, this tree optimized for readability is not meant to do complete regression on a data set, but rather expose patterns that can be discovered and exploited for feature generation. While this research work introduces this technique as a novel approach to improve readability, is very hard to compare against other approaches meant for readability. This is a study that does pure analysis of these techniques based on visual approaches rather than quantifiable. In any case, it is a good idea to see what kind of patterns is discovered if a split is drawn to minimize variance in the greediest way.

While regression trees improve readability over neural networks, they are obviously good for what they were meant for which is regression. A study drawn by Yohannes and Webb (1999) show that regression trees can also be used for outlier detection. Their splitting nature helps separating concepts that present different patterns within the features chosen. In their study, they use data from a famine vulnerability data set. This is meant to identify “outliers” that are targeted as areas of low protection. Some of the features are discrete. The way a regression tree handles discrete features is by creating Boolean attributes for each value of the discrete ones. For example, the attribute DAY_OF_THE_WEEK in their classification data has seven values. Each value represents a day of the week. When preparing these attributes values to be inserted in the regression tree, a pre step approach is used. This involves taking each value (such as Monday, Tuesday, Wednesday etc) and converting it into its own independent attribute. The single discrete DAY_OF_THE_WEEK attribute will now become seven distinct synthetic attributes, each one with its own Boolean value. These will be *IsMonday*, *IsTuesday* and etc, each with its possibility of being true or false.

While exploring attributes information and preprocessing, Yohannes and Webb (1999) also show possible ways of avoiding over-fitting. Like a neural network, it is possible for regression trees to over fit. This happens when the tree either grows too large, or the model themselves at the leaf nodes "memorize" the training data. In CART this occurs when the splitting point is wrongly chosen, without checking its validity instead a validation set. This is one of the solutions that Yohannes suggests. Another interesting solution is the concept of a K-fold cross validation. Cross validation is a method used when there is a lack of training data. While using the entire training data, it makes sure not to oversee the entire data at once. Each piece will take turns into being a validation and a training set. At the end this will dictate the predictability and at the same time take the most advantage of the training data.

Huang and Townshend (2003) propose models that have their applications on subpixel land cover. The problem here is to try to associate land pieces with similar topological properties by just using images. The images contain aerial photos of a rural area around Annapolis Maryland. The purpose here is to allow the machine learning algorithm correctly identify areas of similar topological area properties. These properties include plantation types, type of soil temperature amongst others. Surprisingly the trees formed are not as big, containing only around 6 nodes. The small tree contains several hundred paths, and is able to produce a highly non-linear relationship of the pixels in the image. While the model is not perfect, it demonstrates a low error and good fitting with respect to the test data. The test data involves maps pre-categorized by human beings, based on the same features. For regression, they use a method called Stepwise Linear Regression. This method searches all possible combination of lines performed, and picks the one that minimizes the error with respect to other points in the line. Splitting criteria involves solving SLR (stepwise linear regression) for each possibility of a split. The problem with this technique is time complexity, and this constrains are not analyzed in Huang and Townshend (2003) research work. This type of computation could take hours and is very unpractical. If the dimensionality of the problem is big enough, it can be unfeasible to apply such algorithm.

While the power of piece wise linear regression is in fact very useful for non linear problems, computing a linear regression tree is no easy task. In later chapters further concepts will be explored, motivated by many of the weaknesses shown in these methods. Is it necessary to explore every combination of splitting points? Is it necessary to explore every single point when doing linear regression on a set of points? All these questions lead to the motivation of new methods that use modern randomization techniques to solve these difficult problems.

2.4.4. Regression tree novel approaches

This section focuses on the new approaches that many researchers have taken into optimizing and improving known regression tree algorithms. It was already seen that many fields of study prefer regression trees as their off the shelf algorithms. The biggest reasons being their simplicity and readability and sometimes the speed in which this data structures could be converged.

Izrailev and Agrafiotis (2000) suggest a novel approach involving a controlled randomized algorithm. They use Artificial Ant Colony Systems to find tree structures. This technique is based on the biological and organizational aspects in which an ant colony works. As we know, ant colonies consist of several tunnels and paths leading to the food chambers, yet ants can easily find their way through the most optimal path. Now the problem here is how the ants collaborate to find the shortest path between the food source and the nest. Ants leave a pheromone scent on the ground and the accumulation of this pheromone defines the shortest path.

The algorithm suggested by Izrailev and Agrafiotis (2000) is of a randomized nature. The representation of ants is given by each tree. Each proposed tree represents an ant, and the composition of their leaves represents the path they have taken. On the outside there is a binary topological union of these ants that stores the probability of paths by accumulating the ants finding the correct paths. Paths that lower the error (regression here is done by using average just like CART) will be rewarded with more probability. Those with less will be rewarded with least probability. At the end a sum is made and the topological probabilistic tree will contain an agglomeration of the best paths. It is up to here that either the path with max probability is chosen, or simply picked at random with the respective probabilities.

Results are impressive in a couple of data sets. These data sets happen to be of high dimensionality, so it seems to be a correlation between high dimensionality and guided randomized search. In fact, the reason why the algorithm performed better might be because its comparison was done against a random algorithm (which chooses paths at random). The higher dimensionality, the less correct paths there will be then the random algorithm has a broader set of paths to choose from, reducing the probability equally for everyone. The other baseline algorithm was the RP (recursive portioning), which is a greedy algorithm very similar to CART. At the end results show that in general ant search is better, but not in a degree of significance that can totally discard RP algorithms. Izrailev and Agrafiotis (2000) also failed to present customized toy data sets testing for over fitting. While not really practical, these kinds of toy data sets show a specific feature of the algorithm that might be desired or undesired. It forces the algorithm to explore its weakest and strongest points, which might be hidden in a regular data set.

Randomization techniques are a good choice when exploring a big problem space. Interestingly, Wei Fan et al. (2006) propose a very simple technique. It involves generating a set of random trees and combining them in an average to perform regression on data. There is a mathematical proof showing how entropy influences and at some point combines to produce a factual answer to a problem. The algorithm is very simple. It starts by producing random trees (and by random it is meant that splitting criterion picks random attributes and split points). At the end these random trees combine to produce final regression. Since there is less hypothesis bias, due to the nature of the algorithm, statistically the combination of each random tree reduces the variance in the final outcome. This means that each tree acts as a little helper to find the final regression model to the problem. At the end the trees outputs are combined by doing an average, and this average is the final answer for the regression. Interestingly they showed the random regression tree under several data sets, proven to work for a variety of dimensionalities. The results show that the tree doesn't always performs better than other algorithms, but it does in a great amount of times. The contribution here is that a random regression tree doesn't cost as much when learning it, compared to other algorithms such as CART.

Chapter 3

A Randomized Approach to Regression

This chapter will propose algorithms based in the guided randomized search principle in hopes to overcome these problems. The chapter is divided in two main sections of the study. The first one talks about our proposed randomized linear regression algorithms KGERS. Linear regression takes an important role in the formation of regression trees that perform piece wise linear approximation. The second part will talk about the new algorithms devised for a regression tree, and how KGERS be integrated to be able to perform modeling on non-linear problems.

3.1. Motivation for KGERS

KGERS has two important properties. The first property is the subset nature of the algorithm, by only analyzing part of the data and not the entire set. The second property the randomized nature of KGERS, by only picking random sets of the data through each trial. The motivations for these properties of KGERS are under the main assumption that the input points do follow roughly a linear pattern. By this, it means that there exists a line that minimizes the error against other points, and that this error is relatively small.

The motivation for the subset nature of the algorithm comes from the assumption that we are dealing with a set of points that follow a roughly linear pattern. By roughly linear we mean error relatively low that would make algorithms such as Gradient Descent converge. Assuming we have a perfect data set where all the points are on a hyperplane it doesn't matter which subset is picked. Any subset of size M+1 (where M is the dimensionality) will always produce the same hyperplane. Under this property the motivation for randomized subsets is used.

The next property involves the randomized nature of KGERS. Figure 3.1.A illustrates the subset property. This shows the sample data set, which involves 4 points that roughly follow the shape of a line. To illustrate this example we will use a few data points in a 1 dimensional problem. For this set of 4 points there is a total possibility of six subsets generating lines (subsets must be size 2 since is a 1 dimensional problem). Out of these six lines, only 1 is not similar to the optimal solution.

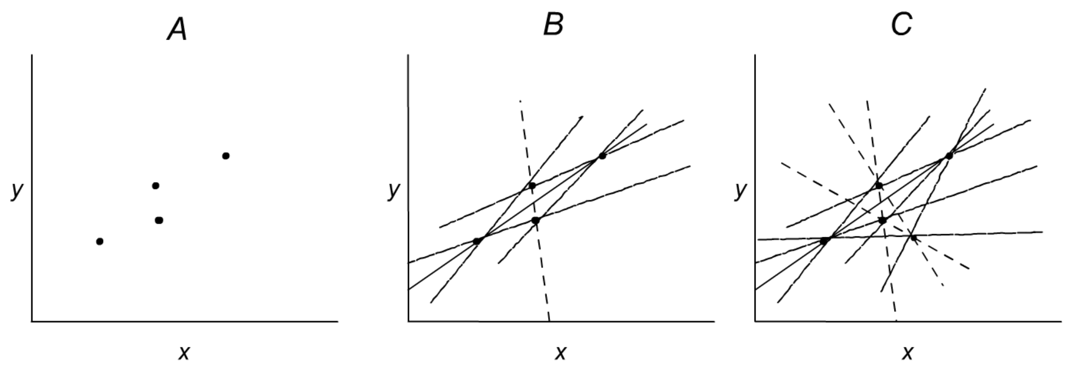


Figure 3.1 Illustration of randomization property from KGERS

Figure 3.1.C of the diagram shows what happens when a new point is added. It only generated two lines (the new dashed lines) that are far from the optimal solution. The probability that a line is picked and is close to the optimal solution is 7/10. Notice that this decremented from an original probability of 4/5 displayed in Figure 3.1.B. Based on these properties the motivation for a randomized subset arrives, due to the likelihood of generating good hyperplanes does not decrease dramatically as more points (that are roughly follow a linear pattern) are added.

3.2. KGERS Algorithm

Solving linear systems is a problem that has many applications. As seen before, it helps modeling hyper-planes of *m* dimensionality when several feature vectors are shown. It is proven that any function can be approximated by splitting it into several linear equations. This process of function discretization into hyper-planes shows good results with toy data sets, and so with the EDInpt data presented in future chapters.

In simple terms, the problem we are trying to solve is that we are given X, which is a set of vectors defined as:

$$X = x_{ij}; i = 1..n; j = 1..m \tag{1}$$

where *m* is the dimensionality, and n the number of points we are given. At the same time, we are also given Y, which is defined as:

$$Y = y_i; i = 1..n \tag{2}$$

This Y is the set of values that we want to do linear regression based on X. At the end we want to figure out the weights of this function:

$$\hat{y}(X_i) = w_0 + \sum_{j=1}^{n} x_{ij} w_j \qquad (3)$$

such that the value of $\hat{y}(X_i)$ approximates y. This chapter will focus on the proposal of KGERS, a newly devised randomized search algorithm that is motivated on the necessities of piece wise linear regression.

Regression and splitting at the leaf nodes of a regression tree requires an algorithm that is less complex, decidable and fast. This means that there is the need for a technically fast algorithm that can still be likely to find a good linear equation to fit. KGERS stands for K-Gaussian Elimination on Randomized Sub Sets.

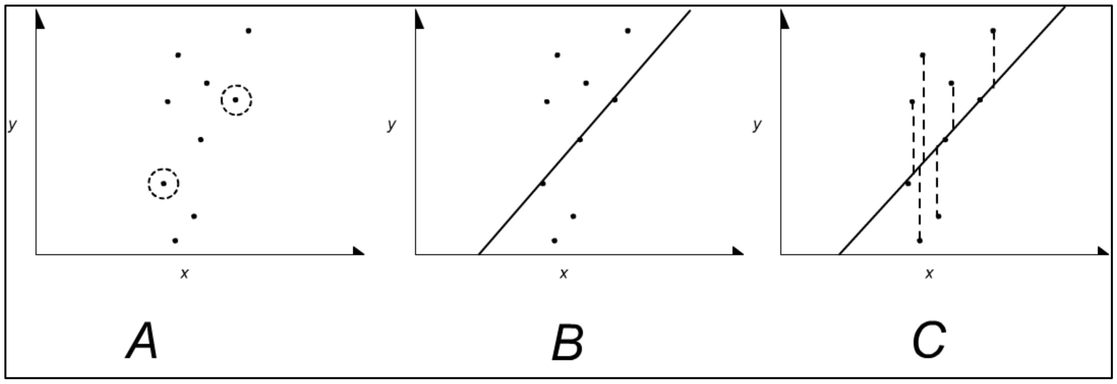


Figure 3.2 KGERS line visualization

Figure 3.2 shows the motivation behind KGERS. This figure is showing a 2 dimensional problem, which requires 2 points for each K random set. This will generate lines and each line will have an error with respect to the rest of the points (Step C). The dashed lines represent the error that the selected subset has against the rest of the points in the figure. Lines with higher error (larger set of dashed lines) will have less impact on the final hyperplane generated. Figure 3.3 shows the pseudo code for KGERS. In step 3, the algorithm uses randomized sets of data (of size M) and arranges them into separate sub sets. Each of these data sub sets will generate a line that has exactly 0 errors with respect to the original points, which is done in step 5 of the algorithm. Finally each of these generated lines will be weighed against its error with respect of a random set of data (of size M again) from the validation set, described in Step 6 and 7 respectively.

```
K-GERS
-inputs: X, Y, K (M is dimensionality)
-outputs: W
1: Begin
2: For k=1 to K
3:   Let S be randomly selected M+1 rows in [XY]
4:   Let S' be [XY] - S
5:   H[k] = GenerateHyperplane(S)
6:   R[k] = CalculateHyperplaneWeight(H[k], S')
7: W = GenerateFinalHyperplane(H, R)
8: Return W
9: END
```

Figure 3.3 K-gers pseudo code

Step 3 of Figure 3.3 makes a random selection of a subset. This random selection will use a uniform distribution for now. The motivation behind this is to have chance for every single point in the training set. The distribution in which the pseudo number generator works could change, but this would be entirely dependent on the input data statistical properties. As a general case a uniform distribution will be used. Sampling is the field in statistics that seeks to extract subsets from individual observations. The way these subsets are picked can follow a specific distribution, such as Gaussian. For the case of this algorithm, a uniform distribution will be used. Step 5 generates a hyper plane using the randomized subsets that were picked. This hyper plane of course will have zero error with respect to the original points that created it, but this is when the validation set enters. Figure 3.4 shows how a hyperplane is generated. Steps 2 through 4 define the elements required in the linear system. In Step 5, Gaussian elimination to solve the linear system, resulting into a hyperplane that has 0 error with respect to these data points used.

```
GenerateHyperplane
-inputs: set S of M + 1 points
-outputs: H (hyperplane H)
1: Begin
2: let H be a hyperplane weight vector of size M
3: let A be the feature matrix from S of size M+1 x M+1
4: let B be the target vector from S of size M+1 x 1.
5: Solve the system AH=B for H
6: Return H
7: END
```

Figure 3.4 GenerteHyperplane pseudocode

To calculate the weight errors, we need a target error function, which is described in step 6 of Figure 3.3. For now is the squared error function. Figure 3.5 explains how to get the error weights.

```
CalculateHyperplaneWeight
-inputs: S (a set of points), H (hyperplane weight vector)
-outputs: r (weight)
1: Begin
2: Let A be the feature values from set S
3: Let B be the target values from set S
4: Let r = 0
5: For i=0 to M
6:   Y_hat = eval(A[rand() % Length(A)],H)
7:   r += 1/(y_hat - B[i])^2
8: Return r
9: END
```

Figure 3.5 CalculateHyperplaneWeight pseudocode

Equation (4) shows the equation representing the calculation of the weights that will be incorporated in step 7 of Figure 3.5.

$$W = \sum_{i=1}^{M+1} \frac{1}{(\hat{y}(x_i) - y_i)^2} \tag{4}$$

Notice the denominator is the error squared. Bigger errors will generate smaller weights, though less influence of that particular line. Once the weights are ready (and normalized), the final step is to aggregate everything by doing a summed weight of the errors, as described by step 7 of Figure 3.3. Figure 3.6 describes this aggregation step formally. Notice how each weight is iterated through in the outer loop in step 2 of Figure 3.6. After this, the most internal loop does an update of each weight respectively by aggregating all weights, in step 4 through 5 of Figure 3.6.

```
GenerateFinalHyperplane
-inputs: H (set of hyperplanes), R (set of hyperplane weights)
-outputs: W
1: Begin
2: Initialize vector W to 0 of size M+1
3: For j=0 to M
4:   For k=1 to K
5:     W[j] += R[k] * H[k][j]
6: Return W
7:   END
```

Figure 3.6 GenerateFinalHyperplane pseudocode

By the end of step 7 in Figure 3.3, the final result should penalize those hyper planes that have a big error with respect to their validation sets. KGERS is an algorithm that doesn't require to access all the data points, and later on this will be shown in its time complexity. Notice how in step 5 of Figure 3.5 the loop goes up to the size of M, which is the dimensionality of the problem. We could ask ourselves why not just get the best hyper plane: that one with the maximum weight? The answer to this question has to do with over fitting. We don't want to discard information given from other corners in the data. Because this randomized algorithm only looks at certain points, it must make the most of it, and assume that actually noisy data points also influence the final hyperplane, but still penalizing them. Notice that if we have data that follows a linear pattern, except for one point, the hyperplanes generated using this point will be heavily penalized and have weights that go to 0.

One advantage of this algorithm is that it uses only one parameter. The only parameter here is K, which can be somehow though as the probabilistic value that a line can be fit. If in the extreme case, the data completely has linear tendencies, this algorithm should be able to find the weights with K = 1 (any set of different points should be able to draw the line). Algorithms such as gradient descent require at least the number of iterations and learning rate. This could be a problem, since in a regression tree each leaf node model might require different parameters to find a proper line. For example, one leaf node clusters data points with a distribution using high numbers, then to find a proper hyper plane the learning rate requires a low value. Conversely, a set of points with low numbers requires a high learning rate. A single learning rate and iterations number will not be able to cover every single sub model in the regression tree. KGERS will be competing against two traditional algorithms, therefore a time complexity analysis is required. This will be shown in the next section.

3.3. Time Complexity Analysis

This section will break up the algorithms into their basic operations, and give a raw time complexity table. This time raw complexity table will include the external algorithm parameters. The next table is the true time complexity, which will reduce the parameters into functions of the input size. KGERS will be a competitor of Gradient Descent, in order to improve the performance in a regression tree. The linear least square algorithm is used to measure how far are the algorithms from reaching a global optimal. Since linear least squares do not guarantee an answer, it is not a good idea to use in the hospital data. It is important however to do some sort of comparison of its time complexity, so this algorithm is used as a baseline.

3.3.1.1. K-GERS Analysis

K-GERS outer most loop goes through the constant K, shown in step 2 of Figure 3.3. Inside is the procedure of solving a linear equation (step 5). This small procedure has a time complexity shown in (5).

$$O(kM^3) \tag{5}$$

The evaluation of errors (step 6 Figure 3.3) has to iterate through M + 1 random example. This involves a time complexity reciprocal to *M*.

$$O(kM^2) \tag{6}$$

Finally is the step of aggregation (step 10 Figure 3.3).

$$O(kM) \tag{7}$$

All these steps are summarized here in respective order of summation:

$$O(kM^3 + kM^2 + kM) \tag{8}$$

The constant K can be usually disregarded, since it's usually much less than *N*. Also, empirical evidence shows that it doesn't have to grow linearly with respect to *N*. As a result, the final time complexity in terms of M and N gives:

$$O(M^3)$$

If the size of K is big enough (for certain data set whose linearity is very low), then it can be said that K grows linearly with N.

3.3.1.2. Gradient Descent Analysis

Gradient descent outer most operation depends on T which is the number of iterations. For each of these iterations, all the examples are traversed. For each time the examples are traversed, each vector component is updated. This gives a raw time complexity of:

$$O(T * N * M) \tag{9}$$

As discussed before, *T* represents the iteration parameter, *N* represents the number of data points, and M represents the dimensionality of each data point. A proper implementation of Gradient descent will require a validation set. Still, we always traverse all the data at least once. Empirical evidence shows that the parameter *T* has to be *at least* the size of N. Smaller values usually don't converge into a solution. A rough estimation suggests that *T* grows linearly with respect to *N*. This means that in terms of dimensionality and N, the time complexity of gradient descent is:

$$O(N^2M) \tag{10}$$

3.3.1.3. Linear Least Squares Analysis

A recapitulation of linear least squares can be found in Figure 3.7.

```
Linear Least Squares
-inputs: X, Y
-outputs: W
1: Begin
2: Initialize matrix A := XTX
3: Initialize vector B := XTY
4: Initialize W to be length M
5: Solve system AW:=B for W
6: Return W
7: END
```

Figure 3.7 Linear Least Squares pseudo code

For step 2 and 3 of Figure 3.7 we need X^T. This means that during this first pre-operation, the time complexity is:

$$O(NM) \tag{11}$$

Matrix multiplication is an expensive operation. On this case, the X^T matrix is of dimensions DxN. This means that the time complexity for the matrix multiplication (X^TX) in step 2 of Figure 3.7 is:

$$O(NM^2) \tag{12}$$

Similarly, step 3 involves the matrix multiplication of X^TY which will give as a time complexity of:

$$O(NM) \tag{13}$$

Added all these previous steps to the solving of the linear equation, adds up to a final time complexity:

$$O(NM^2 + M^3 + 2NM) \tag{14}$$

This is because Gaussian elimination has a cube time complexity. Finally the reduced form will give us a final time complexity shown in (15).

$$O(NM^2 + M^3) \tag{15}$$

3.3.2. Analysis Summary

Table 3.1 shows the summary of time complexities. The last Column displays the final time complexity by approximating the algorithm's parameters to the size of the input vector set.

Algorithm	Time Complexity	Parameter Estimation	Time complexity with parameter estimation	Always Produces a valid solution	Optimal solution
Linear Least Squares	$O(NM^2 + M^3 + 2NM)$	-no parameters-	$O(M^2N + M^3)$	No	Yes (if a valid solution exists)
Gradient Descent	$O(T * N * M)$	$T \sim N$	$O(N^2M)$	Yes	Yes (only if learning rate is sufficiently small, which could take a long time)
KGERS	$O(kM^3 + kM^2 + kM)$	The value of K is uncorrelated to the size of the set, but the linearity of it.	if K is Small $O(M^3)$ If K is large (K ~ N) $O(NM^3)$	Yes	No

Table 3.1. Summary of time complexity

The last two columns in Table 3.1 show the nature of the computational answer of each algorithm. Liner Least Squares might have a matrix that is not invertible, therefore step 5 of Figure 3.7 will fail to give an answer. This means that a valid solution is not guaranteed, given that this matrix is not invertible. The advantage of Linear Least Squares is that if it finds a solution, this solution is guaranteed to be the global optimal. That is the hyperplane with least error for the data set. Gradient Descent and KGERS in the other hand do not follow this rule. These algorithms are capable of achieving the global optimal, but it is not guaranteed. For Gradient Descent it is possible only when the learning rate is sufficiently small, but this might use extended computational time. When dealing with a training set and a test set, KGERS has the probability of doing better than the global optimal due to its randomized nature.

3.4. Regression Tree Algorithms

It was seen before that linear regression is possible through a variety of methods. Techniques go from randomized approaches to decidable linear algebra operations. The problem addressed in this section is regression on non linear functions. The purpose of this study is based on the hospital admission data. We picked a regression tree as the target algorithm for improvements. The reason being is that the nature of a regression tree, as mentioned before, allows seeing actual readable data. This helps to understand the data being analyzed and hopefully catch patterns with the naked eye. It also helps seeing which features of the data are not helping with the model, and which ones are. In this section a new set of splitting criteria, stopping criteria and leaf regressive method is proposed. Recall in chapter 2, the regression tree pseudo code shown in Figure 3.8.

```
RegressionTree
-inputs:
Features, X(inputs), Y(outputs)
-outputs: RegressionTree
1: Begin
2: if STOPPING CRITERIA is true
3:   perform REGRESSION in current X|Y
4: else
5:   perform SPLITTING CRITERIA and split X|Y into two
     subsets (left and right)
6:   Recursively call RegressionTree in Left and right
     Nodes
7: End
```

Figure 3.8 Regression Tree

Typically, the three main steps of a regression tree involve *stopping criteria, splitting criteria* and *regression*.

3.4.1. New splitting criteria

To fit the needs of the hospital data, and the later approaches taken, we have devised a set of modifications to a regression tree algorithm. These modifications are meant to improve time and accuracy specifically for piece wise linear regression.

3.4.1.1. KGERS Splitting

As seen before, variance splitting might help, but the problem with it is the fact that splitting is done by Euclidean distance of clustered points. If at the leaf nodes we want to fit a line, then variance will not do a good job. Variance will work for average regression at leafs, but not so much for a hyperplane. There must be splitting criteria that split the data by its linearity rather than its distance to average. We can run KGERS on each candidate example cluster and then use the error squared as impurity. This is shown in Figure 3.9.

```
PickBestSplit (KGERS version)
-inputs:
Features(F), X(inputs), Y(outputs)
-outputs: (best_f,best_$f_v$)
1: Begin
2: BestGain = 0, (f_best, f_$best_v$) = null;
3: let P be hyperplane fit on X using KGERS
4: AllImpurity = CalcError (X,P,Y)
5: For each f in F
6:  For each split point $f_v$ in f
7:    Split X using (f,$f_v$) into $X_{left}$ and $X_{right}$
8:    Split Y using $X_{left}$ and $X_{right}$ into $Y_{left}$ and $Y_{right}$
9:    Let $P_{left}$ be hyperplane fit on $X_{left}$ using GD
10:    Let $P_{right}$ be hyperplane fit on $X_{right}$ using GD
11:    Gain = Allimpurity - (CalcError ($X_{left}$,$P_{left}$,$Y_{left}$) + CalcError ($X_{right}$,$P_{right}$,$Y_{right}$))
12:    If (Gain > BestGain)
13:     BestGain = Gain
14:     (f_best, f_$best_v$) = (f,$f_v$)
15: Return (best_f,best_$f_v$)
16:  END
```

Figure 3.9 Pseudo code for PickBestSplit by KGERS hyperplane fit

Steps 4 and 8 calculate the errors generated by the new hyperplane with respect to the training data. This function will try to maximize the split that generates the smallest gain, where gain is defined as the squared error of the hyperplane before and after splitting. Impurity can be defined in equation (16).

$$I = (\hat{f}(X) - Y)^2 \tag{16}$$

Impurity could be defined as the squared error of the current hyperplane. What matters is that impurity increases if the model has a lot of error, and vice-versa. Step 11 of Figure 3.9 shows the definition of gain. This idea is very parallel to that of splitting by variance, but instead optimizes the data for linear regression at the leaf nodes. KGERS is very fast too, so this splitting criterion method will be efficient and practical.

3.4.1.2. Gradient Descent Splitting

This technique is the same as shown in Figure 3.9 but using Gradient Descent to fit a line instead of KGERS. While this technique will certainly help achieving a good splitting pattern for linear regression at leafs, it is slow (high time complexity) and each run of Gradient Descent requires several parameters. Because during a split of data the distribution of Y values might change this could mean that the same learning rate will not be sufficient to achieve convergence. Instead of using Gradient Descent, KGERS can be used. It has to be kept in mind that when choosing a splitting method, every single feature and every single threshold value of this feature has to be explored for possible splitting points. If the algorithm that calculates the impurity of splitting points is computational expensive then the splitting criteria decision itself might be non-solvable. The final and biggest advantage on KGERS over gradient descent is the usage of a single parameter. With this, KGERS split should be the splitting criteria of choice when using linear regression at leaf nodes.

3.4.2. New Stopping Criteria

The new stopping criteria devised is to make it more automatic, without the need to know the data distribution or any other measurement directly related to the data. The only parameter required for the following is a single confidence interval (a threshold) that if bigger, means a deeper tree with more leafs and if smaller a more compact tree.

3.4.2.1. T-Test on Error

The motivation for the T-Test Error stopping criteria is the splitting scenario. At the end, we want to split only if this is going to improve the performance on error and we want to stop otherwise. For this, we can see if the changes in error from splitting and stopping are significant. The T Statistic test is a measurement that tells how different two averages are. Depending on the confidence interval, measurements above the T threshold will indicate a high difference in the distributions. Low values indicate no such difference.

This measurement can be used to measure error differences, as shown in (17).

$$D = \frac{(\delta_{after} - \delta_{before})}{\sqrt{(\sigma_{before}/N_{before}) + (\sigma_{after}/N_{after})}} \qquad (17)$$

If the D measurement is bigger than T it means that the two distributions are different. The symbol δ represents the average error of the current node (on its own subset of data). This means that δ_{before} is the average error of the current node before splitting and δ_{after} is the average error after splitting. The symbol σ represents the standard deviation. Depending on the state, standard deviation of this distribution is from before and after splitting. If these two distributions have a value of D that passes the threshold T (derived from a confidence interval table), it means that it should continue splitting, since there is a significance change in error. If the D value is lesser than T, then the algorithm should stop splitting since further splitting doesn't decrease the error significantly (from a statistical point of view).

It has to be clear, this stopping criteria is generic and can be applied to any Regression Tree with any particular regression model at the leaves. As long as there is the calculation of error before and after splitting, this stopping criterion will work.

3.4.2.2. T-Test on shape

The motivation for this is that if the shape of two hyperplanes fitted is different and doesn't change, then there is no reason to keep dividing. If the hyperplane candidates generated before are no different from the candidate hyperplanes generated after splitting, then splitting would be silly and wouldn't really change the nature of the regression, therefore a stopping condition is met. If the opposite happens, the shape of the hyperplanes is very different before and after splitting, it means that the algorithm is potentially discovering new ways of fitting a model, and that the model below certainly lacks of a linear behavior, therefore the splitting continues. This stopping criterion can only be applied to those trees with linear regressive model at the leaves (KGERS or Gradient Descent).

The statistical test on shape of the split data looks at the linearity shape instead of taking an error perspective. It is also a post splitting stopping criteria so requires an analysis of before and after splitting (just like T-Test on Error). The same equation in (17) applies, but there is a difference in the meaning of the average and variance. The average in this case is the average cosine similarity measure. This measure is defined in (18).

$$CosSim(A,B) = \frac{A \cdot B}{||A|| \, ||B||} \tag{18}$$

The cosine similarity is a measurement that dictates how similar two vectors are. If the value of *CosSim* is 1, it means that both vectors are equal; otherwise -1 means that they are the complete opposite of each other.

The T-Test is done on the average cosine similarity of all the KGERS hyperplane candidates fitted before and after splitting. The vectors used in the cosine similarity function are the weight vectors of the candidate hyperplane. For all the possible pair of vectors in each group, an average cosine similarity mean and variance is extracted, being these the two averages.

3.4.3. Regression at Leaf Nodes

KGERS represents a viable new regression algorithm. It's an algorithm that requires much smaller time complexity and only one parameter independent of the Y's distribution.

The regression methods are not limited to linear regression; neural networks can be also applied, but then lays the question of splitting criteria, which seems a little bit out of scope of this study. For the purpose of this study the focus will be on piece wise linear approximation. KGERS at the leaf nodes will be ultimately compared against gradient descent and a mean.

3.4.4. Summary of regression tree algorithms

Table 3.2 summarizes the sources of each algorithm. It also clarifies the current proposed ones as a contribution to the computer science field.

Source	Splitting criteria	Stopping Criteria	Regression
Yohannes and Webb (1999)	Variance	Number of Nodes	Average
D. Vogel et al (2007)	Clustering	Levels	Gradient Descent
Huang and Townshend (2003)	Variance	Number of Nodes	Gradient Descent
Proposed approach	KGERS Split	T-Test on Error	KGERS Regression
Proposed approach	KGERS Split	T-Test on Shape	KGERS Regression

Table 3.2. Summary and Sources of regression tree algorithms

3.5. Summary

This chapter proposed a new algorithm for linear regression, KGERS. The objective of this algorithm is to be faster than regular approaches, like Linear Least Squares and Gradient Descent. It was shown through time complexity analysis that KGERS is not dependant on N, a crucial variable when estimating time complexity. Although KGERS still depends on K, which is the main iterations parameter, and this parameter is straightly linked to the linearity of the data. If the data tends to be linear, K should be lower. The main purpose of KGERS is to be used in a regression tree as both, splitting criteria and regressive model. Two methods for stopping criteria are proposed. These are performing T statistic tests on Error and shape of potential curves.

