Université de Montréal

Financial Time Series Analysis with Competitive Neural Networks

par

Maxime Roussakov

Département de mathématiques et de statistique
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Statistique
orientation Mathématiques Appliquées

August 7, 2017

© Maxime Roussakov, 2017

SOMMAIRE

L'objectif principal de mémoire est la modélisation des données temporelles non stationnaires. Bien que les modèles statistiques classiques tentent de corriger les données non stationnaires en différenciant et en ajustant pour la tendance, je tente de créer des grappes localisées de données de séries temporelles stationnaires grâce à l'algorithme du « self-organizing map ». Bien que de nombreuses techniques aient été développées pour les séries chronologiques à l'aide du « self-organizing map », je tente de construire un cadre mathématique qui justifie son utilisation dans la prévision des séries chronologiques financières. De plus, je compare les méthodes de prévision existantes à l'aide du SOM avec celles pour lesquelles un cadre mathématique a été développé et qui n'ont pas été appliquées dans un contexte de prévision. Je compare ces méthodes avec la méthode ARIMA bien connue pour la prévision des séries chronologiques. Le deuxième objectif de mémoire est de démontrer la capacité du « self-organizing map » à regrouper des données vectorielles, puisqu'elle a été développée à l'origine comme un réseau neuronal avec l'objectif de regroupement. Plus précisément, je démontrerai ses capacités de regroupement sur les données du « limit order book » et présenterai diverses méthodes de visualisation de ses sorties.

Mots Clés: Self-Organizing Map, Limit Order Book, Réseau Neuronal, Analyse en Composantes Principales, Classification Hiérarchique, Stationnarité, Prévisions

SUMMARY

The main objective of this Master's thesis is in the modelling of non-stationary time series data. While classical statistical models attempt to correct non-stationary data through differencing and de-trending, I attempt to create localized clusters of stationary time series data through the use of the self-organizing map algorithm. While numerous techniques have been developed that model time series using the self-organizing map, I attempt to build a mathematical framework that justifies its use in the forecasting of financial times series. Additionally, I compare existing forecasting methods using the SOM with those for which a framework has been developed and which have not been applied in a forecasting context. I then compare these methods with the well known ARIMA method of time series forecasting. The second objective of this thesis is to demonstrate the self-organizing map's ability to cluster data vectors as it was originally developed as a neural network approach to clustering. Specifically I will demonstrate its clustering abilities on limit order book data and present various visualization methods of its output.

Keywords: Self-Organizing Map, Limit Order Book, Neural Network, Principal Component Analysis, Hierarchical Clustering, Stationarity, Forecasting

CONTENTS

Sommaire ii

Summary iii

List of figures 1

List of tables 2

Remerciements 1

Introduction 2

Chapter 1. Preliminaries 6

1.1. Time Series 6

1.1.1. Stationarity 7

1.2. ARIMA Model 8

1.3. Principal Component Analysis 10

1.4. Hierarchical Clustering 13

1.5. Limit Order Book 15

Chapter 2. Neural Networks 19

2.1. Learning Methods 20

2.1.1. Competitive Learning 20

2.2. Self Organizing Maps 21

2.2.1. Unsupervised Maps 21

2.2.2. Supervised Maps 29

2.3. Overfitting 31

2.3.1. Cross-Validation 34

Chapter 3. Forecasting 36

3.1. Maximum Likelihood Forecast 38
3.1.1. KPSS Test 39
3.1.2. AIC Criteria 40
3.2. VQTAM Model 41
3.3. Rule Extraction 42
3.3.1. Min/Max Method 43
3.3.2. Confidence Interval Method 43
3.4. Double Vector Quantization Model 44
3.5. Out of Sample Tests 46
3.6. Error Criterion 47
Chapter 4. Analysis of Cross Sectional Data 49
Chapter 5. Analysis of Sequential Data 64
5.1. Dow Jones Industrial Average Index 66
5.2. Apple 68
Chapter 6. Conclusion 71
Bibliography 73
Appendix A. R Code .... A-i

LIST OF FIGURES

4.1 Validation curve for k-means 52

4.2 Validation curve for the SOM with a Bubble neighborhood function 53

4.3 Validation curve for the SOM with a Gaussian neighborhood function 53

4.4 Progression of MSE with each iteration 54

4.5 Magnitude of each variable in each neuron 55

4.6 Number of training vectors mapped to each neuron 55

4.7 Dendrogram 56

4.8 Supercluster representation of SOM map 57

4.9 Bid-Ask Volume Imbalance Bar Plots 60

4.10 Price vs Time based on SC representation of BAVI 61

4.11 Variance of Principal Components 63

4.12 Projection on first 2 PCs 63

5.1 X-Y Fused SOM Model-Dow Jones 67

5.2 Double VQTAM Model-Dow Jones 67

5.3 ARIMA Model-Dow Jones 68

5.4 X-Y Fused SOM Model-Apple 69

5.5 Double VQTAM Model-Apple 70

5.6 ARIMA Model-Apple 70

LIST OF TABLES

4.1 Fixed Parameters for SOM 52
4.2 Frequencies-SuperCluster 1 59
4.3 Frequencies-SuperCluster 2 59
4.4 Frequencies-SuperCluster 3 59
4.5 Frequencies-SuperCluster 4 59

5.1 Error Criteria-Dow Jones 66
5.2 Error Criteria-Apple 69

REMERCIEMENTS

First and foremost I would like to thank my graduate adviser, Manuel Morales, for guiding me through the completion of this thesis, who always seemed ready to meet with me and provide me with advice on any issues that I had. I truly couldn't have done it without you.

I would also like to thank my family for supporting me all this time and the professors in the Mathematics and Statistics department who taught me courses throughout my graduate studies.

I would also like to thank my colleagues at IPSOL Capital, where I completed a 4 month internship through the Mitacs program, in collaboration with the University. The financial support and insights into the world of financial modeling were very useful for my report. Particular thanks to Luc Gosselin whose insights into the world of finance were invaluable in the presentation of my results. Finally, I would like to thank my friend and colleague, Jean Hounkpe, who also provided me with valuable insights into mathematical modelling and never hesitated to lend me a helping hand in formulating my thoughts during the writing of this thesis.

INTRODUCTION

Since the creation of the stock exchange, experts from a wide variety of fields have been implementing methods to increase investment returns. When it comes to making predictions in the stock market there are two broad approaches to achieving this, fundamental and technical analysis. Fundamental analysis refers to the method of analyzing the performance of the companies underlying each stock through the use of various metrics. Technical analysis, on the other hand, involves the use of historical data to build models that are able to capture patterns and trends. My study will involve the use of technical analysis to build models that can be used by experts to engineer portfolios of financial products that provide a guaranteed profit through the continuous implementation of said models, a concept known as statistical arbitrage.

Classically, the modeling of financial time series has been done based on the assumption of an arbitrage free market through the use of geometric Brownian motion models with estimated drift and volatility components. However, when it comes to short term market movements it has become quite evident that due to exogenous causes created by human involvement in the market, a single differential equation is not quite accurate in modeling the movement of financial time series. Additionally, even in the long term, it has been observed that financial markets can follow different behaviors over time due to such causes such as overreaction and mean reversion. This issue can be explained by the fact that financial time series are non-stationary, meaning their mean, variance and autocorrelations are not constant through time.

Methods for modeling non-stationary times series generally rely on models that use a rolling window based on past observations of the series to build forecasts. Such models base themselves on specific assumptions on the statistical distribution of the error terms to model parameters. An example outside the scope of my study would be the use of factor models and principal component

analysis incorporated into regression models (Avellaneda and Lee, 2008.) In my study, I will be using the ARIMA model for comparative purposes, which is a model for non-stationary time series in which the value of a series at any given point in time is a function of its values in a specified preceding window. The ARIMA model manages to model non-stationary time series by applying a differencing operator to the series at each point in time. In addition to relaxing the assumption of stationarity, the advantage of moving away from geometric Brownian motion models is that the assumption of arbitrage free markets can be let go and forecasting models that create arbitrage opportunities can be developed.

The methods to model non-stationary time series that I plan to address involve the use of models that learn from data and adapt to grow and change when exposed to new data sets, so-called machine learning algorithms. Machine learning algorithms differ from classical approaches in that the models do not need to be explicitly programmed and are thus significantly more robust than classical methods. Specifically, I will be implementing neural networks, which are a branch of machine learning algorithms in which a set of interconnected neurons is used to visualize data as well as extract inferences, similar to a human brain. Now, machine learning algorithms in general can be divided into supervised and unsupervised variants. The supervised variant is meant to extract input-output relationships from data. By identifying a set of input (independent) variables and output (dependent) variables, the algorithm is meant to extract a relationship between the two. Once this relationship is defined it can be applied on a separate data set in order to draw relevant conclusions. Unsupervised machine learning algorithms, on the other hand, have no explicit input or output variables defined but are rather techniques for classifying large data sets in order to improve interpretability.

The purpose of using neural networks in my study will be to model financial time series data. Specifically, I will use a neural network approach to clustering to model a time series of financial product prices and build corresponding forecasts. By clustering vectors of current and lagged prices formed by moving a fixed length window through the series, my aim is to show that the clusters can essentially be interpreted as local models for the series. Since this approach can develop local representations of a non stationary time series, by using the local models independently for forecasting, adjustments to non-stationarity as in the ARIMA model are no longer necessary. In the context of time series forecasting

with neural networks, the input vectors or the independent variables, are the current and lagged prices. The output vector, the dependent variable, can thus be seen as the one step ahead price on the particular financial product. Additionally, in order to demonstrate such a neural network's ability to cluster characteristics of a time series, I will use one to model intra day snapshots of limit order book data, a log of prices and volumes that individuals are willing to buy and sell a particular stock at, in order to gain insight into the behavior of financial market participants. The specific neural network that I will use to achieve this will be the self-organizing map. A self-organizing map is a neural network that uses an unsupervised training algorithm that configures a set of neurons into a representation of the original data. SOM essentially reduces a multi-dimensional data set to a lower-dimensional map of neurons. I will be using it in its original unsupervised implementation to cluster characteristics of the limit order book of a particular stock. The supervised variants of this algorithm that I plan to use for forecasting purposes are the double vector quantization method proposed by Simon et al. (2004) and the X-Y fused SOM proposed by Melssen et al. (2006).

Previous work on time series clustering and forecasting has been explored from various perspectives. As far as implementing unsupervised clustering of high frequency financial data, Blazejewski and Coggins (2006) use self-organizing maps to cluster variables extracted from the limit order books of ten stocks on the Australian stock exchange with the largest market capitalization in order to test a set of hypotheses about intra-day price movements. In the realm of supervised learning, Deboeck and Kohonen (2000) use self-organizing maps in the prediction of interest rates by modeling distributions of interest rate shocks conditional on interest rate structure classes. Barreto (2007) presents several approaches to time series prediction using self-organizing maps which include the vector quantization method, the double vector quantization method, local AR models from clusters of data vectors, on-line learning of local linear models as well as time-varying local AR Models from Prototypes. Koskela et al. (1998) used a temporal variant of the SOM, called the Recursive SOM, to cluster financial time series data. The research performed by Sanchez-Marono et al. (2003) use SOM for data partitioning, while local models in each cluster are built using functional networks. As a final mention, Dablemont et al. (2003) propose a method in which vector quantization is done on a set of input vectors and a corresponding set of output vectors. These are then combined in a probabilistic way, using Radial Basis Function Networks, to build a prediction. In the vector quantization method, Barreto

(2006) uses a rather simplistic approach for transforming the original unsupervised self-organizing map algorithm to a supervised variant. The work done in this thesis builds on his method by incorporating X-Y fused SOMs, which is a more sophisticated variant of the supervised self-organizing map, which is then compared to the double vector quantization method.

The first chapter of this thesis will consist of presenting all the concepts relevant to the framework of my study. A formal definition of time series and stationarity will be presented as well as the concepts of principal component analysis and hierarchical clustering which will be relevant in the interpretation of the output of the self-organizing map. Additionally, I will present the concept of a limit order book and the manner in which it is used to match buyers and sellers of a particular stock. In the second chapter, I will present the concept of a neural network as well as the specifics behind training and testing such a network. Additionally, the self-organizing map algorithm in both its original unsupervised implementation as well as the supervised variants will be presented. I will also be discussing the concept of overfitting, which is vital in scenarios where testing a model is done on a data set separate from the training set. I will also introduce the concepts of cross validation which is necessary for selecting the optimal tuning parameters of a neural network. The third chapter will address forecasting, in which I will present the concept of forecasting a time series using a model trained with a rolling window, a variant of the cross validation method. I will also present the various methods explored in terms of training and testing SOMs as well as the error criteria that I will use to compare these methods. In the fourth chapter, I will present the results of the analysis conducted in R involving the clustering of high frequency cross-sectional data from the limit order book of Apple stock. Finally in the fourth chapter, I will present the results of forecasting obtained from the prices of the Dow Jones Industrial Average Index as well high frequency prices of Apple stock extracted from the same limit order book. I will compare the results obtained through the X-Y fused SOM implementation of the SOM as well as the results of the double vector quantization model and the ARIMA model. A conclusion will complete this thesis.

Chapter 1

PRELIMINARIES

In this section, I present the essential concepts needed to understand the nature of our analysis. We begin by presenting the notion of a time series as well as stationarity. Following this, we present the well-known ARIMA process which is an integrated combination of the AR and MA processes, which will also be presented. Although we limit our discussion of the ARIMA model to the scope of our analysis, further insights can be found in Wei (2006.) Additionally, the concepts of hierarchical clustering and principal components, which will be used in conjunction with our neural network approach to time series analysis, will be presented. Additional information pertaining to hierarchical clustering can be found in Everitt et al. (2011) and that pertaining to principal component analysis in Jolliffe (2002.)

1.1. Time Series

In order to understand the concept of a time series an understanding of a stochastic process must be established. A stochastic process is a family of random variables $X(\omega, t)$ in which ω belongs to a sample space, representing the set of all possible outcomes, and t belongs to an index set.

Definition 1.1.1. *A time series can be defined as the realization of a stochastic process $X(\omega, t)$, as a function of t, for a given ω.*

In stochastic processes, the collection of all possible realizations of a time series is known as an ensemble. In our analysis, we assume that that the index set for a stochastic process consists of integers, $\{X(\omega, t) : t \in \mathbb{Z}\}$ and we denote a realization of a stochastic process, a time series, as $\{X(t) : t \in \mathbb{Z}\}$. Additionally, we represent $\{X(t_1), X(t_2), ..., X(t_n)\}$ as random variables, which are defined as

a set of variables in a stochastic process for fixed times $t_1, t_2, ..., t_n$.

The mean function of a stochastic process can be defined as:

$$\mu(t) = E(X(t)).$$

The variance function of a stochastic process can be defined as:

$$\sigma^2(t) = E(X(t) - \mu(t))^2.$$

The covariance function between $X(t_p)$ and $X(t_q)$ can be defined as:

$$\gamma(t_p, t_q) = E[(X(t_p) - \mu(t_p)) \cdot (X(t_q) - \mu(t_q))].$$

1.1.1. Stationarity

Stationarity is an important concept in time series analysis and in order to understand its implications we define the n-dimensional distribution function for a stochastic process $\{X(\omega, t) : t \in \mathbb{Z}\}$ as:

$$F_{X(t_1),X(t_2),...,X(t_n)}(x_1, x_2, ..., x_n) = P(X(t_1) < x_1, X(t_2) < x_2, ..., X(t_n) < x_n).$$

Definition 1.1.2. *A stochastic process is said to be strictly stationary if its joint distribution function does not depend explicitly on time. For $n \in \mathbb{Z}$, we must have $F_{X(t_1),X(t_2),...,X(t_n)}(x_1, x_2, ..., x_n) = F_{X(t_1+k),X(t_2+k),...,X(t_n+k)}(x_1, x_2, ..., x_n)$ for n-tuple $(t_1, t_2, .., t_n)$ and $k \in \mathbb{Z}$.*

In time series analysis the joint distribution function of an observed stochastic process and therefore strict stationarity is difficult to actually verify. For this reason we define a concept that is closely related but easier to verify in practice, so called second order weak-stationarity.