Chapter 4

Estimating Hospital Admissions

As seen in the introduction of this study, one of the main goals is to successfully model hospital data to solve the staffing problem. This chapter will describe the data being used, along with some of its features. The contributions in this section will help with the modeling of demand in emergency department admissions and specific care unit admissions. As recalled in previous chapters, there exist two sub problems. The first one involves the estimation of admissions from the emergency department (ED) into the main hospital. This can be thought as the patients going out of the emergency department into a hospital room for a span of time. The second sub problem involves the estimation of demand into one of the three types of care units, which will be described later.

4.1. Hospital admissions data

The hospital admission data consists of four specific sets. These were kindly provided by Holmes Regional Medical Center, located in Melbourne Florida. This is important since the results discussed in this chapter do not necessarily reflect the behavior of patients around the nation, but from a specific town. These four sets can be divided into two groups. The first group is concerned with sub problem A and the second to sub problem B. These sub problems are explained in detail in the introduction of this study. The sets are summarized in Table 4.1. The first problem is the admission of patients from the emergency department to the hospital internal beds (EDInpt which stands for Emergency Department Inpatients). As explained before, this is a big problem in hospitals nationwide. If a hospital knew in advance the number of people who will be admitted, or at least an estimate, staffing nurses and doctors would be a fairly simple task. The second sub problem deals with the actual supply of care units that need to be given. A care unit is the composed by a bed and the additional devices and items that a patients need. Since this data is unavailable, we will use the demand for the specific care units (which is available) assuming that the hospital can match the supply if it knows the demand.

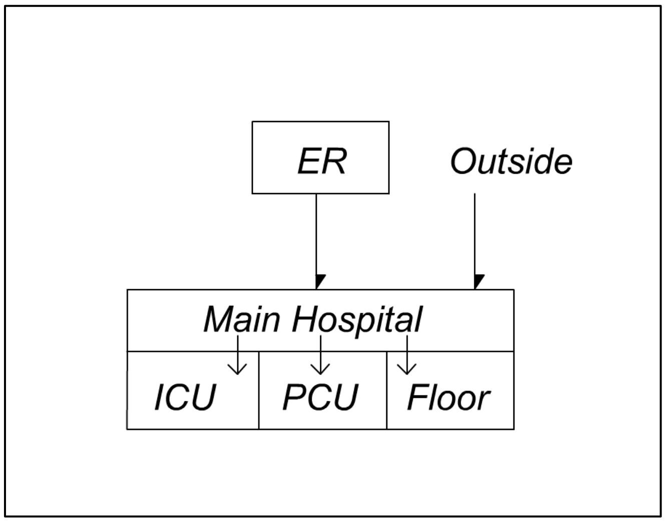


Figure 4.1 Outline of Hospital data

Figure 4.1 shows the main outline of the hospital data studied in this chapter. People admitted from the ER to the main hospital are represented by the arrow going from the ED box to the main hospital. The other arrow (coming from outside) represents the demand coming from other sources that are not the emergency room. These will not be studied on this research work. Once inside the main hospital, both demands "merge". The total demand is then split into three types of bed units, PCU ICU and Floor. Notice that these units include both, demand coming from the inside and the ER.

Data set	Training Set	Test Set
Hospital Admissions from Emergency Room (EDInpt)	April 2008 – May, 2009	April 2010 – May 2010
Demand of ICU beds	April 2008 – May, 2009	April 2010 – May 2010
Demand of PCU beds	April 2008 – May, 2009	April 2010 – May 2010
Demand of Floor beds	April 2008 – May 2009	April 2010 – May 2010

Table 4.1 Training and Test sets for hospital data

Table 4.1 shows the distribution of each bed in the hospital. Here a training set will be used solely for model building. The test set is meant as an evaluation reference point that will help measuring the accuracy of the algorithm. Staffing requires calling personal ahead of time, and assigning them specific beds depending on the workflow of the hospital. Treating the problem as a time series is a very promising strategy due to the prior studies done in the entire data sets. The next three data sets deal with the actual type of bed used from this EDInpt demand plus external demand. These are numbers that tell how many types of beds are taken. As seen before there are three types of bed and each type of bed is described in Table 4.2. Each bed's description and type is an important part of staffing. Each type requires different kind of personal and equipment. Knowing them in advance will certainly proof useful for the hospital personal.

Type of bed	Meaning	Description
ICU	Intensive Care Units.	Beds with patients who are in critical condition. These beds are most costly and require special equipment. Nursing ratio is 1 nurse for 2 beds.
PCU	Progressive Care Units.	This care units present less critical patients, with a nursing ratio of 1 nurse for 4 beds.
Floor	Floor units (other).	General patients, nursing ratio being 1 nurse for 6 beds.

Table 4.2 Hospital bed types.

The EDInpt data consist of a set of rows. Each row represents a patient, with its respective admission time to the hospital. Admission times are already in six daily hour intervals; each interval being of four hours each. PCU, ICU and Floor come also as row data, but this time, without the hour. This means that they come packed in terms of hours. The goal is to treat the problem as a time series, so at the end the raw data needs to be processed and put into buckets. For EDInpt each number in the time series will represent the number of admissions that happened in a particular hour. This means that each bucket in EDInpt represents a 4 hour interval. On the other hand, PCU, ICU and Floor data are sampled in a daily bases. This means that each bucket from these data will represent the number of PCU/ICU/Floor beds used in the particular day. Just like the previous chapter, each data set consists of a training set and a validation set.

4.2. Experiment procedures

In this chapter one set of experiments will be driven on the EDInpt data, ICU, PCU and floor untis. These will provide evidence on accuracy and time complexity of the newly devised regression tree on the hospital data. A summary of the algorithms used is available in Table 4.3, each with a particular description.

Name	Algorithm	Description
Neural network	Feed Forward Neural network trained with Back Propagation	Most common “off the shelf” neural network algorithm. Will be used to be contrasted in computational power and accuracy against regression trees.
Regression Tree KGERS	Regression Tree with KGERS for splitting, t-Test on shape stopping criteria and KGERS for regression.	Algorithm proposed. Aims to provide faster and similar results to those of Regression tree with Gradient Descent.
Regression Tree Mean	Regression Tree with Variance splitting, t-Test on Error for stopping criteria and average for regression.	Control algorithm, since it takes no parameters it will be used as reference due to its minimalistic nature.
Regression Tree Gradient Descent	Regression Tree with Gradient Descent splitting, t-Test on shape stopping criteria and Gradient Descent for regression.	Typical algorithm that performs piecewise linear regression, with usually good results. Aims to be compared and contrasted with Regression Tree using KGERS.

Table 4.3 algorithms for non linear regression experiments

Each data set will contain three types of analysis. The first one is an evaluation of computation time; this is done by plotting a chart of CPU time versus size of input (the N variable in time complexity). The second sub experiment is an evaluation of error versus value of the algorithm’s main parameter. This will be done for each algorithm respectively. Finally the last sub experiment is an analysis of CPU time versus error for each algorithm. This third part should be able of identifying the strongest algorithms that take the most advantage of computational power. Later in Chapter 5 we will look into the algorithms from a computer science point of view and evaluate those using synthetic data. In this chapter the algorithms will be evaluated for performance on regression of hospital data, and not in other criteria such as resistance to noise.

4.3. Evaluation Criteria

The main evaluation criteria used for the experiments is the root mean squared error (RMSE) of the regression. The RMSE summarizes the error at about 90% confidence interval, and displays the units by which the regression is off. Other measurements include using the RMSE as a percent error, by diving it to the total mean of the Y values in the experiment. Equation (1) summarizes the RMSE of a model.

$$RMSE(f,x,y)=\sqrt{\left(\frac{\sum_{i=1}^{N}(f(x_i)-y_i)^2}{N}\right)} \qquad (1)$$

The RMSE takes three inputs. The first one is *f*, this represents the model (the output of our regression algorithms in this case). The model can be thought as a function itself. The next input is *x*. This is the respective test set for the model. Adjunct to this is *y*, which represents the actual outputs of the test set. The RMSE is simply the average error amplified using the quadratic property. This RMSE can curiously be greater than the average values of the Ys.

For the hospital's interest, the Mean Absolute Error is also calculated. This can be described in equation (2).

$$MAE=\frac{1}{n}\sum_{i=1}^{n}\left|f(x_i)-y_i\right| \qquad (2)$$

The MAE describes exactly in average by how much is the model diverging from the test set. Notice that the difference operation is not exponential; this naturally causes the MAE to be lesser than the RMSE since it doesn't "exaggerate" errors.

Another evaluation criterion is represented by the CPU clock cycles. The experiments were performed on an *Intel Core Duo©* machine, running *Microsoft Windows 7* 32 bit operating system. The CPU clock cycle function was extracted using the c runtime library *clock()* function, declared in the *<time.h>* header file. We use CPU as a measurement of how well the algorithms take advantage of the computing power, and therefore measure the most efficient one using this empirical evidence.

Finally, for the hospital data set, each algorithm will be evaluated using the RMSE measurement. In the case of randomized algorithms (such as Regression Tree KGERS) the RMSE used will be the minimum amongst all its trials.

4.4. Preliminary analysis for feature selection

Before drawing any features and start doing regression, we need a prior study; a guide that will tell us the data's trends and possible features that could be derived. We have chosen to first sample the data as a time series. This is done by counting the patients being admitted into buckets. Once the counting is done, the result is a list of numbers, which represents a time series. Each number in the list will represent the number of admissions or beds in the time span that a bucket takes. As mentioned before, EDInpt uses 4 hour buckets, while PCU/ICU and Floor use daily buckets.

Once the time series is at hand, the next step is to perform an FFT analysis of such. Fast Fourier Transform is a method of decomposing a signal into its primitive cycles, helping to identify any cyclical patterns into the data. A summary of the FFT analysis values is summarized on Figure 4.2. While running the FFT analysis on the EDInpt data, three patterns were discovered. The three cycles are listed below:

- Daily cycle: hour of the day (out of the 6 possible hours) shows similarities.
- Yearly cycle: same days of years show similarities.
- Weekly cycle: Same days of the week share similar patterns.

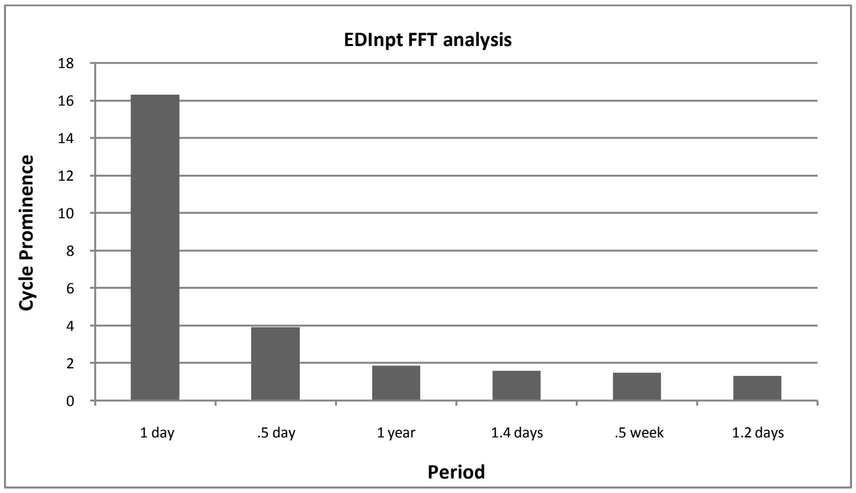


Figure 4.2 FFT analysis cycle prominence of EDInpt data.

Figure 4.2 shows six different bars. Each bar represents the prominence of a cycle in the EDInpt data. As seen here, the first two bars show a high prominence of daily cycles. The next biggest cycles happen to be the yearly repetitions. Finally the weakest one, but still present is the weekly cycle. All these cycles are normalized against an epsilon measurement which represents no cycle at all. So while the weekly cycle happens to appear very low, it is in fact still prominent, in the exact proportion shown with respect to the daily cycle.

Figure 4.3 shows the FFT analysis performed on the other three data sets. We decided to plot them on the same chart since their sampling is different from that one on the EDInpt data.

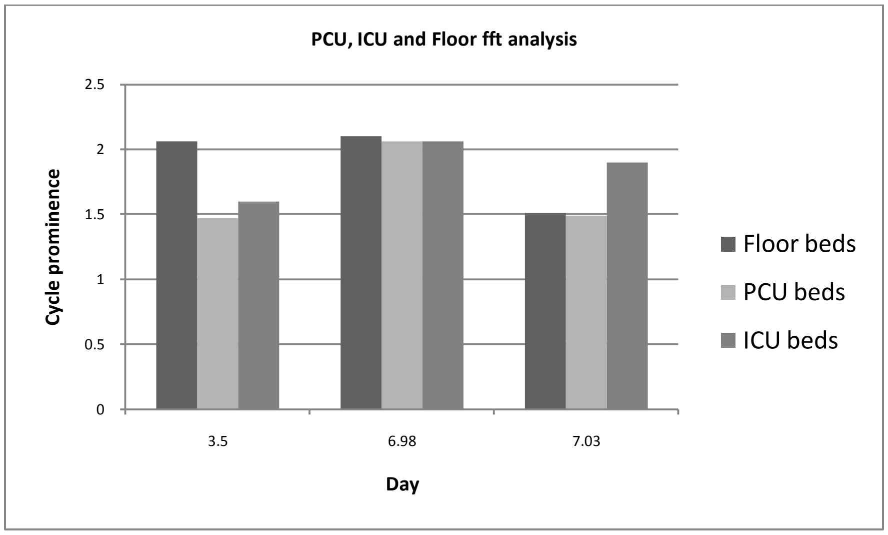


Figure 4.3 FFT analysis cycle prominences of PCU, ICU and Floor data.

The PCU, ICU and Floor data related to *subproblem B* clearly show a prominence of weekly cycles. The only cycles identified were those in the span of a week. This means that days will be showing a specific pattern that is repeated throughout the data.

As seen before, there are three major cycles being prominent in the EDInpt data. The next step is to plot these cycles and analyze the respective shapes that each one present. Figure 4.4 shows the respective shape of daily cycles in data.

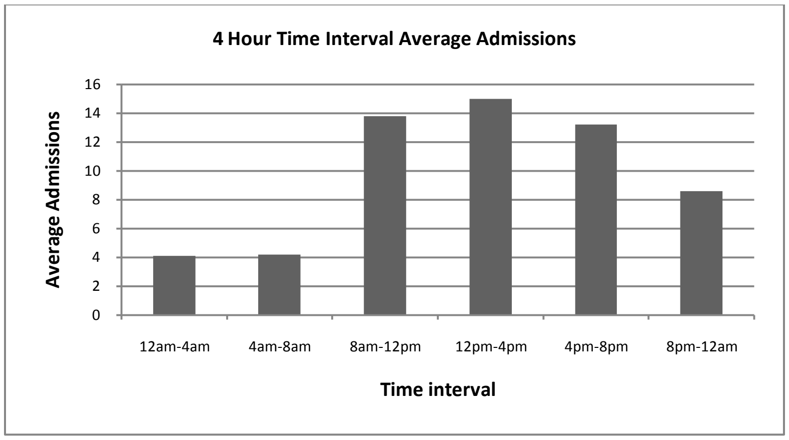


Figure 4.4 Shape of daily cycles.

Each entry in the X axis represents the given time interval. This data is plotted as the average a number of admissions that each time interval (or bucket as we mentioned before in the time series manner) possesses. As it can be appreciated, the busiest times for the target hospital are always close to noon and afternoon. Early hours in the morning seem not to be as demanded. A logical explanation is that usually people sleep during the night. The likelihood of getting injured while sleeping is less than actually during active hours. But this is just speculation and the causes could be different; however it seems very logical that the highest hours of hospital admissions happen to be in the rates of the mid day.

Figure 4.5 shows the admission pattern for monthly cycles. That is, using the original sampled data, now displays the monthly average for the entire data set. Figure 4.5 summarizes the years into their twelve respective months, each with its average number of admissions. It shows that months around early in the year (spring in the case of the United States) is a choke point of high admissions while months closer to the summer have less admission.

There exists a considerable difference in month averages, strictly saying that the least demanding month differs for at least 12 patients from higher ranked months. Finally Figure 4.6 shows the behavior of the weekly cycle, which is the least powerful amongst the data. It can be appreciated that Figure 4.6 displays two waves in one. The first one starts around the early days of the week, achieving its peek on Monday and a recessive period on Wednesday.

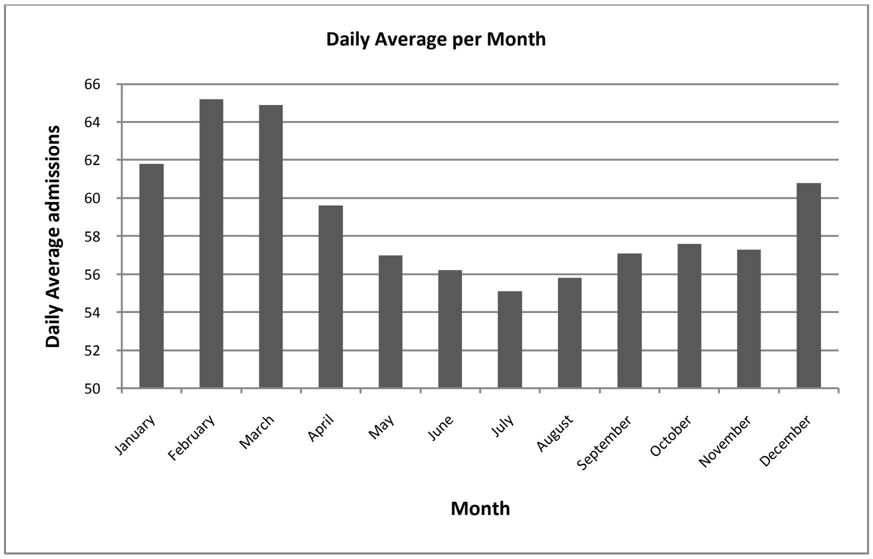


Figure 4.5 Shape of yearly cycles.

It can be noted that after this recessive period, the admission rates picks up and then drops suddenly during the weekend. It's very surprising that not many admissions occur during the weekend. A possible logical explanation is the way people act during the weekends, and sometimes the lack of activities they are involved in. During the week, it is for sure that transportation moves predominately in the local area, and this is the case in the city of Melbourne.

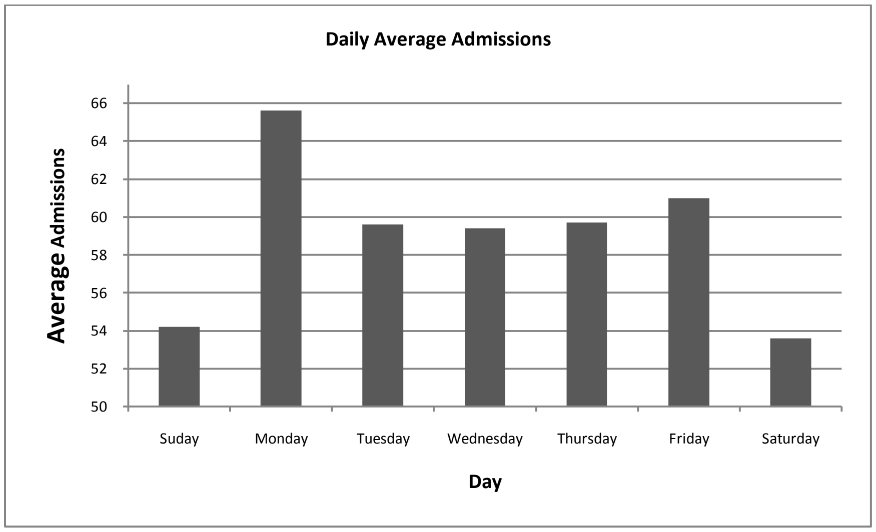


Figure 4.6 Shape of weekly cycles.

As it was seen before, the three remaining hospital data sets also show a weekly pattern, which seems to be the most prominent amongst the cycles. This pattern is graphically shown in Figure 4.7.

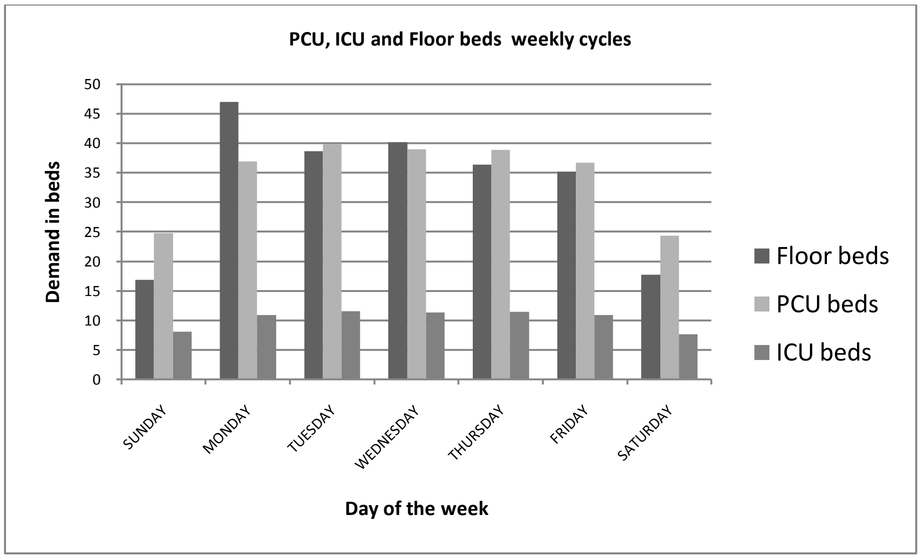


Figure 4.7 PCU, ICU and Floor weekly cycle shape

The weekly pattern of the bed data seems to have its peek around the middle of the week, especially Monday. It can be appreciated that each data set (Floor, PCU and ICU) has different rate of admissions, floor beds being the most abundant. ICU beds are the most costly and hard to maintain. Usually patients residing in these beds require more nurses and more life support equipment, which is expensive and scarce. Floor beds seem to be the most abundant since usually the hospital gets overcrowded easily, and patients overflowing form the emergency department have to be sent through here.

Finally this complete cyclical analysis can lead to feature investigation and extraction. The algorithms discussed in the previous chapters will need at the end a set of vectors for each point in the data (where each point is a bucket sitting in a time series). Each one of these points will have a vector associated with it that will summarize information of the past. The goal is to be able to forecast the future using the past, and surely the FFT analysis with combination of the time series bucket plots will give us hints to do so.

4.5. Processing of data and Feature Vectors

In order to be able to use the data into the machine learning algorithms mentioned before, the data must take the form of feature vectors. Each point should consist of a vector with its respective regression value. The case of the EDInpt data, the data comes as 6 time intervals per day. This means that each day consists of 6 numbers that represent 4 hour time intervals during the day. PCU, ICU and Floor data sets have one number per day instead; data is not sampled by hour but by day for these last 3 data sets.

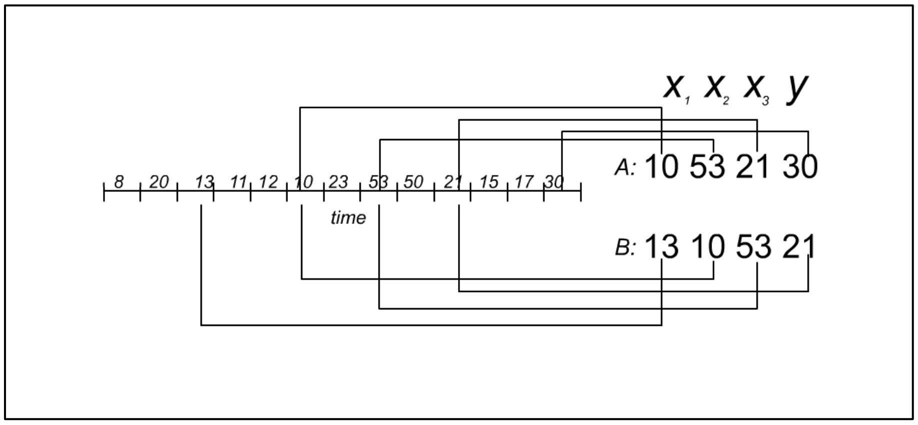


Figure 4.8 Feature Extraction from time series

Figure 4.8 shows the process of extracting features from an admission time series. The vertical timeline represents the admission numbers on each bucket for the respective time series. This is the raw version of the data given by Holmes Regional Medical Center. The feature extraction process consists of extracting a window time frame as shown in Figure 4.8. For this example two feature vectors are used, point A and B. Each point has an association with the respective time line, acting like a window that traverses time. This "time window" uses data from the past to derive features specifically. The next section will describe exactly what each element in the feature vectors represents.

4.6. Feature selection

The feature vector designed for the EDInpt data is entirely based on the prior studies and FFT analysis explained before. These features will be extracted from the timeline data. To recap the entire process, we first start with rows of patients. After the data is sampled into buckets each bucket being on a specific time. For the EDInpt data, each bucket represents a 4 hour span during the day. The next step is extracting features for each possible prediction date. For example, if we want to know the number of admissions to the hospital next Monday in the second bucket (that is, from 4 am to 8 am), we will use data from the past and feed it into our regression algorithm, that supposedly already has created a model. Figure 4.9 shows the derivation of the first feature.

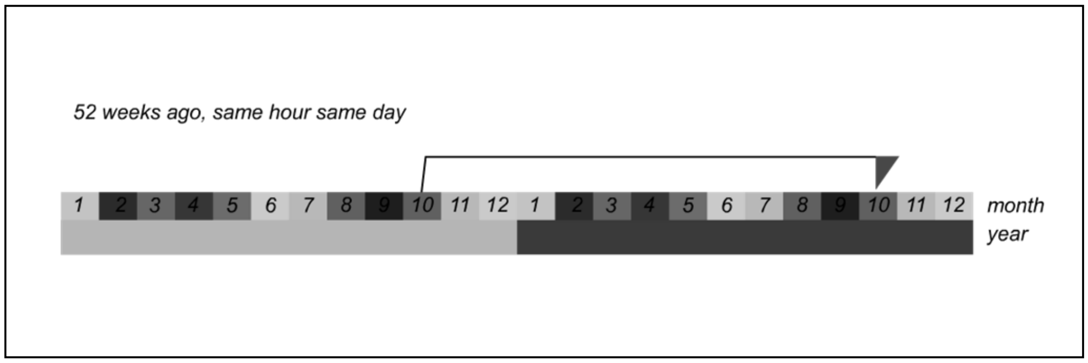


Figure 4.9 First feature of EDInpt data

The first feature involves the same hour, 52 weeks ago (which is a year ago, but the same day of the week) using the same hour. Figure 4.9 shows a Monday 3rd of October from 4 am to 8 am (second bucket of the day) as the green arrow. From now on, every feature will always involve the same day of the week and the same hour bucket of the day. The green arrow is the target prediction date. The end of the arrow shows the same Monday, but 52 weeks ago in October, same hour. This will be the first element of the feature vector. Figure 4.10 shows the second element of features. The second feature uses the average of the four days surrounding the past day. This is based on the assumption that days surrounding the past might also influence the future prediction which occurs a year after. It also helps smoothing out the influences of days. We use two weeks span since a month shows consistency in earlier FFT analysis. It is out of the interests to see this span.

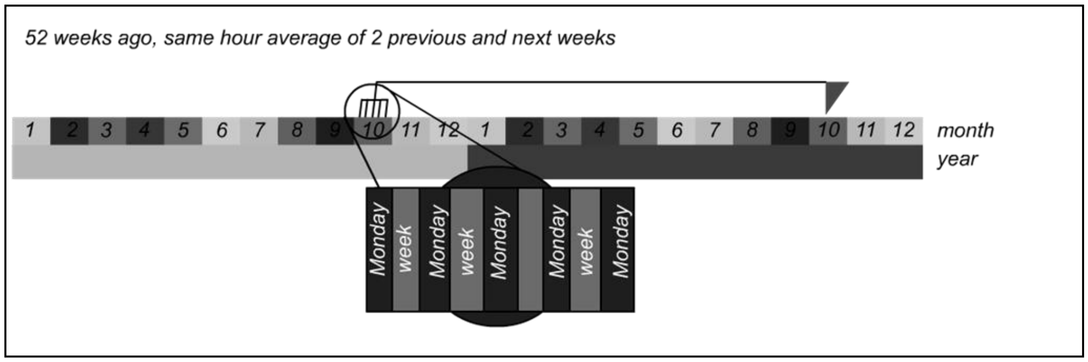


Figure 4.10 Second feature of EDInpt data

Figure 4.11 shows the third feature of the EDInpt data. This feature consists of 365 days ago. The reason being is that a year ago means the same date. Consequently holidays like the 4rth of July are likely to present the same amount of inflow in patient data. We believe that such patterns do influence potential prediction and modeling.

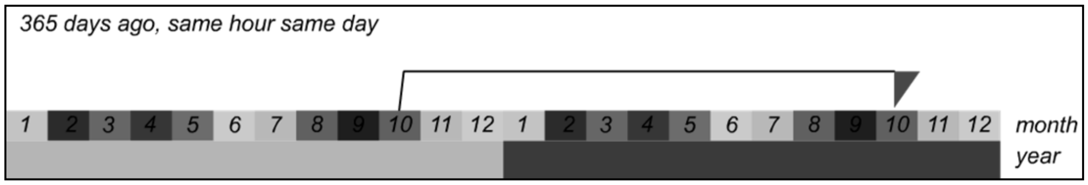


Figure 4.11 Third feature of EDInpt

Figure 4.12 shows the fourth feature. The fourth feature uses the average of the last three exact days in a span of 3 weeks. This means that it will use 1 week exactly before the prediction, in addition to the previous week, and the week before the previous week to do an average.

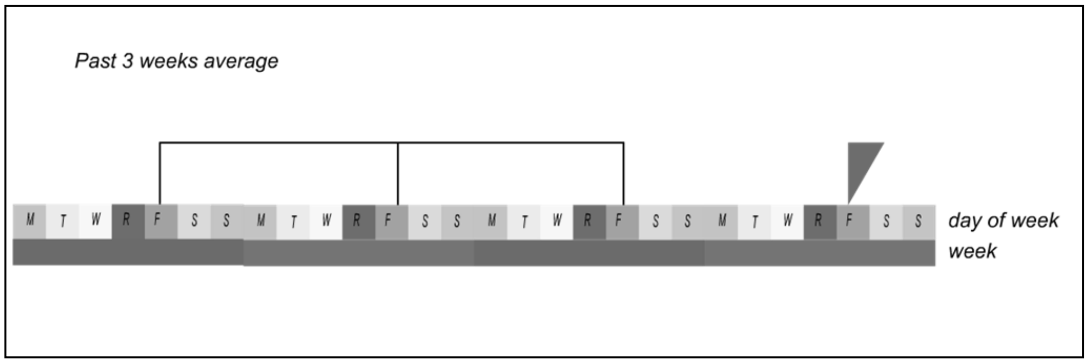


Figure 4.12 Fourth feature of EDInpt data

Again, the reason to pick the average is the assumption that we are smoothing the influence patterns (potentially smoothing noise) and using adjacent days that could influence the prediction. The fifth feature is shown in Figure 4.13.

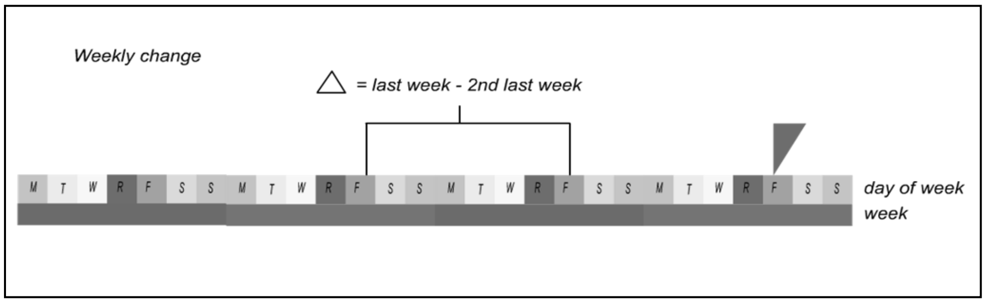

Weekly change
△ = last week - 2nd last week
M T W R F S S M T W R F S S M T W R F S S M T W R F S S
day of week
week

Figure 4.13 fifth feature of EDInpt Weekly change feature

The weekly change represents the step that a week took. It works as a delta, a result from the subtraction of the previous week minus the week before the previous, same hour and same day respectively. Finally, Figure 4.14 shows the last feature of the EDInpt data. This feature represents the yearly change, and it is a delta between the last week (because this is the most recent day we have) minus exactly 52 weeks ago, same hour and same day of the week.

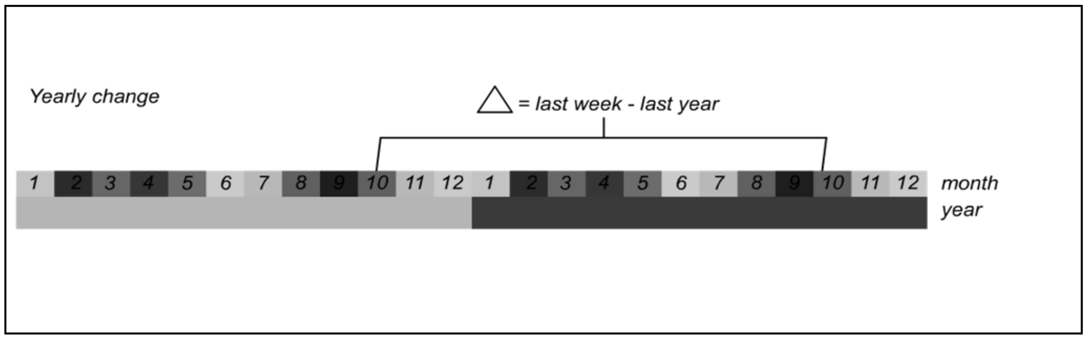


Figure 4.14 sixth feature of EDInpt Yearly change feature

At the end we have a total of 6 features, each one covering a pattern that was found during the FFT analysis.

Since the PCU, ICU and Floor data present the same cyclical patterns in the prior FFT analysis, they will all contain the same set of features. The first feature of PCU ICU and Floor is described in Figure 4.15.

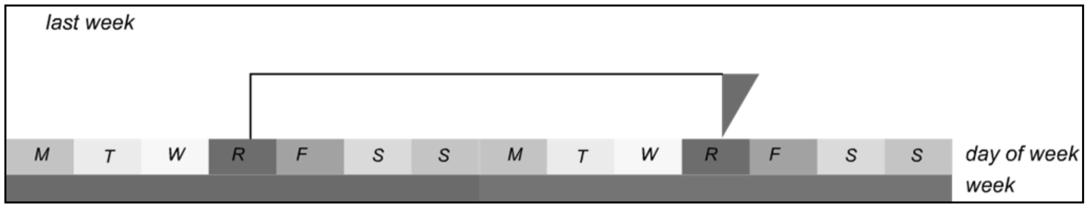


Figure 4.15 First PCU ICU Floor feature, last week value.

The first feature utilizes the concept of a weekly cycle. Simply extracts the last week’s value and uses it as its first vector value. The second feature is displayed in Figure 4.16.

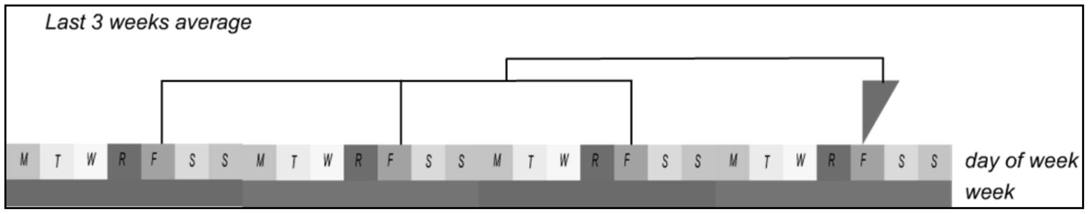
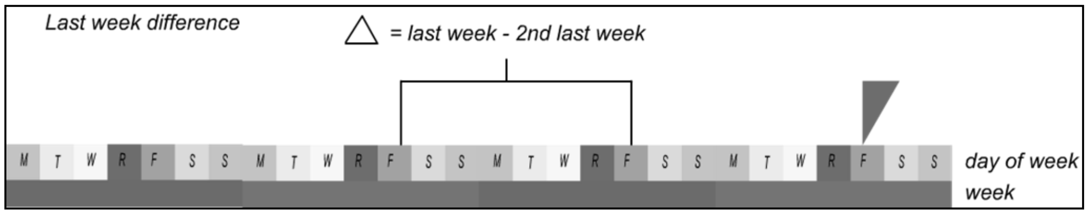


Figure 4.16 Second PCU, ICU and Floor feature, last 3 weeks average.

This involves the average of the past three weeks. We have to keep in mind that all of these three data sets are sampled in one day intervals (single day buckets). Therefore there is no specific hour like in the EDInpt data set. The third feature involves the weekly delta, which can be seen in Figure 4.17.


Last week difference
△ = last week - 2nd last week
M T W R F S S M T W R F S S M T W R F S S M T W R F S S
day of week
week
Figure 4.17 Third PCU ICU Floor feature, weekly difference.

As seen before, this data set also represents the weekly difference, but this time it is done with a single day because of the nature in which the data is sampled. This difference should give information to the algorithm about potential differences between weekly jumps. Finally the last feature can be appreciated in Figure 4.18. This feature is exactly the amount of beds used 52 weeks ago.

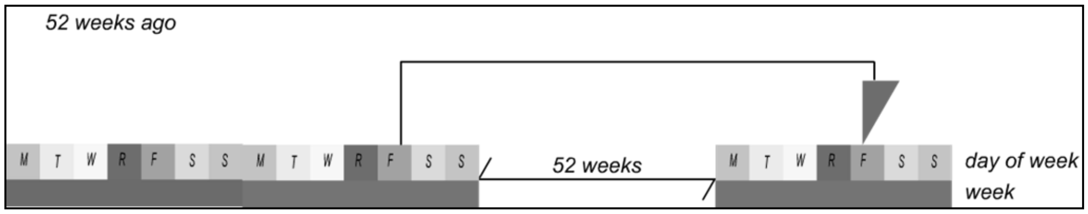


Figure 4.18 Fifth PCU ICU Floor feature, 52 weeks ago

We have to keep in mind that 52 weeks ago represents exactly one year (falls into the same year, unlike 365). Throughout the feature extraction process it has to be kept in mind that leap year could exist. In any case, in such situation the respective time corrections should be done. As explained, all these features will play an important role, and they are the result of an evolutionary process of trial and error. We found out that these are the features that work probably the best from the spectrum explored.

Vector id	Feature Description	Figure
X_1	52 weeks ago, same hour same day	4.9
X_2	52 weeks ago, same hour average of 2 previous and next weeks	4.10
X_3	365 days ago, same hour same day	4.11
X_4	Past 3 weeks average	4.12
X_5	Weekly change	4.13
X_6	Yearly change	4.14

Table 4.4.a Summary of EDInpt feature vector

Vector id	Feature Description	Figure
X_1	Last week value	4.15
X_2	Last 3 weeks average	4.16
X_3	Last week difference	4.17
X_4	52 weeks ago	4.18

Table 4.4.b Summary of ICU, PCU and Floor features.

4.7. Empirical Results

This section will show the experimental results on the hospital data. For each data set there are 4 figures. The first three figures will deal each one with a particular algorithm applicable to that data set. The algorithms are Regression Tree Gradient Descent, Neural Network and Regression Tree KGERS. Each graph will be of error vs main parameter of the respective algorithm. The fourth Figure will illustrate the Error vs CPU time that each algorithm performs.

4.7.1. Performance on EDInpt data set

Figure 4.19.a shows the performance of a Regression Tree Gradient Descent.

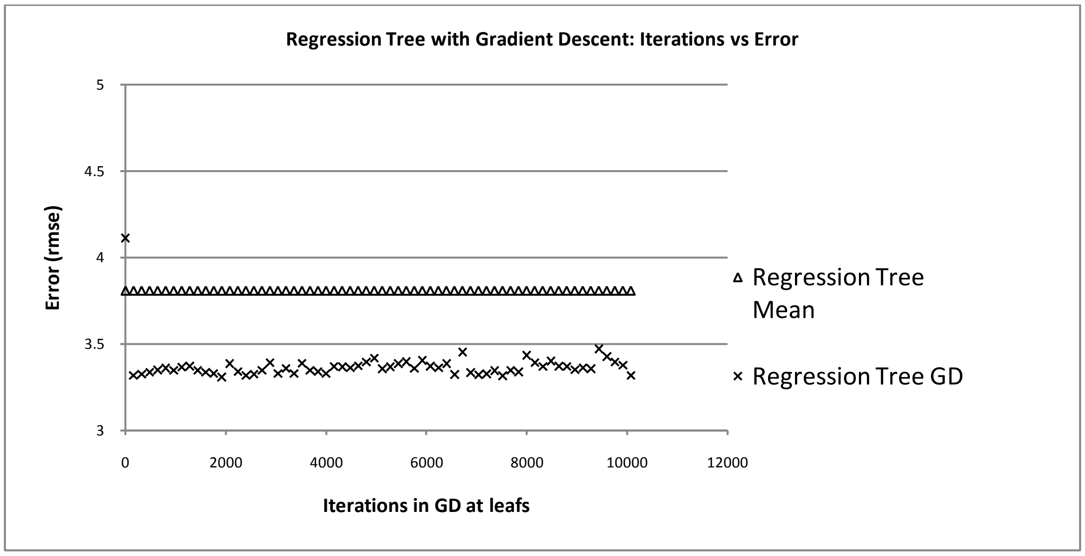


Figure 4.19.a Regression Tree Gradient Descent on EDInpt data

As it can be appreciated, these first batch of experiments deal with the EDInpt data. Gradient Descent surprisingly shows a stable result even with a low value of iterations. One possible explanation for this is that the splitting criteria performs well enough, such that the entries at the leaf nodes don’t need many updates.

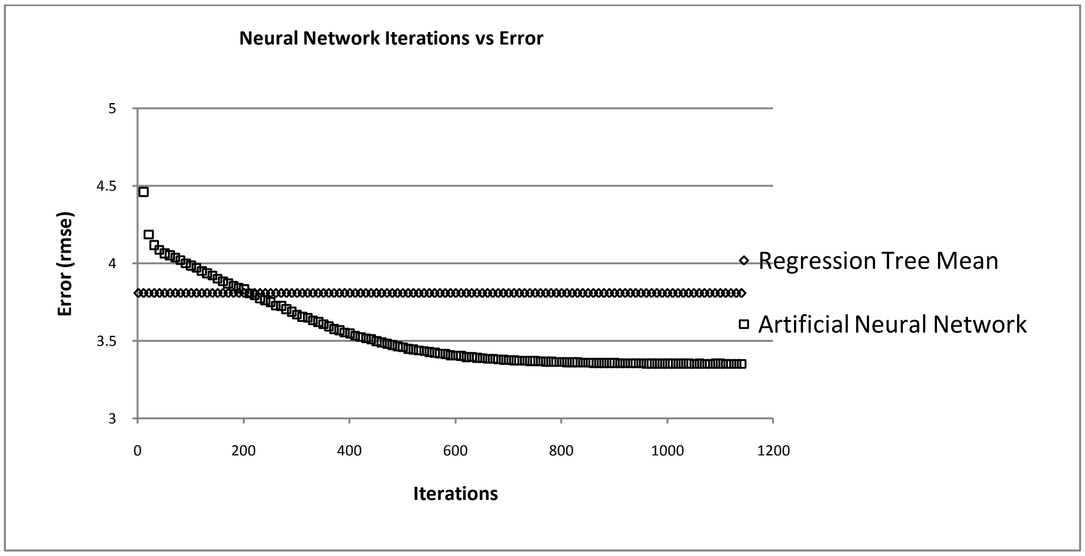


Figure 4.19.b Neural network on EDInpt data

Figure 4.19.b shows the convergence of the neural network as its iterations increase. This is expected, due to the nature of this machine learning algorithm. Notice though, how the neural network requires as many iterations as the amount of data present. This shows that the iterations parameters in back propagation is proportional to the size of the input data.

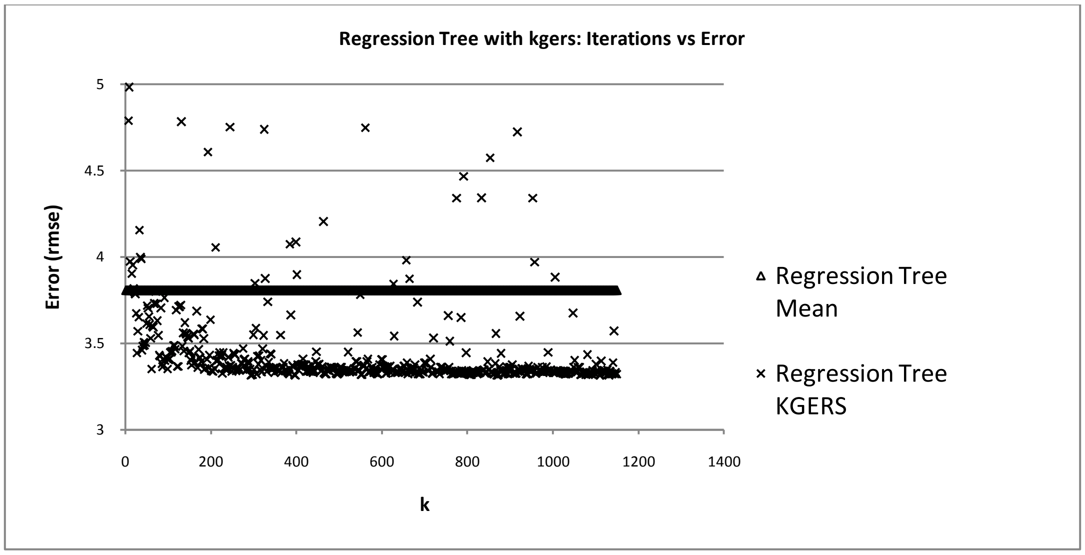


Figure 4.19.c Regression Tree KGERS on EDInpt data.

Figure 4.19.c shows the performance of a Regression Tree KGERS under the EDInpt data. KGERS converges very quickly and stabilizes at the very end. This sort of stabilization shows that the splitting criteria have done a fairly good job and that KGERS can find the correct answer quickly.

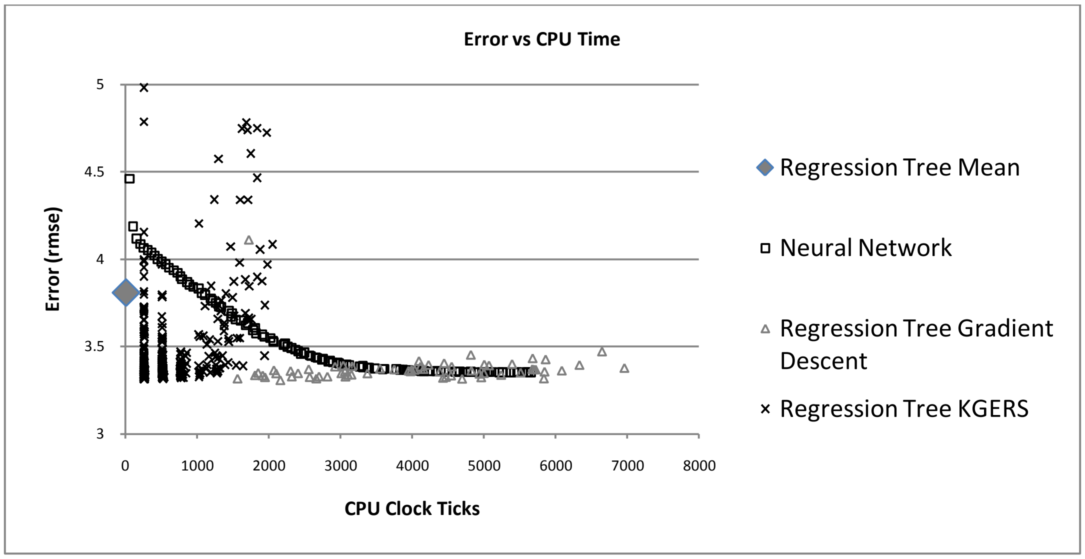


Figure 4.19.d Error vs CPU time analysis for algorithms under EDInpt data

Finally, it can be shown in Figure 4.19.d the real cpu speed for each of the mentioned algorithms under the EDInpt data. Figure 4.19.e shows a zoomed in version of the error displayed. It can be seen that KGERS has a faster approach than the other algorithms and the lowest error, unlike the Regression Tree Gradient Descent, which seems not appearing in the graph.

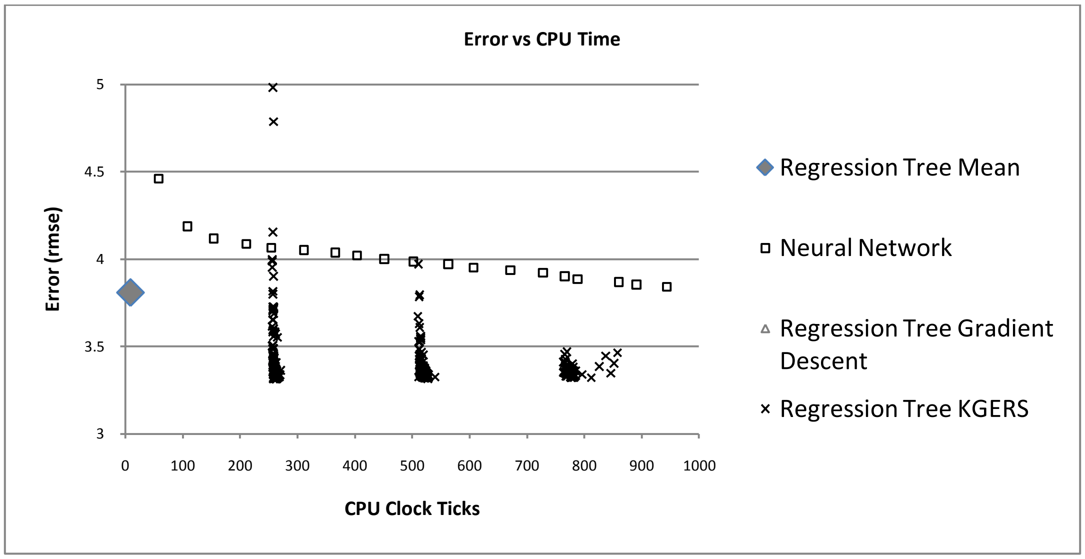


Figure 4.19.e Error vs CPU time analysis for algorithms under EDInpt data, ZOOM

Regression Tree KGERS is the algorithm that converges the fastest amongst the candidates, while Gradient Descent takes some more time to reach this error. The Neural Network performs also fairly slow compared to the regression trees. It can be shown that while the Regression Tree Mean provides the best time complexity, it over fits and doesn't converge to the lowest possible error.

It can be noted that the CPU cycle performance of the Regression Tree is falling into 3 buckets, there are around 250, 550 and 800. The reason this happens is explained by the time complexity nature of the algorithm. Regression Tree KGERS' most expensive operation is the splitting criteria. This is when the algorithm has to choose amongst all the possible splitting points the one which maximizes gain. Because KGERS is independent of the input size, varying K doesn't necessarily produce a higher CPU consumption. Instead, the randomized nature of the splitting criterion discovering patterns is what causes the algorithm CPU consumption to be between these buckets.

Figure 4.19.f shows how the CPU cycles increase as the number of leaf nodes in the tree increase.

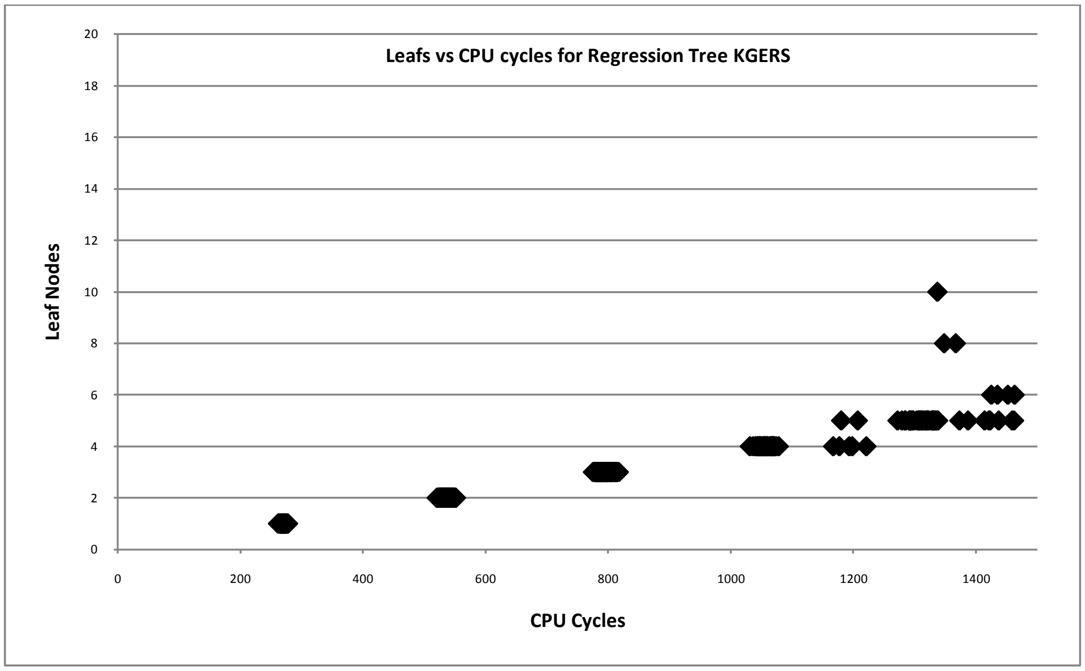


Figure 4.19.f Leaf Nodes vs CPU cycles

Notice how the number of leafs and the CPU cycles used is almost a linear relationship. This explains the bucket concentration in Figure 4.19.e. Table 4.5 summarizes the CPU cycles and the number of leaf nodes that each Regression Tree KGERS has on average.

CPU Cycle Bucket	Number of Leaves
200-300	1
500-600	2
700-800	3

Table 4.5 Average Leaf sizes with respect to CPU time

Table 4.6 shows a comparison of algorithms in terms of RMSE.

Algorithm	Best RMSE
Regression Tree KGERS	3.31
Regression Tree Mean	3.75
Regression Tree Gradient Descent	3.33
Neural Network	3.35

Table 4.6 Error comparison of Algorithms under EDInpt data

The algorithm producing the best Root Mean Square Error is the Regression Tree KGERS. It has to be shown though, that these errors do not necessarily show a more powerful contribution. But still, for the sake of forecasting and estimation, this information is useful for the hospital. The special property about KGERS is the freedom it possesses when being trained due to its randomized nature. This certainly helps into giving flexibility to the algorithm when exploring the solution space. The difficulties are that KGERS' regression tree is still tied up to greedy partitioning, and as a consequence of being a randomized search algorithm it shows several signs of instability. Still, stability can be verified in Figure 4.19.f, where it is shown that the probability of error falls thoroughly.

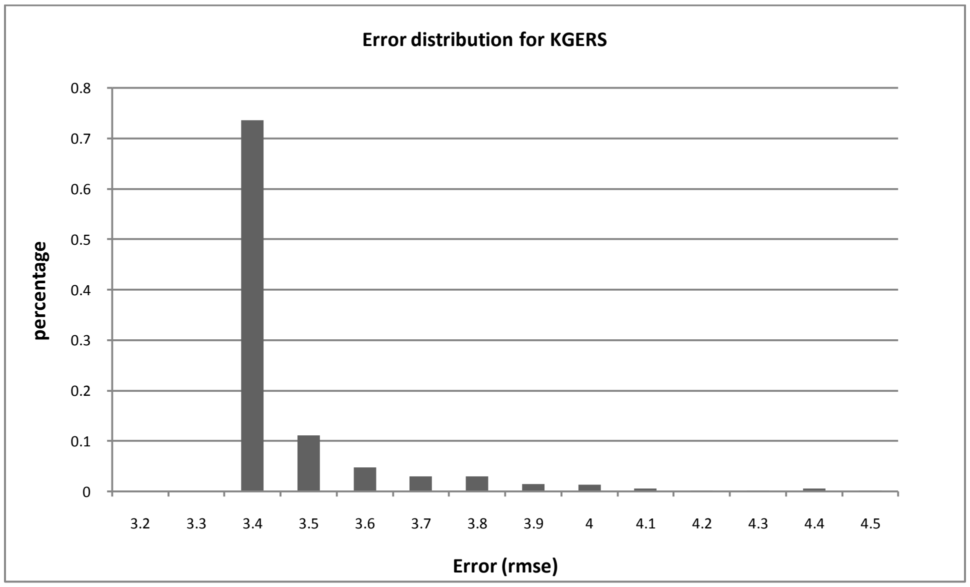


Figure 4.19.f Error distribution in KGERS for EDInpt

Because KGERS is randomized, several trials are required to know the best error. It happens that most of the trials fall under the 3.4 RMSE bin value. This means that for this data set the algorithm behaves very stable, and the majority of trials hint towards the minimum error.

4.7.2. Performance on ICU data set.

This section will perform the experiments on the ICU data presented on prior sections.

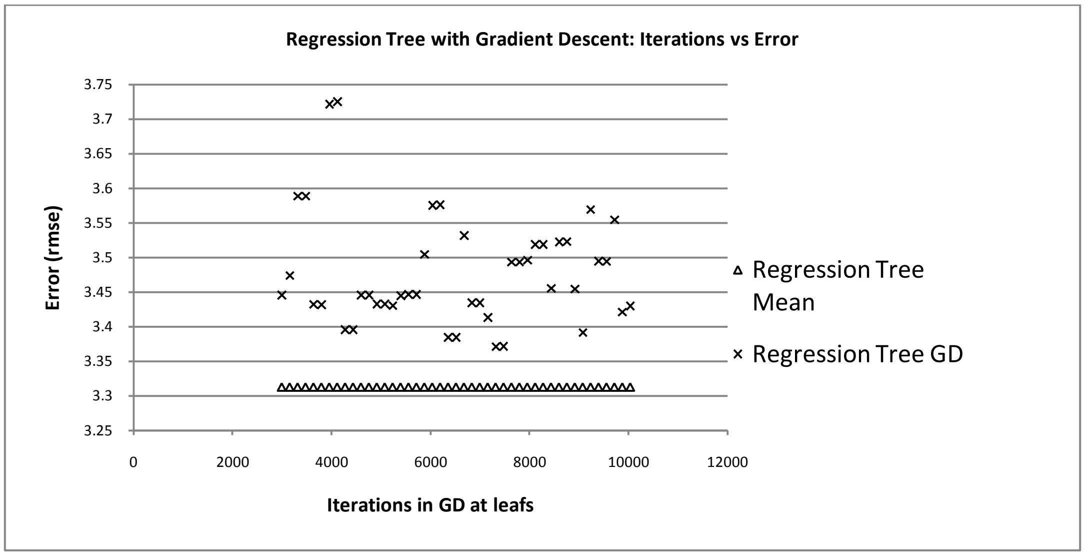


Figure 4.20.a Regression Tree Gradient Descent on ICU data

Results for the Regression Tree Gradient Descent are not very pleasing, these are shown in Figure 4.20.a. The algorithm takes much more time than Regression Tree Mean, and still performs worse. The nature of the data shows a lot of noise present. This sort of noise can confuse the splitting criteria and therefore generate a very unstable model.

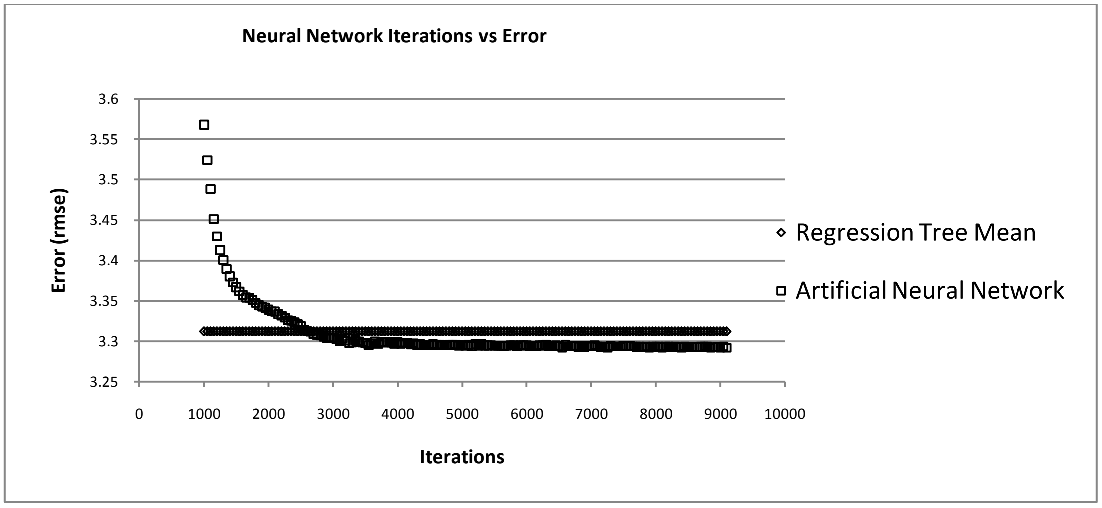


Figure 4.20.b Neural Network on ICU data

Figure 4.20.b shows the performance of the Neural Network on the ICU data. The Neural Network quickly converges to an error value, and seems to stay under that range. This is an indicator that the Neural Network does perform well on the data. The Regression Tree Mean has a similar RMSE to that one found by the Neural Network.

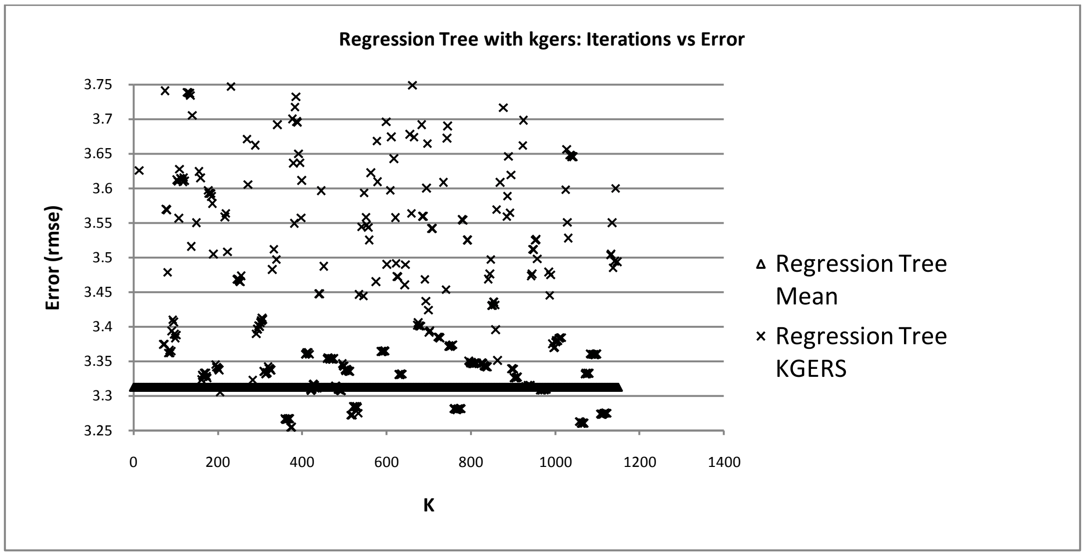


Figure 4.20.c Regression Tree KGERS leaves on ICU data

The Regression Tree KGERS at the leaf nodes seems very unstable. Although unstable, is the algorithm that got the lowest RMSE amongst the others. Instability can be due to the low mean and high variance of the data. Usually when the variance is really high, this confuses the algorithm due to its randomized nature. Splitting criterion doesn't seem to help much, since the Neural Network could fit the problem without any trouble.

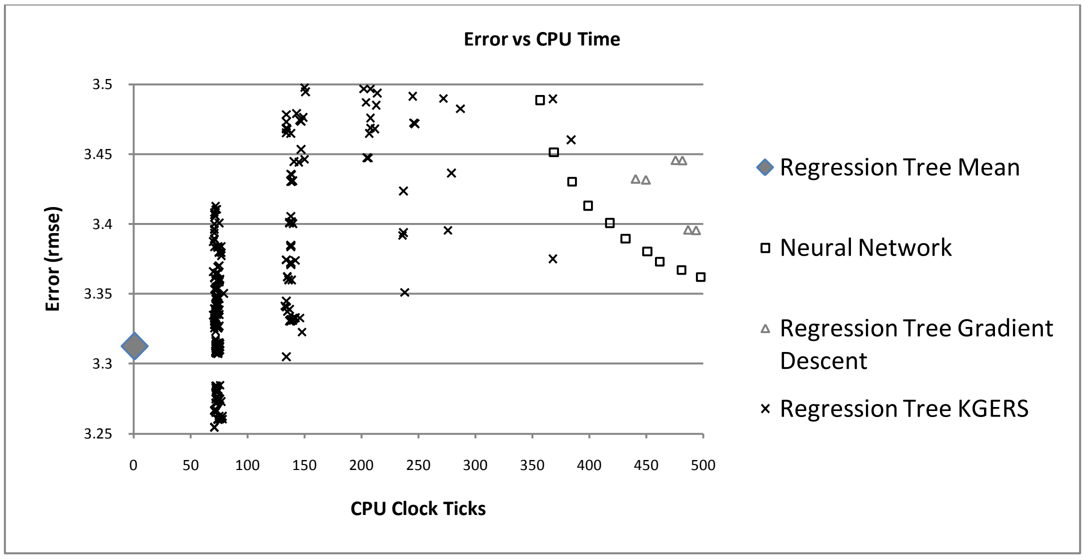


Figure 4.20.d Error vs CPU time analysis for algorithms under ICU data

Finally, Figure 4.20.d shows a summary of the results. The three algorithms converge to similar errors, with Regression Tree Mean being the one with least computational time. Figure 4.20.e shows a closer view to the Error vs CPU on the ICU data set.