Definition 1.1.3. *A process is second order weekly stationary if it is strictly stationary up to order to 2, $F_{X(t_1),X(t_2)}(x_1, x_2) = F_{X(t_1+k),X(t_2+k)}(x_1, x_2)$ for $k \in \mathbb{Z}$ and whose first two moments are finite, $E(|X(t)|) < \infty$ and $E(X^2(t)) < \infty$.*

Several consequences follow from a process exhibiting 2nd order weekly stationarity

- the mean function is constant for all t: $\mu(t) = \mu$
- the variance function is also constant for all t: $\sigma^2(t) = \sigma^2$

- the covariance between $X(t)$ and $X(t+k)$ depends only on the time difference k: $\gamma(t, t+k) = \gamma(k)$.

1.2. ARIMA Model

When it comes to analyzing time series, in practice, the assumption of stationarity is often not met. For this reason, it becomes crucial to develop models that correct for non-stationarity in order to fit a robust model to a time-series. The most common violation to stationarity is that of a non-constant mean and there are two scenarios under which this can happen.

The first is the presence of a deterministic trend in time, for which the mean of the process will be a deterministic function of time. Although the original process is not stationary, the deviation from the mean, $X(t) - \mu(t)$, is a stationary process. We denote the original process $X(t)$ a trend-stationary process. In such cases, there are various models that can be developed to represent the non-constant mean (Wei, 2006.) The second scenario is when a time-series behaves similarly at various windows in time after having adjusted for differences in local means. In this case, $\Delta(X)$ is stationary and the original process $X(t)$ is denoted a homogeneous non-stationary process.

In my study, I will present models that correct for homogeneous-nonstationarity through a process known as differencing. In order to present models that incorporate this procedure, we must first define several time series processes.

Definition 1.2.1. *The autoregressive process of order P, AR(P), can be defined as*

$$\dot{X}(t) = \phi_1 \dot{X}(t-1) + \phi_2 \dot{X}(t-2) + ... + \phi_p \dot{X}(t-p) + \epsilon(t)$$

or

$$\phi_p(B)\dot{X}(t) = \epsilon(t),$$

where $\phi_p(B) = 1 - \phi_1 B - \phi_2 B^2 - ... - \phi_p B^p$,
$\dot{X}(t) = X(t) - \mu$,

B^d refers to the dth order backshift operator i.e $B^d(X(t)) = X(t-d)$ for $d \in \mathbb{N}$

and $\epsilon(t)$ represents a sequence of uncorrelated random variables from a distribution with constant mean and variance known as a white noise process.

We refer to Wei (2006) in his demonstration that the AR(p) process is homogeneous-stationary if the roots of $\phi_p(B) = 0$ lie outside of the unit circle.

Definition 1.2.2. *The moving average process of order q, MA(q), can be defined as*

$$\dot{X}(t) = \epsilon(t) - \theta_1\epsilon(t-1) - \theta_2\epsilon(t-2) - ... - \theta_q\epsilon(t-q)$$

or

$$\dot{X}(t) = \theta(B)\epsilon(t),$$

where $\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - ... - \theta_q B^q$,

$\dot{X}(t) = X(t) - \mu$

and $\epsilon(t)$ represents a white noise process.

We again refer to Wei (2006) in his demonstration that the MA(q) process is always homogeneous-stationary.

Definition 1.2.3. *The autoregressive moving average model, ARMA(p,q), is defined as*

$$\phi_p(B)\dot{X}(t) = \theta_q(B)\epsilon(t)$$

or

$$\phi_p(B)X(t) = \theta_0 + \theta_q(B)\epsilon(t),$$

where $\theta_0 = (1 - \phi_1 B - \phi_2 B^2 - ... - \phi_p B^p)\mu$.

With these definitions, we can now begin to develop a model that incorporates homogeneous non-stationarity. One manner in which homogeneous non-stationarity can be corrected is by computing differences between consecutive observations in order to stabilize the time series. By incorporating such differencing into the previously defined ARMA model, we obtain a fairly robust time series model for non-stationary data.

Definition 1.2.4. *The autoregressive integrated moving average model, ARIMA(p,d,q), is defined as*

$$\phi_p(B)(1 - B)^d X(t) = \theta_0 + \theta_q(B)\epsilon(t).$$

For values of $d \geq 1$, θ_0 represents the deterministic trend term and is omitted from the model unless the data has a clear deterministic trend that we are interested in modeling.

1.3. Principal Component Analysis

Principal component analysis is a data visualization method whose goal is to reduce the dimensionality of a data set by performing a transformation on the original variables. These transformed variables, the principal components, are uncorrelated and are ordered so that the first few retain most of the variation present in the original variables.

Definition 1.3.1. *Principal components* $\boldsymbol{Y} = (Y_1, Y_2..., Y_N)$ *that result from the transformation of a data set composed of N-dimensional vectors* $\boldsymbol{X} = (X_1, X_2..., X_N)$ *are defined to be vectors that result from a transformation in which* $Var(Y_i)$ *is maximized for i=1,...,N subject to the constraint that* $Cov(Y_i, Y_j) = 0$ *for j<i.*

Since the variability present in the original data is translated directly to the variability of the principal components, the dimensionality reduction aspect can be achieved by retaining $p < N$ principal components that explain the majority

of the variance present in the original data set. The remaining $N-p$ principal components can simply be discarded. It is this property of principal components that make it such a powerful data visualization tool.

Theorem 1.3.1. *For i=1,...,N the kth principal component, denoted* $Y_k = \boldsymbol{\alpha}_i \cdot \boldsymbol{X}$*, has a loading vector* $\boldsymbol{\alpha}_i$ *equal to the eigenvector corresponding to the kth largest eigenvalue,* λ_k *of the variance-covariance matrix,* $\Sigma = Var(\boldsymbol{X})$*. Furthermore, if* $\boldsymbol{\alpha}_i$ *has unit length (*$\boldsymbol{\alpha_i'} \cdot \boldsymbol{\alpha_i} = 1$*), then* $Var(Y_k) = \lambda_k$*.*

Proof. We begin with the first principal component Y_1. The goal is to maximize

$$Var(Y_1) = Var(\boldsymbol{\alpha_1}\mathbf{X}) = \boldsymbol{\alpha_1'}Var(\mathbf{X})\boldsymbol{\alpha_1'} = \boldsymbol{\alpha_1'}\boldsymbol{\Sigma}\boldsymbol{\alpha_1}$$

with respect to $\boldsymbol{\alpha_1}$, subject to the constraint that $\boldsymbol{\alpha_1'} * \boldsymbol{\alpha_1} = 1$. We resort to the method of Lagrange multipliers to achieve this end.

The function we wish to maximize reduces to

$$f(\boldsymbol{\alpha_1}) = \boldsymbol{\alpha_1'}\boldsymbol{\Sigma}\boldsymbol{\alpha_1} - \lambda_1(\boldsymbol{\alpha_1'}\boldsymbol{\alpha_1} - 1).$$

Taking the derivative with respect to $\boldsymbol{\alpha_1}$ and setting the result to 0 yields

$$f'(\boldsymbol{\alpha_1}) = \boldsymbol{\Sigma}\boldsymbol{\alpha_1} - \lambda_1\boldsymbol{\alpha_1} = 0$$
$$\boldsymbol{\Sigma}\boldsymbol{\alpha_1} = \lambda_1\boldsymbol{\alpha_1},$$

which can be recognized as an equation where $\alpha_{\mathbf{1}}$ corresponds to an eigenvector of $\boldsymbol{\Sigma}$ and $\lambda_{\mathbf{1}}$ to the eigenvalue. The question that remains is which eigenvector of $\boldsymbol{\Sigma}$ do we choose? To answer this question, we return to the function that we originally intended on maximizing, which we can now write as:

$$Var(Y_1) = \boldsymbol{\alpha_1'}\boldsymbol{\Sigma}\boldsymbol{\alpha_1} = \boldsymbol{\alpha_1'}\lambda_1\boldsymbol{\alpha_1} = \lambda_1\boldsymbol{\alpha_1'}\boldsymbol{\alpha_1} = \lambda_1$$

We now observe that in order to maximize $Var(Y_1)$ we chose $\lambda_{\mathbf{1}}$ to be the largest eigenvalue of $\boldsymbol{\Sigma}$ and $\boldsymbol{\alpha_1}$ the corresponding eigenvector.

By repeating this process, starting with Y_1 and ending with Y_N, each time choosing the next largest eigenvalue and corresponding eigenvector, we obtain principal components $\mathbf{Y} = (Y_1, Y_2..., Y_N)$ with progressively decreasing variances, $Var(Y_1) = \lambda_1 > Var(Y_1) = \lambda_2 >, ..., > Var(Y_N) = \lambda_N$.

Additionally, because the matrix Σ is a variance-covariance matrix and hence a symmetric matrix, its eigenvectors can be chosen to be orthogonal and hence have a dot product of 0, $\boldsymbol{\alpha_i'}\boldsymbol{\alpha_j} = 0$ for $i \neq j$. Since $Cov(Y_i, Y_j) = Cov(\boldsymbol{\alpha_i'}\mathbf{X}_i, \boldsymbol{\alpha_j'}\mathbf{X}_j) = \boldsymbol{\alpha_i'}\boldsymbol{\Sigma}\boldsymbol{\alpha_j} = \boldsymbol{\alpha_i'}\lambda_j\boldsymbol{\alpha_j} = \lambda_j\boldsymbol{\alpha_i'}\boldsymbol{\alpha_j}$, $Cov(Y_i, Y_j) = 0$ for orthogonal eigenvectors $\boldsymbol{\alpha_i}$ and $\boldsymbol{\alpha_j}$.

□

In order to reduce the dimensionality of our data set from N to $P < N$ we can simply decide the proportion of explained variance we wish to retain, which here can be represented by $(\lambda_1 + ... + \lambda_P)/(\lambda_1 + ... + \lambda_N)$ and only keep the corresponding components $Y_1, ..., Y_P$.

Now, despite the inherent statistical nature of principal components, the derivation shows that principal components are the result of an orthogonal linear transformation of a set of vectors optimizing certain algebraic criteria. As a result of this, several algebraic properties result from this derivation, the best known of which is certainly the spectral decomposition theorem.

Theorem 1.3.2. *The variance-covariance matrix of vectors* $\boldsymbol{X} = (X_1, X_2..., X_N)$ *can be written*

$$\Sigma = \lambda_1\boldsymbol{\alpha_1} \cdot \boldsymbol{\alpha_1'} + \lambda_2\boldsymbol{\alpha_2} \cdot \boldsymbol{\alpha_2'} + ... + \lambda_p\boldsymbol{\alpha_N} \cdot \boldsymbol{\alpha_N'}. \tag{1.3.1}$$

Proof. From the derivation of principal components we have directly that

$\Sigma A = A\Lambda$,

where

A is a $p \cdot N$ matrix whose columns correspond to the loading vectors α_k,

Λ is a diagonal matrix whose kth diagonal element is λ_k.

We can therefore write $\Sigma = A\Lambda A'$ and the result follows directly.

□

From the spectral decomposition theorem it becomes clear that the variance-covariance of the original variables can be reconstructed with knowledge of the values of the loading vectors (eigenvectors) as well as the variances of the principal components (eigenvalues).

Another point to be made about principal component analysis is that it is in many cases desirable to first standardize the original variables before deriving the associated principal components, thereby deriving the principal components of the correlation matrix of the original variables. In other words, it can be desirable to derive the principal components $Z = A'\mathbf{X}^*$ where ${X_i}^* = \frac{X_i}{\sqrt{Var(X_i)}}$ for $i = 1, ..., N$. The major advantage of standardizing the variables before the application of principal component analysis is to work with unitless measures. Due to different units of measure that could potentially be used for the various components of the original variables the results become difficult to interpret. Additionally, large differences between the variances of the components of $\mathbf{X}$ can cause those variables whose variances are largest to dominate the first few PCs. By standardizing the components relative to each vector, and hence working with the decomposition of the correlation matrix, these issues can be circumvented.

1.4. Hierarchical Clustering

Hierarchical clustering is a clustering method which is divided in two categories: Divise and agglomerative.

Definition 1.4.1. *Divisive clustering is a top down strategy based on starting with a single cluster which consists of all n data vectors and successively breaking it down into finer groups until n clusters are formed, each containing one data vector.*

Definition 1.4.2. *Agglomerative clustering is a bottom up strategy which consists of fusing n data vectors into successively larger clusters until a single cluster is formed containing all n data vectors.*

Since agglomerative methods are more commonly used and will be applied in this study in conjunction with self-organizing maps, they will be the focus of my analysis.

Agglomerative clustering consists of the following procedure:

(1) The clusters are initialized by assigning each vector to its own cluster, creating n separate clusters, each with one data point
(2) A distance measure between all the clusters is computed
(3) The two clusters with the smallest distance between them are merged together to form a new single cluster
(4) Step 2 is repeated until there is a single cluster remaining, containing all n vectors.

Differences between various agglomerative hierarchical clustering methods arise from the choices of inter-cluster proximity measures which we aim to minimize at each step of the process. Following is a table with the most common proximity measures used in hierarchical clustering and the minimization criterion for fusing clusters at each step of the process.

Method	Inter-cluster proximity
Single linkage	Smallest inter-cluster distance between all containing data vectors
Complete linkage	Largest inter-cluster distance between all containing data vectors
Average linkage	Average inter-cluster distance between all containing data vectors
Centroid linkage	Inter-cluster distance between means of all vectors belonging to each cluster
Median linkage	Weighted inter-cluster distance between means of all vectors belonging to each cluster
Ward's method	Increase in the total within-cluster error sum of squares

The distinction between the various proximity measures has been elegantly summarized by Lance and Williams (1967) via a single parametrized measure.

Definition 1.4.3. *The distance between a group k and a group (ij) formed by the fusion of two groups (i and j) can be written as*

$$d_{k(ij)} = \alpha_i d_{ki} + \alpha_j d_{kj} + \beta d_{ij} + \gamma |d_{ki} - d_{kj}| \tag{1.4.1}$$

where d_{ij} is the distance between groups i and j.

Now, following is a table with the most common distance measures used in hierarchical clustering along with their descriptions and their parameters in Lance

and Williams (1967) generalized distance measure.

Method	α_i	β	γ
Single linkage	1/2	0	-1/2
Complete linkage	1/2	0	1/2
Average linkage	$n_i/(n_i+n_j)$	0	0
Centroid linkage	$n_i/(n_i+n_j)$	$-n_in_j/(n_i+n_j)^2$	0
Median linkage	1/2	-1/4	0
Ward's method	$(n_k+n_i)/(n_k+n_i+n_j)$	$-n_k/(n_k+n_i+n_j)$	0

Now, when agglomerative hierarchical clustering is performed the end result is one large cluster that contains all the data points. Considering this, the issue that arises is when to cease the process in order to obtain a representative amount of clusters. The most straightforward approach would be visual inspection of a dendrogram, which is essentially a tree diagram representing the clusters that form at every iteration of the algorithm. Cutting the tree at a point at which recombining clusters would seem to be redundant in the context of the problem would be one way to approach this issue. Other methods have been developed that add some objectivity to cutting the tree. Dynamic tree cutting proposed by Langfelder et al. (2008) permits the cutting of the tree at different levels for different branches. The concept revolves around combining and decomposing clusters over many iterations until a stable state is achieved. Due to the multitude of other methods available as well as the vast array of parameters for the cut heights and the cluster sizes that must be chosen, the study remains very much subjective.