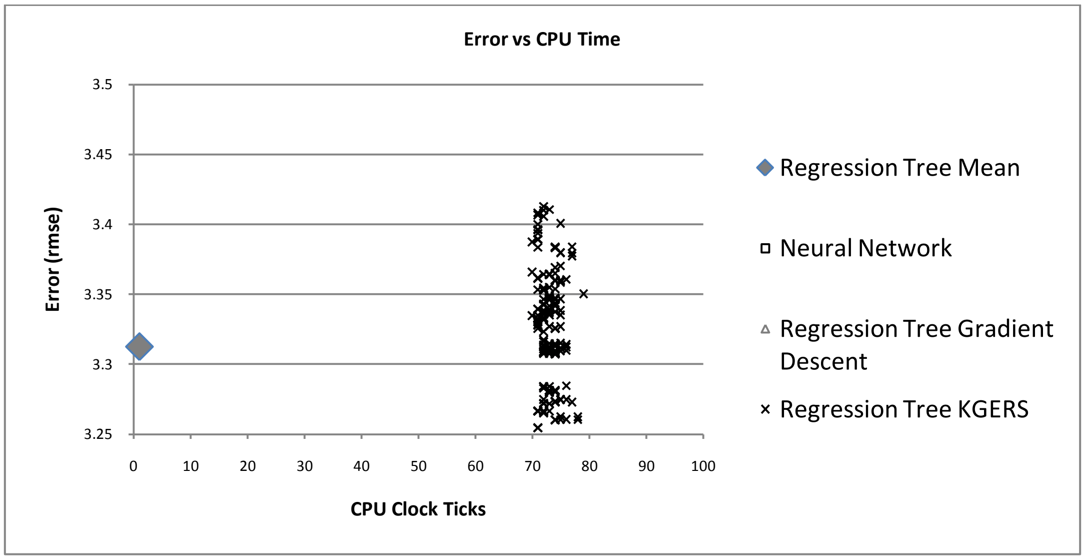


Figure 4.20.e Error vs CPU time analysis for algorithms under ICU data, ZOOM

Regression Tree KGERS shows a significant improvement into the error results. These show the potential of KGERS to be into the least amount of errors within a respectable range, doing better than any other algorithm with low computational time.

Algorithm	Best RMSE
Regression Tree KGERS	3.27
Regression Tree Mean	3.31
Regression Tree Gradient Descent	3.39
Artificial Neural Network	3.29

Table 4.7 Error comparison of Algorithms under ICU data

Table 4.7 shows a contrast of the performance of the algorithms with respect to this data set. Although KGERS performed better error-wise, this low error quantity came from a very unstable distribution. The probability of KGERS landing into such error is relatively small due to its instability. Out of all the algorithms the most reliable for this data set appears to be a Neural Network. Noise is perhaps too big and confuses general split in regression trees.

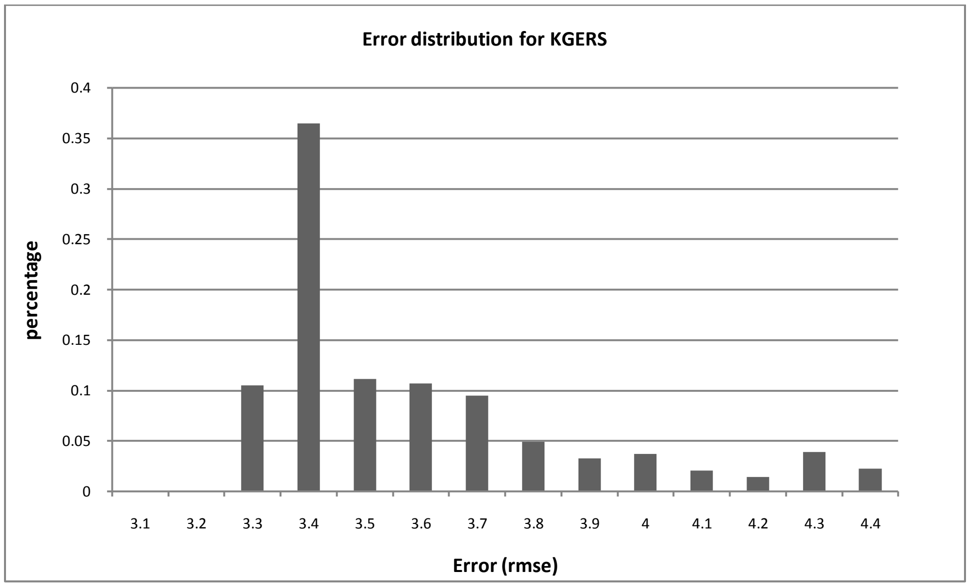


Figure 4.20.f Error Distribution for KGERS on ICU data

Figure 4.20.f shows that the probability of landing into a small error (3.3) is very small. This is reflecting that the Regression Tree KGERS tends to land in an error bucket of around 3.4.

4.7.3. Performance on PCU data set.

The next set of experiments will include results of the algorithms using the PCU data set. The first algorithm ran into this data set is show in Figure 4.21.a.

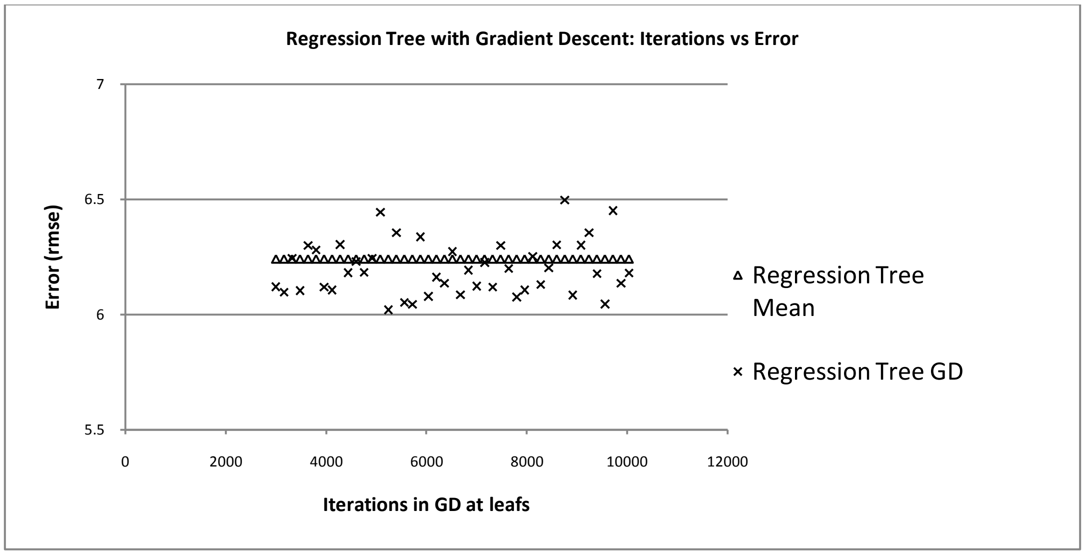


Figure 4.21.a Regression Tree Gradient Descent on PCU data

The performance of the Regression Tree Gradient Descent on the PCU data is not satisfactory. It seems that the tree is having trouble splitting and fitting gradient descent at the leaves. This can be seen by the fact that the algorithm seems very unstable, varying sometimes even more than the error of the Regression Tree Mean.

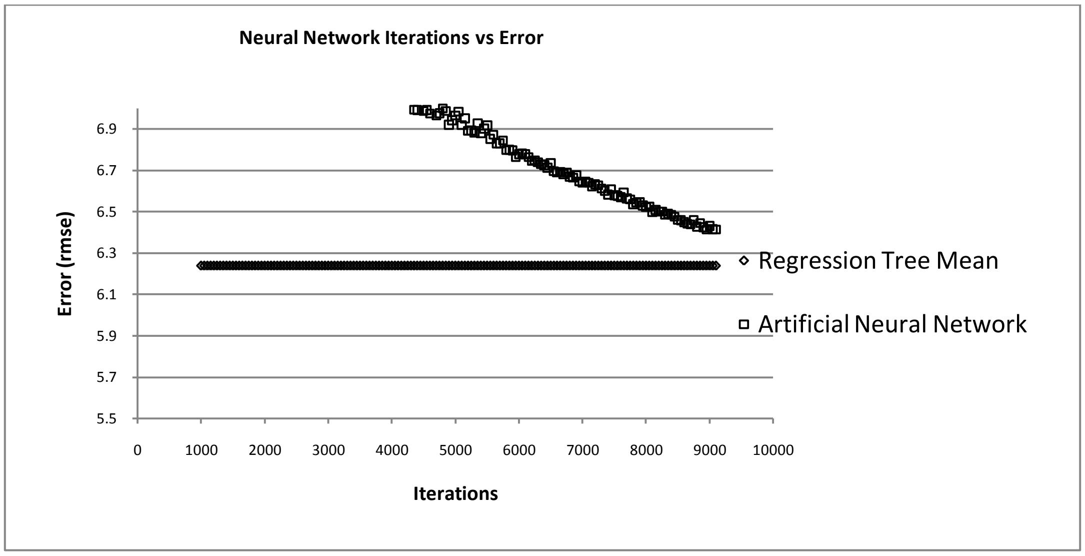


Figure 4.21.b Neural Network on PCU data

As typically seen, Neural Networks trained on back propagation tend to find their way into local minima. This can be seen in Figure 4.21.b where the Neural Network converges. Surprisingly the Regression Tree Mean seems to be doing a better job, with a difference of 0.1 in error. So much better that the neural network can't converge easily below this value. A possible explanation for this is noisy data. Having an average could be better in some sense, since its very similar to the idea of blurring an image. These properties can also show that the error in the data tends to have a Normal distribution, meaning that the Regression Tree Mean is capable of finding that measurement that can be in between this error.

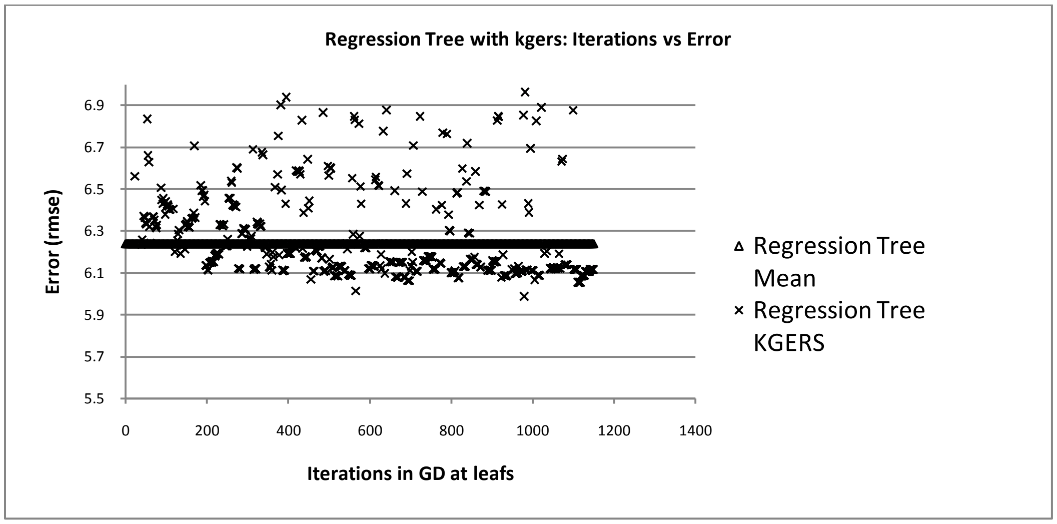


Figure 4.21.c Regression Tree KGERS on PCU data

Figure 4.21.c shows the performance of KGERS on top of the PCU data. This Regression Tree performs significantly better than the Regression Tree Gradient Descent. Interestingly, this algorithm shows more stability than gradient descent. A possible reason for this is the way splitting and stopping criterion is being used. Splitting is done through KGERS and stopping criteria is done strictly through t test on shape. The RMSE discovered by this algorithm shows a better settlement to that one of the previous two algorithms.

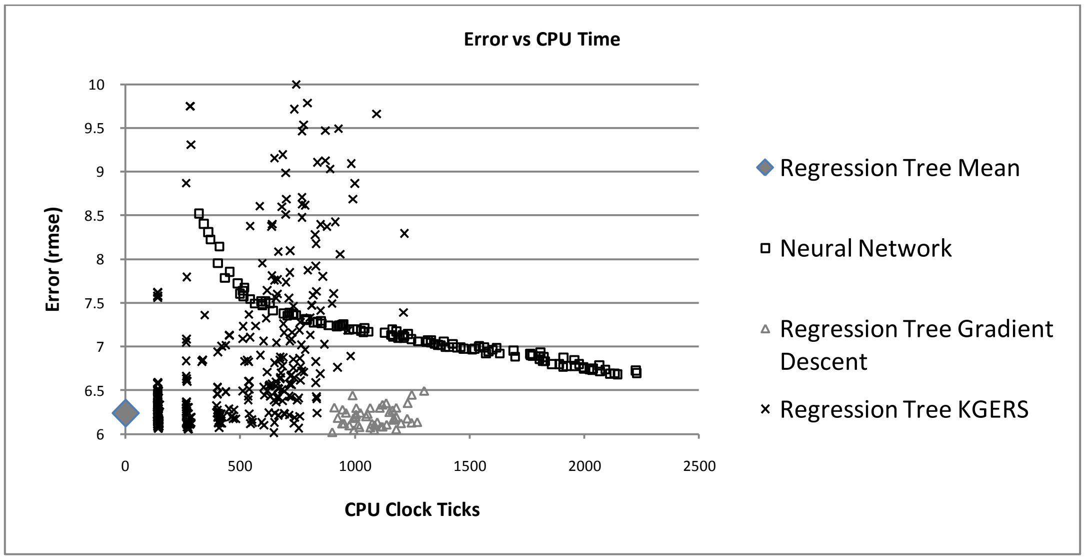


Figure 4.21.d Error vs CPU time analysis for algorithms under PCU data

Figure 4.21.d shows the average cpu power in clock ticks versus the error discovered by the particular algorithm. A closer view can be appreciated in Figure 4.21.e.

This closer view reveals that the error on PCU coming from the Regression Tree KGERS is low. Computational complexity is also the lowest amongst the other algorithms. However the error is very close to that one of the Regression Tree Mean.

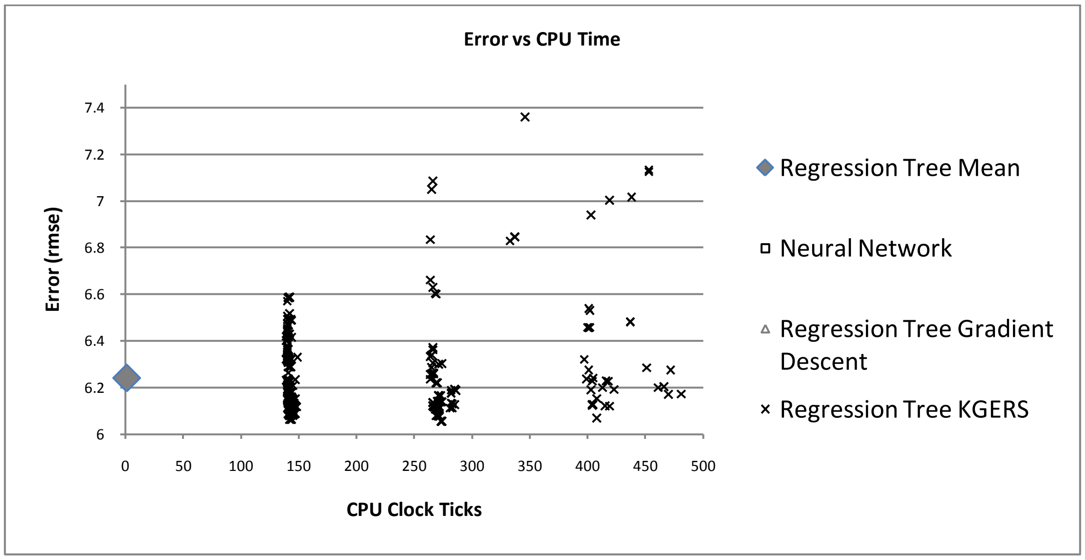


Figure 4.21.e Error vs CPU time analysis for algorithms under PCU data, ZOOM

Like previous results, Regression Tree Gradient Descent seems to be doing better using less CPU power. The neural network takes too long and doesn't converge to a really small error.

Algorithm	Best RMSE
Regression Tree KGERS	6.01
Regression Tree Mean	6.23
Regression Tree Gradient Descent	6.12
Artificial Neural Network	6.41

Table 4.8 Error comparison of Algorithms under PCU data

Table 4.8 shows a summary of the algorithms final performance on the PCU data. Notice that the PCU data has an average of 36.6 admissions in a 4 hour period. PCU beds are very common since they involve usually walk in patients. Therefore the demand of these is higher. The Regression Tree KGERS performed better in terms of error, and was able to retrieve the least error amongst the other algorithms. Figure 4.21.f shows the error distribution that the Regression Tree KGERS generated.

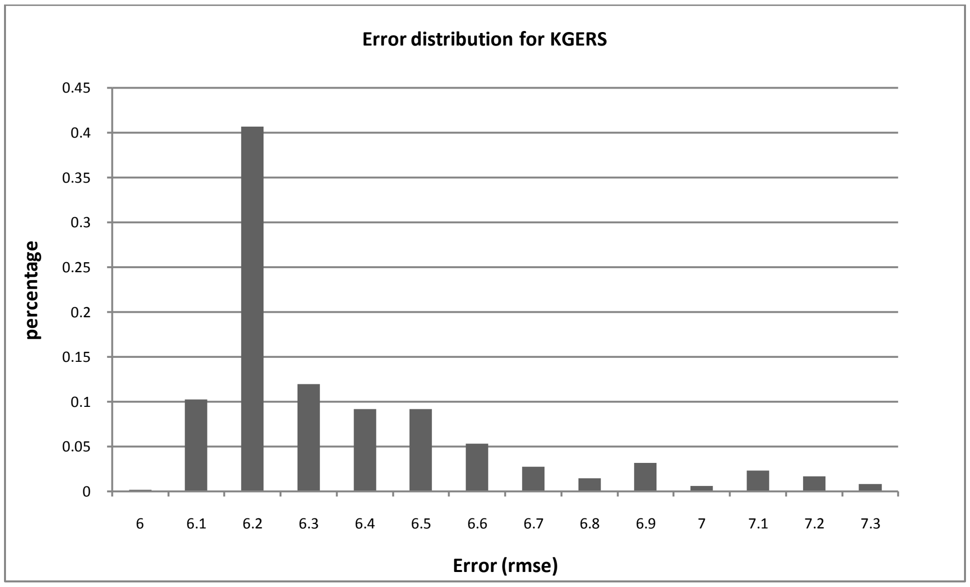


Figure 4.21.f Error Distribution for KGERS on PCU data

The error distribution shows a high probability of the error to be on the 6.2 bucket from the Regression Tree KGERS. This shows that this algorithm is very stable for this data set, and tends to find a proper answer.

4.7.4. Performance on Floor data set.

The final data set to be analyzed is the floor admissions. The next set of experiments will run the procedures on the Floor data set.

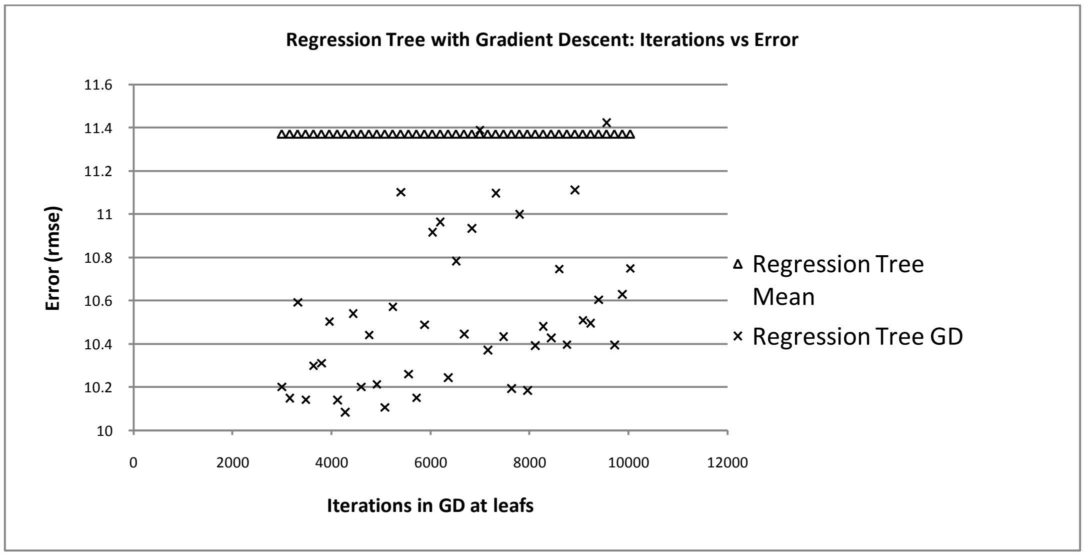


Figure 4.22.a Regression Tree Gradient Descent on Floor data

Figure 4.22.a shows the Regression Tree not converging into a specific RMSE error. This shows an untied relationship between iterations and level of precision that the tree possesses. Error is significantly lower than that one of Regression Mean.

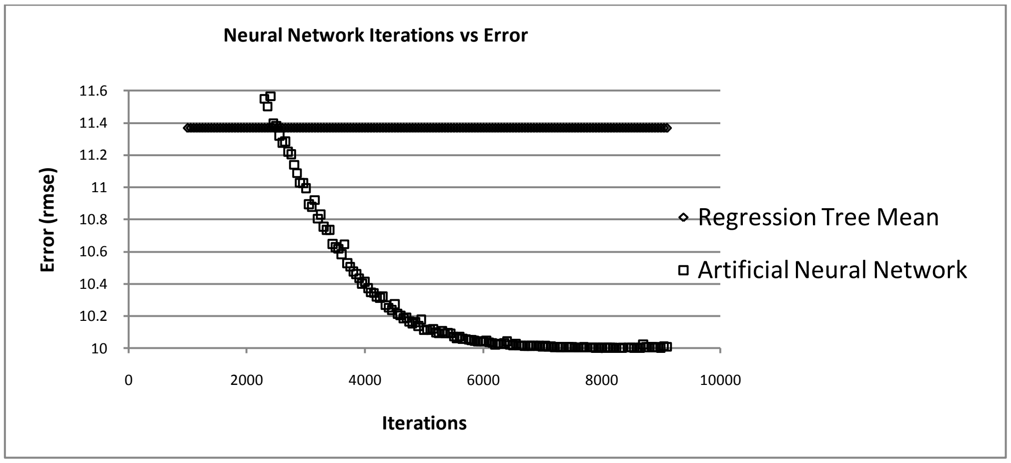


Figure 4.22.b Neural Network on Floor data

Figure 4.22.b shows that the Neural Network converges perfectly into a low error. The advantage of this is the stability shown by the algorithm, which is superior than that of the previous regression tree.

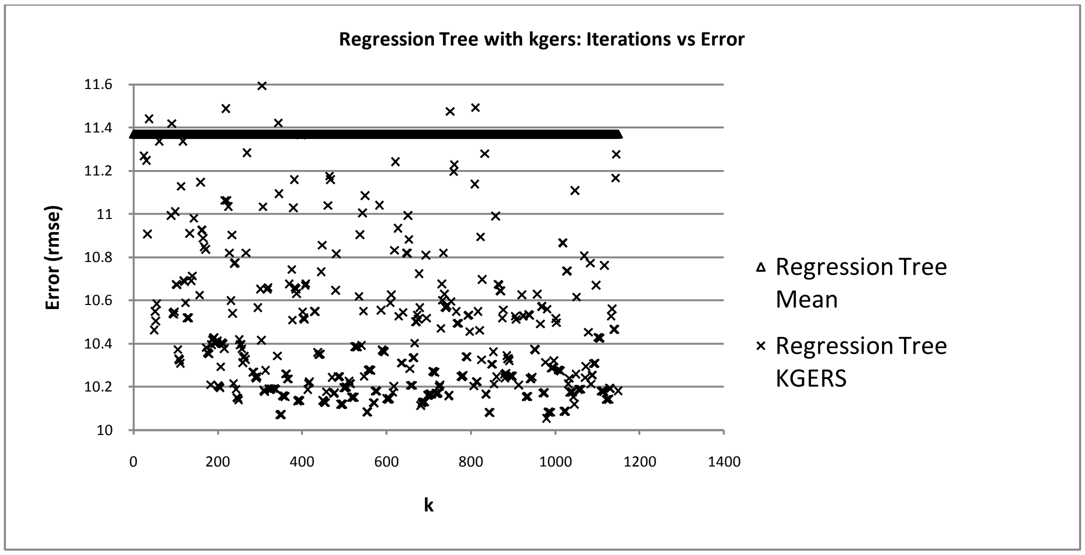


Figure 4.22.c Regression Tree KGERS leaves on Floor data

Figure 4.22.c shows the behavior that the Regression Tree KGERS presents on the FLOOR data. While not perfect the algorithm still presents a better level of stability than that of Regression Tree Gradient Descent. This figure also shows that while increasing K, the algorithm starts converging slowly.

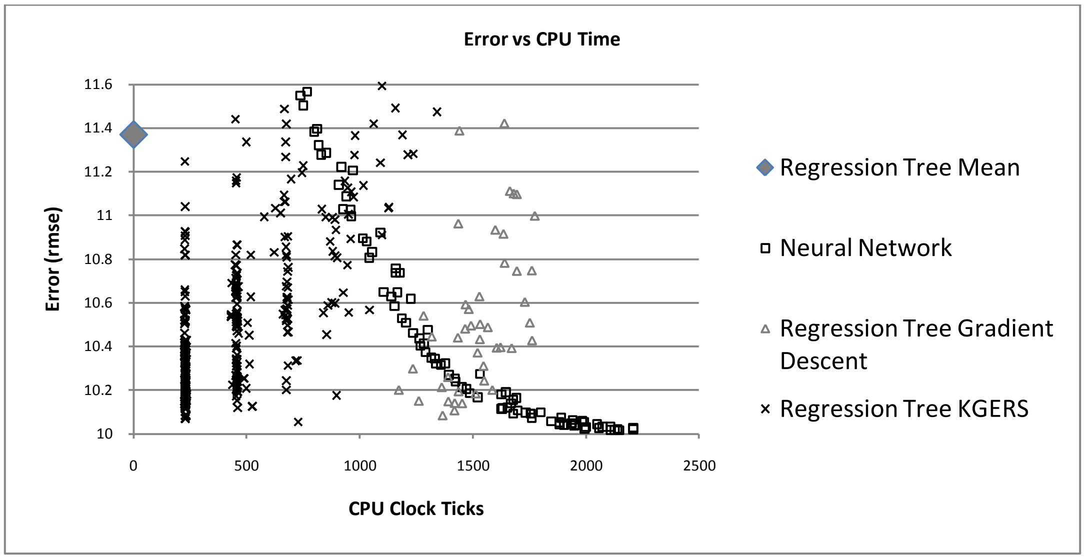


Figure 4.22.d Error vs CPU time analysis for algorithms under Floor data

Finally Figure 4.22.d shows the relationship between the algorithms and CPU time usage. As it can be appreciated the Regression Tree KGERS has the least cpu power and a low error. The neural network however seems to perform better. The Regression Tree Mean leaf nodes doesn’t perform very well, and is unable to give a low error. However, when zooming in into figure 4.22.e, it can be appreciated that the Regression Tree KGERS is still the one capable of achieving the lowest error in the least amount of time, even if this error difference is not significant.

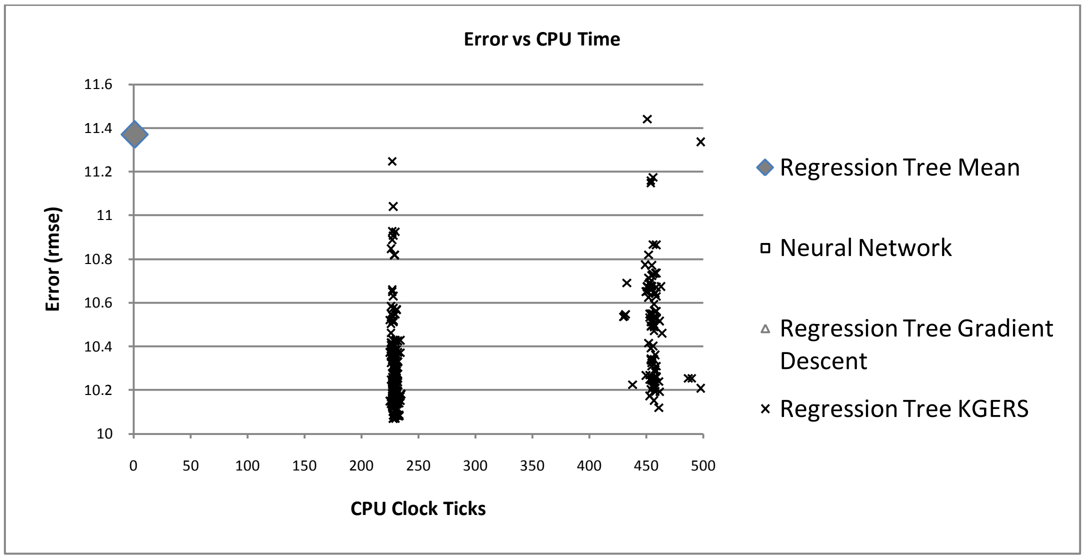


Figure 4.22.e Error vs CPU time analysis for algorithms under Floor data, ZOOM

Algorithm	Best RMSE
Regression Tree KGERS	10.08
Regression Tree Mean	11.37
Regression Tree Gradient Descent	10.14
Artificial Neural Network	10.08

Table 4.9 Error comparison of Algorithms under Floor data

From these algorithms, Regression Tree KGERS for leaf nodes and the Neural Network were able to get a minimum error. KGERS presents an unstable pattern and lower CPU consumption, in contrast to the other algorithms, but is still able to get into a low error. The Regression Tree Gradient Descent for the leaf nodes seems to do better than Regression Tree mean at the leaf nodes. The way noise is distributed in this data set shows the different behaviors of these algorithms. Another key factor for this is the mean and variance of the data, both are high thus providing more instability to the Regression Trees.

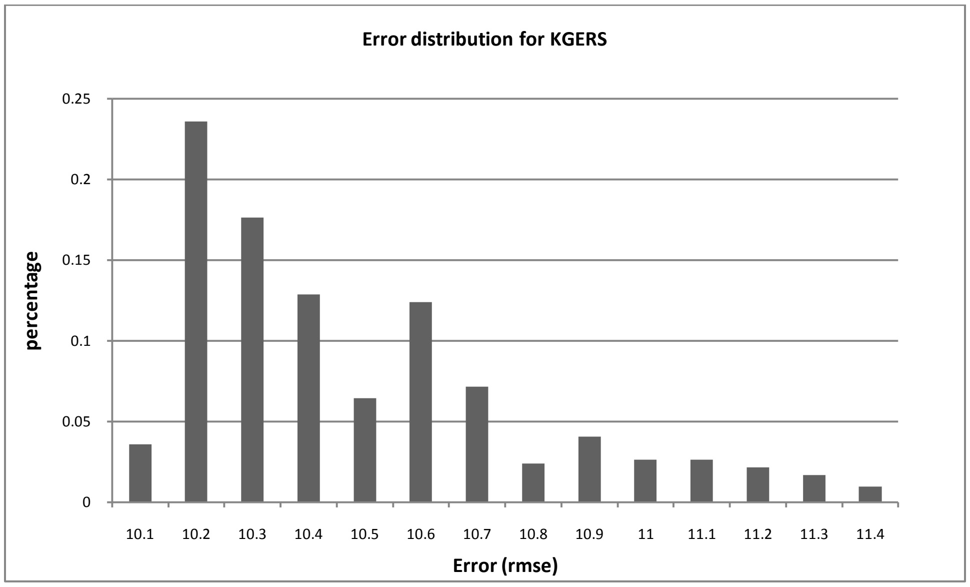


Figure 4.22.f Error Distribution for KGERS on Floor data

Figure 4.22.f shows the error distribution for the Regression Tree KGERS on the Floor data. It shows that this algorithm has high degree of instability due to the distribution portrayed here. While the algorithm is able to fall into the lowest possible error (which is an RMSE of 10.1) it has higher possibilities to fall under the 10.2 through 10.4 bins. This shows that the data might contain a lot of noise, therefore making it hard for the Decision Tree to achieve a low error.

4.8. Contributions to hospital

Previous algorithms show the result of the actual prediction executed in real hospital admissions data from Holmes Regional Medical Center in Melbourne Florida. During the following weeks of presenting these results to the hospital, staff and directors were pleased with the results and proposed to use the models in weekly bases. So far three weeks have been forecasted and they have been using the data successfully. As a start, only the EDInpt admission data has been processed for real application and staffing purposes. A summary of the results can be seen in Table 4.10.

Week time	RMSE	Mean Absolute Error	90% confidence interval	% error
Feb 25th – Mar 3rd	3.6	2.8	-6.2 to 5.2	26%
Mar 4th – Mar 10th	3.3	2.7	-6.2 to 5.2	24%
Mar 11th – Mar 17th	3.6	2.9	-6.0 to 5.2	26%
Mart 18th – Mar 24th	3.8	3.2	-6.0 to 5.2	28%
Mar 25th – Mar 31st	3.1	2.5	-6.0 to 5.2	23%
Apr 1st – Apr 7th	4.0	3.4	-6.0 to 5.2	30%
Apr 8th – Apr 14th	3.0	2.4	-6.0 to 5.2	19%

Table 4.10 Hospital runs using models with EDInpt data

This summary of results shows that for now errors tend to be high. Nevertheless, the hospital is mostly interested in the confidence interval calculated for such runs. Before this the hospital used to run simple regression techniques involving only looking at the exact value of that admission bucket one year ago. This provided noisy results with higher error than those of the models shown here. Holmes Regional Medical Center has future plans with these models and potentially put them in operation.

4.9. Summary

As seen before, previous algorithms show a general good performance in the hospital data. Noisy data such as ICU present a difficulty for randomized algorithms such as KGERS. Nevertheless, the Regression Tree KGERS at the leaf nodes showed more accuracy, especially in PCU and EDInpt data. The Regression Tree KGERS also is able to get the least RMSE in the FLOOR and ICU data. The only problem with these is the instability of the algorithm, showing a lot of spikes. For these previous ones, the Neural Network would be a better fit, since it's more stable and able to perform as close. The hospital is more interested in the absolute error and the confidence intervals. To their terms, these are very satisfactory and could meet the necessary qualities to make the models operational, and help with the staffing problem.

Chapter 5

Empirical Evaluation with Synthetic Datasets

This chapter will cover the experimental procedures that evaluate and compare both, linear and non linear regression algorithms on the synthetic data set. As seen before, each algorithm presents advantages and disadvantages. In other words, there can't be an algorithm that follows the "one size fits them all" rule. To discover the advantages and disadvantages of the algorithms used, it will be required to evaluate carefully and assess each one of them with a respective data set. The synthetic data set was prefabricated to test certain aspects of these machine learning algorithms such as accuracy under noise and over fitting. While these aspects might not be present in the hospital data set, they serve as a guide to see how the algorithms would behave against other types of noise. This data set will be referred as synthetic data set. As mentioned before, the objective of this chapter is to evaluate the potential contributions that the newly devised *algorithms* have on time complexity, accuracy of prediction and over fitting in other algorithms that show different characteristics from the hospital data set. The following experiments will show the robustness of the new algorithms with respect to a data set that degrades in quality.

5.1. Data Sets Overview

There are 6 data sets used to evaluate the performance of the algorithms. These data sets can be divided in two groups. These six synthetic data sets contain two separate groups. Because the experiments are divided into two sub sections (nonlinear and linear regression), there exists to flavors of this synthetic data set group. One is on top of a multi dimensional linear function; the second one is in a non linear function. Later on, the purpose and application of this data set will be explained.

It must be noted, that there will be a training set and validation set for each data set.

- *Training Set:* The set of vectors used as training data. The model will be generated from these set of points.
- *Test Set:* The set of vectors used as evaluation criteria. The test set is never seen by the algorithm and doesn't take any part from the model. The model's task is to use the test set as verification on its accuracy and fitness.

The two previous points are standard protocols of machine learning that are necessary for evaluation and the scientific process. As argued in earlier chapters, when evaluating algorithms, it is always important to test several aspects that might jeopardize the performance of a machine learning algorithms. Such criteria are things such as over fitting and accuracy under noise. We will see later in the experimental section that there is the evaluation of two types of algorithms: linear and non linear regression algorithms. Although different, the Synthetic data sets contain the same variations. The variation list is summarized in Table 5.1, in conjunction of a brief explanation of the purpose.

Synthetic data set variation	Description	Motivation
No error	Data contains no error in training set and test set. Derived strictly from a custom function.	None, used for control.
Low Error	Data strictly derived from a custom function, with 1% of noise. Noise follows a normal distribution.	Accuracy under noise
Medium Error	Data strictly derived from a custom function, with 13% of noise. Noise follows a normal distribution.	Accuracy under noise
High Error	Data strictly derived from a custom function, with 30% of noise. Noise follows a normal distribution.	Accuracy under noise
High Error on train, no error on test	Data strictly derived from a custom function, with 30% of noise. Noise follows a normal distribution. Noise only in training set, not in test set.	Accuracy under noise and over fitting
One irrelevant feature	One of the features is completely irrelevant to the target value.	Accuracy under noise and over fitting
1/3 of irrelevant features	One third of the features are completely irrelevant to the target value.	Accuracy under noise and over fitting

Table 5.1 Variations of SYNTHETIC data sets.

The variation of Synthetic data sets helps comparing the two main weaknesses present in most of machine learning algorithms. It helps not only to show their strength and precision, but also under what circumstances are they more feasible to use. This is a good contribution and could certainly be applied to other fields and applications.

The synthetic data set for the linear features consists of a simple hyper plane of 16 dimensions. It can be summarized in equation (1).

$$y = \sum_{i=1}^{14} w_i x_i + w_0 + \varepsilon \qquad (1)$$

The weights chosen are described in Table 5.2. Here the algorithms task is to figure out the weights of the hyperplane. Later on we will see the accuracy of these algorithms and explanations of why some of them take longer to converge. The contents of the non linear Synthetic data set is a simple sine function, it is meant to provide a simple base, in which all non linear regression algorithms can converge and represent. With a base like this, it will be easier to see the reactions and potential disadvantages derived from noise and over fitting.

Weight	Value
W_0	8
W_1	.25
W_2	23
W_3	-7
W_4	0.25
W_5	6
W_6	10
W_7	40
W_8	30
W_9	2
W_{10}	7
W_{11}	89
W_{12}	10
W_{13}	2
W_{14}	8

Table 5.2 List of weights used by linear function.

The exact function used for non linear regression is described in equation (2).

$$y = \alpha_0 + \alpha_1 \sin(x) + \varepsilon \qquad (2)$$

The respective weights for equation (2) are described in Table 5.3.

Weight	Value
α_0	2.0
α_1	7.0

Table 5.3 List of weights used by non linear function.

Notice that these two functions have a ε symbol at the end, that is, equations (1) and (2). It happens that this is going to be the noise added during feature alteration described in Table 5.1. This noise will have a normal distribution, and depending on the intensity the mean and variance parameters are adjusted. A normal distribution is defined in equation (3).

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad (3)$$

This is a well known probability of density function, where x is the desired noise, and the output is the probability that it would occur. The two parameters are σ^2 which represents the variance of the noise distribution and μ the mean. As mentioned before, noise percentage contains three levels. These are displayed in Table 5.1, and these levels will be the desired mean used for that particular level of noise. The variance remains constant, is represented by a 20% of the error mean. This helps in exploring a broad range of the total noise spectrum.

5.2. Experiment procedures

There exist two sets of the experiments. The first set will evaluate the performance on linear functions. The second set will focus on the non linear algorithms. The reason why there are two groups of experiments is because there is a level of granularity for piece wise linear regression. The ultimate goal is to propose a regression tree that performs piece wise linear regression in the most optimal way, targeted towards hospital data. In order to have this, linear regression must be explored at its lower level. The algorithms proposed for the leaf nodes of the regression tree must be compared to those already existent. This is done to demonstrate that the advantages behind the motivations for KGERS are empirically valid. For the second group of experiments, the proposed regression tree will now be tested using the newly devised linear regression algorithms. This comparison is done against other methods of regression. The algorithms chosen for the linear regression experiments are described in Table 5.4.

Algorithm	Description
KGERS	Algorithm proposed. Aims to provide same and closer results to those of Least Squares. Empirical evidence will have to proof that its time complexity is lower, and that it can sometimes achieve global optimal.
Gradient Descent	Most common linear regression method. This algorithm will be compared to KGERS and empirical evidence should contrast advantages and disadvantages.
Least Squares	Algorithm that returns global optimal. The problem with this algorithm is that it might not be decidable, as in one of the matrixes might not be invertible, therefore unable to find a solution. It is used as a benchmark on accuracy on the regression.

Table 5.4 algorithms for linear regression experiments

The second experimental part contains the same procedures mentioned in chapter 4, section 4.2. This will be on the non linear regression algorithms on the second set. The algorithms used are summarized in table 4.3 of chapter 4.

5.3. Linear Regression Experiment results

Each run of KGERS is the average of 5 runs itself. Same process applies to the CPU usage of KGERS. Since KGERS is considered a randomized algorithm, an average is always required to know the correct inclinations under different parameters. Given this overview, the subsections here are going to deal with the experimental results directly.

5.3.1. Computation time

Figure 5.1 shows the comparison of the three algorithms in terms of N. Notice that the scale of the Y axis is logarithmic. The X axis is the size of the training of the Synthetic data with no error associated with it. The Y axis represents the total CPU time spent by each algorithm. The CPU time is measured in clock cycles, meaning that each cycle is a burst of instructions being executed. All the experiments in this study were run on the same machine, a dual core Pentium. The machine was not running any extra CPU intensive software and was hardened to the most. It is important to know that still the operating system has other tasks in between that could be the source of the spikes. In general, the tendency is likely remain constant and this kind of noise can be ignored. As expected, by the theoretical complexity analysis done in previous sections, Gradient descent grows linear, while KGERS and Least Squares remain constant. Equation (5) reviews Gradient Descent's time complexity.

$$O(N^2M) \tag{5}$$

We have to remember that while equation (5) shows a quadratic growth of N, in the experiments the iterations parameter remains constant. This means that the complexity of Gradient Descent in the experimental data is O(N) instead of O(N^2). Figure 5.1.a shows this behavior. Dimensionality also remains constant so that's why the mentioned complexity remains the same.

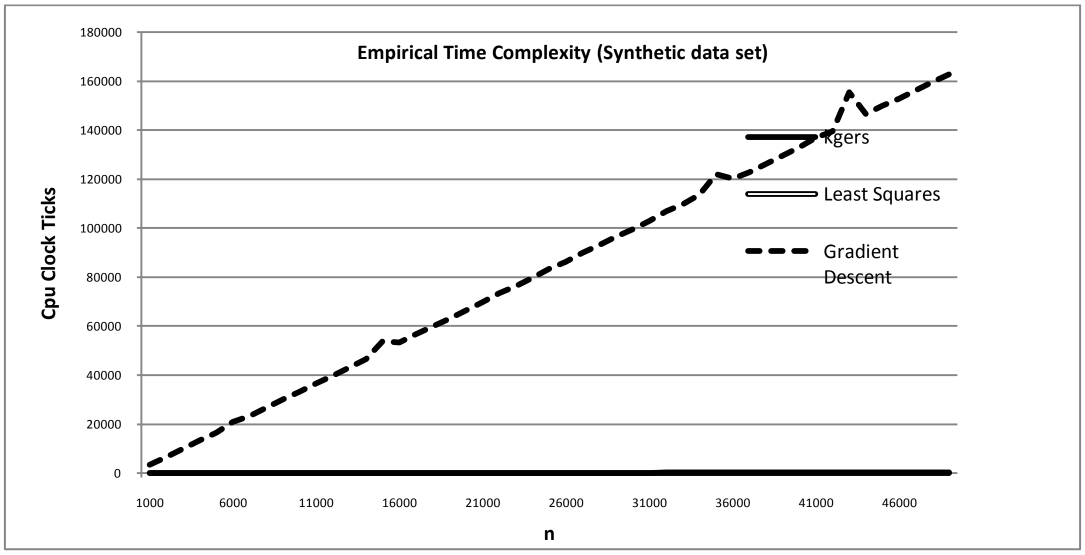


Figure 5.1.a Linear regression algorithm speed analysis

A closer look in Figure 5.1.b. reveals the gradual increase of KGERS and Least squares. It has to be noted that the reason why KGERS shows an increase in clock ticks is because the algorithm still needs to parse and process the data. The more data, the more time this overhead takes. This overhead is already included in the other algorithms.

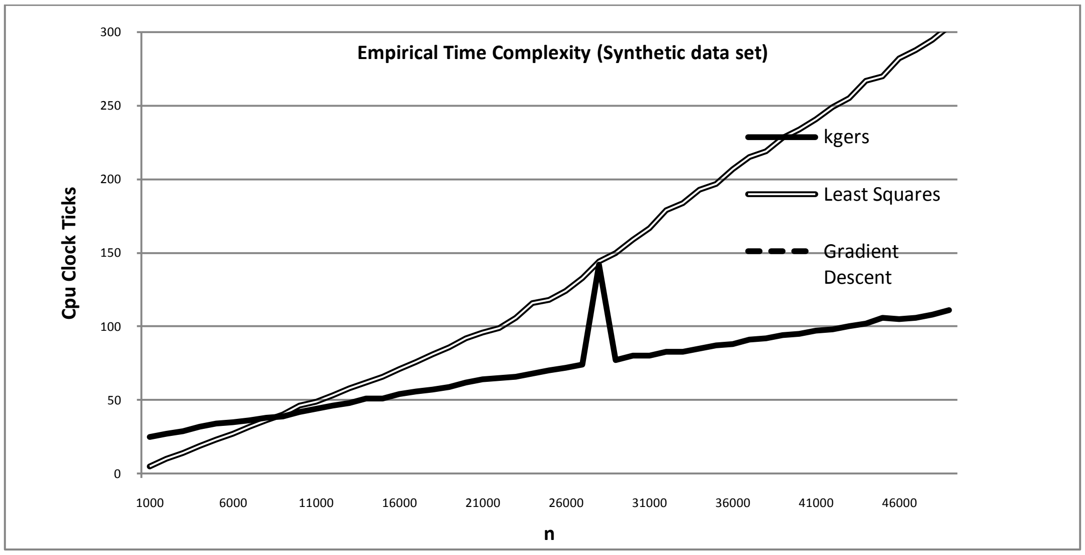


Figure 5.1.b Linear regression algorithm speed analysis zoom

As expected, the time complexity for Gradient Descent grows much faster than that of KGERS and Linear Least Squares. Interestingly, Linear Least Squares also shows trends of linear increase with respect of the size of N. The only difference is that this growth is much smaller. Also, Linear Least Squares contains a matrix multiplication operation, which causes a very subtle increase with respect to N. After 11000 examples used Least Squares CPU clock tick usage passes that one of KGERS. KGERS remains almost at a constant speed, reason being that KGERS is independent of N. If KGERS wants to aim for accuracy, it will have to fix the K parameter depending on how linear the data tends to be.

5.3.2. Performance on noisy data

As mentioned before, this experiment will try to explain the performance of algorithms and behavior based on varying noise. There are three levels of noise that have been recorded. Before comparing the levels of noise though, a control experiment is needed.

5.3.2.1. No Noise

Figures 5.2.1.a, 5.2.1.b and 5.2.1.c. show that the algorithms can ultimately converge and perfectly perform regression on a line data. KGERS doesn't show any sign of error, gradient descent converges very fast and linear least squares can achieve an error of 0.

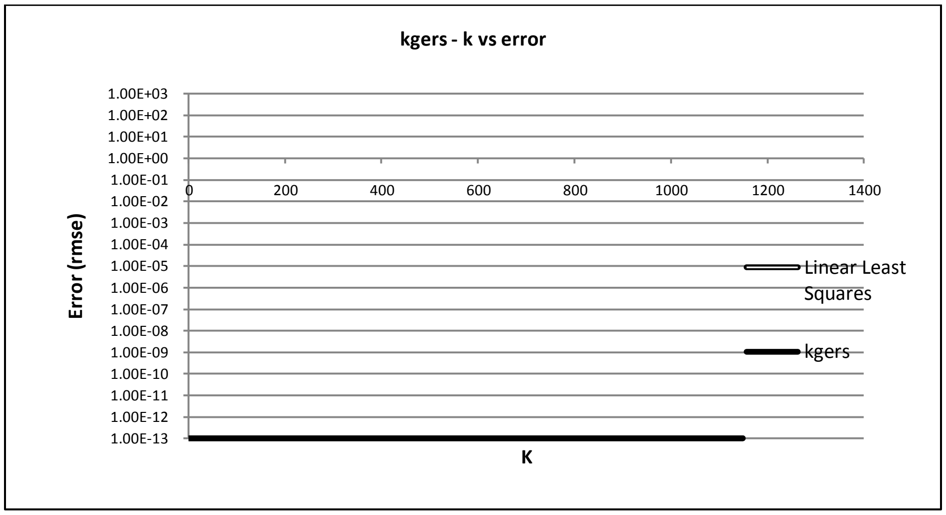


Figure 5.2.1.a Kgers on Synthetic data, no noise.

As seen in Figure 5.2.1.a, KGERS can easily achieve a perfect regression if there is no noise. This is because KGERS extracts chunks of the data and tries to fit a line. If the data contains no error, this means that any m+1 points (where m is the dimensionality) will generate a perfect hyperplane.

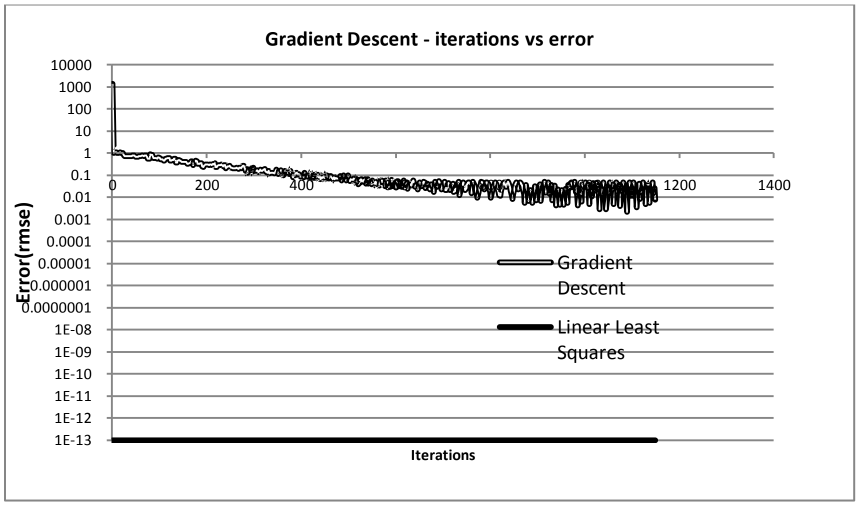


Figure 5.2.1.b Gradient Descent on Synthetic data, no noise.

Gradient Descent requires some CPU power to converge. Because it keeps updating weights step by step, the error will be directly dictated by how many iterations the algorithm experiences. The more iterations, the closer it gets to the answer. Notice how it converges since the scale in the Y axis is logarithmic. After several iterations the error stays the same. Interestingly, the algorithm gets really close to the answer but not quite, unlike KGERS and linear least squares, which immediately get to the answer.

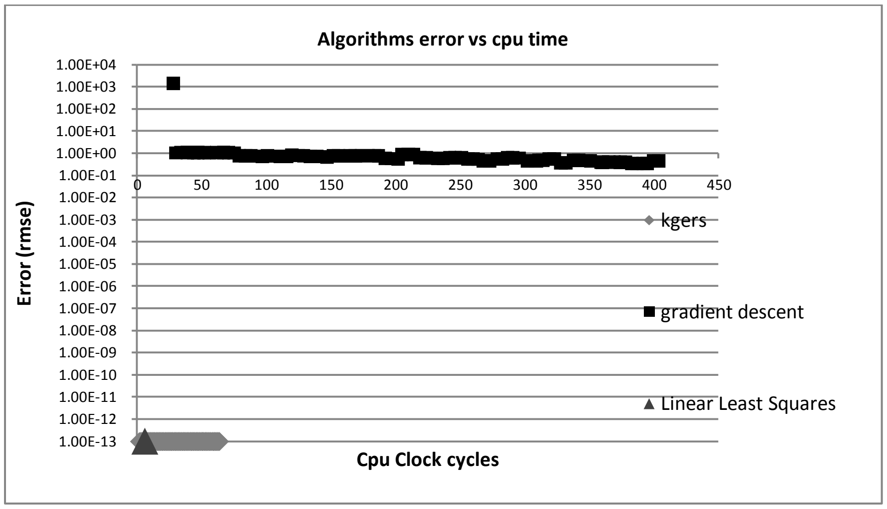


Figure 5.2.1.c Algorithm comparison on synthetic data without noise.

Notice how KGERS and Linear Least Squares perform much better with the data. They show straight convergence and a good answer.

5.3.2.2. Low Noise

Figures 5.2.2.a, 5.2.2.b and 5.2.2.c show the performance of the algorithms with low noise.

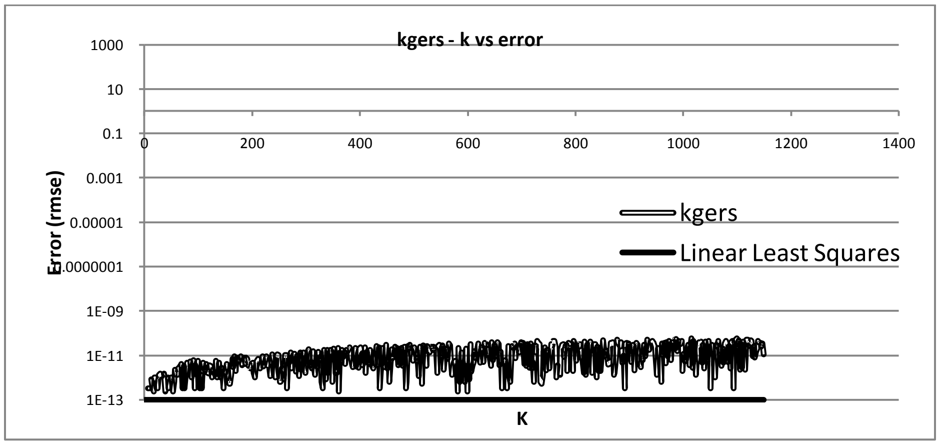


Figure 5.2.2.a Kgers on Synthetic data, low noise.

Figure 5.2.2.a show that KGERS gains some inaccuracy with a little noise, although this quantity is very small, we are talking about exponent to the -12. This is surely a tiny number, and can be thought as if KGERS actually converges to 0 in error.

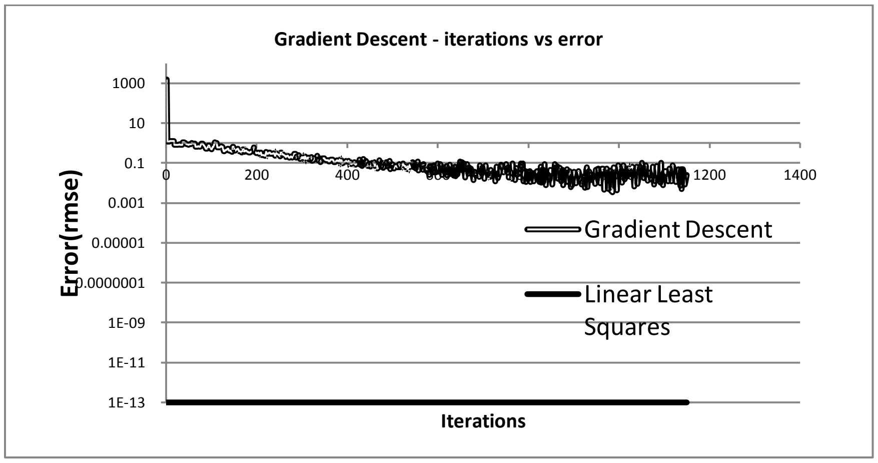


Figure 5.2.2.b Gradient Descent on Synthetic data, low noise.

Gradient descent converges into not the global optimal, which is dictated by Linear Least squares in Figure 5.2.2.b. Although the noise and error it gets is very small, is considerably bigger than that of KGERS.

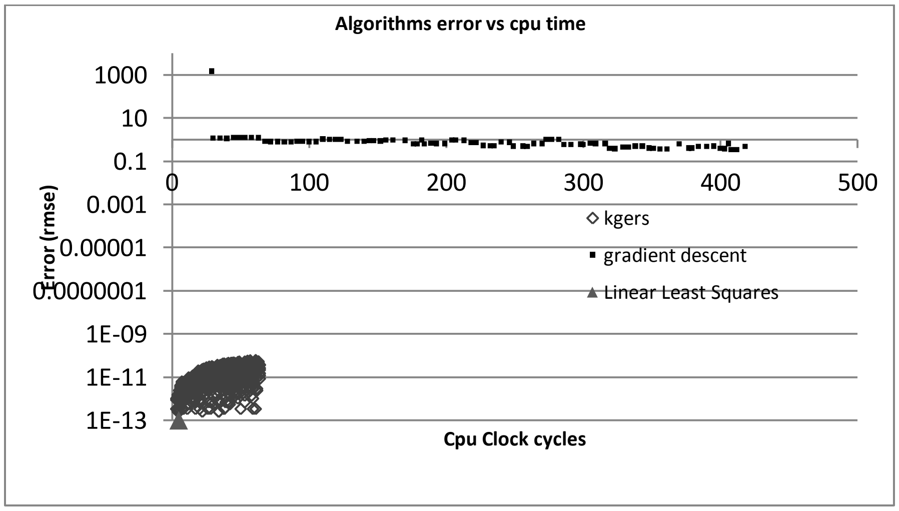


Figure 5.2.2.c Algorithm comparison on synthetic data, low noise

The CPU Time analysis in Figure 5.2.2.c shows that KGERS is slightly slower than Linear Least Squares, although not significantly. It does show that with little noise, KGERS is capable of finding a better answer than that of Gradient Descent.

5.3.2.3. Medium Noise

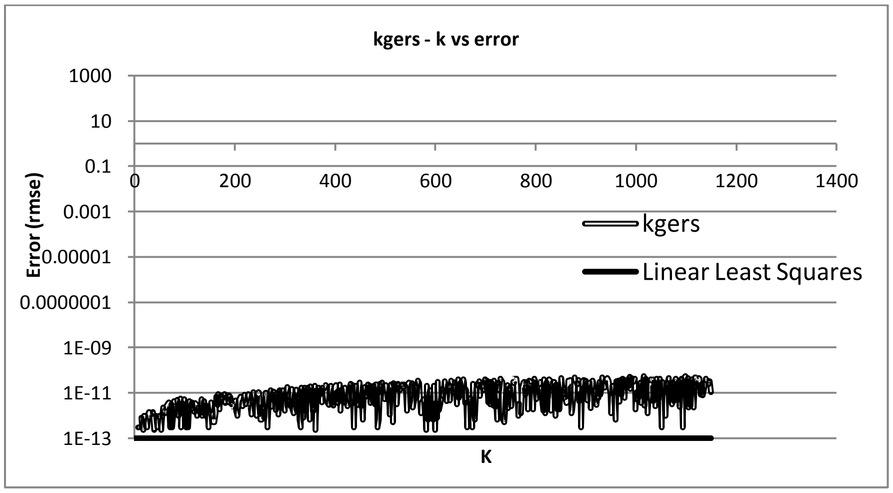


Figure 5.2.3.a KGERS on Synthetic data, medium noise.

Figure 5.2.3.a shows a very similar pattern for KGERS. With medium level noise it is capable to converge into small values. This shows that small amounts of noise do not affect KGERS in a big manner. The slight upward convergence of KGERS shown in this graph displays that KGERS's inaccuracy is growing linearly with the value of parameter K. This means that in some way, with small noise, KGERS could over fit. This however can be discarded since the RMSE reported by KGERS is very small, close to 0.

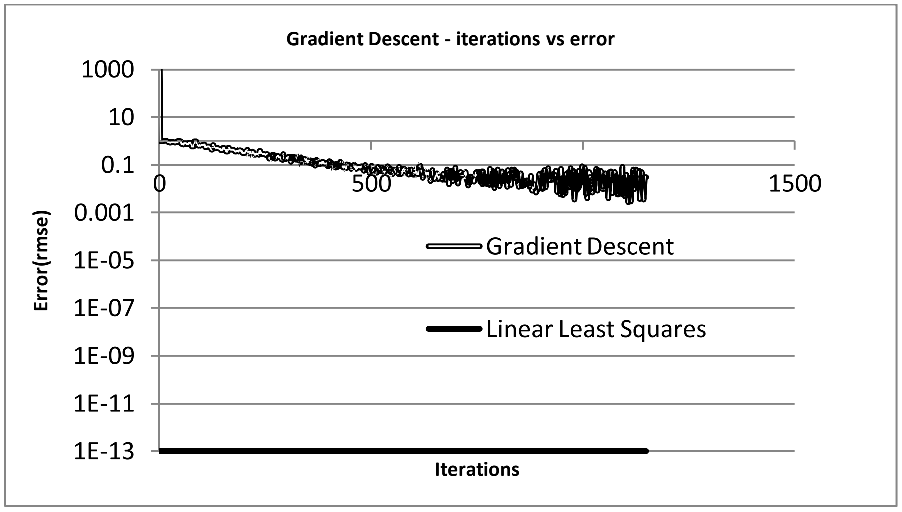


Figure 5.2.3.b Gradient Descent on Synthetic data, medium noise.

Gradient Descent shows a very similar convergence to a local minimum than that of low noise data. Figure 5.2.3.b shows this, and it seems that Gradient Descent cannot find a lower error than that found by KGERS.

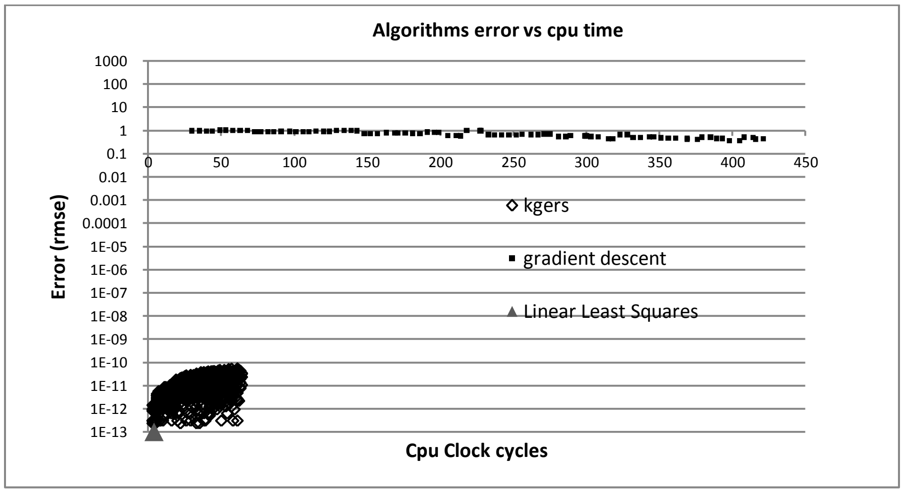


Figure 5.2.3.c Algorithm comparison on synthetic data, medium noise

The CPU analysis displayed in Figure 5.2.3.c shows that KGERS is still close to the global optimal found by Linear Least Squares. Gradient Descent seems much slower and converges into a larger error for medium noise. These results are very similar to those mentioned in Figure 5.2.2.c, which is CPU speed analysis of low noise data.