1.5. Limit Order Book

The limit order book is a trading method that is used by exchanges that matches customers submitted orders to buy or sell a stock at a particular price and volume on a time priority basis. Essentially, the limit order book consists of two queues, the bid side (the buy side) and the ask side (the sell side). The bid side consists of orders submitted by buyers specifying the desired volume of shares for purchase and a price constraint specifying the maximal price at which a trade can occur. The ask side, on the other hand, consists of orders submitted by sellers specifying the desired volume of shares for sale and a price constraint specifying the minimal price at which a trade can occur.

At a given time t, the limit order book will contain a queue of unexecuted ask orders at prices $\alpha_1(t), \alpha_2(t), ..., \alpha_n(t)$ and another queue of unexecuted bid orders at prices $\beta_1(t), \beta_2(t), ..., \beta_n(t)$ each waiting to be matched with incoming orders. Additionally, we have that the following must hold

$$\beta_n(t) \leq, ..., \beta_2(t) \leq \beta_1(t) < \alpha_1(t) \leq \alpha_2(t), ..., \leq \alpha_n(t). \tag{1.5.1}$$

The bid with the highest limit price, β_1 is called the best bid and the ask with the lowest limit price, α_1 is called the best ask. The best bid and the best ask have priority to trade first and the difference between their corresponding prices constitutes the bid ask spread.

Bids and asks are entered into the limit order book throughout the trading day and are stored in the book until they are removed or traded. A trade takes place either when a bid arrives with a limit price equal to or higher than the price of the best ask or when an ask arrives with a limit price equal to or lower than the price of the best bid. This triggers an order matching mechanism in which one or several limit orders are executed on a price-time priority basis, based on the volume desired by the trade initiator. We denote $V_{\beta_n}(t)$, the volume of a stock at time t at the nth level on the bid side of the limit order book corresponding to the order of price β_n and $V_{\alpha_n}(t)$ the volume of a stock at time t at the nth level on the ask side of the limit order book corresponding to the order of price α_n. The matching mechanism functions as follows:

If the new order is an ask limit order for a volume V_α at the price α then:

- If $\alpha \leq \beta_1$ and $V_\alpha \leq V_{\beta_1}$
  - The new ask is matched with the best unexecuted bid and the two are executed at the bid price β_1.
- If $\alpha \leq \beta_1$ and $V_\alpha > V_{\beta_1}$
  - V_{β_1} of the total ask volume is matched with the best unexecuted bid and executed at the bid price β_1.

- The remaining volume $V_\alpha - V_{\beta_1}$ is then matched against the subsequent bids until an $m \in \mathbb{Z}$ is reached such that $V_\alpha \leq \sum_{n=1}^{m} V_{\beta_n}$ under the constraint that $\alpha \leq \beta_m$.

- If for $m \in \mathbb{Z}$, $\beta_{m+1} < \alpha \leq \beta_m$ and $V_\alpha > \sum_{n=1}^{m} V_{\beta_n}$, $\sum_{n=1}^{m} V_{\beta_n}$ is executed at the corresponding bid prices and the remaining volume $V_\alpha - \sum_{n=1}^{m} V_{\beta_n}$ joins the queue of unexecuted asks in the order book.

- If $\alpha > \beta_1$, no match is possible and the new ask joins the queue of unexecuted asks in the order book

If the new order is an bid limit order for a volume V_β at the price β then:

- If $\beta \geq \alpha_1$ and $V_\beta \leq V_{\alpha_1}$
  - The new bid is matched with the best unexecuted ask and the two are executed at the ask price α_1.
- If $\beta \geq \alpha_1$ and $V_\beta > V_{\alpha_1}$
  - V_{α_1} of the total bid volume is matched with the best unexecuted ask and executed at the ask price α_1.

  - The remaining volume $V_\beta - V_{\alpha_1}$ is then matched against the subsequent asks until an $m \in \mathbb{Z}$ is reached such that $V_\beta \leq \sum_{n=1}^{m} V_{\alpha_n}$ under the constraint that $\beta \geq \alpha_m$.

  - If for $m \in \mathbb{Z}$, $\alpha_m \leq \beta < \alpha_{m+1}$ and $V_\beta > \sum_{n=1}^{m} V_{\alpha_n}$, $\sum_{n=1}^{m} V_{\alpha_n}$ is executed at the corresponding ask prices and the remaining volume $V_\beta - \sum_{n=1}^{m} V_{\alpha_n}$ joins the queue of unexecuted bids in the order book.
- If $\beta < \alpha_1$, no match is possible and the new ask joins the queue of unexecuted bids in the order book

We also note that if two bids (asks) are submitted at the same price, $\beta_n(t) = \beta_m(t)(\alpha_n(t) = \alpha_m(t))$, trade priority goes to those that were first entered into the queue. The trade is called buyer-initiated if the initiating order was a buy order (execution of ask limit orders), and seller-initiated if the initiating order was a sell order (execution of bid limit orders) and the intraday quoted price of a stock corresponds to the last known execution price in the limit order book. Additionally, during trading, a limit order at a competitive price will be immediately executed against the best available bid or ask just as though it were a

market order. Consequently, using the previously defined terminology we define the price of execution of a market order, which is an order to buy (sell) a specified quantity of a stock at the best price currently available, as follows:

Definition 1.5.1. *A market order to sell a volume of V_α of a stock is executed at a price of:*

$$\alpha_{MO} = \sum_{n=1}^{m-1} \beta_n \cdot V_{\beta_n} + (V_\alpha - \sum_{n=1}^{m-1} V_{\beta_n}) \cdot \beta_m \tag{1.5.2}$$

$$for$$

$$\sum_{n=1}^{m-1} V_{\beta_n} \leq V_\alpha < \sum_{n=1}^{m} V_{\beta_n}.$$

Definition 1.5.2. *A market order to buy a volume V_β of a stock is executed at a price of:*

$$\beta_{MO} = \sum_{n=1}^{m-1} \alpha_n \cdot V_{\alpha_n} + (V_\beta - \sum_{n=1}^{m-1} V_{\alpha_n}) \cdot \alpha_m \tag{1.5.3}$$

$$for$$

$$\sum_{n=1}^{m-1} V_{\alpha_n} \leq V_\beta < \sum_{n=1}^{m} V_{\alpha_n}.$$

Chapter 2

NEURAL NETWORKS

A neural network, the class of machine learning algorithm that I will be implementing in my study, is a model that consists of a collection of neurons that attempts to artificially reproduce the manner in which cognition works in the human brain. Similar to the brain, an artificial neural network has the ability to store knowledge and then make it available for use. Analogous to the human brain, an artificial neural network has a collection of neurons with synaptic weights that are used to acquire knowledge from its environment, which is known as the training phase of the network. Through these weights, the network stores this information and is then capable of generalizing and drawing conclusions on previously unseen information presented to the neural network. This stage of implementing the network is known as the testing phase.

Neural networks are powerful tools used in modeling as they are capable of capturing non-linear relationships in data. These non-linear relationships can be captured by the neural network's capability of building input-output mappings without specifying any assumptions for the statistical model on the input data. From this perspective, using neural networks to model data falls in the paradigm of non-parametric statistical inference.

In this section, a description of the learning method known as competitive learning will be presented as this will be the general learning method I will be using to train the network. A detailed description of training and testing such a network will be presented. Following this, I will present the self-organizing map and a description of its learning algorithm in both its original unsupervised implementation as well as a supervised variant known as the X-Y fused SOM. Finally, I will define the issue of overfitting and tests that can be used to test for such a phenomenon.

2.1. Learning Methods

In the context of neural networks, learning is defined by the manner in which the synaptic weights of the network are adapted to the data. The process of learning is comprised of a neural network first being stimulated by a set of input data, undergoing changes in its weights and then responding to a new set of input data through these newly developed weight values. A well defined set of rules which a neural network follows during the training phase characterizes the learning method of the network. In my study, I will focus on a type of learning method known as competitive learning.

2.1.1. Competitive Learning

Competitive learning is a type of learning algorithm for neural networks that is well suited for finding clusters within a data set. As stated by Haykin (1999) a competitive learning algorithm is characterized by the following:

Definition 2.1.1. *Competitive learning is characterized by three basic features:*

- *The neurons in the network are indistinguishable except for a random initialization of the synaptic weights that allow them to respond differently to input data.*
- *There is a limit to the strength of the synaptic weights of the neurons.*
- *The neurons in the network compete for the right to respond to a subset of the input data in a manner that only one neuron wins the competition, the so called "winner take all" neuron.*

A competitive neural network consists of a single layer of output neurons, each of which is connected to every single input value. These connections from the input values to the neurons are known as feed forward connections as they stimulate the neurons to become active. The neurons themselves are connected to each other through lateral connections that permit them to interact during the training phase.

Each neuron in the network is assigned a synaptic weight denoted $\mathbf{m}_j = < m_{j1}, m_{j2}, ..., m_{jN} >$ and the competition takes place by computing, for each input

vector $\mathbf{X}_i =< X_{i1}, X_{i2}, ..., X_{iN} >$, a similarity measure between the input and each neuron. The winning neuron is then determined as the one with the largest similarity measure with a given input vector. For input vector $\mathbf{X}_i$, the output of winning neuron m is then:

$$m_{ji} = \begin{cases} 1, & \text{for input vector } \mathbf{X}_i \\ 0, & \text{otherwise.} \end{cases} \tag{2.1.1}$$

Additionally, the standard competitive rule for updating the synaptic weight vectors of the neurons in the network is the following:

$$\Delta \mathbf{m}_j = \begin{cases} \alpha \cdot (\mathbf{X}_i - \mathbf{m}_j), & \text{for winning neuron } \mathbf{m}_j \\ 0, & \text{otherwise.} \end{cases} \tag{2.1.2}$$

The parameter α is known as the learning-rate parameter.

2.2. Self Organizing Maps

A self-organizing map is a type of competitive neural network in which the neurons are positioned at the nodes of a lattice and compete for the right to be activated by the input vectors. A defining characteristic of the self-organizing map, as put by Kohonen (1990), is that through its implementation a topographic map is created for the input patterns in which the locations of the neurons in the lattice are indicative of features and patterns contained in the input vectors. This formulation of the self-organizing map which consists of a two-dimensional map in which there is no specific distinction between dependent and independent variables in the input vectors is the original, deemed unsupervised, variant of the algorithm.

2.2.1. Unsupervised Maps

The steps to training a self-organizing map can be broken down into three components: The competitive phase, the cooperative phase, and synaptic adaptation phase. We now describe each step in detail.

Competitive Phase

During the competitive phase of training a self-organizing map, the input vectors are presented to the network one at a time and a similarity measure is computed with each neuron in the network. The neuron whose synaptic weight

maximizes the value of the similarity measure is deemed the winning neuron. Depending on the context either the location on the grid of the winning neuron $i(x)$ or its associated synaptic weight is deemed the response of the network. In my study, I will use the inverse of the Euclidean distance as the similarity measure. In other words, the winning neuron is that whose synaptic weight $\mathbf{m}_i$ minimizes the Euclidean distance with the input vector $\mathbf{X}_j$. The Euclidean distance can be defined as:

Definition 2.2.1. *The Euclidean distance between vectors* $\boldsymbol{X} =< X_1, .., X_N >$ *and* $\boldsymbol{Y} =< Y_1, ..., Y_N >$ *is defined as*

$$d(\boldsymbol{X}, \boldsymbol{Y}) = \sqrt{\sum_{i=1}^{N}(x_i - y_i)^2} \tag{2.2.1}$$

Cooperative Phase

Once the winning neuron is determined, the next phase of the process consists of updating its synaptic weights. An additional level of sophistication to that of a standard competitive learning algorithm that the self-organizing map has is that the neurons in the vicinity of the winning neuron $i(x)$ also have their synaptic weights updated. The winning neuron can be seen as the neuron that constitutes the center of the neighborhood of cooperating neurons. To this end, the cooperative phase consists of determining which neurons are updated and quantifying a relationship between them. The definition of a neighborhood function becomes necessary to define the extent to which neighbors of the winning neuron $i(x)$ are updated.

Definition 2.2.2. *A neighborhood function for winning neuron* $i(x)$ *is defined as:*

$$h_{j,i(x)} = f(d_{j,i(x)}) \tag{2.2.2}$$

where $d_{j,i(x)}$ *denotes the lateral distance between winning neuron* $i(x)$ *and neuron* j *in a defined neighborhood,* σ*, of* $i(x)$ *and* $h_{j,i(x)}$ *must satisfy the following conditions:*

- $h_{j,i(x)}$ *attains a maximum value for winning neuron* $i(x)$
  i.e $max(h_{j,i(x)}) = h_{i(x),i(x)} = f(0)$.

- $h_{j,i(x)}$ *is a monotonically decreasing function of* $d_{j,i(x)}$
  i.e for $d_{j_1,i(x)} \leq d_{j_2,i(x)}$, $h_{j_1,i(x)} \geq h_{j_2,i(x)}$.

We note that the lateral distance between nodes $i(x)$ and j, $d_{j,i(x)}$ is a function of the topology of the network. For neurons $i(x)$ and j at positions $(x_{i(x)}, y_{i(x)})$ and (x_j, y_j) on the grid, $d_{j,i(x)}$ is taken as the Euclidean distance between these positions i.e. $d_{j,i(x)} = \sqrt{(y_j - y_{i(x)})^2 + (x_j - x_{i(x)})^2}$. In my study, I will attempt to model the neighborhood function with two functions that satisfy the previous requirements. I will then analyze the trade-off between accuracy of prediction and computational efficiency between the two approaches.

- Bubble neighborhood function

$$h_{j,i(x)} = \begin{cases} 1, & \text{for neurons j whose lateral distance } d_{j,i(x)} < \sigma \\ 0, & \text{otherwise.} \end{cases} \tag{2.2.3}$$

- Gaussian neighborhood function

$$h_{j,i(x)} = \begin{cases} e^{-d^2_{j,i(x)}/2\sigma^2}, & \text{for neurons j whose lateral distance } d_{j,i(x)} < \sigma \\ 0, & \text{otherwise.} \end{cases} \tag{2.2.4}$$

Adaptive Phase

The adaptive phase of the self-organizing map algorithm consists of the winning neuron and its topological neighbors within a vicinity σ updating their synaptic weights based on an updating rule. The updating rule for the self-organizing map can be seen as a modified rule, with an additional level of sophistication, when compared to standard competitive learning, by replacing the learning parameter α with $\alpha \cdot h_{j,i(x)}$.

$$\Delta \mathbf{m}_j = \begin{cases} \alpha \cdot h_{j,i(x)}(\mathbf{X}_i - \mathbf{m}_j), & \text{for neurons } j \text{ whose lateral distance } d_{j,i(x)} < \sigma \\ 0, & \text{otherwise.} \end{cases} \tag{2.2.5}$$

In discrete time, if the synaptic weight of neuron j is denoted $\mathbf{m}_j(t)$ the adaptive phase of the algorithm can be denoted as:

$$\mathbf{m}_j(t+1) - \mathbf{m}_j(t) = \begin{cases} \alpha(t) \cdot h_{j,i(x)}(t)(\mathbf{X}_i - \mathbf{m}_j(t)), & \text{for neurons } j \text{ whose lateral distance } d_{j,i(x)} < \sigma(t) \\ 0, & \text{otherwise.} \end{cases} \tag{2.2.6}$$

In this formalism of the adaptive phase of the self-organizing map, $\alpha(t)$, $\sigma(t)$ and consequently $h_{j,i(x)}(t)$ are monotonically decreasing functions of t.