5.3.2.4. High Noise

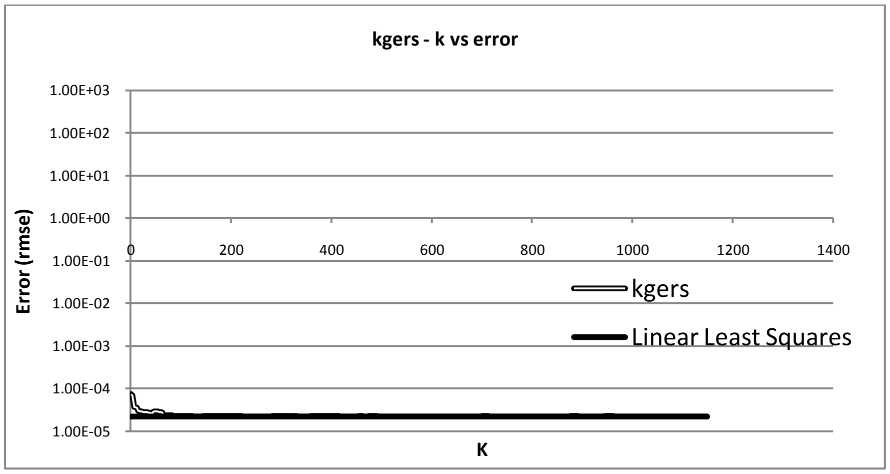


Figure 5.2.4.a KGERS on Synthetic data, high noise.

High noise data imposes a new error threshold, much larger than those displayed in the Figures 5.2.2.x. Figure 5.2.4.a shows that KGERS is able to stick up with the global minimum imposed by Linear Least Squares, even in large amounts of errors.

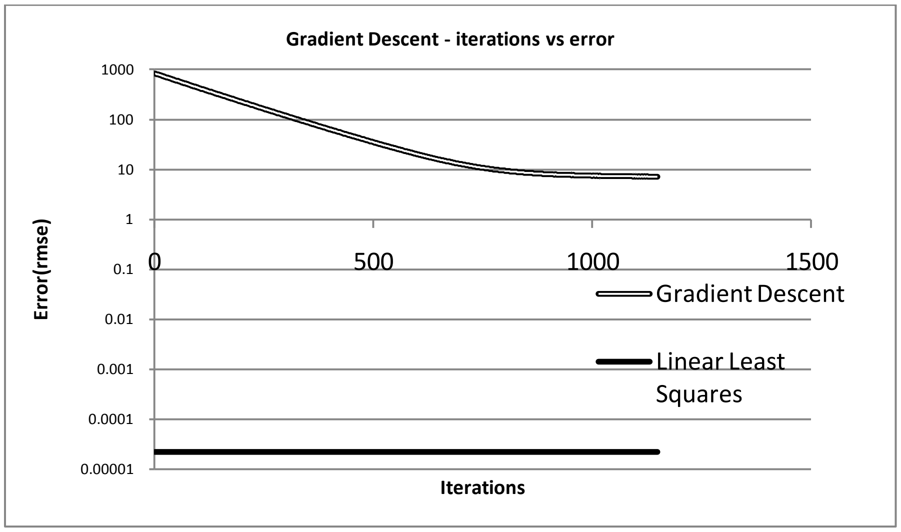


Figure 5.2.4.b Gradient Descent on Synthetic data, high noise.

Figure 5.2.4.b shows that Gradient Descent is converging to a much larger error, this error which rounds around 18%. As seen here, Gradient Descent is again very far from the global optimal error imposed by Linear Least Squares.

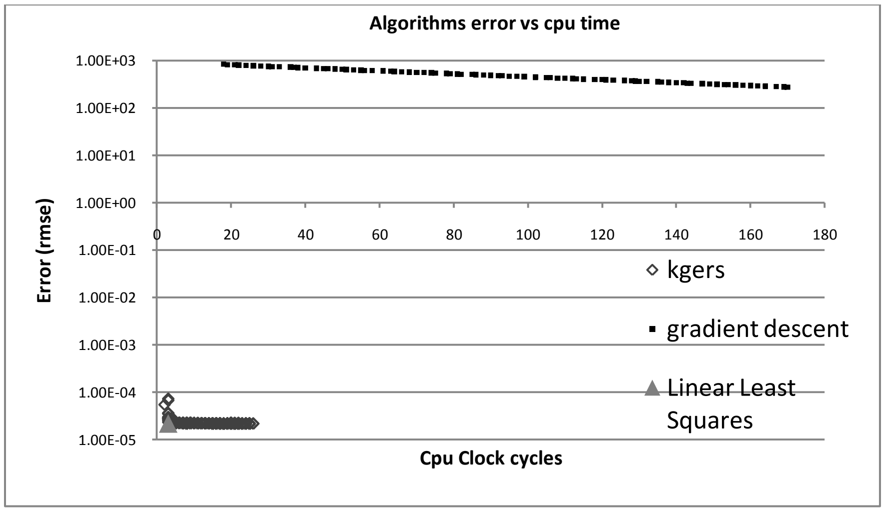


Figure 5.2.4.c Algorithm comparison on synthetic data, high noise

High noise error on the synthetic data shows that KGERS is very well accurate, by preserving at rmse almost constant at 1e-12. Figure 5.2.4.c. shows that KGERS is able to resist this noise, and completely perform as well as Linear Least Squares. On top of this, unlike Linear Least Squares, KGERS is guaranteed to return an answer (as long as there are enough points).

5.3.3. Performance on Over fitting

Previous experiments show how KGERS is able to successfully resist noise and maintain itself at a good RMSE rate. Even with high noise, we saw how Gradient Descent scales up in its RMSE, while KGERS remains constant and true to the global optimal. The next set of experiments performed is meant to test over fitting and performance analysis using the linear synthetic data.

5.3.3.1. Noise Only on Test Set

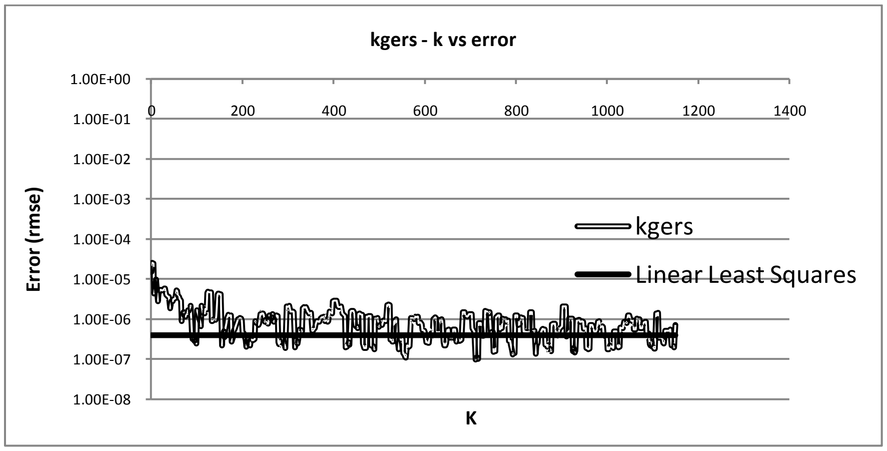


Figure 5.3.1.a KGERS on synthetic data, Noise only on Test Set

Figure 5.3.1.a shows KGERS converging to an RMSE of 12e-7, and not over fitting. The noise level in the training sets is very high, around 30%. This displays that KGERS can even perform better than Linear Least Squares, almost half of the time.

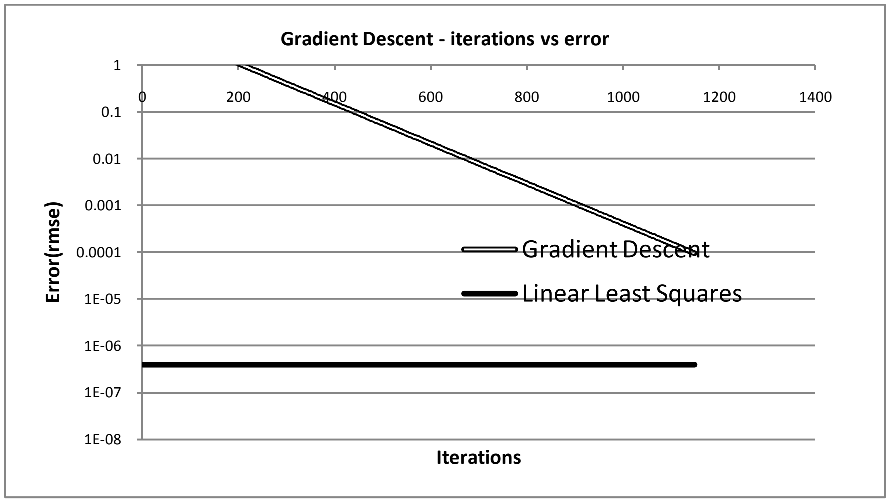


Figure 5.3.1.b Gradient Descent on synthetic data, Noise only on Test Set

Figure 5.3.1.b shows that gradient descent converges (notice that the scale in the Y axis is logarithmic). This means that gradient descent has found a local minima to fall into, and will not reach the global optimal proposed by Linear Least Squares.

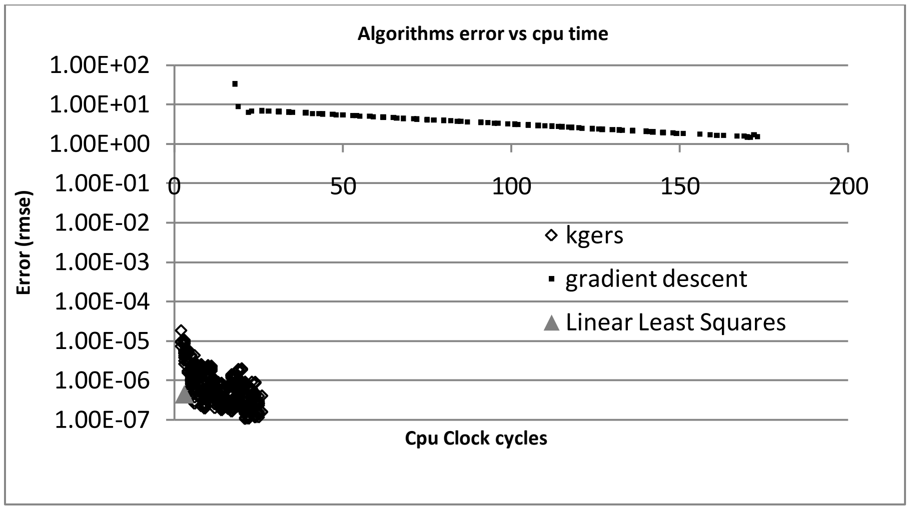


Figure 5.3.1.c Algorithm comparison on synthetic data, high noise only on training data

An overview of the CPU performance of these algorithms on top of the noisy training data is shown in Figure 5.3.1.c. It can be appreciated that KGERS can sometimes surpass the error discovered by Linear Least squares.

5.3.3.2. One Irrelevant Feature

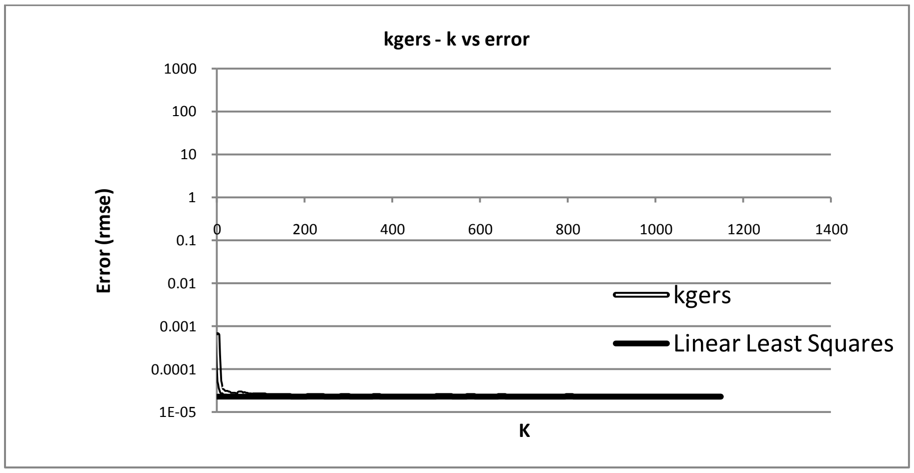


Figure 5.3.2.a KGERS on synthetic data, one irrelevant feature

Figure 5.3.2.a shows the behavior of KGERS under synthetic data with one irrelevant feature. In the case of this linear regression, one of the weights is unused. The algorithm should be able to set this weight to 0 in the hyperplane, since it is irrelevant. KGERS shows a quick convergence to the global optimal.

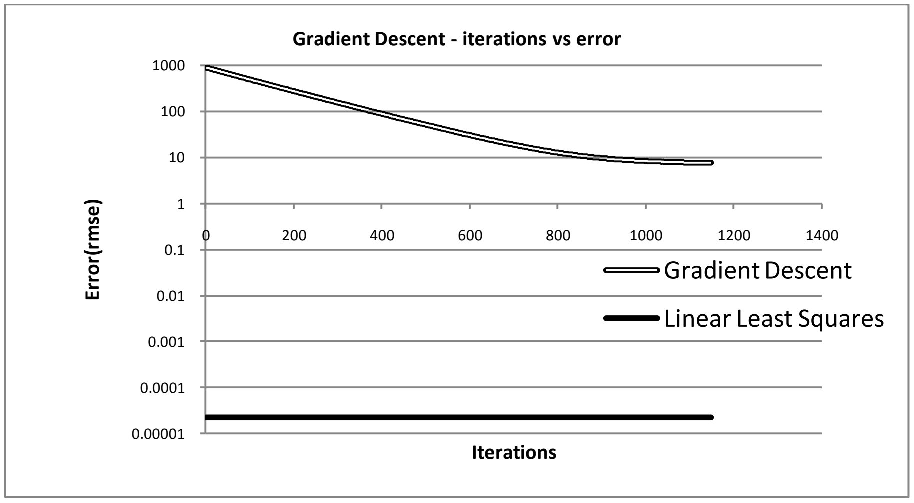


Figure 5.3.2.b Gradient Descent on synthetic data, one irrelevant feature

Again, in Figure 5.3.2.b, gradient descent shows similar patterns. These display the algorithm converging to a high error, being very sensible with only 1 irrelevant attribute.

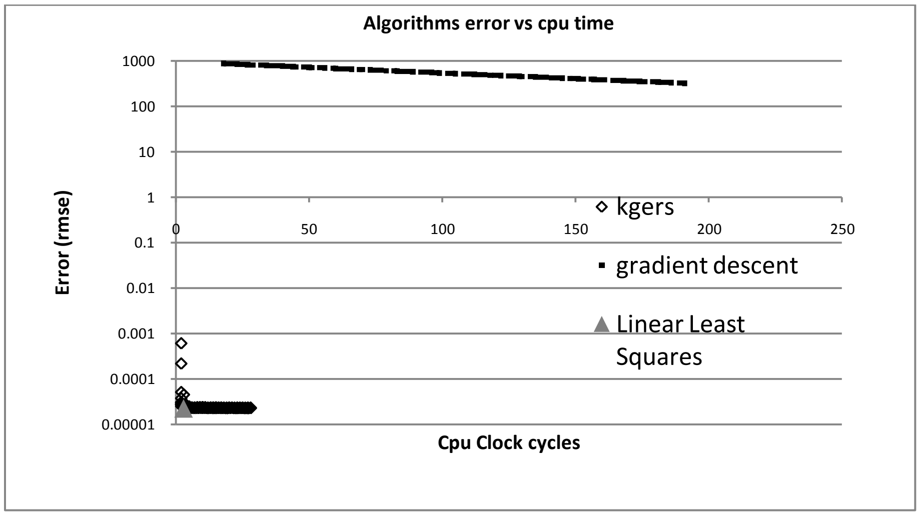


Figure 5.3.2.c Algorithm comparison on synthetic data, one irrelevant feature.

Finally the CPU performance for this data set reveals that KGERS can match and stay at the same level of Linear Least Squares. Reasoning behind this is the random nature of KGERS, which allows for the algorithm to punish those sets with high error with respect to the rest of the data.

5.3.3.3. 1/3 Irrelevant Features

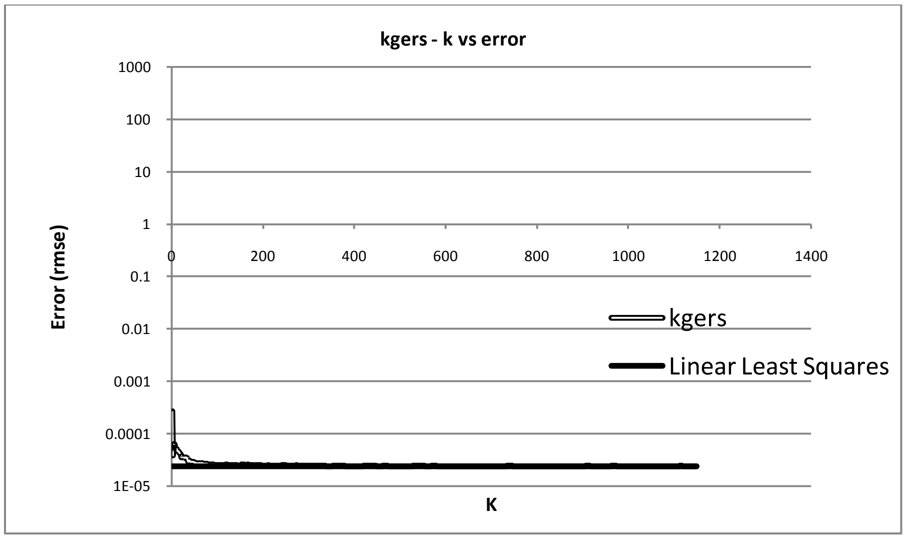


Figure 5.3.3.a KGERS on synthetic data, 1/3 irrelevant features

Figure 5.3.3.a shows that KGERS resists to the extra noise against one third of the weights being irrelevant. Also it shows that KGERS manages to do even better than Linear Least Squares in a couple of points. This means that KGERS is not as greedy as other algorithms, and can actually explore areas of the solution space that any other non-randomized algorithm wouldn't.

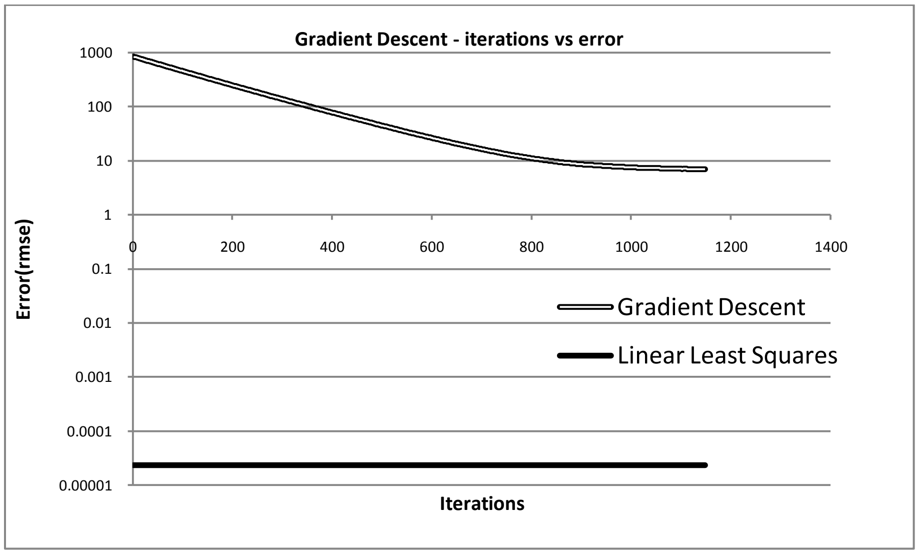


Figure 5.3.3.b Gradient Descent on synthetic data, 1/3 irrelevant features

Gradient Descent is showing worst results as noise and more irrelevant features are added to the synthetic data pool. Figure 5.3.3.b. shows this. Although smooth convergent, the curve that Gradient Descent displays is incapable of hitting a very low error, unlike Linear Least Squares.

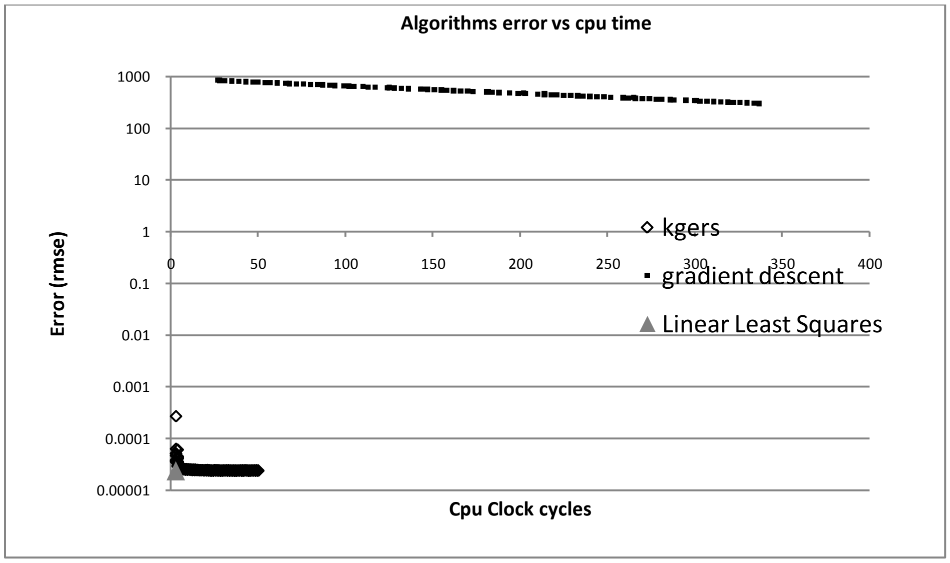


Figure 5.3.3.c Algorithm comparison on synthetic data, 1/3 irrelevant features.

The final analysis of CPU time and error for each algorithm, shown in Figure 5.3.3.c, demonstrates that the initial speculated motivations behind KGERS design are correct. KGERS is able to converge, and in similar CPU time like Linear Least Squares. KGERS certainly offers some contributions based on speed, accuracy and resistivity to noise and over fitting. All these features mentioned are essential for the correct construction of a Regression Tree. One disadvantage of KGERS is its instability. As we see in Figure 5.2.2.a, KGERS shows an increase in instability when noise is present. This all depends on the size of K; the bigger K is the more likely the algorithm can find its way to a global optimal. This of course, comes of the price of CPU power.

5.4. Non-linear Regression – Experiment results

This research has come a long way to arrive to this point. In this section the new devised Regression Tree will be compared against other off the shelf algorithms that perform regression. These algorithms are described in Table 4.3 of the previous chapter. This section includes a similar structure to that one of linear regression experimental results; and the objective is the same. We want these algorithms compared, and being able to show the main advantages and disadvantages that they offer.

5.4.1. Computational Time

Time complexity analysis shows a constant time for a regression tree using KGERS. As we can see, in Figure 5.4.a, time increases as the size of the input set increases. This is expected, since KGERS at the leaf nodes doesn't utilize the entire example set to learn the proper hyperplane. Interestingly, gradient descent and the Forward Feed Neural network portray a similar rate of growth. It can be observed from this figure that the Neural Network is multiple magnitudes beyond CPU clock ticks than than the other algorithms. Figure 5.4.b shows a closer version for Regression Tree KGERS, Regression Tree Gradient Descent and Regression Tree Mean. Amongst these, Regression Tree Mean shows the least growth, however Regression Tree KGERS shows almost a constant time in CPU clock ticks. This is again due to the fact that KGERS doesn't use the entire input space to resolve a solution and produce regression. The opposite can be said with Regression Tree Gradient Descent, which shows a more rapid growth.

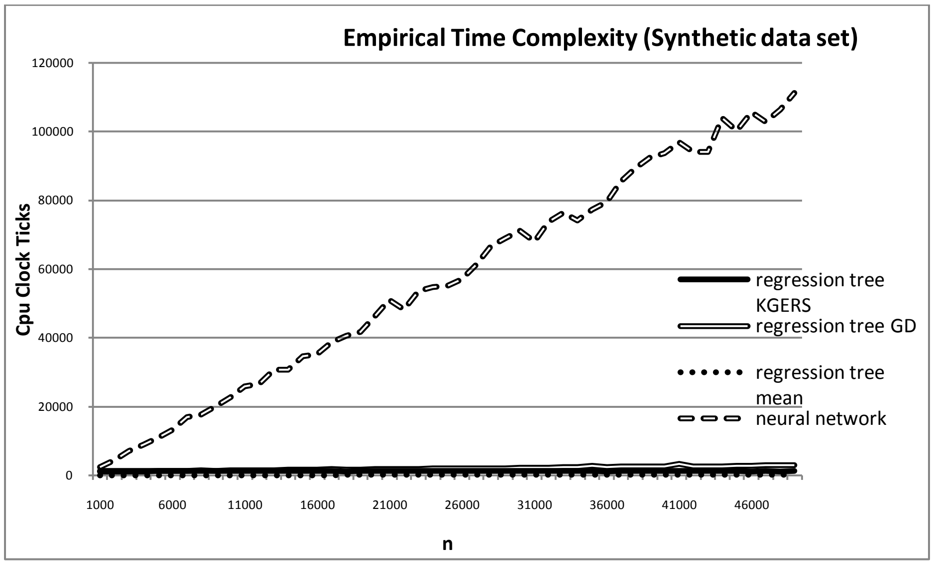


Figure 5.4.a non linear regression algorithms speed analysis

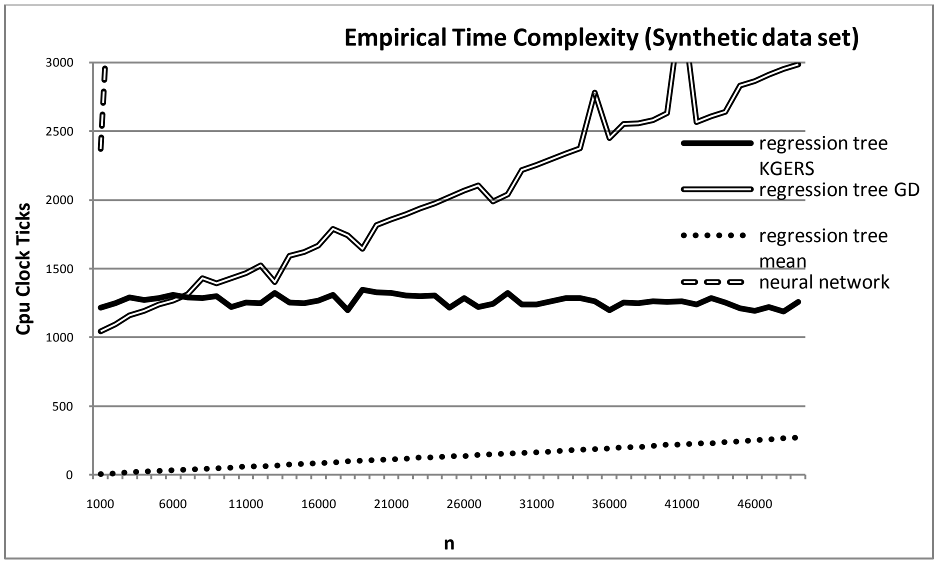


Figure 5.4.b non linear regression algorithms speed analysis zoom.

As seen in this figure other algorithms increase linearly with respect to the size of the input. KGERS will not continue growing as the input size grows since its time complexity is independent of N. The real data in later experiments presents a larger set, meaning that the growth line of these algorithms will keep increasing linearly with respect to the input size. The spikes are caused due to irregularities in the CPU tasks. Memory allocation and compiler optimizations all take part into this noise. Finally we see that the Regression Tree with mean at leaf nodes performs very fast, and doesn't seem to increase as much as the others, however it has to be remembered that KGERS split is simple and requires a lot of tree levels in order to fit a non linear function correctly.

5.4.2. Performance on noisy data

The following set of experiments will give us an idea of the performance that the proposed KGERS Regression Tree has over other algorithms. It should also be able to expose possible flaws that such algorithm possesses.

5.4.2.1. No Noise

Figures 5.5.1.a, 5.5.1.b, 5.5.1.c and 5.5.1.d show the performance of these algorithms on a data set with no noise.

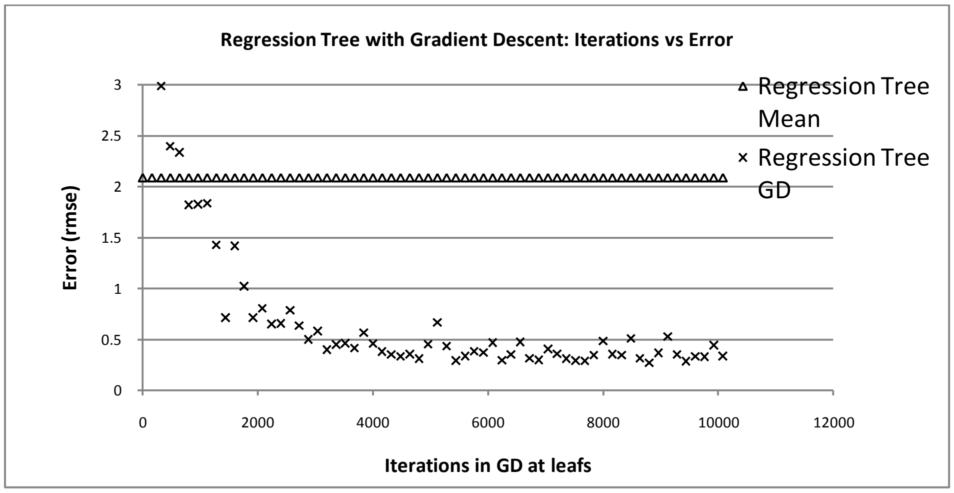


Figure 5.5.1.a RT Gradient Descent Analysis on Synthetic data with no noise

Figure 5.5.1.a shows that the gradient descent algorithm in the regression tree converges close to zero. The convergence is reached fast and is limited by the size of the tree. Keep in mind that the function is a sine wave, thus piece wise linear regression will still have some traces of error.

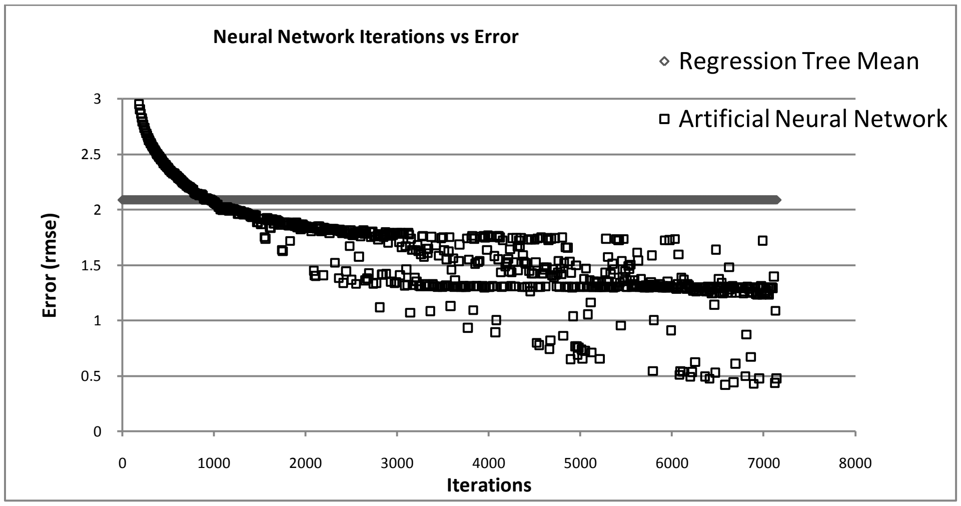


Figure 5.5.1.b Neural Network Analysis on Synthetic data with no noise

Figure5. 5.1.b shows the results of neural network acting on Synthetic data (a simple sine wave) with no error. As it can be appreciated, the neural network tends to converge, but gets stuck into a seamless cycle of noise. A reason for this is that the learning rate is constant. Certain problems require an adjustable learning rate which will achieve better convergence.

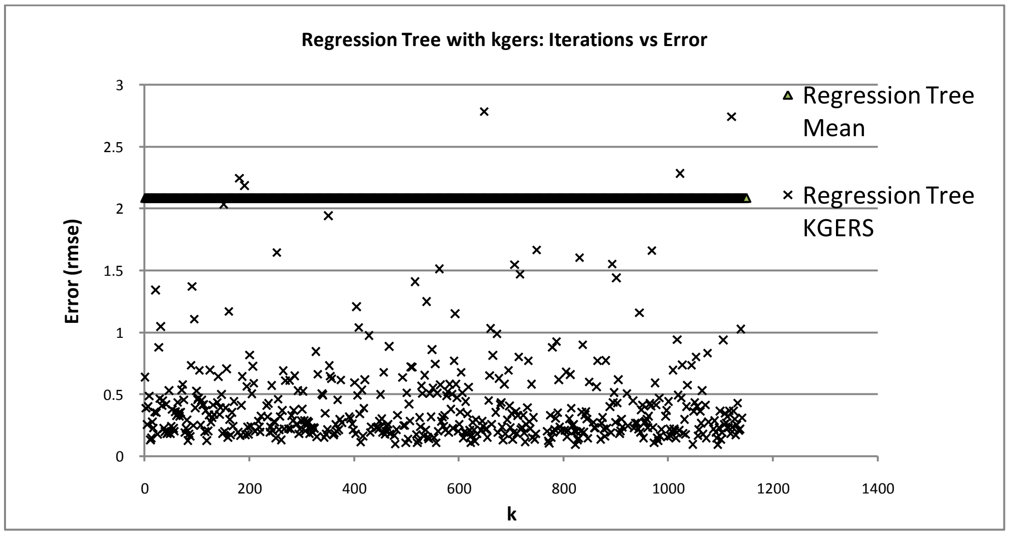


Figure 5.5.1.c RT KGERS on Synthetic data with no noise

Regression tree with KGERS is shown in Figure 5.5.1.c. Because the splitting criteria discovered already the best places to do linear regression, KGERS doesn't need a significant number of iterations (parameter K) to find a line. Remember that if an example set of points fits perfectly in a line, KGERS would only need *m+1* (where *m* is the dimensionality) in order to find a line that fits perfectly with respect to all the points.

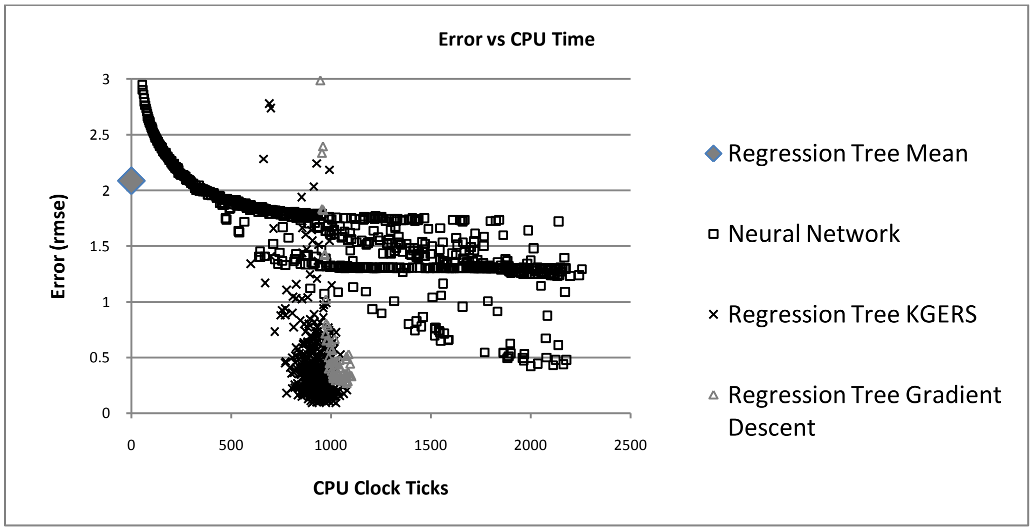


Figure 5.5.1.d CPU Analysis on Synthetic data with no noise

Figure 5.5.1.d presents the basis for this error study. As we can see, KGERS and the Gradient Descent Regression tree achieve the lowest levels of errors with least CPU time consumption. On the other hand, the neural network requires some extra time to get closer to the convergent answer. As expected the Regression Tree using the mean didn’t do a good job, this is because of the inflexibility at its leaf nodes. The next set of experiments take place with low error data. These will be shown in Figures 5.5.2.a, 5.5.2.b, 5.5.2.c and a complete comparison in Figure 5.5.2.d.

5.4.2.2. Low Noise

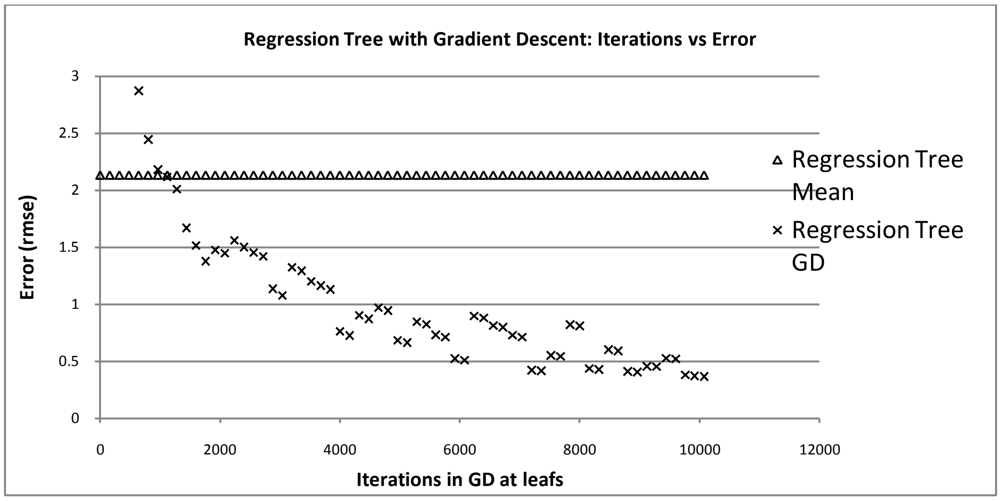


Figure 5.5.2.a RT Gradient Descent Analysis on Synthetic data with low noise

Figure 5.5.2.a shows the error convergence of a Regression Tree using Gradient Descent in the leaf nodes. The error achieved here is still not significantly high, and very similar to that of the data set without errors.

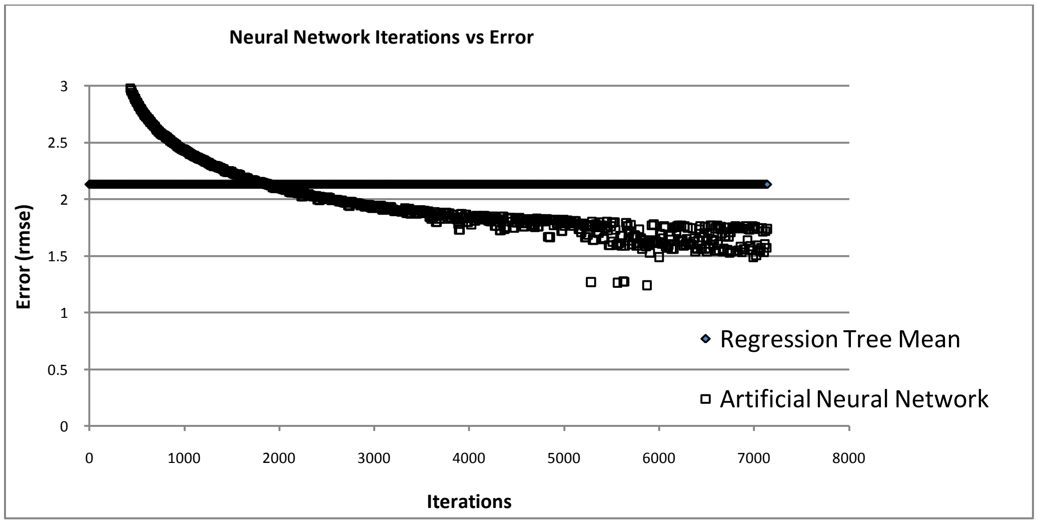


Figure 5.5.2.b Neural Network Analysis on Synthetic data with low noise

As seen in Figure 5.5.2.b and 5.5.1.b the neural network converges to a medium error. The problem cause for this is a bigger need of intermediate neurons to completely be able to plot the target function. In Any case, it seems that it is affected by low noise quantities.

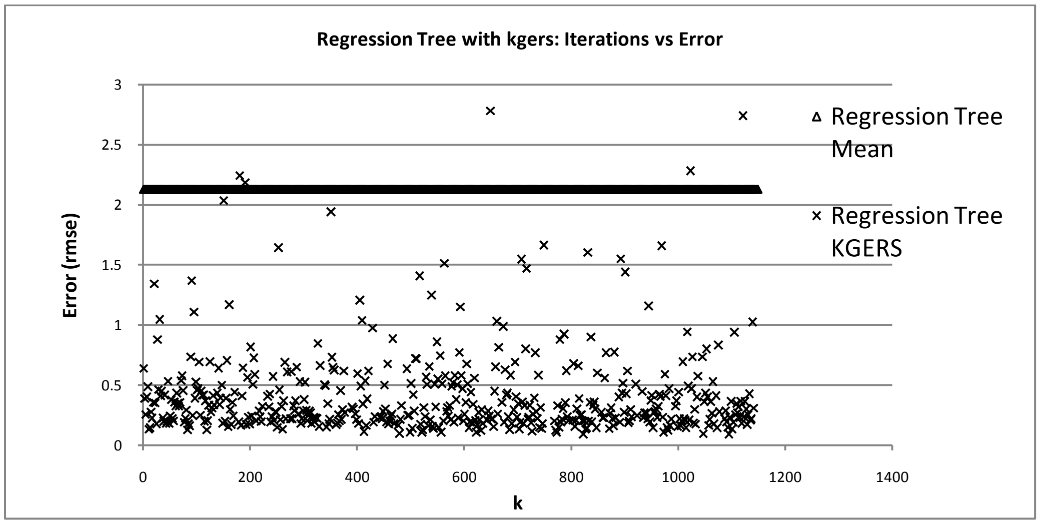


Figure 5.5.2.c RT KGERS on Synthetic data with low noise

The KGERS Regression Tree shown in Figure 5.5.2.c shows that KGERS seems unaffected to noise, and keeps a high accuracy on the data. As mentioned before, the splitting criteria does a good job in finding the correct splitting points to find a line, making KGERS use less iterations.

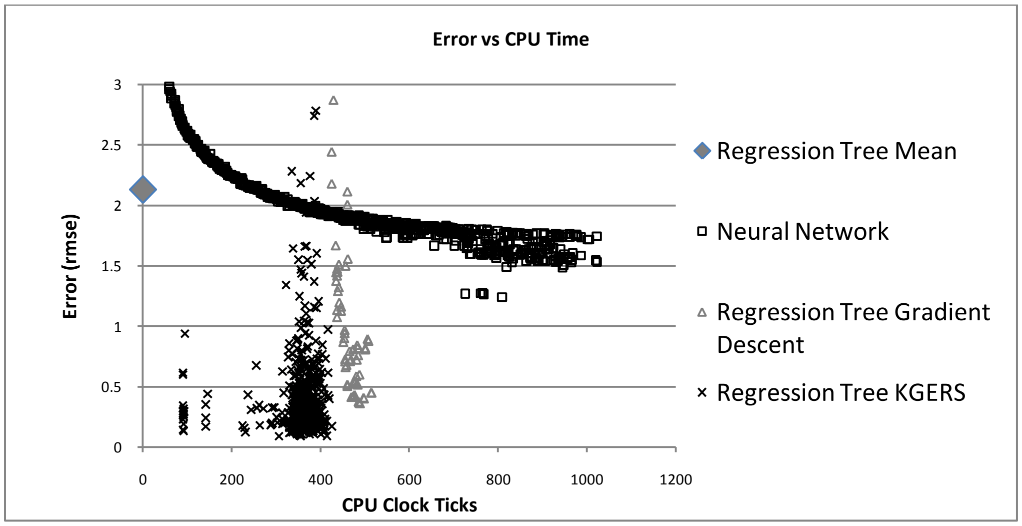


Figure 5.5.2.d CPU Analysis on Synthetic data with low noise

Finally, Figure 5.5.2.d shows that KGERS has the best performance amongst all algorithms with respect to synthetic data with low errors.

5.4.2.3. Medium Noise

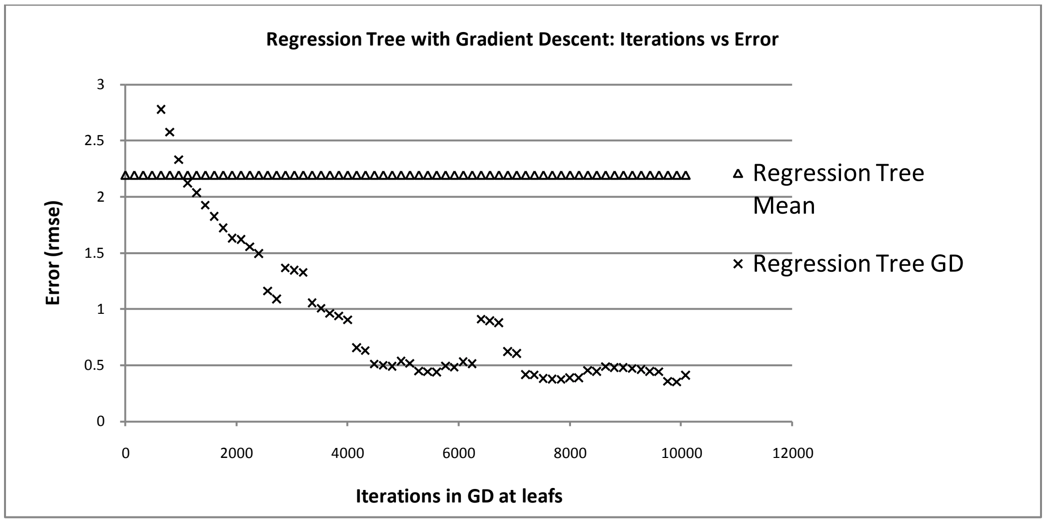


Figure 5.5.3.a RT Gradient Descent Analysis on Synthetic data with medium noise

Figure 5.5.3.a shows now the experiment with medium error. Again, Gradient Descent is able to converge properly after several iterations. The gain again in inaccuracy is insignificant to that compared with synthetic data with low error.

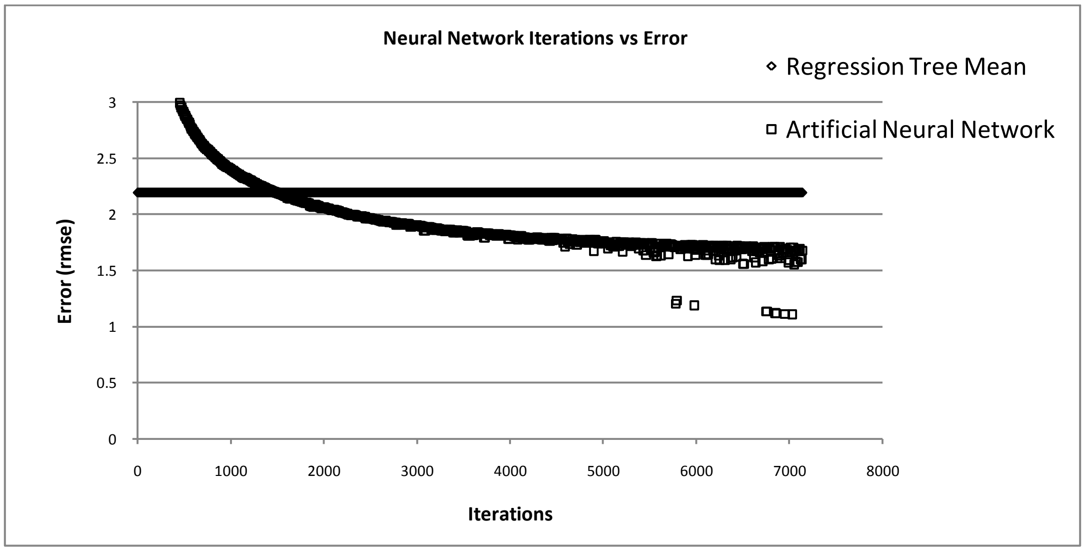


Figure 5.5.3.b Neural Network Analysis on Synthetic data with medium error

Figure 5.5.3.b shows that the neural network does suffer from an increase in error, surprisingly more than a regression tree with medium error.

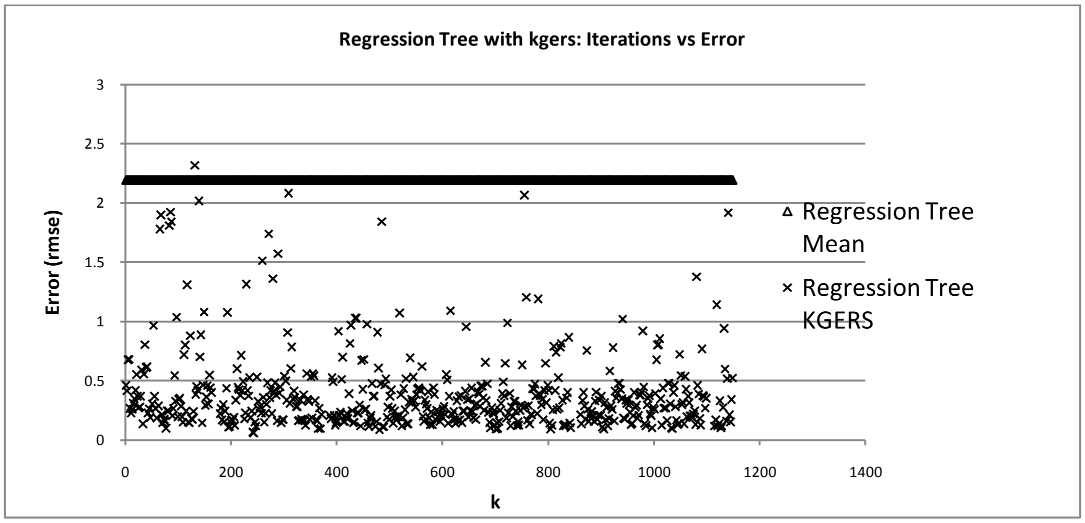


Figure 5.5.3.c RT KGERS on Synthetic data with medium error

Regression Tree with KGERS in Figure 5.5.3.c shows a KGERS keeping accuracy having an average RMSE of 0.27. The splitting criteria does well enough that K doesn't vary much even after several iterations.

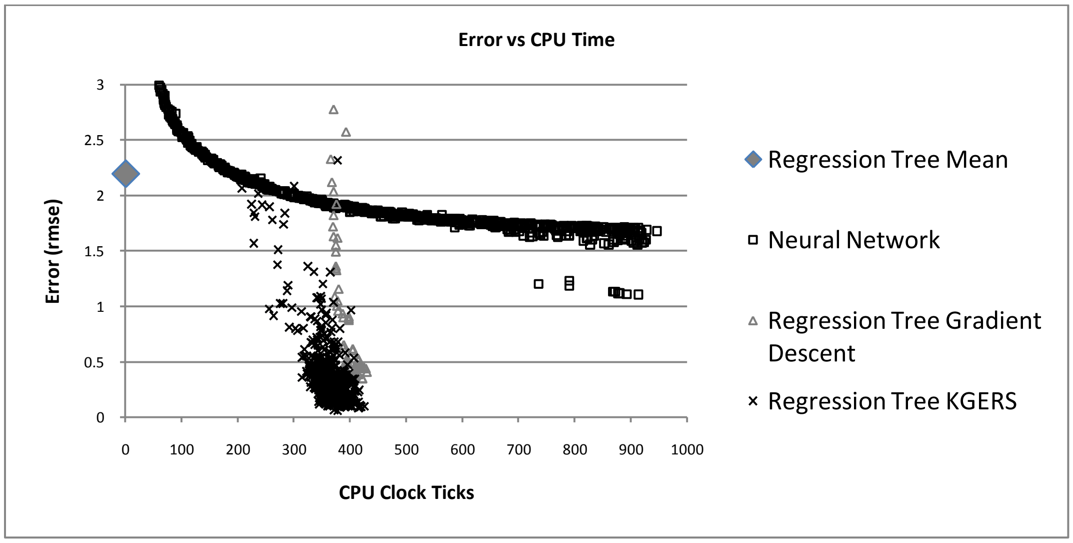


Figure 5.5.3.d CPU Analysis on Synthetic data with medium error

The final summary for medium error is found in Figure 5.5.3.d. It shows that again KGERS emerges victorious amongst the other algorithms with respect to noise. A possible nature is the randomized nature of KGERS, and that the splitting criteria does a very good job identifying the most "linear" areas of the graph.

5.4.2.4. High Noise

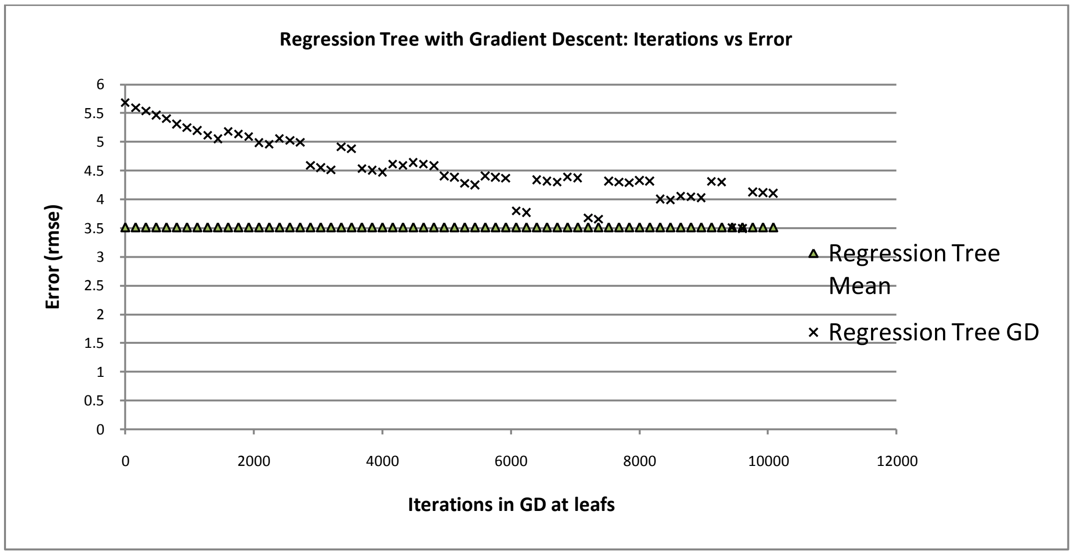


Figure 5.5.4.a RT Gradient Descent Analysis on Synthetic data with high noise

Figure 5.5.4a shows a Regression Tree using Gradient Descent that doesn't perform better than a Regression Tree with mean leafs. Surprisingly, a lot of noise will really alter the splitting criteria and not let the tree develop good lines.

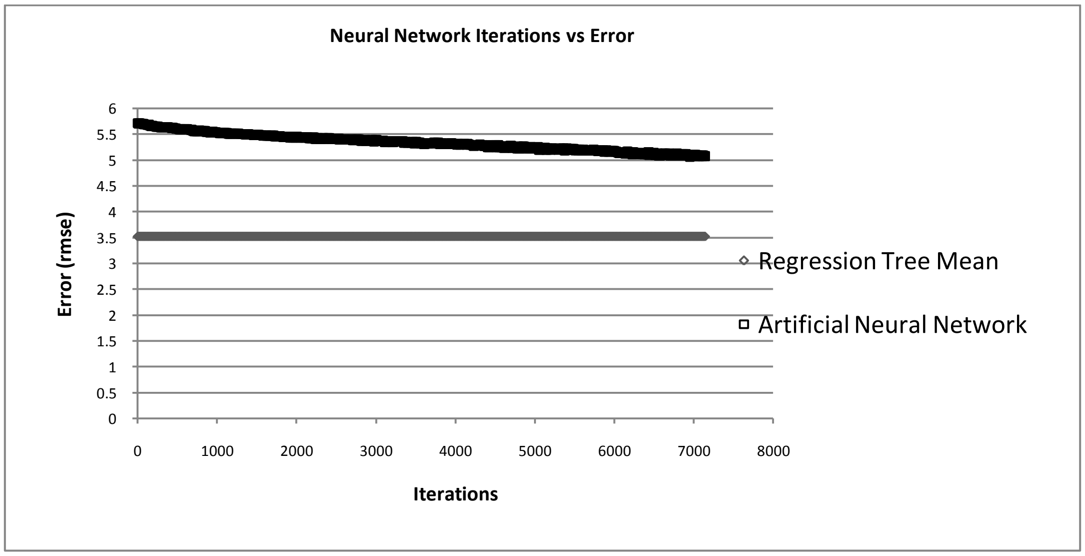


Figure 5.5.4.b Neural Network Analysis on Synthetic data with high noise

The same can be said with the neural network in Figure 5.5.4.b. This analysis shows that the Neural Network doesn't converge in a low error value, even worse than the mean. Because the noise has a Normal distribution, it is intuitive that the mean performs very well. The mean is probably one of the lowest values of that can generate error for a particular sample that has a Normal distribution.

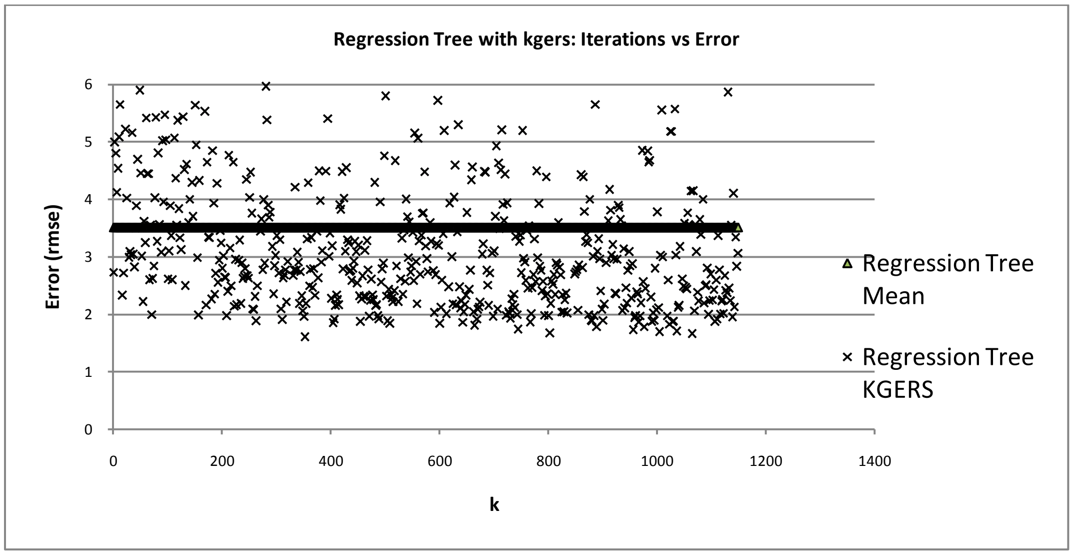


Figure 5.5.4.c RT KGERS on Synthetic data with high noise

Figure 5.5.4.c shows that KGERS maintains the lowest error amongst the other algorithms with least computational time consumption. It performs almost 50% better than the Regression Tree with mean. It has to be noticed that there is an increase of instability that doesn't seem to converge for KGERS compared to the previous data set.

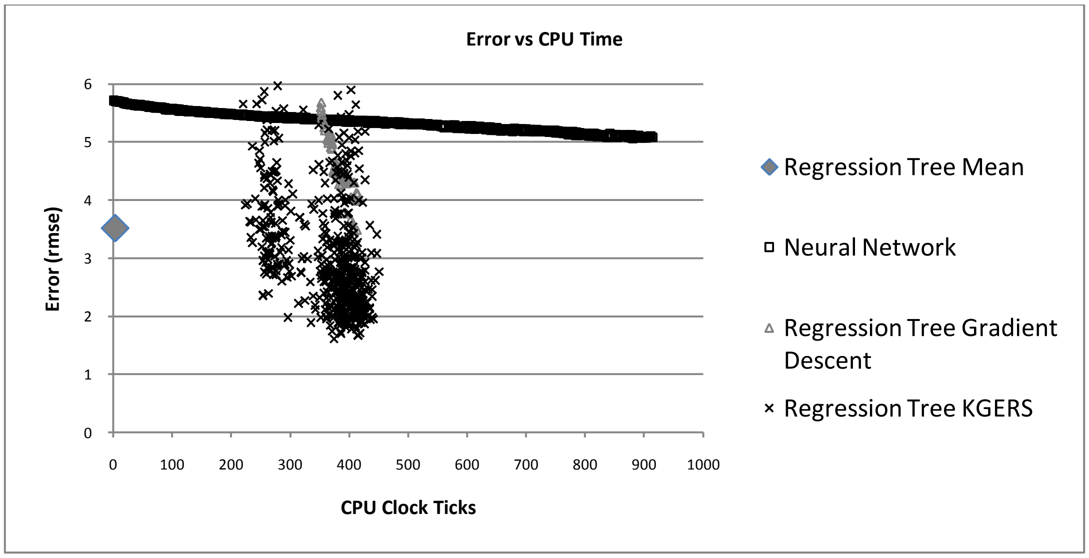


Figure 5.5.4.d CPU Analysis on Synthetic data with high noise

5.4.3. Performance on Over fitting

The next set of experiments is meant to test over fitting for each algorithm. Starting with Figure set 5.5.5 these show the reaction of the algorithms in error with respect to synthetic data designed to over fit.

5.4.3.1. No Noise on Test Set

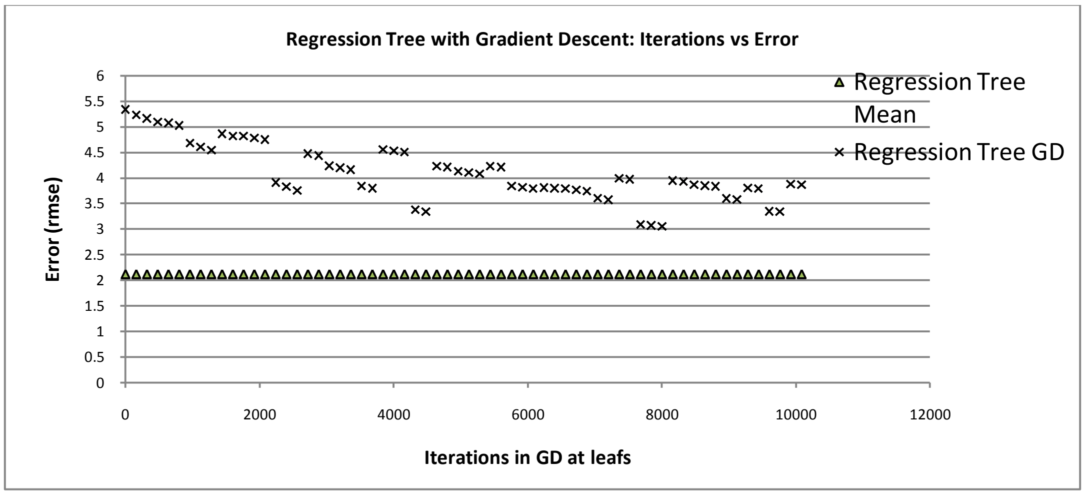


Figure 5.5.5.a RT Gradient Descent Analysis on Synthetic data with no noise on Test Set

Figure 5.5.5.a shows the Regression Tree with Gradient Descent performing with greater error than that of a Regression Tree with mean error. A reason for this is the fact that the noise has a Gaussian distribution which might bias this kind of mean regression to perform better.

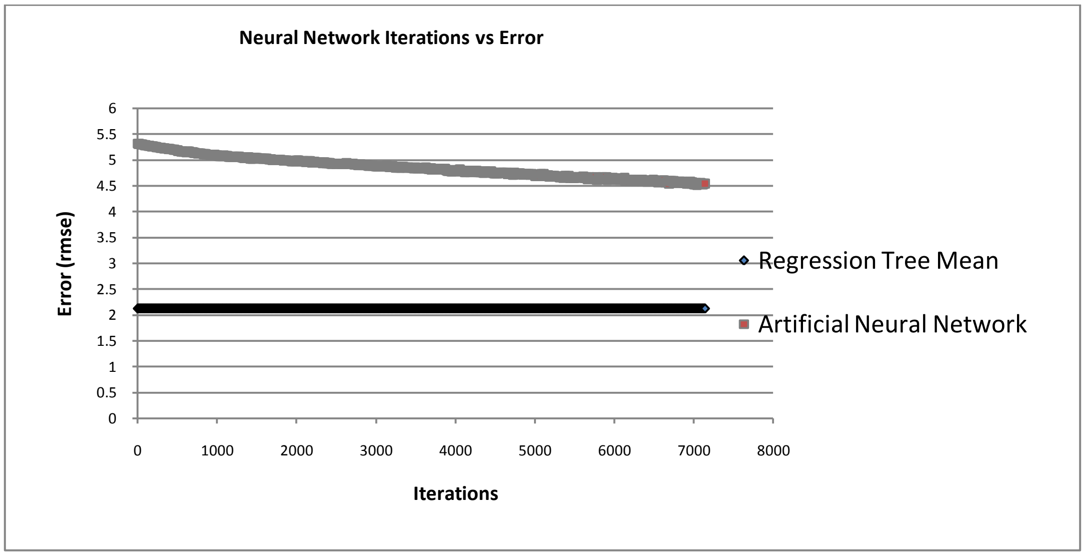


Figure 5.5.5.b Neural Network Analysis on Synthetic data with no noise on Test Set

The neural network in Figure 5.5.5.b shows a relatively high error with respect to the Regression Tree with mean at leaf nodes. Using a test set with no error, and a training set with high error, does some damage to greedy algorithms like back propagation.