The testing phase of the self-organizing map can then be seen as a non-linear mapping from the continuous input space η defined by the relationship of the input vectors and the discrete output space α defined by the relationship of the synaptic weights vectors arranged at the nodes of a lattice:

$$\phi : \eta \rightarrow \alpha \tag{2.2.7}$$

For a given set of input vectors $\mathbf{X}_i \in \eta$ the SOM algorithm first performs the non linear mapping ϕ through the determination of the winning neuron $i(x)$. The synaptic weight belonging to this neuron can then be viewed as a pointer back into the input space. In this way the SOM algorithm is able to provide a small set of synaptic weight vectors $\mathbf{m}_j \in \alpha$ that provide a good approximation of the probability distribution of the input space, which consists of a larger set of input vectors $\mathbf{X}_i \in \eta$. While techniques such as principal component analysis can achieve this for data forming a plane in the input space, due to the topological ordering property of the self-organizing map, it can be seen as a generalization

to non-linear density estimation.

The steps of the SOM leaning algorithm can be summarized as follows:

Summary

(1) **Initialization** The number of clusters is chosen and the map's synaptic weights, $\mathbf{m}_j(0) \in R^N$, are initialized such that $\mathbf{m}_j(0) \neq \mathbf{m}_k(0)$ for $j \neq k$

(2) **Sampling** The data set $\mathbf{X}_i \in R^N$ is presented to the network one vector at a time through random sampling of the inputs.

(3) **Competitive Phase** For each $\mathbf{X}_i$, distances between $\mathbf{X}_i$ and all the synaptic weights are computed. The winning neuron is chosen as the neuron whose synaptic weight $\mathbf{m}_{i(x)}$ has the smallest Euclidean distance to $\mathbf{X}_i$, in other words $\mathbf{m}_{i(x)}$ is chosen amongst all $\mathbf{m}_j$ such that $D = ||\mathbf{X}_i - \mathbf{m}_j||$ is at a minimum

(4) **Cooperative Phase** A radius $\sigma(t)$ is established around the winning neuron $i(x)$ which in turn is assigned a neighborhood function $h_{j,i(x)} = f(d_{j,i(x)})$, where $h_{j,i(x)}$ is a monotonically decreasing function of $d_{j,i(x)}$, the distance between winning neuron $i(x)$ and neurons j, for all neurons j within a distance $\sigma(t)$ of $i(x)$.

(5) **Adaptive Phase** The winning neuron and its topological neighbors are updated by being moved closer to the input vector in the input space. The rule used to update the vectors in the vicinity of the winning neuron is the following:

$$\mathbf{m}_j(t+1) = \mathbf{m}_j(t) + \alpha(t) \cdot h_{j,i(x)} \cdot (\mathbf{X}_i - \mathbf{m}_j(t)), \tag{2.2.8}$$

where $\alpha(t)$ is the learning rate.

Note: $\alpha(t)$ and $\sigma(t)$ are monotonically decreasing functions of t

(6) The time step t is increased and the next vector is presented to the network. This process is repeated for as many iterations as required until

convergence.

It must be noted that the SOM algorithm was formalized from an intuitive perspective and not from the derivation of a specific error function. As put by Kohonen et al. (1991), it just so happens to be true that the optimization of an error functional leads to the SOM algorithm and the heuristically established algorithm reflects topological relationships between clusters of data without an explicit mathematical formalization. However, we follow Kohonen et al. (1991) in his approach to establish a formalism and derive a slightly modified SOM algorithm. In order to follow this approach several definitions must be established.

Definition 2.2.3. *Let $\boldsymbol{X}_i \in R^N$ be a vector and $\boldsymbol{m}_{i(x)} \in R^N$ the associated synaptic weight vector with smallest Euclidean distance amongst all synaptic vectors. We define the locally weighted mismatch of $\boldsymbol{X}_i$ with respect to the winner as*

$$E = \sum_j h_{ji(x)} ||\boldsymbol{X}_i - m_j||^2 \tag{2.2.9}$$

where $h_{ji(x)}$ describes the interaction of winning synaptic weight vector $m_{i(x)}$ and synaptic weight vectors m_j during the training phase.

The locally weighted mismatch for $\mathbf{X}_i$ is essentially a sum of the distances between the input vector $\mathbf{X}_i$ and all synaptic weight vectors m_j, weighted by the neighborhood function of closest synaptic weight vector to $\mathbf{X}_i$ applied to m_j.

Now, the self-organizing map originated from a concept known as vector quantization. This concept developed in the context of signal processing and consists of partitioning a space of vector valued input data into a finite number of regions, each of which is represented by a single model vector, the centroid. As such, it can be seen as a non-neural network generalization of the self-organizing map algorithm as it is essentially, also, a clustering algorithm used to reduce high-dimensional data. The synaptic weight vectors can therefore be seen as the equivalent of the centroids in vector quantization algorithms.

Since the training phase of the self-organizing map is based upon this principle we define the mean quantization error.

Definition 2.2.4. *We denote the N dimensional input vectors as $\boldsymbol{X}_i \in R^N$ and $\boldsymbol{m}_{i(x)} \in R^N$ the associated centroid. The mean quantization error can then be defined as*

$$E(||\boldsymbol{X}_i - \boldsymbol{m}_{i(x)}||)^2 = \int_i ||\boldsymbol{X}_i - \boldsymbol{m}_{i(x)}||^2 p(\boldsymbol{X}_i) d\boldsymbol{X}_i \tag{2.2.10}$$

where

$p(\boldsymbol{X}_i)$ is the probability density of $\boldsymbol{X}_i$
and $d\boldsymbol{X}_i$ is a differential hyper volume element in the $\boldsymbol{X}_i$ space.

The optimization of this objective function leads to centroids $m_{i(x)}$ whose spatial locations and values can be used to construct density estimators of the corresponding vectors $\mathbf{X}_i$. Now, a type of vector quantization closely related to the SOM is k-means clustering which aims to partition the vectors into k groups such that the sum of squares from points to the assigned cluster centers is minimized.

Definition 2.2.5. *For $i \in \mathbb{Z}$, the k-means clustering algorithm for vectors $\boldsymbol{X}_i$ i=1,...,n is defined by centroids $m_{i(x)}$ defined the minimization of the following objective function*

$$E(||\boldsymbol{X}_i - \boldsymbol{m}_{i(x)}||)^2 = \sum_{i=1}^{n} ||\boldsymbol{X}_i - \boldsymbol{m}_{i(x)}||^2 / N. \tag{2.2.11}$$

We note that there are a wide variety of algorithms that minimize (2.2.11) such as those developed by MacQueen (1967), Lloyd (1957) and Forgy (1965). In order to draw a parallel to the self-organizing map, I present Kohonen's algorithm, which leads to the formation of centroids that minimize (2.2.11) with respect to $i(x)$.

Theorem 2.2.1. *The optimal values of $\boldsymbol{m}_{i(x)}$ that minimize (2.2.11) are the values of $\boldsymbol{m}_j$ that result from the convergence of the following iterative algorithm*

$$\boldsymbol{m}_j(t+1) = \boldsymbol{m}_j(t) + \alpha(t) \cdot \delta_{ji(x)} \cdot (\boldsymbol{X}_i - \boldsymbol{m}_j(t)) \tag{2.2.12}$$

where $\delta_{ji(x)}$ is the kronecker delta,
$0 < \alpha(t) < 1$,

$\sum_{t=0}^{\infty} \alpha(t) = \infty \quad \textit{and} \quad \sum_{t=0}^{\infty} \alpha(t)^2 < \infty.$

Proof. For a formal proof using both a stochastic approximation method as well as the true steepest descent optimization we refer to Kohonen et al. (1991).

□

We note that in this formulation, the only distinction between the k-means and the SOM is the width of the neighborhood kernel, which reduces to 1 for the neuron whose synaptic weight vector is closest to $\mathbf{X}_i$ and 0 elsewhere in the k-means algorithm.

By combining the locally weighted mismatch function with the mean quantization error we now obtain the objective function that we are then interested in minimizing to obtain the weights for the self-organizing map. The difference between this objective function and that of the k-means is the distance of each input from all of the synaptic weight vectors instead of just the closest one is taken into account, weighted by the neighborhood function.

$$E = \int_i \sum_j h_{ji(x)} ||\mathbf{X}_i - \mathbf{m}_j||^2 p(\mathbf{X}_i) d\mathbf{X}_i \tag{2.2.13}$$

Kohonen et al. (1991) showed that the asymptotic values m_∞ of the following iterative algorithm define the set of $m_{i(x)}$ that globally minimize E.

$$\mathbf{m}_j(t+1) = \begin{cases} \mathbf{m}_j(t) + \alpha(t) \cdot [h_{i(x)i(x)} \cdot (\mathbf{X}_i - \mathbf{m}_{i(x)}(t)) - \frac{1}{2} \sum_{k \neq i(x)} h_{i(x)k} \cdot (\mathbf{X}_i - \mathbf{m}_k(t))] \\ \mathbf{m}_j(t) + \alpha(t) \cdot h_{ji(x)} \cdot (\mathbf{X}_i - \mathbf{m}_j(t)) \ for \ j \neq i(x) \end{cases} \tag{2.2.14}$$

The difference between this algorithm and that of the SOM is that on each iteration the neuron whose synaptic weight vector is closest to $\mathbf{X}_i$ and its topological neighbors are updated based on slightly different principles. Due to this, the asymptotic value of the m_j in the original SOM algorithm do not coincide perfectly with those that minimize (2.2.13). However, numerical simulations have shown that the practical differences between the two versions of the SOM are negligible particularly if the time invariant neighborhood function is replaced by a time variant counterpart that decreases monotonically with each iteration.

We now refer to properties of the self-organizing map as presented by Haykin (1999). These properties result both from the author's derivation of the map from first principles as well as empirical results. For a formal proof of the following properties I invite the reader to consult Haykin (1999).

(1) **Approximation of the Input Space**
The feature map $\phi : \eta \to \alpha$, represented by the synaptic weight vectors in the output space, $\mathbf{m}_j$, provides a good approximation to the input space η.

(2) **Topological Ordering**
The map approximates the input space η through synaptic weight vectors $\mathbf{m}_j$ in such a way that the spatial location of the corresponding neurons in the lattice, (x_j, y_j), correspond to a particular domain or features in the input space.

(3) **Density Matching**
Regions in the input space η from which sample vectors $\mathbf{X}$ are drawn with high probability of occurrence are mapped onto larger domains of the output space α.

(4) **Feature Selection**
Given data from an input space with a non-linear distribution, $f(\mathbf{X})$, the self-organizing map is able to select a set of best features, $\mathbf{m}_{i(x)}$, for approximating the underlying distribution.

Additionally, I refer to computer simulations conducted by Haykin (1999) in which the properties of the SOM were studied in the context of a two dimensional map driven by a two dimensional distribution. The author simulates vectors $\mathbf{X}$ drawn from a two-dimensional uniform distribution on [-1,1] and then clusters the vectors using a SOM algorithm driven by a $10 \cdot 10$ map. Visual inspection of the vectors $\mathbf{X}$ and well as $\mathbf{m}_{i(x)}$ in the input space showed that the statistical distribution of the neurons in the map approached that of the input vectors.

2.2.2. Supervised Maps

Now that the fundamentals of the self-organizing map have been presented, my focus now shifts on developing a supervised variant of the SOM for predictive modeling. The idea behind using supervised self-organizing map formulations is to capture non-linear relationships between input and output variables, while

in addition preserving the topology present in the data, properties that classical supervised learning models such as multiple linear regression fail to possess. Additionally, as I will present in chapter 3, supervised SOM variants enable the modeling of non-stationary time series data without the need for any adjustments.

In classical supervised regression models, parameters, $\beta_1, ..., \beta_n$, are estimated using a training set of data to develop a function that quantifies the relationship between the set of independent and dependent variables. These are usually estimated using assumptions of statistical distributions on the error terms and applied to the independent variables in a testing set of data to get an out of sample estimate of the dependent variable. The use of such models requires having a preconceived notion of the type of relationship between the independent and dependent variables and in most cases this involves the assumption of linearity. Since a self-organizing map is a type of machine learning algorithm, no such assumptions are necessary and the parameters that are estimated, the synaptic weight vectors, can be used to build a predictive model free of these constraints.

X-Y fused SOM

While the models presented in chapter 3 are variations of the classical SOM algorithm found in literature specifically used for time series forecasting, here I present a model developed by Melssen et al. (2006) whose use extends to any regression or classification problem, the X-Y fused SOM. The X-Y fused SOM involves simultaneously training two separate maps of equal dimensions: an input X-map for the independent variables and an output Y-map for the dependent variable, which are concatenated together, forming a combined input-output map. To take into account the non-linear relationship between input and output variables, in the X-Y fused model, the training of the maps is guided by a fused similarity measure. For an input-output pair $(\mathbf{X}_i, Y_i)$, the similarity obtained for an input $\mathbf{X}_i$ and the synaptic weights in the X-map, $\mathbf{m}_{j_X}$, is combined with the similarity corresponding to the output Y_i and the synaptic weights in the Y-map, m_{j_Y}, to drive the formation process of the maps. This similarity measure can be defined as follows:

Definition 2.2.6.

$$D = \gamma \cdot ||\boldsymbol{X}_i - \boldsymbol{m}_{j_X}|| + (1 - \gamma) \cdot ||Y_i - m_{j_Y}||, \tag{2.2.15}$$

where γ regulates the relative weight on the input and output mappings.

The common winning unit in the X and Y maps, $(\mathbf{m}_{i(x)}, m_{i(y)})$ is taken as the joint $(\mathbf{m}_{j_X}, m_{j_Y})$ that minimize 2.2.6. The updating of synaptic weight vectors in both maps is done simultaneously and in a manner analogous to that of the unsupervised SOM. Since in such a setup, information present in the inputs and output is used during the update of the weights of the neurons, the formation of the concatenated map is driven by X and Y in a truly supervised way. Additionally the relative weight parameter γ can be adjusted based on the dimension of the vector $\mathbf{X}_i$.

The idea is then to use a testing set of independent variables $\mathbf{X}$, treat the synaptic weight vectors of the winning neurons, $m_{X_{i(x)}}$ in the input space as point estimators of their respective input vectors and the corresponding synaptic weights in the Y space, $m_{Y_{i(y)}}$ are then the estimated predictions.

2.3. Overfitting

In machine learning, the generalization capabilities of a learning algorithm from the data set that is used to estimate its parameters, the training set, is measured by its performance on an independent data set, the testing set. Overfitting is a phenomenon that occurs in this context when a model memorizes the training data as opposed to learning from it and is then unable to generalize on previously unseen data. When overfitting occurs, a statistical model describes random noise in the data instead of the underlying relationship. The contributors to an overfitted model are the number of data points presented to train the model as well as the number of parameters chosen. Underfitting, on the other hand, occurs when a model is unable to capture the relationships even in the training data. Underfitting is a sign that the model is mis-specified and a fundamental change in the specification of the model becomes necessary.

When it comes to fitting models to time-series data, overfitting occurs when patterns in the time series are present in the testing set but not present in the training set. The data in the training set that does not appear in the testing set, on the other hand, is noise and a robust learning algorithm is one that is efficient at reducing the fitting of noise to the trained model. In classical predictive modeling, given an input vector $\mathbf{X}_i$, a response variable Y_i, and a prediction

$\hat{Y}_i$, the expected loss function over a test set for a given training set τ for measuring the errors between Y_i, and $\hat{Y}_i$, can be written as $Err_\tau = E(L_\tau(Y_i, \hat{Y}_i)|\tau)$. The corresponding expected loss function for a test set over arbitrary training sets, in other words averaged over the randomness in a particular training set, can then be written as $Err = E(Err_\tau) = E(L_\tau(Y_i, \hat{Y}_i))$. An effective model in machine learning can minimize this expected loss function through optimization of so called tuning parameters.

Definition 2.3.1. *This error is often represented by the expected square error defined as:*

$$E(L_\tau(Y_i, \hat{Y}_i)) = E((Y_i - \hat{Y}_i)^2) \tag{2.3.1}$$

To better understand the phenomenon of overfitting, particularly in the context of neural networks we begin with a definition of the bias-variance trade off. The bias variance trade off can be understood as the decomposition of the expected error as:

Theorem 2.3.1.

$$Err = E((Y_i - \hat{Y}_i)^2) = Bias(\hat{Y}_i)^2 + Var(\hat{Y}_i) + \sigma^2 \tag{2.3.2}$$

where

$$Bias(\hat{Y}_i) = E(\hat{Y}_i - Y_i)$$

$$Var(\hat{Y}_i) = E(\hat{Y}_i^2) - E(\hat{Y}_i)^2$$

σ^2 *represents the irreducible error variance and can also be seen as a lower bound on the expected error on unseen testing sets*

Proof. $E(Y_i - \hat{Y}_i)^2 =$

$$E(Y_i^2 + \hat{Y}_i^2 - 2 \cdot Y_i \cdot \hat{Y}_i) = E(Y_i^2) + E(\hat{Y}_i^2) - 2 \cdot E(Y_i \cdot \hat{Y}_i) =$$

$$Var(Y_i) + E(Y_i)^2 + Var(\hat{Y}_i) + E(\hat{Y}_i)^2 - 2 \cdot E(Y_i \cdot \hat{Y}_i) =$$

$$Var(Y_i) + Var(\hat{Y}_i) + (E(Y_i^2) + E(\hat{Y}_i)^2 - 2 \cdot E(Y_i \cdot \hat{Y}_i))=$$

$$\sigma_{Y_i}^2 + Var(\hat{Y}_i) + Bias(\hat{Y}_i)^2$$

□

As a model increases in complexity it can better adapt to the data presented in the training set, thus increasing variance but decreasing bias. On the other hand, a less complex model is less adapted to the training data, producing less variance, although increasing bias. Thus, high variance produces overfitting while high bias produces underfitting. The goal is to determine an intermediate model complexity that gives minimum expected test error while producing results within a reasonable computation time.

When it comes to competitive neural networks, however, we are not modeling a predictive relationship but rather applying a neural network approach to clustering using a set of synaptic weights. The synaptic weights can be seen as point estimators for the input vectors contained by their corresponding neurons. In the case of unsupervised self-organizing maps the synaptic weight vectors, $\mathbf{m}_{i(x)}$ can be seen as estimators for vectors $\mathbf{X}_i$ and in the case of supervised SOMs $(\mathbf{m}_{i(x)}, m_{i(y)})$ are estimators for $(\mathbf{X}_i, Y_i)$.

Drawing an analogy to the previous definition of bias variance trade off, in the case of competitive learning algorithms, the inputs are the $\mathbf{X}_i$, the outputs are the best matching synaptic weight vectors $\mathbf{m}_{i(x)}$ and the error functional can be defined as,

$$Err = E(X_{ij} - \mathbf{m}_{i(x)j})^2 = Bias(X_{ij}^2) + Var(m_{i(x)j}) + \sigma_{X_i}^2$$

for m-dimensional vectors $\mathbf{X}_i$ and $m_{i(x)}$ and $j = 1, ..., m$.

For supervised learning we add the additional term:

$$Err = E(Y_i - m_{i(y)})^2 = Bias(Y_i^2) + Var(m_{i(y)}) + \sigma_{Y_i}^2$$

In both supervised and unsupervised context the balance between bias and variance for self-organizing maps is one that cannot be tackled from a theoretical standpoint due to the fact that there is no closed form that relates the error to the the tuning parameters of the model. However, empirical tests can be conducted

to determine the effect of varying tuning parameters on both the training and testing error, thereby measuring the trade off between variance and bias.

2.3.1. Cross-Validation

The method that I plan to use to control for overfitting is that of cross validation. Cross validation is a technique used to estimate the expected out of sample error $Err = E(Err_{\tau})$. Cross validation consists of partitioning a data set into k roughly equal sized groups, withhold one group and then fit the model to the points belonging to the remaining $k-1$ groups. Following this, testing is done on the withheld group and a loss function computed. This process is then repeated until all k groups have been tested on and the loss function is averaged over the test sets.

Mathematically, let $\hat{Y}^{-k}$ denote the fitted function, computed with the kth part of the data removed. The cross validated prediction error can then be defined as:

Definition 2.3.2.

$$CV(f^{hat}) = 1/N \sum_{i=1}^{N} L(Y_i, \hat{Y}^{-k}) \tag{2.3.3}$$

In the context of competitive neural networks the loss is between the input vectors and best matching synaptic vectors. The neighborhood function, $h_{j,i(x)}$ as a tuning parameter, can be made to vary to determine its effects on the cross-validation error. Additionally, the number of neurons, ψ, being a tuning parameter can be also made to vary to determine its effects on the cross-validation error. We can therefore define the testing error which we are interesting in minimizing, in relation to the training error as:

Definition 2.3.3.

$$CV(f^{hat}, h_{j,i(x)}, \psi) = 1/N \sum_{i=1}^{N} L(Y_i, \hat{Y}^{-k}(h_{j,i(x)}, \psi)) \tag{2.3.4}$$

By varying the number of neurons as well as the neighborhood function for a fixed number of training and testing vectors, we can then observe an empirical

representation of the training error in relation to the testing error, giving an idea of the effect the tuning parameters have on over and under fitting.

Chapter 3

FORECASTING

Forecasting time series has been a topic that has been explored using many different techniques and approaches. The process of forecasting time series involves the use of historical data in order to build a model that can predict future observations. My study will focus specifically on one-step ahead forecasting, meaning, building models that predict one time step into the future.

Definition 3.0.1.

Given a stochastic process $X(\omega, t)$ defined at times $t_1, t_2, ..., t_n$:

$$\{X(t_1), X(t_2), ..., X(t_n)\}$$

The one-step ahead forecast of the process at time t_{n+1} can be defined as

$$\hat{X}(t_{n+1}) = E_\omega(X(t_{n+1})|\{X(t_1), X(t_2), ..., X(t_n)\}). \tag{3.0.1}$$

In this chapter, I will begin by presenting forecasting using the ARIMA model. The ARIMA model is one of the most commonly used time series models used in forecasting future values of a series. For this reason, I will present the manner in which it is implemented, starting with an explanation of the minimum mean square error approach to estimate the parameters. Additionally, I will present the KPSS test, which is used to determine the optimal amount of times differencing is required to negate the effects of homogeneous non-stationarity. Additionally, I will present the AIC criterion, which is used to compare several ARIMA models with varying levels of the parameters p and q in order to chose the most robust model. Following this, I will present an alternative approach to time series prediction, the use of neural networks, specifically the self-organizing map.

The use of self-organizing maps in time series prediction can be seen as a crossover of function approximation and vector quantization. By applying the use of the self-organizing map, which was originally designed as a vector quantization algorithm to time series prediction, which can be seen as a function approximation task, we are venturing into a domain that has been scarcely explored. The main advantages of using vector quantization algorithms in time series prediction mainly revolve around the fact that there is no need to specify any distributional assumptions about the underlying data. That being said, there are certainly further advantages to using the self-organizing map in time series prediction as well.

Firstly, referring to the properties of the map presented in section 2.2, a self-organizing map is a non-linear, topologically preserving mapping from a continuous vector-valued input space η to a discrete space of synaptic weight vectors α. The probability distribution of the weight vectors can thus be seen as matching the probability distribution of the corresponding vectors in the input space. Therefore, for vectors $\mathbf{X}^{in}(t) =< X(t), X(t-1), ..., X(t-h) >$, $t = 1, ..., n$ belonging to cluster i, $F(\mathbf{m}_{i(x)}) \approx F(\mathbf{X}^{in}(t)) \approx F(\mathbf{X}^{in}(t+h)) \approx ...\ for\ \mathbf{X} \in \eta$ for all $h \in \mathbb{R}$ such that $\mathbf{X}^{in}(t+h)$ belongs to cluster i. Referring to the feature selection property, which according to Haykin (1999) is a culmination of the previous properties, the self-organizing map can essentially be seen as detecting regions in the input space, η, where a time series possesses a similar underlying distribution. Following this, the SOM approximates the value of this distribution function through the topology of the map as well as the values of the synaptic weight vectors, $\mathbf{m}_{i(x)}$, which can in turn be used together with a similarity measure, for forecasting. In this sense, the clusters represent localized stationary models which can then be independently used during the testing phase to determine the synaptic weight vector that best represents an out-of-sample series. An approach such as the X-Y fused SOM can then be used to create an additional mapping from the X component of synaptic weight vectors $\mathbf{m}_{i(x)}$ in the $\mathbf{X}$ space to the Y component of synaptic weight vectors $m_{i(y)}$ in the Y space. $m_{i(y)}$, in turn, is assumed to have a marginal probability distribution approximating that of $Y^{out}(t) = X(t+1)$ and therefore the following estimate can be made: $m_{i(y)} = \hat{\mathbf{X}}(t+1)$. In addition to this, the self-organizing map offers tuning parameters which can be adjusted based on the nature and scope of the data, namely the learning rate $\alpha(t)$, the neighborhood function, $h_{i(x)j}(t)$ and the radius $\sigma(t)$.

Although the self-organizing map was originally designed as an unsupervised machine learning algorithm, several attempts have been made to implement it as a supervised variant in time series prediction. I will present the Vector-Quantized Temporal Associative Memory model (VQTAM) in its original implementation as well as the Double Vector Quantization model. Following this, I will show how an X-Y fused SOM is a more generalized model of the VQTAM that should yield improved results and propose its use for time series prediction over the former.

3.1. Maximum Likelihood Forecast

The maximum likelihood forecast is an approach to estimating the parameters $\phi_p(B)$ and $\theta_q(B)$ of the ARIMA model. In order to understand this approach, a definition must be established.

Definition 3.1.1. *An ARIMA process, $\phi_p(B)(1-B)^d X(t) = \theta_q(B)\epsilon(t)$, is said to be invertible if it can be represented in the form:*

$$\pi(B)X(t) = \epsilon(t)$$

where $\pi(B) = 1 - \sum_{j=1}^{\infty} \pi_j B^j = \phi(B)(1-B)^d/\theta(B)$ and $\sum_{j=1}^{\infty} |\pi_j| < \infty$.

We refer to Wei (2006) for a proof of the fact that an ARIMA time series model is invertible if the roots of $\theta_q(B) = 0$ lie outside of the unit circle. Under this condition, we can therefore represent the ARIMA model as:

$$X(t) = \sum_{j=1}^{\infty} \pi_j X(t-j) + \epsilon(t).$$

This representation is crucial in order to be able to make forecasts, since the current value of a time series represented by an ARIMA model can be written as a linear combination of past observations with an additive error term. Additionally, for an invertible process, the π_j weights converge and one could simply take the most recent observations that account for a desired degree of accuracy of the model.

For building a one step ahead forecast we can use the model:

$X(t+1) = \sum_{j=1}^{\infty} \pi_j X(t+1-j) + \epsilon(t+1),$

where $\epsilon(t+1) \sim N(0, \sigma^2)$.

Now, once invertability has been established, to estimate the parameters, $\phi_p(B)$ and $\theta_q(B)$ in an ARIMA model, we adopt a slightly different representation:

$X(t+1) = \Psi_1 X(t) + ... + \Psi(p+d)X(t+1-p-d) + \epsilon(t+1) - \theta_1\epsilon(t) - ... - \theta_q\epsilon(t+1-q).$

Taking $\hat{\epsilon}(t+1) = 0$ and $\epsilon(n) = \hat{\epsilon}(n) = E(\epsilon(n)|X_1, ..., X_t)$ for $n \leq t$ we obtain:

$\hat{X}(t+1) = \Psi_1 X(t) + ... + \Psi(p+d)X(t+1-p-d) - \theta_1\hat{\epsilon}(t) - ... - \theta_q\hat{\epsilon}(t+1-q),$

where $\Psi(B) = \phi(B)(1-B)^d$.

With a representation that allows us to calculate the one-step ahead forecast of an ARIMA (p,d,q) model, the next step would be to determine the values of the unknown parameters in the model, namely, σ^2, θ and ϕ. The method I use in my study is that of the maximum likelihood estimation.

Definition 3.1.2. *For $\epsilon(t) \sim N(0, \sigma^2)$, we choose as σ^2, θ and ψ those values that maximize the joint gaussian density subject to the constraints of the parameters, also known as the likelihood function:*

$L(\sigma^2, \theta, \phi) = f_{\sigma^2,\theta,\phi}(X(1), X(2), ..., X(N)).$

The representation of the likelihood function, as well as the methods used in its maximization are topics of discussion in their own regard. We refer the reader to Dufour (2008) for a detailed discussion on the topic.

Now, when it comes to fitting an ARIMA model to time series data, we now present different approaches to determining the optimal parameters of p,d and q.

3.1.1. KPSS Test

The KPSS test, named after Kwiatkowski, Phillips, Schmidt and Shin, is a test to determine the minimal value of d necessary to incorporate into an ARIMA

model in order to negate the effects of homogeneous non-stationarity. We begin by fitting the following model to the data:

$$X(t) = \xi(t) + \epsilon(t),$$

where $\epsilon(t)$ is stationary and $\xi(t) = \xi(t-1) + v(t)$ and $v(t) \sim IID(0, \sigma_v^2)$.

We use the following as null and alternative hypothesis:

$$H_0 : \sigma_v^2 = 0$$

$$H_A : \sigma_v^2 > 0$$

We note that under the null hypothesis $\sigma_v^2 = 0$, $\xi(t) = \xi(0)$, $X(t) = \xi(0) + \epsilon(t)$ and X_t is stationary.

In applying this test to a time series at time T, we begin by fitting the following estimated model using linear regression: $X_t = \hat{\xi}(t) + \hat{\epsilon}(t)$.

Using the following test statistic to determine whether or not we should reject the null hypothesis:

$$KPSS = 1/T^2 \cdot \sum_{t=1}^{T} S_t^2 / \hat{\sigma}^2,$$

where $S_t = \sum_{s=1}^{t} \hat{\epsilon}_s$ and $\hat{\sigma}^2$ is an estimator of the variance of $\hat{\epsilon}(t)$.

The test determines for a given significance level, the smallest integer value of d needed to not reject H_0 and applies it on the time series data. In my study, I use a significance level of α=0.05.

3.1.2. AIC Criteria

The Akaike information criterion (AIC) is a measure of strength of a given model. We begin by establishing its definition.

Definition 3.1.3. $AIC = -2log(L) + 2k$,

where k is the number of free parameters to be estimated

and L is the previously defined likelihood of the model.

In the case of an ARIMA model, the number of free parameters to estimate corresponds to p θ parameters q ϕ parameters and σ^2, for a total of p+q+1 parameters. The AIC criteria therefore becomes:

$$AIC = -2log(L) + 2(p + q + 1)$$

In my study I will apply ARIMA models to my data with a grid of values for p and q, each varying from 0 to 5 and chose that with the lowest value of the AIC criteria.