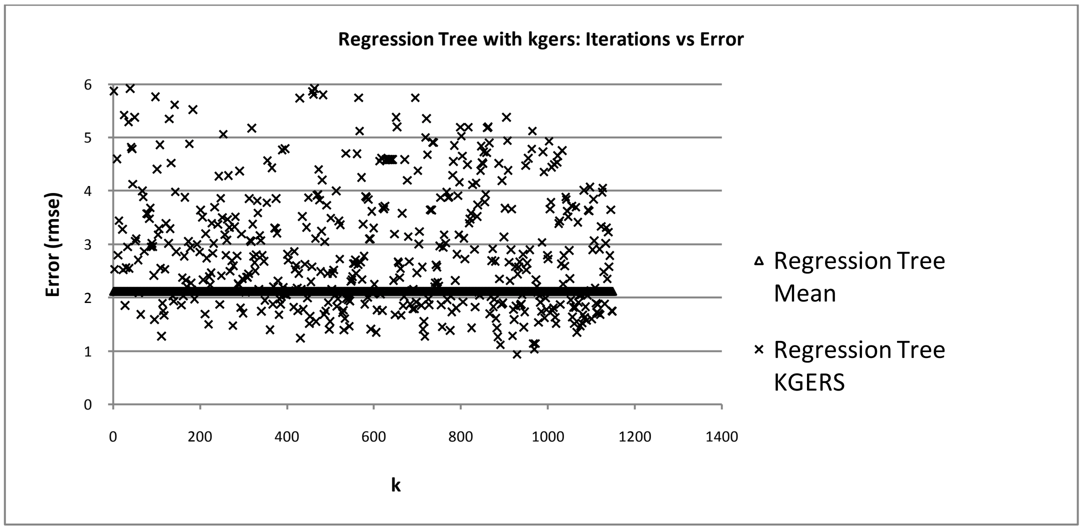


Figure 5.5.5.c RT KGERS on Synthetic data with no noise on Test Set

Figure 5.5.5.c shows that the Regression Tree with KGERS turns a little more unstable, with an increase in error. Still it is capable of reaching lower error rates than those by the neural network and the Regression Tree with Gradient Descent.

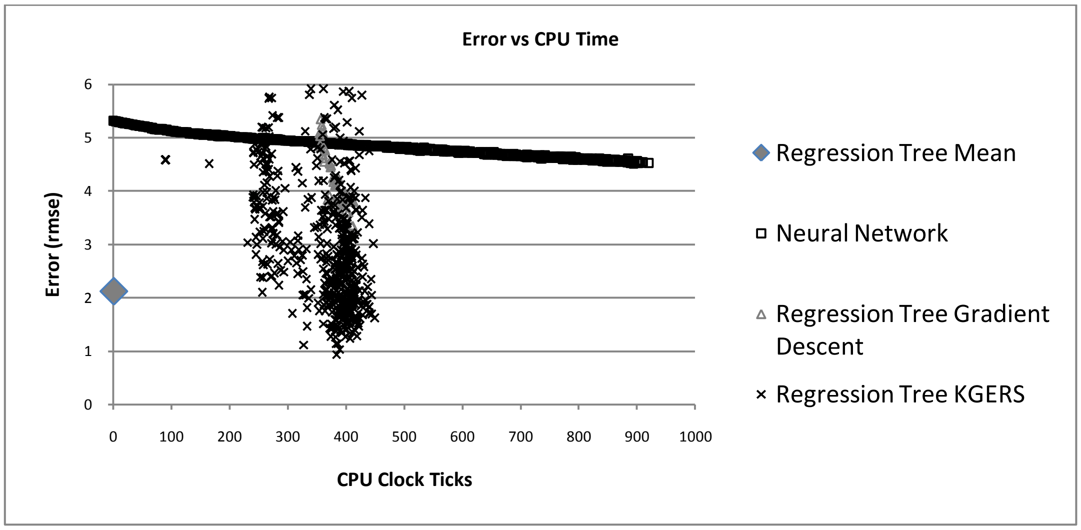


Figure 5.5.5.d CPU Analysis on Synthetic data with no noise on Test Set

An overall view of these can be appreciated in Figure 5.5.5.d. In overall, the Regression Tree with Mean leafs presents a faster and better approach for highly noisy data. Of course, this should be verified with other type of error distribution, but again, this direction might diverge from the initial objective, which is to find a good algorithm for the hospital data.

5.4.3.2. One Irrelevant Feature

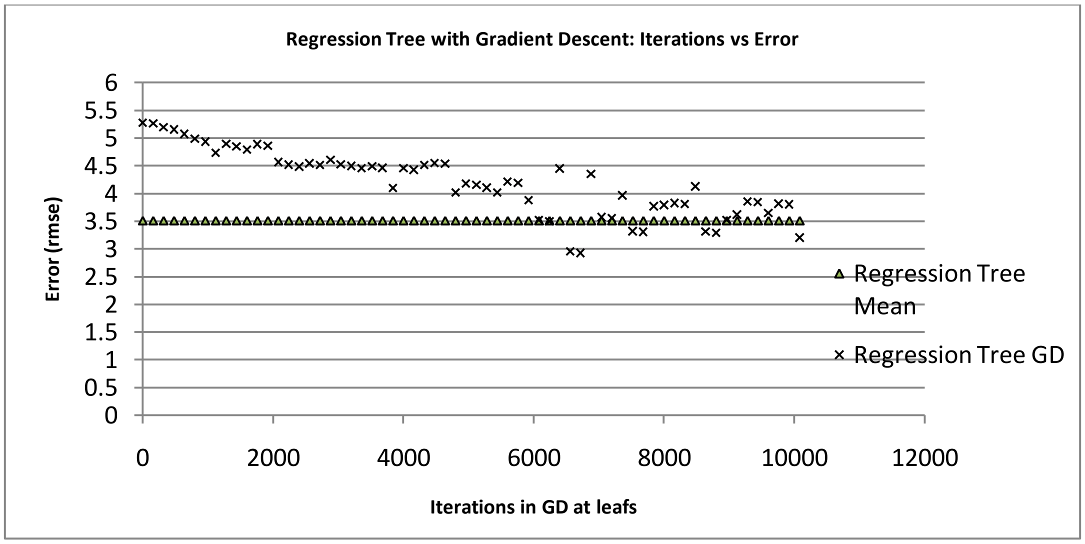


Figure 5.5.6.a RT Gradient Descent Analysis on Synthetic data with one extra irrelevant feature

The next batch of experiments covers one extra irrelevant feature on the synthetic data. Figure 5.5.6.a shows that the Regression Tree with Gradient Descent leafs resists to this noise, and can converge to a lower error than that of a simple mean.

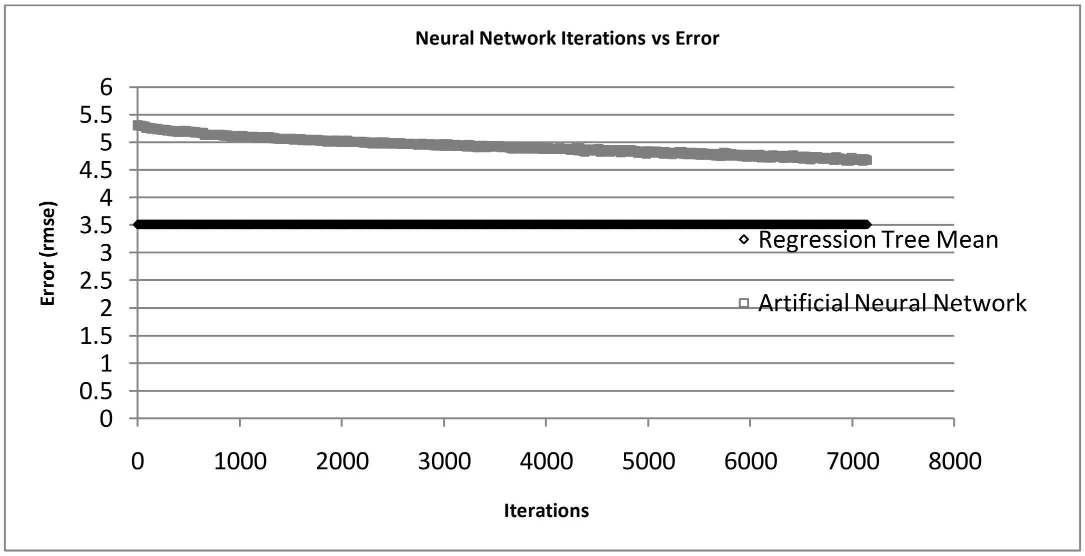


Figure 5.5.6.b Neural Network Analysis on Synthetic data with one extra irrelevant feature

Figure 5.5.6.b shows the Neural Network, still stuck in a high error range. As mentioned before, back propagation tends to get stuck in the local minimum solution, and sometimes is hard to find the global optimal.

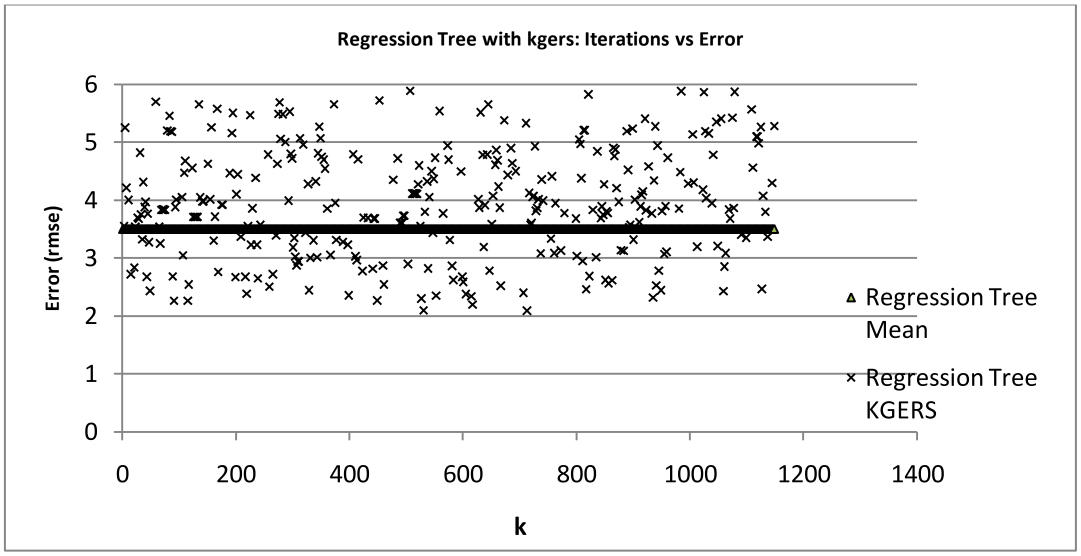


Figure 5.5.6.c RT KGERS on Synthetic data with one extra irrelevant feature

Randomized algorithms such as the Regression tree with KGERS seem to be able to have low error (2.1 at the best case) to this noise, compared to the Regression Tree mean which does around 3.5 of RMSE. Some evidence of KGERS CPU advantage can be appreciated in Figure 5.5.6.c.

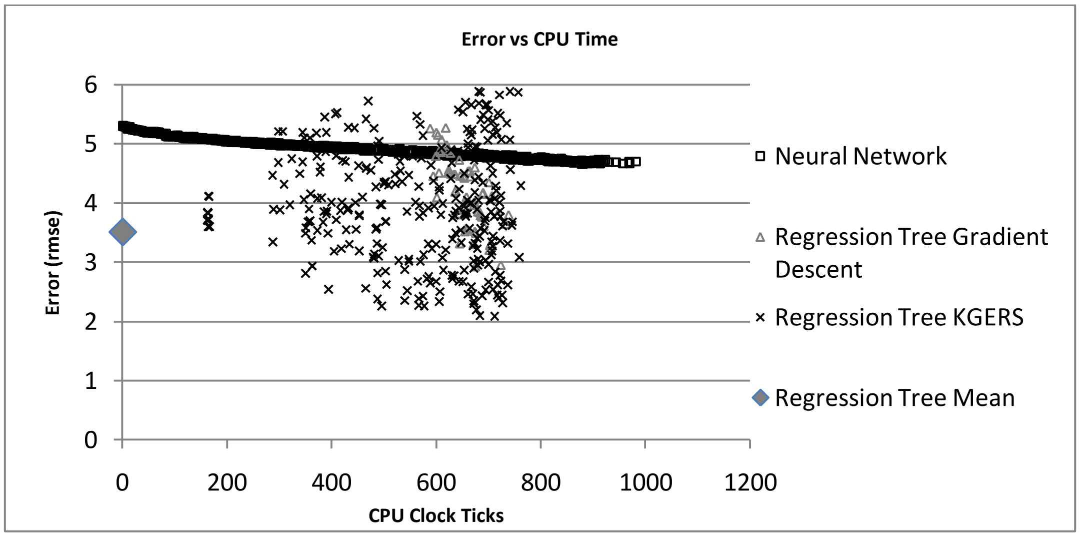


Figure 5.5.6.d CPU Analysis on Synthetic data with one extra irrelevant feature

The summary of the extra irrelevant feature is shown in Figure 5.5.6.d. The CPU cycles required to achieve a low error are higher than usual spanning up to 400 CPU clock ticks, and KGERS shows signs of high instability.

5.4.3.3. Two extra Irrelevant Features

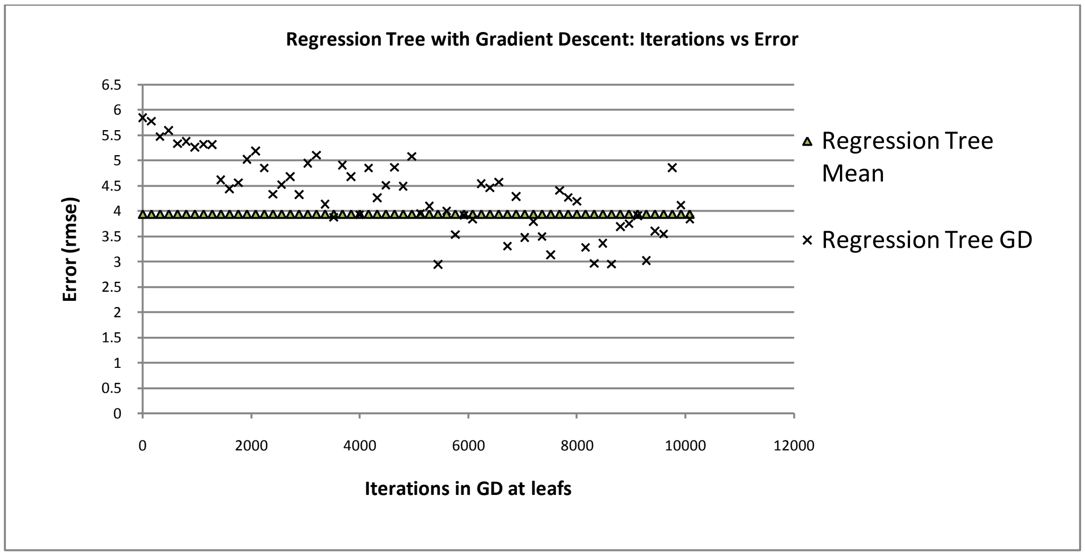


Figure 5.5.7.a RT Gradient Descent Analysis on Synthetic data with two extra irrelevant features

The final batch of experiments involves using two irrelevant features. The first algorithm, which is the Regression tree with Gradient Descent at the leafs shows a very similar error curve.

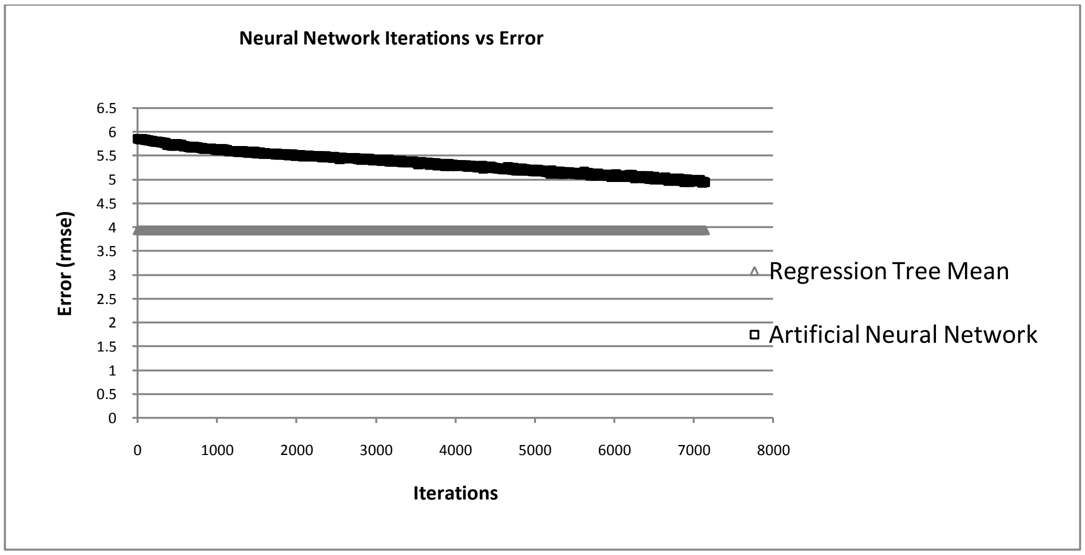


Figure 5.5.7.b Neural network on Synthetic data with two extra irrelevant features

Figure 5.5.7.b Neural Network has a very similar curve to that one of Figure 5.5.7.a. The neural network seems to converge into a high quantity of error.

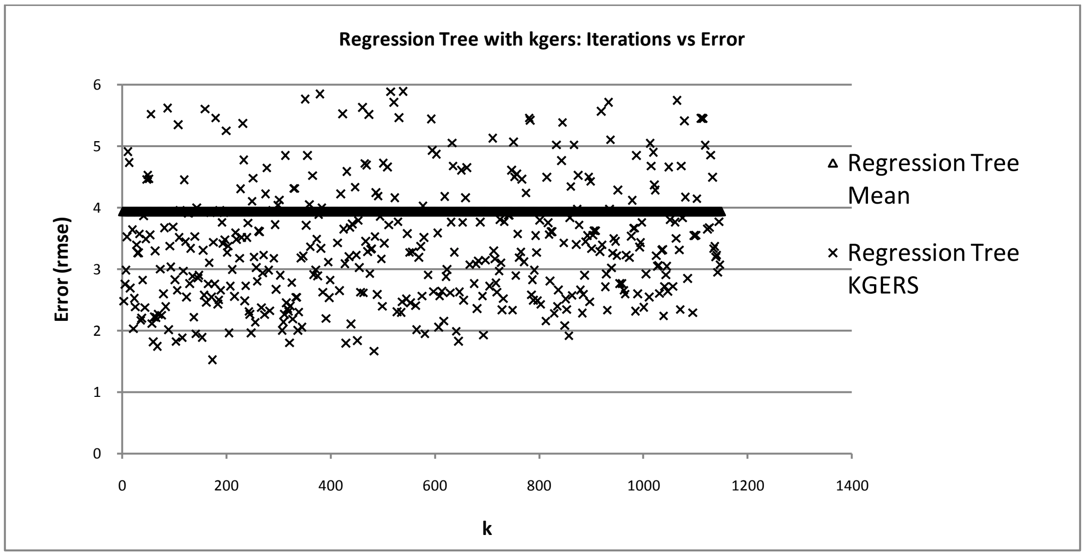


Figure 5.5.7.c RT KGERS on Synthetic data with two extra irrelevant features

The Regression Tree with KGERS in Figure 5.5.7.c shows that this algorithm starts to get very unstable. The more irrelevant attributes, the harder it is for these regression tree algorithms to find a good tree. The result is a very large tree with too many parameters. This can be thought as over fitting. Still regression is well done at the leaves, and a lot of times this error is better than that one of the neural network.

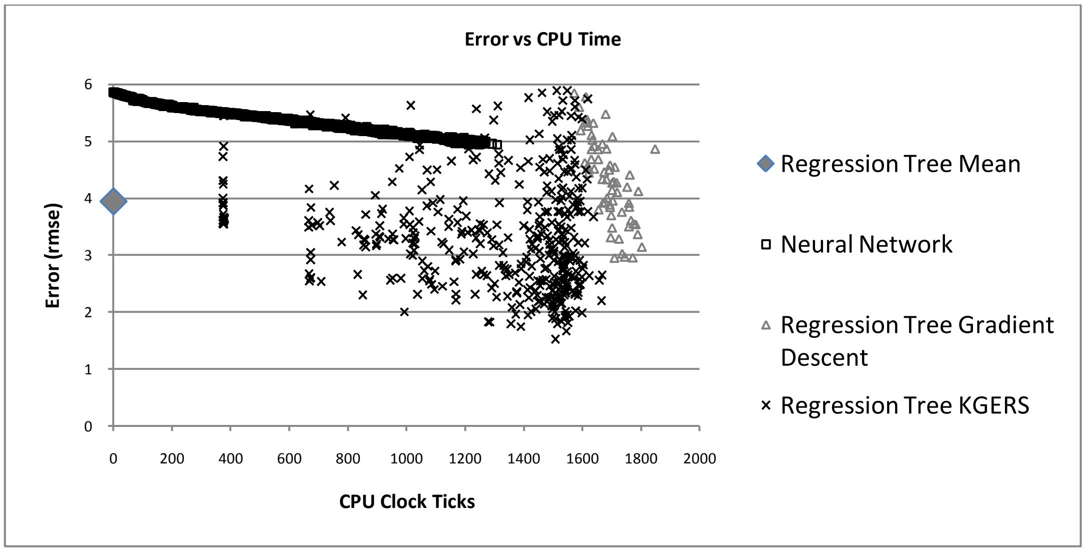


Figure 5.5.7.d CPU Analysis on Synthetic data with two extra irrelevant features

Finally, Figure 5.5.7.d shows the summary of CPU power versus error for each algorithm in this last batch of experiments. The pattern found here is that the more noise is injected to the data set, the more likely to over fit. The Regression tree using KGERS trades this over fitting with instability. This is a great advantage, since instability would mean that there is something wrong with the training set. This could come from either the features or noise itself. Algorithms like the Neural Network don't present any signs of trouble when doing this, and silently fit the noisy data.

As shown before, KGERS seems to be a better fit for a Regression Tree seeking for linear approximation. We can see that during the presence of noise it can preserve accuracy (degrading at most by 30% of MAE error) and can ultimately show signs of instability instead of fitting on the error.

5.5. Summary

This chapter shows the behavior of the new linear regression algorithm KGERS when facing regression with a data set with degrading noise. The algorithm becomes more unstable, however it still has the possibility of converging to a correct global optimal. Other algorithms such as Gradient Descent and Linear Least Squares tend to over fit. Surprisingly Linear Least Squares does hold its position and shows resistivity into noise although not always. The next section shows the behavior of the new devised regression tree which uses KGERS splitting, T-Test Shape stopping criterion and KGERS regression. This new algorithm is compared to standard off the shelf algorithms, such as gradient descent Regression Tree and the Feed Forward Neural Network. Experiments show that the KGERS Regression Tree does show some sign of robustness. Inconsistency is a problem; the randomized algorithm becomes more unstable as there is more noise instead of over fitting. It was shown that KGERS Regression Tree is significantly faster than the Feed Forward Neural Network and the Gradient Descent Regression Tree. Most of these advantages are due to its independence of the input size N.

Chapter 6

Conclusions

The empirical evidence in this study shows two kinds of contributions. The first contribution is the nature of KGERS as linear regression and as a composition of Regression Tree algorithms. KGERS shows lower error from other algorithms (around 3% less), and faster regression (75% faster than Neural Networks on average) on the hospital data. This lead to the trial usage of the models to estimate the admissions from the ED data for 3 weeks. The second set of contributions deal with general algorithmic properties of KGERS, which show more efficient regression when exposed to noisy data. One of the main limitations is the estimation of the K parameter in KGERS, which was done manually for this study. New improvements would include exploring systematic ways of estimating K.

6.1. Contributions

The first of contributions talks about the medical informatics problem that Melbourne Holmes Regional Medical Center is facing. The second set of contributions deal with the algorithmic evaluation results, and the new devised techniques used for piecewise linear regression.

6.1.1. Medical Informatics Contributions

The first contribution is the accuracy on the estimations of future admissions from the EDInpt data. Currently, Holmes Regional Medical Center is using the models proposed on this study to estimate admissions. While the error still goes around 16% (in terms of MAE), the absolute error is lower and allowable within the hospital interests. There is satisfaction on the staff side. As a result there has been 3 weeks of trial usage of the models presented in this research to estimate admissions. Currently the research's direction might go into official operation on the hospital side.

The second contribution deals with the rest of admissions data. Estimation of the other 3 hospital data sets (PCU, ICU, and Floor) does present low error (around 16% in terms of MAE). Because of time constrains these results have not been presented to the hospital, however they show similar levels of errors to those of the EDInpt data set. This means that the forecasting with these data sets also present desirable results that might help with the holding problem.

The final contribution is the fact that the Regression Tree using KGERS at the leaf nodes presents higher accuracy than the other traditional algorithms. This confirms the initial suspicion that the other algorithms are more likely to over fit and not perform as well under noisy circumstances.

6.1.2. Algorithmic Contributions

The Regression Tree using KGERS for splitting and leaf nodes is the first algorithmic contribution from this research. Experiments confirm that the time complexity is independent from the input size, making it faster than the other traditional algorithms.

The second algorithmic contribution is the fact that KGERS linear regression does guarantee a solution, unlike Linear Least Squares. Linear Least Squares only gives a solution when its input matrix is invertible, otherwise it will fail. KGERS can perform as fast as this algorithm and guarantee a solution at the same time.

KGERS in the leaf nodes shows that is maintains low error (around 20%) when exposed to high noise (33% of noise) for non linear regression. While the traditional algorithms over fit, or sometimes don't even converge, KGERS seems to keep the same error and not completely diverge from low error quantities.

Linear regression using KGERS presents a time complexity of $O(M^2)$, where M is the dimensionality of the problem. This shows that KGERS is independent from N, providing a faster way of doing splitting and regression for Regression Tree models. Other algorithms such as Neural Networks, need to analyze the entire input space.

KGERS is a good choice for piece wise linear regression since at the leaf nodes it is almost guaranteed by design that the data presents a linear pattern. This means that if the input space is split into very linear chunks, KGERS is a great choice since it doesn't need to look at the entire input space to generate a low error hyperplane. This can even potentially be done with low values of K.

6.2. Limitations and Potential improvement

The disadvantages of KGERS can be observed from the experimental data in chapter 5. The regression tree using KGERS presents problems when there is low error on the data. The jump from no error to a small quantity of error is big (around a 14% of MAE increase in noise). Although this is countered by the fact that this algorithms keeps the rate of error from noise small compared to other algorithms like back propagation in the neural network.

Since KGERS is a randomized search algorithm, it can present instability problems when data is very noisy. This means that the likelihood of getting a low error decreases. Future work in this area involves exploring techniques and heuristics that might decrease this erratic probability when the noise is high. Estimating the correct value for K (number of sub sets that KGERS does) is done manually. New statistical ways can be explored to automate this, and let the algorithm pick the correct value for K.

KGERS algorithm uses an EPS design (equal probability of selection). This EPS design could be a potential limit in the process of learning. Using sampling methods could improve the performance if applied carefully. The problem with an EPS design with respect to the hospital data is that several points occur more often than others. For example, the likelihood of having 25 patients is lower than that of having only 13 patients a time slice in the EDInpt. Sampling theory states that a biased population requires a biased sampling. Amongst the existent techniques are stratified sampling and clustering sampling. Stratified sampling will extract candidate subsets from those feature vectors that share similar semantics, in terms of their Y value. Clustering sampling can use the feature vectors, cluster them in groups and extract samples of these subgroups with similar probability of proportion. Biasing the training data in terms of sampling, so it reflects the true population, can potentially give more honest results.

Another possibility for future work involves looking at Box Jenkins models (Wei Yin Loh 2008), such as ARIMA. These models also use information about the distribution of noise in the data, calling it random shock modeling. Finally, on the hospital side, there is still work to be done regarding modeling of the holding problem. The EDInpt, PCU, ICU and Floor models output could be combined to perform regression on the average holding time of patients. There is still plenty of work on this area and we believe that the contributions presented in this study will help improving the efficiency of hospitals.

Bibliography

Andrea Matsunaga, Jose A.B. Fortes (2010), "On the Use of Machine Learningng to Predict the Time and Resources Consumed by Applications," ccgrid, pp.495-504, 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, 2010.

Bagnall, A. J. and Janacek, G. J. (2004). Clustering time series from ARMA models with clipped data. In Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '04). ACM, New York, NY, USA, 49-58. 2004.

Buja, A., Lee Y-S. (2001), "Data mining criteria for tree-based regression and classification", ACM Special Interest Group on Management of Data, pp. 27 – 36.

Chetan Gupta, Abhay Mehta, Umeshwar Dayal (2008), "PQR: Predicting Query Execution Times for Autonomous Workload Management," icac, pp.13-22, 2008 International Conference on Autonomic Computing.

D. Montgomery, Johnson L. (1976), Forecasting and Time Series Analysis. Washington, DC: McGraw-Hill. Chp 4, 5.

Dobra, A., Johannes, G.(2002), "SECRET: A Scalable Linear Regression Tree Algorithm", proceedings of the 8th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '02). ACM, New York, NY, USA, 481-487, 2002.

Golding, D, Ricardo, Mardales, George Nagy (2006) "In Search of Meaning for Time Series Subsequence Clustering: Matching Algorithms Based on a New Distance Measure" , pp.347-356, Proceedings of the 15th ACM international conference on Information and knowledge management, 2006.

Huang, C.; Townshend, J. R. G. (2000). "A stepwise regression tree for nonlinear approximation: applications to estimating subpixel land cover"

Izrailev, S., Angrafiotis, D. (2000), "A Novel Method for Building Regression Tree Models for QSAR Based on Artificial Ant Colony Systems", J. Chem. Inf. Comput. Sci. 2001, 41, pp. 176-180.

Kohonen, T. (2008). Data management by self-organizing maps. In Proceedings of the 2008 IEEE world conference on Computational intelligence: research frontiers (WCCI'08), Jacek M. Zurada, Gary G. Yen, and Jun Wang (Eds.). Springer-Verlag, Berlin, Heidelberg, 309-332.

Leegon, J., Joens, I., Lanaghan, K., Aronsky, D., (2006) Predicting Hospital Admission in a Pediatric Emergency Department using an Artificial Neural Network, Proc. of American Medical Informatics Association (AMIA) p. 1004 Dep. of Biomedical Informatics Vanderbilt University, Nashville, TN, USA.

Li, J., Guo, L., (2009) Hospital Admision Prediction Using Pre-hospital Variables IEEE International Conference on Bioinformatics and Biomedicine, p 283-286
Mitchell, T. (1997), Machine Lerning, McGraw-Hill.

Nikolov, Ventsislav (2010). "Optimizations in Time Series Clustering and Prediction" , pp.528-533, International Conference on Computer Systems and Technologies, 2010.

Perlich, C., Rosset, S., Lawrence, R., Zadrozny, B., (2006) High-Quantile Modeling for Customer Wallet Estimation and Other Applications. Industrial and Government Track Paper. IBM T.J. Watspn Research Center & Universidade Federal Fluminense, p 977-985
Pissarenko, D. (2002). Neural Networks For Financial Time Series Prediction: Overview Over Recent Research. BSc thesis. pp 25-32. 2002.

Specht, D.F. (1991); , "A general regression neural network," Neural Networks, IEEE Transactions on , vol.2, no.6, pp.568-576, Nov 1991 .

Vogel, D., Asparouhov, O., Scheffer, T. (2007), "Scalable Look-Ahead Linear Regression Trees" , International Conference on Knowledge Discovery and Data Mining, pp.757-764, 2007

Wei, F., Joe, M., Philip, S. Yu. (2006), "A general framework for accurate and fast regression by data summarization in random decision trees.", Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '06). ACM, New York, NY, USA, 136-146, 2006.

Wei-Yin Loh (2008), " Classification and Regression Tree Methods" , Encyclopedia of Statistics in Quality and Reliability, Wiley, pp.315-323, 2008
Yohannes Y., Webb P., Classification and Regression trees, CART: A user manual for identifying indicators of vulnerability to famine and chronic food insecurity, Washington.