3.2. VQTAM Model

The Vector-Quantized Temporal Associative Memory model proposed by Barreto (2007) is a method for predicting the one-step ahead return of a time series using the self-organizing map algorithm. This method was devised as a generalization to time-series data of a supervised SOM-based associative memory technique used for static data in the domain of robotics, also implemented by Barreto (2003.) The method first involves creating N-p regressor vectors, each with p consecutive observations of a given time series. The vectors should be formed by moving a time window by one time step for each consecutively observed vector. In other words, the N-p vectors formed can be denoted:

$$< X(p), ..., X(1) >$$

$$< X(p + 1), ..., X(2) >$$

$$...$$

$$< X(N - 1)...X(N - p) > .$$

These vectors are denoted the input vectors $\mathbf{X}^{in}(t)$ $t = p..., N-1$ in the application of our model. Additionally, for each vector the one-step ahead observation X(t+1) is denoted the associated output value $Y^{out}(t)$. We can therefore denote the vectors that we are clustering as well as the synaptic weight vector for neuron i as follows:

$$\mathbf{X}(t) = (\mathbf{X}^{in}(t)\ Y^{out}(t))$$

$$\mathbf{m}(t) = (\mathbf{m}_{i(x)}(t)\ m_{i(y)}(t)).$$

The concept behind using the VQTAM model in time series prediction is first using the input vector $\mathbf{X}^{in}(t)$ in determining the best matching weight. In other words the best matching synaptic weight is chosen as the vector $\mathbf{m}_{i(x)}(t)$ with the smallest Euclidean distance to $\mathbf{X}^{in}(t)$. However for updating the weights both the input and associated output vectors are used,

$$\mathbf{m}_{i(x)}(t+1) = \mathbf{m}_{i(x)}(t) + \alpha(t) \cdot h_{i(x)j}(t) \cdot (\mathbf{X}^{in}(t) - \mathbf{m}_{i(x)}(t)) \qquad (3.2.1)$$

$$m_{i(y)}(t+1) = m_{i(y)}(t) + \alpha(t) \cdot h_{i(y)j}(t) \cdot (Y^{out}(t) - m_{i(y)}(t)). \qquad (3.2.2)$$

In order to perform a one-step ahead prediction for testing vector $\mathbf{X}^{in}(t)$, the parameter vector $\mathbf{m}_{i(x)}$ with the smallest Euclidean distance is computed and the output component $m_{i(y)}$ is used as the one-step ahead prediction. In other words, for an input vector $\mathbf{X}^{in}(t)$ the one-step ahead prediction is:

$$\hat{Y}^{out}(t) = \hat{X}(t+1) = m_{i(y)}.$$

It is important to note here that in the VQTAM model, the flow of information is directed from the X-map towards the Y-map. In other words, the information present in the output $Y^{out}(t)$ is not taken explicitly into account during the formation process of the driving X-map. The X-Y fused SOM is almost identical to the VQTAM model but offers the additional benefit that a fused similarity measure between the input and output variables drives the updating of the synaptic weights. The relative weight γ also permits the adjustment of this measure based on the dimension of the input vectors. In this model, during the testing phase, the mapping between $m_{i(x)}$ and $m_{i(y)}$ for a testing vector $\mathbf{X}^{in}(t)$ is truly driven by the relationship between the independent and dependent variables. For this reason, I decide to use the more sophisticated X-Y fused SOM for time series forecasting in my results and compare it to the existing methods presented in this chapter.

3.3. Rule Extraction

As mentioned previously the probability distribution of the synaptic weight vectors, $\mathbf{m}(t) = (\mathbf{m}_{i(x)}(t),\ m_{i(y)}(t))$ can be seen as approximating the probability distribution of the associated vectors $\mathbf{X}(t) = (\mathbf{X}^{in}(t)\ Y^{out}(t))$. For this reason, the components of the weight vectors $\mathbf{m}_{i(x)}(t)$ should approximate the components of associated input vectors $\mathbf{X}^{in}(t)$. I therefore present two additional methods that can be used to determine the degree to which the probability distribution of the weight vectors approximates the probability distribution of the inputs.

The concepts here are based on the work of Barreto (2007), who attempted to build rule extraction procedures for determining intervals for the one-step ahead forecast.

3.3.1. Min/Max Method

The min-max method for determining whether or not a prediction is reliable is as follows:

(1) First the SOM model is trained using the input vectors $\mathbf{X}^{in}(t) =< X(t), ..., X(t-p+1) >$ and the associated output vectors $Y^{out}(t) = X(t+1)$.
(2) We denote χ^i the set of all input vectors mapped to neuron i and we denote the jth such vector $\mathbf{X}_j^{in} =< X_{j1}, ..., X_{jn} >$.
(3) For each set χ^i we denote

$$X_l(min) = min(X_l)\ \forall j\ l = 1, ..., n$$

$$X_l(max) = max(X_l)\ \forall j\ l = 1, ..., n$$

(4) Now, for each input vector in the testing set, we determine which neuron i is closest based on Euclidean distance and denote the testing vector $\mathbf{X}_j^{test} =< X_{j1}^{test}, ..., X_{jn}^{test} >$. The one-step ahead return $\hat{X}(t+1) = m_{i(y)}$ is then reliable if all the following conditions hold:

$$X_1(min) < X_{j1}^{test} < X_1(max)$$
$$X_2(min) < X_{j2}^{test} < X_2(max)$$
$$...$$
$$X_n(min) < X_{jn}^{test} < X_n(max).$$

3.3.2. Confidence Interval Method

The confidence interval method that I propose for determining reliable predictions is similar to the min-max method but incorporates the distribution of the component vectors in each cluster of the self-organizing map. This method offers additional sophistication to the method of Barreto (2007) in that a confidence interval is a more accurate representation of the distribution of the components of $\mathbf{X}^{in}(t)$ over a minimum and maximum, which can be heavily influenced by outliers.

The confidence interval method for determining whether or not a prediction is reliable is as follows:

(1) First the SOM model is trained using the input vectors $\mathbf{X}^{in}(t) =< X(t), ..., X(t-p+1) >$ and the associated output vectors $Y^{out}(t) = X(t+1)$.

(2) We denote χ^i the set of all input vectors mapped to neuron i and we denote the jth such vector $\mathbf{X}_j^{in} =< X_{j1}, ..., X_{jn} >$.

(3) For each set χ^i we denote

$$X_l(lowerbound) = \bar{X}_l - t_{0.975} * s_l/\sqrt{m_i} \ \forall j \ l = 1, ..., n$$

$$X_l(upperbound) = \bar{X}_l + t_{0.975} * s_l/\sqrt{m_i} \ \forall j \ l = 1, ..., n$$

where $\bar{X}_l$ and s_l represent the mean and standard deviation of component vector l in cluster i,
m_i is the number of vectors in cluster i,
t_α represents the α percentile of the student t distribution.

(4) Now, for each input vector in the testing set, we determine which neuron i is closest based on Euclidean distance and denote the testing vector $\mathbf{X}_j^{test} =< X_{j1}^{test}, ..., X_{jn}^{test} >$. The one-step ahead return $\hat{X}(t+1) = m_{i(y)}$ is then reliable if all the following conditions hold:

$$X_l(lowerbound) < X_{j1}^{test} < X_1(upperbound)$$
$$X_l(lowerbound) < X_{j2}^{test} < X_2(upperbound)$$
$$...$$
$$X_l(lowerbound) < X_{jn}^{test} < X_n(upperbound).$$

3.4. Double Vector Quantization Model

The Double vector quantization method proposed by Simon et al. (2004) is an implementation of the self-organizing map algorithm in which two separate SOM maps are trained and combined in order to produce a forecast. The concept was developped in order to produce long term forecasts, although in our implementation, we will use it to produce the one-step ahead forecast. Again, the method starts by creating *N-p* regressor vectors, each with p consecutive observations of a given time series.

$$< X(p), ..., X(1) >$$
$$< X(p+1), ..., X(2) >$$
$$...$$
$$< X(N-1), ..., X(N-p) > .$$

Following this, we create deformations, which consists of subtracting the components of consecutive vectors. The deformation associated with the vector $\mathbf{X}^{in}(t)$

is:

$$\mathbf{\Delta X}(t) = \mathbf{X}^{in}(t+1) - \mathbf{X}^{in}(t)$$

Following this, two separate SOM algorithms are trained, one on the N-p regressor vectors and the other on the associated deformations, each with its own set of neurons. Once this is done, the conditional transition probability matrix between cluster i in the regressor vector space and cluster j in the deformation vector space is calculated, based on the following definition:

$$P(j|i) = P(\mathbf{\Delta X}(t) \in m_j | \mathbf{X}(t) \in m_i) \quad \forall j.$$

In other words, the conditional transition probability matrix for a vector $\mathbf{X}^{in}(t)$ mapped to neuron i consists of counting all the associated deformation vectors mapped to all neurons j and dividing it by the total number of associated regressor vectors in neuron i.

Now, in order to compute the one-step ahead prediction, the following steps are conducted:

(1) For a testing vector $\mathbf{X}(t)$, the parameter vector $\mathbf{m}_{i(x)}(t)$ with the smallest Euclidean distance is computed.
(2) Choose a random deformation $\mathbf{m}_j$ (according to the conditional distribution $P(j|i(x))$.
(3) Add the deformation to $\mathbf{X}(t)$, resulting in

$$\hat{\mathbf{X}}(t+1) = \mathbf{X}(t) + \mathbf{m}_j(t).$$

The one step-ahead prediction is then the first component of that vector.

This method is based on exploiting the mean-reverting nature between a return and a variance. The idea is that the probability of a large deformation is high when the initial return is low and vice-versa. By clustering the deformations separately from the returns, the SOM hopes to capture that mean-reverting nature.

Theoretically, it would also be possible to predict an arbitrary time-horizon into the future with the same initial testing vector by repeating the procedure and each time replacing the testing vector with the one-step ahead prediction. For this, a Monte Carlo procedure would have to be used to draw from the the conditional transition probability matrix at each iteration. Additionally, the process would be repeated a significant number of times with the same initial

testing vector and the results averaged. It would then even be possible to build a distribution of returns for a given initial testing vector. Kohonen and Deboek (2000) did exactly this with both short and long interest rate structures. However, my study will focus primarily on one-step ahead predictions since it would be comparable to the other methods presented.

3.5. Out of Sample Tests

Now, an important topic in forecasting is the manner in which the data are split between the training set and the testing set. It is important that the data set used to estimate the parameters of the model, the training set, be disjoint from the data set used to assess the accuracy of the model. Otherwise, the model is simply memorizing training data as opposed to learning to generalize from it, as previously discussed in the context of overfitting. When it comes to forecasting, since we want to predict future values of a given time series, the k-fold cross validation method discussed in section 2.3 cannot be applied since the vectors in testing set must be drawn from a later time period than the vectors used to train the model. There are various ways which can be implemented to train and test a model applied to a time series and we refer the reader to Tashman (2000) for an in depth analysis of the various methods available to separate a data set into a training and testing set as well as various techniques used for updating and recalibrating the model. In my study, I will borrow the method addressed in this paper titled rolling window forecasting.

Definition 3.5.1. *The rolling window method of forecasting involves training a model on* $X(t)$,$X(2)$,...,$X(t+M-1)$ *and testing the model on* $X(t+M)$ *for* $t = 1, .., N-M$,

where N *denotes the size of the data set and*

M *denotes the number of points desired in the training set.*

This method essentially involves first making a one-step ahead forecast using a given initial portion of data, calculating the forecasting error by comparing it with the actual one-step ahead value. Following this, the window is rolled one step forward to reestimate the parameters of the model and the process is repeated until the end of the data set. Since the self-organizing map involves vectorizing the data, each $X(t)$ in the previous definition is replaced by

$X^{in}(t) =< X(t), X(t-1), ..., X(t-p+1) >$ and $t = p, ..., N - M$ for vectors of length p.

One setback that the rolling origin forecast for one-step ahead predictions has is it leaves the model susceptible to so called data snooping biases. Since we are vectorizing the input data, some of the values that were used to train the model are used in the one-step ahead forecast. The details of this phenomenon are explained in detail in Chapados (2009.) However, in order to maintain comparability between the ARIMA and the self-organizing map forecasts, I leave explorations of this issue for further studies.

3.6. Error Criterion

Once a model has been trained, a sequence of one-step ahead estimates is computed, $\hat{X}(t+1)$. This sequence is then compared to the actual, known, one-step ahead values, $X(t+1)$, using an error criterion. The choice of error criterion is important as its comparison across multiple forecasting methods as well as various time series will permit the identification of optimal models to retain. One type of error that I will be using is an error measured in percentage terms, a scale independent measure, called the Mean Absolute Percentage Error.

Definition 3.6.1. *The mean absolute percentage error (MAPE) can be defined as:*

$$MAPE = \sum_{t'=t+1}^{N-M+1} |(X(t') - \hat{X}(t'))/X(t')| * (100/N - M - t + 1),$$

where N is the number of terms in the testing set.

One disadvantage that the MAPE has is that positive error terms are penalized more heavily than the negative error terms. For this reason, an additional error criterion that I plan on using is that of the Mean Squared Error.

Definition 3.6.2. *The Mean Squared Error (MSE) can be defined as:*

$$MSE = \sum_{t'=t+1}^{N-M+1} (X(t') - \hat{X}(t'))^2/(N - M - t + 1).$$

The Mean Square Error is an important criteria in the evaluation of the out of sample forecast since the training phase of the self-organizing map involves minimizing the euclidean distance between the inputs $\mathbf{X}_i$ and synaptic weight vectors $\mathbf{m}_{jX}$ in addition to Y_i and m_{jY} for the X-Y fused SOM. Since minimizing

euclidean distance is equivalent to minimizing the MSE, the latter is an excellent indicator of the out of sample generalization capabilities of the SOM.

Chapter 4

ANALYSIS OF CROSS SECTIONAL DATA

We begin our analysis of cross-sectional data by analyzing the limit order book of Apple stock trading on the NASDAQ exchange for the trading day of 2012-06-21 between the hours of 9:30 and 16:00 . The limit order book data was based on the official NASDAQ Historical TotalView-ITCH data which supplies information on limit order events that change the state of the order book. LOBSTER in turn, provides reconstructed limit order book data based on this feed that shows the evolution of the order book throughout the trading day up to a specified depth. Specifically, the data that is provided are the evolution of the limit order book up to a depth of ten on the bid and ask sides as well as information on the event causing an update of the order book. All events are time stamped to seconds after midnight, with millisecond decimal precision.

The types of events that can cause an update of the limit order book are:

- submission of a new bid order at price β and volume V_β or submission of a new ask order at price α and volume V_α
- cancellation of a bid or ask limit order
- execution of a bid or ask limit order through the mechanics discussed in section 1.5.

After each update of the limit order book, data are provided, at each level and on both the bid and ask sides, of the price and volume of unexecuted trades. In my analysis, I will only analyze the state of the limit order book after an update caused by an order execution. This will give insight into the behavior of market participants between executions, as far as submitting and removing limit orders during these intervals. Using the terminology discussed in section 1.5, the volume at each depth $i \in 1, ..., 10$ can be defined as follows:

Definition 4.0.1.
For a bid limit order,

$$BV_i = \sum_{n_i \in \mathbb{Z}} V_{\beta_{n_i}} \tag{4.0.1}$$

such that $\beta_{n_{i1}} = \beta_{n_{i2}} = ... \quad \& \quad \beta_{n_i} < \beta_{n_{i-1}} < \beta_{n_{i-2}}...$

For a sell limit order

$$AV_i = \sum_{n \in \mathbb{Z}} V_{\alpha_{n_i}} \tag{4.0.2}$$

such that $\alpha_{n_{i1}} = \alpha_{n_{i2}} = ... \quad \& \quad \alpha_{n_i} > \alpha_{n_{i-1}} > \alpha_{n_{i-2}}...$

My goal will be to study market participants behavior through the volume imbalance between the bid and ask sides. For this reason, I define the variable:

Definition 4.0.2.

$$Bid\ Ask\ Volume\ Imbalance_i(t) = log(AV_i(t)/BV_i(t))\ for\ i = 1, ..., 4 \tag{4.0.3}$$

Additionally, in my study, I will be interested in executions of market orders, which show up as a series of executions of limit orders that take place within small time windows of each other. For this reason, I define a series of blocks in the limit order book, which are comprised only of limit order executions, on both the bid and ask sides, that happen within 100 milliseconds of each other. As soon as a limit order is executed within more than 100 milliseconds of the previous execution, it marks the beginning of the following block. I assume that within each block there are only complete market order executions. The bid-ask volume imbalances are then taken exclusively after the last limit order execution of a block. Using this structure I then define the following variable:

Definition 4.0.3. *Letting $t_{B_n}(1), ..., t_{B_n}(T)$ denote the times of limit order executions in block B_n for $n \in \mathbb{Z}$*

Execution Price Direction=

$$\begin{cases} 1, & for\ \alpha_E(t_{B_n}(1)) > \alpha_1(t_{B_{n-1}}(T))\ (\beta_E(t_{B_n}(1)) > \beta_1(t_{B_{n-1}}(T))) \\ 0, & for\ \alpha_E(t_{B_n}(1)) = \alpha_1(t_{B_{n-1}}(T))\ (\beta_E(t_{B_n}(1)) = \beta_1(t_{B_{n-1}}(T))) \\ -1, & for\ \alpha_E(t_{B_n}(1)) < \alpha_1(t_{B_{n-1}}(T))\ (\beta_E(t_{B_n}(1)) < \beta_1(t_{B_{n-1}}(T))) \end{cases} \tag{4.0.4}$$

where $\alpha_E(t)$ ($\beta_E(t)$) denotes the execution price at time t of a sell (buy) limit order.

The total number of vectors after the definition of such a structure is 10 329. We then proceed by separating these vectors into three equal groups of size 3443. The first two groups will be used to determine the optimal tuning parameters for the self-organizing map through cross validation. Using these parameters, the remaining group will be used to train the self-organizing map, which can then be used for testing on a live basis to develop trading strategies. The reason I define such a structure is that in the case of competitive neural networks, the number of neurons is itself a tuning parameter. In my study, I consider the optimal number of neurons to be largely dependent on the number of training vectors, and by performing 2-fold cross validation on 2/3 of the data vectors, we end up finding the optimal number of neurons for 1/3 of the data vectors. Since the neighborhood function $h_{j,i(x)}$, is also a tuning parameter, I proceed by performing 2-fold cross validation on data vectors from 9:30 to 14:21 (6446 points) using both the Gaussian and Bubble neighborhood functions using grid sizes that vary from 5*5 to 15*15. Then, for both neighborhood functions, I present the evolution of the training error and validation error, as measured by mean square error from the trained vectors $\mathbf{X}_i$ to their closest synaptic weight vectors $\mathbf{m}_{i(x)}$ for increasing map sizes. I also present additional parameters that were used to train the SOM that were simply taken as their defaults in the *kohonen* package in R. *rlen* is defined as the number of times the full data set $\mathbf{X}_1, .., \mathbf{X}_n$ is presented to the network.

In table 4.1 $t = 1, ..., rlen$ and $\sigma(0)$ is taken to be a value that covers 2/3 of all map units. Additionally, when t' is reached such that $\sigma(t') < 1$, only the winning unit gets updated.

Table 4.1. Fixed Parameters for SOM

Parameters	Values
rlen	100
$\alpha(t)$	0.05-0.04/(rlen-1)*($\lceil t/N \rceil$ -1)
$\sigma(t)$	$\sigma(0)$ -$\sigma(0)$/(rlen-1)*($\lceil t/N \rceil$ -1)
$m_j(0)$	Initialized through random sampling of X_i
Topology	Hexagonal

Referring to figures 4.1 and 4.2, we note that validation curves for the k-means and the SOM with a Bubble neighborhood function are quite similar. However, figure 4.3 shows that a Gaussian neighborhood function has a significant impact on the shape of the validation curve. This shows that applying a radius around the winning neuron in the cooperative phase of training only has a significant impact if a Gaussian weighting is applied to the neurons within this radius. In my study, I am interested in selecting a grid size that reduces overfitting by minimizing the distance between the training and validation errors whilst selecting a sufficient number of neurons that keeps both these errors sufficiently low. For this reason, I chose the Gaussian function as the optimal neighborhood function. Additionally, we note that for map sizes larger than $10 \cdot 10$ the validation error still decreases but by a slower rate than the training error. In order to optimize the computation time, we limit our choice for the number of neurons to a $10 \cdot 10$ map size for 3443 vectors.

Figure 4.1. Validation curve for k-means

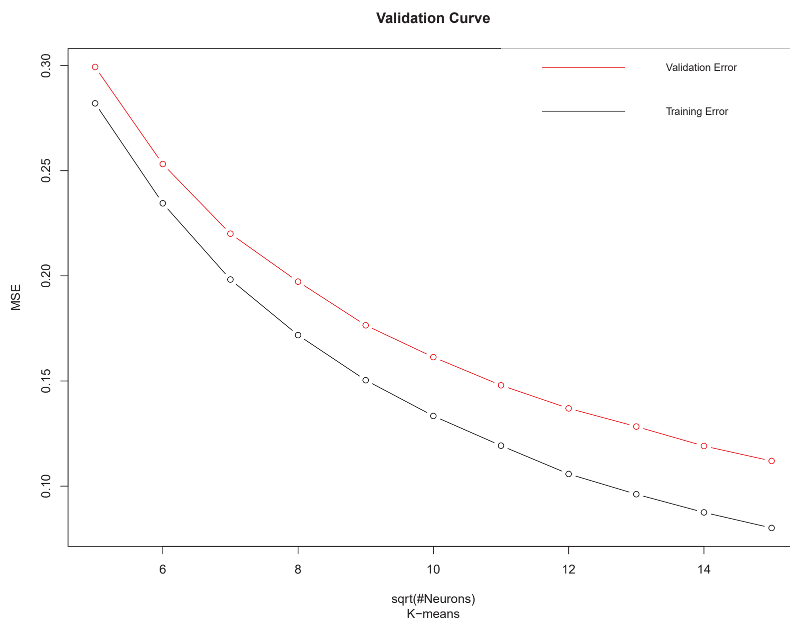


FIGURE 4.2. Validation curve for the SOM with a Bubble neighborhood function

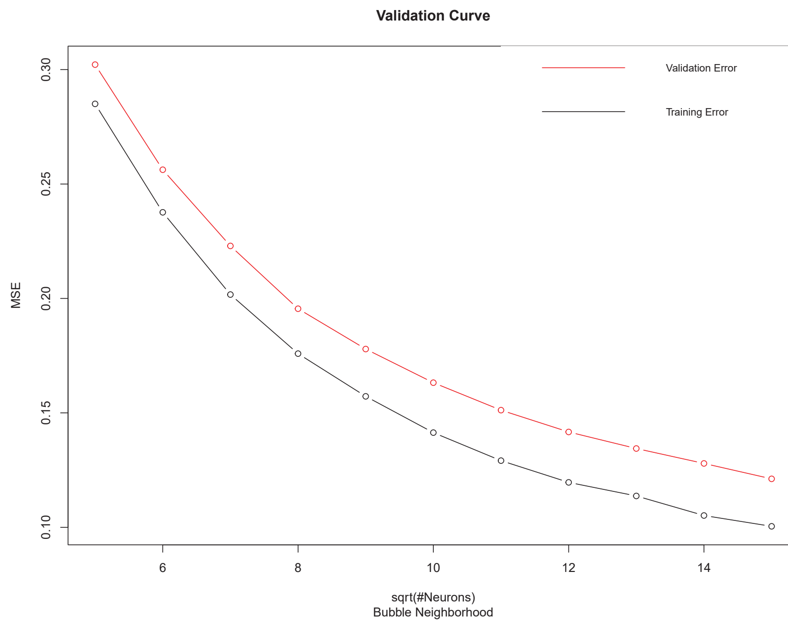
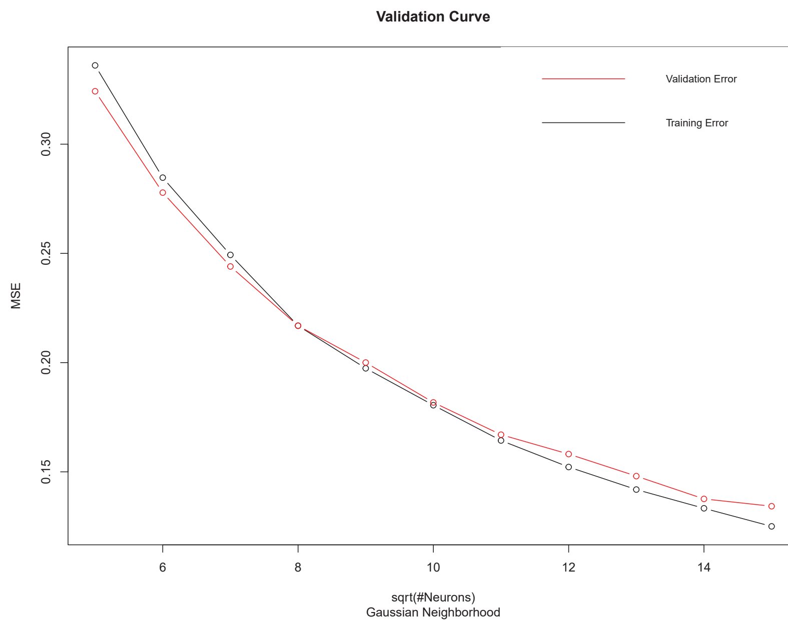


FIGURE 4.3. Validation curve for the SOM with a Gaussian neighborhood function



Now, we train the map using the remaining 3443 data vectors that represent volume imbalances of the order book between 14:21 and 16:00 using the determined optimal tuning parameters.

Referring to figure 4.4, we are able to study the convergence of the algorithm as measured by the mean distance of the training vectors to the synaptic weight vectors of their closest neurons, after each iteration. Figure 4.4 plots t^* on the x-axis and $\sum_{i=1}^{n}(d(X_i(t*), m_{i(x)})^2/n$ on the y-axis where $t* = 1, ..., rlen$ and the distance function represents Euclidean distance.

In this context, an iteration $t*$ is the presentation of the full data set, one vector at a time, to the self-organizing map algorithm. Around the 80th iteration, we note the error drops drastically and remains between 0.006 and 0.008, indicating that 100 iterations is an optimal number of iterations for the size of our data set as increasing this value will increase computation time, without significant decreases to the error.

Following this, we draw our attention to figures 4.5 and 4.6 which are presented for visualization purposes. A major advantage of the self-organizing map (and neural network clustering algorithms) is that is permits efficient and clear representations of the states of its neurons after training is completed. The count plot depicts the number of training vectors mapped to each neuron with a sliding color scale and the codes plot depicts individual fan representations of the magnitude of each variable in the synaptic weight vector, for each neuron. Together, these plots enable us to obtain a visual representation of the underlying distribution of the input vectors.

FIGURE 4.4. Progression of MSE with each iteration

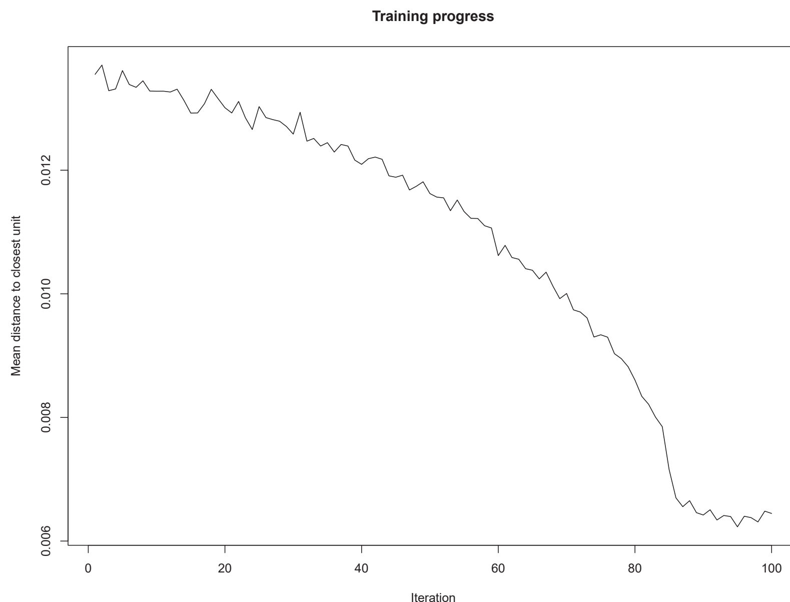


FIGURE 4.5. Magnitude of each variable in each neuron

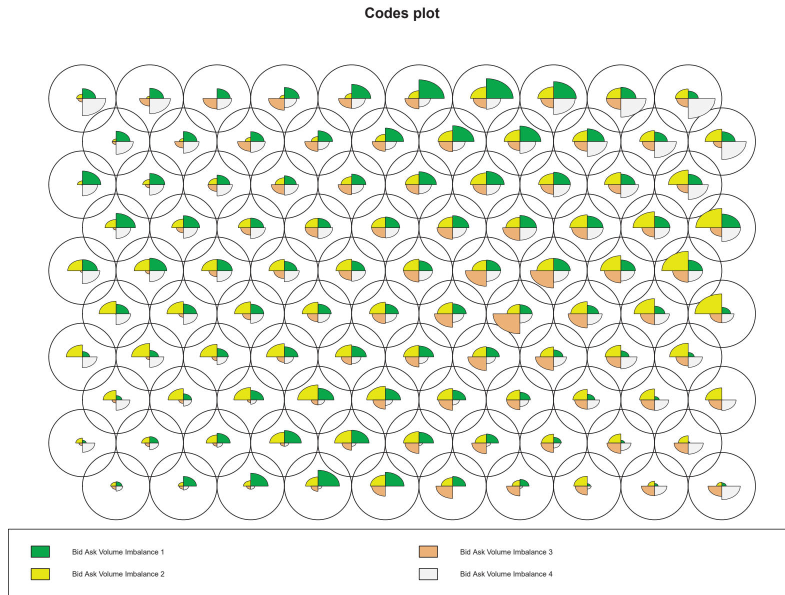
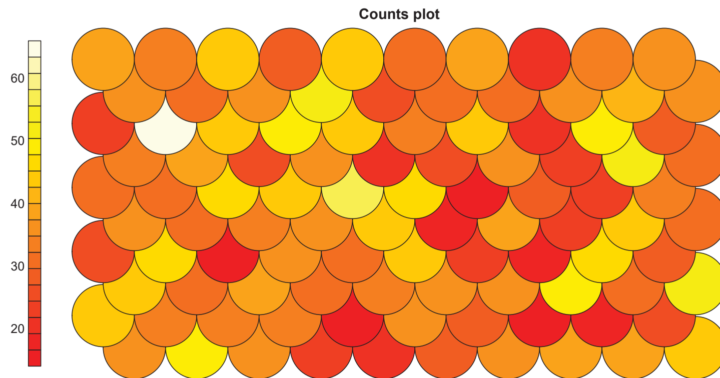


FIGURE 4.6. Number of training vectors mapped to each neuron



Now, after training the self-organizing map, I proceed with hierarchical clustering of the synaptic weight vectors in order obtain a smaller number of clusters, each of which will be a unique representation of the the state of the limit order book, as characterized by the bid ask volume imbalances. With this approach, I am able to benefit from both the neural network approach to clustering of the

SOM as well as the agglomerative approach of hierarchical clustering.

Figure 4.7 represents a dendrogram, which is a tree diagram used to illustrate the arrangement of the clusters produced by hierarchical clustering. At the bottom of the dendrogram we have the individual synaptic weight vectors. Through the agglomerative clustering approach, the synaptic weight vectors are then combined until one large cluster is formed, as represented by the top branch in the dendrogram. The distance between merged clusters is monotone increasing with the level of the merger and the height in the plot on the y-axis represents the distance between two merged clusters. In my study, the complete linkage method is used as a distance method, and the distance measure used in the complete linkage method is Euclidean. I invite the reader to consult section 1.4 for a detailed description of this method. Through visual analysis of figure 4.7, I chose to cut the dendrogram at the height of 2, producing 4 clear clusters. In figure 4.8, we also see a representation of the SOM neurons by these "superclusters" with the 4 superclusters given unique colors.

Figure 4.7. Dendrogram

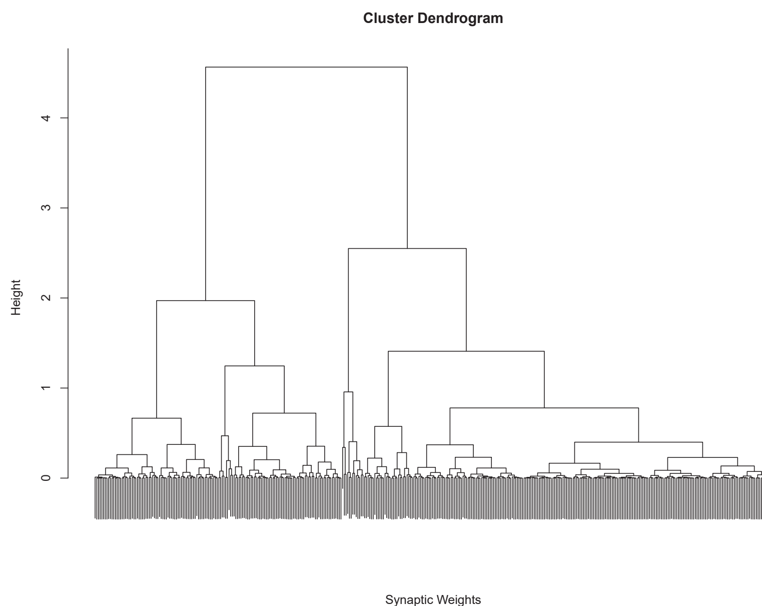
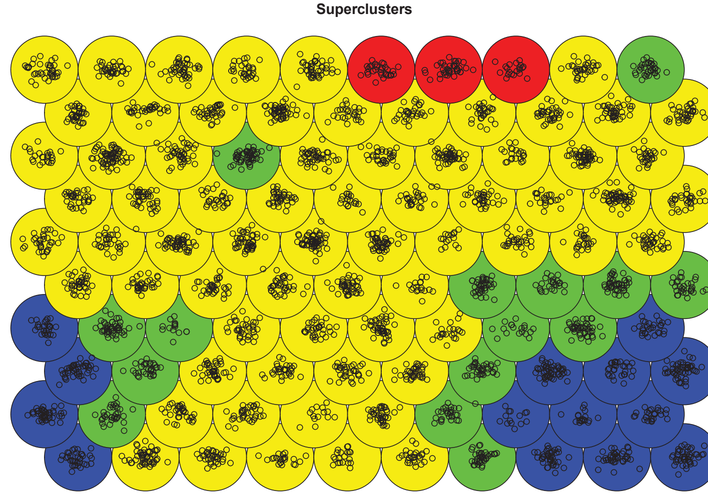


FIGURE 4.8. Supercluster representation of SOM map



Following the training of the SOM algorithm, the idea is that real time snapshots of the market, as defined by volume imbalances in the limit order book following the execution of a market order, can be presented to the network for testing, and the supercluster that it falls in gives insight into the current state of the market. By studying the *Execution Price Direction* variable, we can obtain insight into the manner in which volume will be added into the order book, on both the bid and ask sides. This enables the development of strategies in which a desired volume of stocks is obtained at the optimal price. In tables 4.2-4.5, I present the frequency distribution of the variable *Execution Price Direction* for both buy and sell limit orders, for all 4 superclusters. In figure 4.9, I then present bar plots of the mean Bid Ask Volume Imbalances for all 4 superclusters.

Referring to tables 4.2 and 4.3, we begin by noting that $P(Execution\ Price\ Direction = 1|BuyLO)$ is largest in the first supercluster with a value of 61.9% and $P(Execution\ Price\ Direction = -1|SellLO)$ is largest in the second supercluster with a value of 63.0%.

Drawing our attention to figure 4.9, we note that the first supercluster exhibits the lowest mean values for all 4 levels of volume imbalances amongst the superclusters. This indicates that when $BV_i(t)$ is much larger than $AV_i(t)$, market participants are likely to submit bid limit orders at a price $\beta > \beta_1(t)$ in an effort to have their orders executed in a timely manner and not be put at the end

of a long queue.

Alternatively, referring to figure 4.9, we note that it is one of two superclusters exhibiting a positive volume imbalance on the first level, the other one being the fourth supercluster, as indicated by figure 4.9. This indicates that when $AV_1(t)$ is much larger than $BV_1(t)$, market participants are likely to submit ask limit orders at a price $\alpha < \alpha_1(t)$. Similarly to the previous situation, this is likely due to market participants submitting orders such that they end up executed in a timely manner. Although, referring to 4.9, the fourth supercluster exhibits a larger volume imbalance at the first level, despite $P(Execution\ Price\ Direction = -1|SellLO)$ being at the lower value of 55.3%, we note that there are simply less training vectors that fall within this supercluster. Therefore, during the testing phase the fourth supercluster should not guide any trading strategies.

Finally, referring to table 4.4 and figure 4.9, we note that in the third supercluster all four levels of the bid ask volume imbalance are negative, although larger than those in supercluster 1. Appropriately, we note that $P(Execution\ Price\ Direction = 1|BuyLO)$ is 57.2%, a value slightly less than the corresponding value in the first supercluster.

Ideas for further research might involve modelling the time between the execution of the last limit order in a block and the submission of volume into the limit order book, separately for each supercluster. This would give insight into the patterns of volume submission into the book between $t_{B_{n-1}}(T)$ and $t_{B_n}(1)$. One might imagine a starting point would be the application of a Poisson stochastic process to model this time, which would give a market participant insight not only into the manner in which volume is added but the time until it is added. This would enable the developments of algorithms that submit opposing market orders based on both the current state of the book and time to submission.

Table 4.2. Frequencies-SuperCluster 1

		Next Limit Order Execution			
		Sell		Buy	
		Count	Column N %	Count	Column N %
PriceDirection	-1	103	48.1%	4	1.40%
	0	89	41.6%	102	36.7%
	1	22	10.30%	172	61.90%
	Total	214	100.0%	278	100.0%

Table 4.3. Frequencies-SuperCluster 2

		Next Limit Order Execution			
		Sell		Buy	
		Count	Column N %	Count	Column N %
PriceDirection	-1	617	63.00%	70	5.10%
	0	319	32.6%	549	40.00%
	1	44	4.50%	754	54.90%
	Total	980	100.0%	1373	100.0%

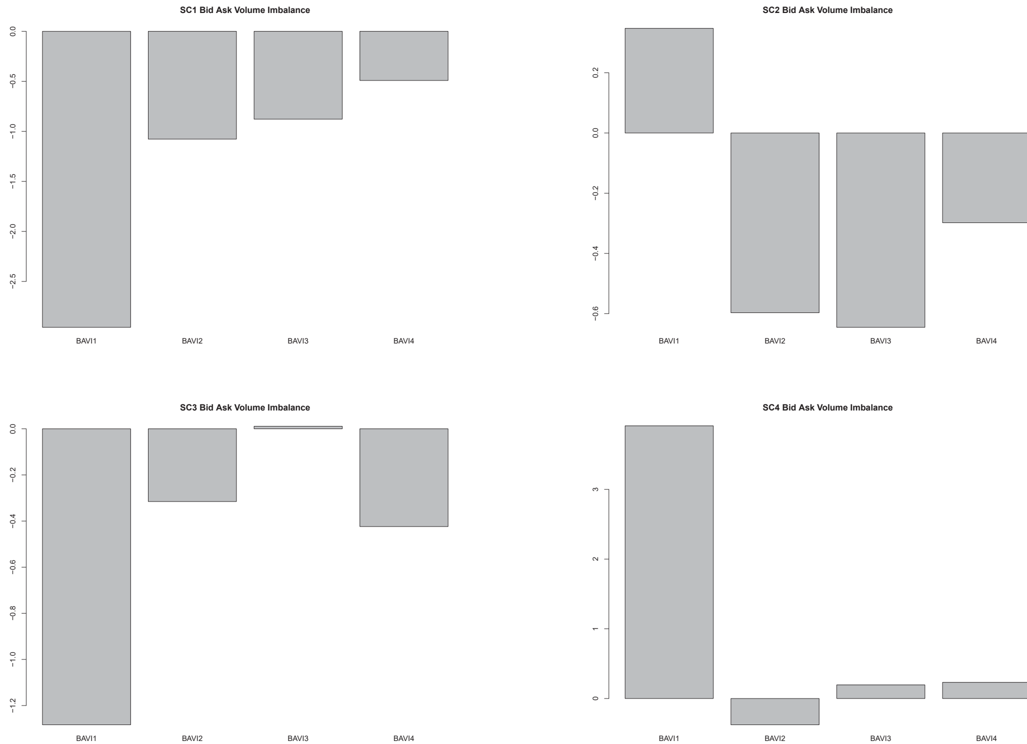
Table 4.4. Frequencies-SuperCluster 3

		Next Limit Order Execution			
		Sell		Buy	
		Count	Column N %	Count	Column N %
PriceDirection	-1	116	57.40%	7	2.3%
	0	67	33.20%	124	40.50%
	1	19	9.40%	175	57.20%
	Total	202	100.0%	306	100.0%

Table 4.5. Frequencies-SuperCluster 4

		Next Limit Order Execution			
		Sell		Buy	
		Count	Column N %	Count	Column N %
PriceDirection	-1	21	55.3%	5	9.6%
	0	16	42.10%	21	40.4%
	1	1	2.60%	26	50.00%
	Total	38	100.0%	52	100.0%

Figure 4.9. Bid-Ask Volume Imbalance Bar Plots



Now, referring to figure 4.10, we can also analyze the vectors representing bid ask volume imbalances based on the time of day that their corresponding superclusters occurred in. In the figure, we see the first 100 vectors in the testing set, which represent order executions between 14:21 and 14:25 color coded by the superclusters to which they belong. The y-axis represents the execution price of the trade that caused the update of the limit order book at time t and the x-axis, an index corresponding to an ordering in time of the last executed trade of a block. This representation can give insight into the relationship between the execution price of a stock and the evolution of state of the bid-ask volume imbalance throughout the day.

Another type of representation of the superclusters is through principal component analysis. In order to work with unitless measures and the correlation matrix of the original variables, we again standardize the bid-ask volume imbalances within each supercluster with the respective means and variances. Then, in the following figures, I use principal component analysis to graphically represent the bid ask volume imbalance vectors in each supercluster. Figure 4.11 represents the variance (y-axis) associated with each of the principal components (x-axis).

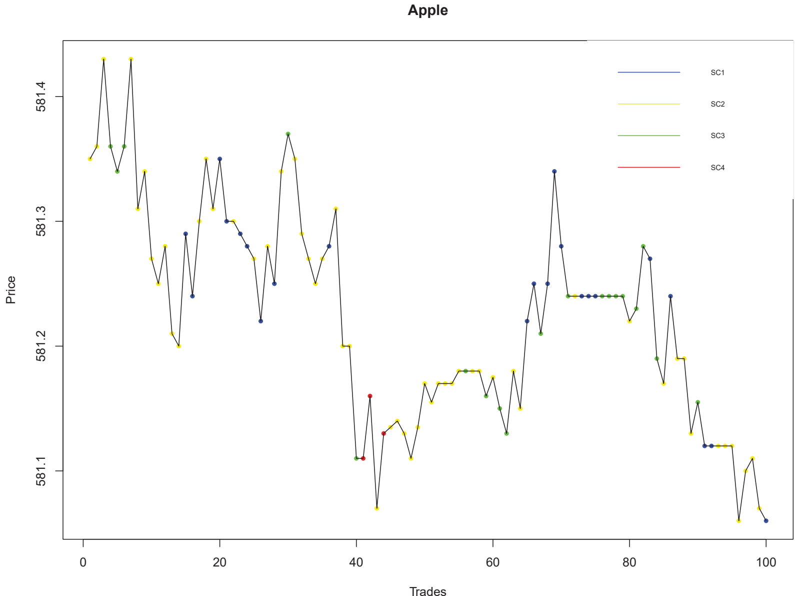
To better illustrate the data, we choose to retain the first two PCs for each supercluster, which we note represents a sufficient amount of explained variance. In figure 4.12, I then present a two dimensional plot of the bid ask volume imbalance vectors projected on to a plane defined by the first two principal components. On the x-axis, we see the first principal component as well the percentage of explained variance that it represents and on the y-axis we see the same for the second principal component. Additionally, the planes embedded on each of principal component planes represent a projection of the axes in the original, higher dimensional data. In other words, given $PC_k = \alpha_{k1} * BAVI_1 + .. + \alpha_{k4} * BAVI_4$ for $k = 1, 2$ the vectors representing bid ask volume imbalance in the plane can be represented as:

$$\mathbf{BAVI}_i = \alpha_{1i} * \mathbf{I}_x + \alpha_{2i} * \mathbf{I}_y$$

where $\mathbf{I}_x$ and $\mathbf{I}_y$ are unit vectors pointing in the direction of the X and Y planes respectively.

This type of representation can be used in conjunction with tables 4.2 to 4.5 to develop trading strategies. As previously mentioned, principal component analysis can be seen as a linear dimensionality reduction technique. The advantage of applying principal component analysis after the application of the self-organizing

FIGURE 4.10. Price vs Time based on SC representation of BAVI



map is that the latter has the ability of mapping high-dimensional non-linear data. In other words, data in n-dimensional space that cannot be projected onto a hyperplane in a lower dimensional space. Since the underlying relationship of bid ask volume imbalances at four separate levels is assumed to be non-linear, we make use of the topological ordering property of the SOM to create clusters with similar topologies before applying principal component analysis within these clusters. While it is true that the data within the clusters is not expected to be distributed over a plane and hence not ideal to be projected onto a lower dimensional hyperplane, less complex topographical relationships in input data for principal component analysis still produces clearer representations of the output. Additionally, the superclusters represent states of the limit order book and our goal is to obtain a visualization of individual observations within each supercluster.

Further research into dimensionality reduction in limit orders can entail application of principal component analysis and the SOM on the original data vectors and comparison of the output would give insight into the topographical relationships in the data. I refer to Annas et al. (2007) for a comparative study between the self-organizing map and principal component analysis in dimensionality reduction. In my study, Figure 4.12 can be used as a visualization tool that gives insight into the relative similarity of bid ask volume imbalances within each supercluster. Since principal components are simply the direction in space along which projections of the data have the largest variance it is an excellent tool for outlier detection.

In addition to being used as a visualization tool, using this strategy, regression models can be developpped in which the principal components can be used as regressors for a desired dependent variable i.e $Y_i = \beta_0 + \beta_1 * PC_1 + ... + \beta_4 * PC_4 + \epsilon_i$ with the advantage that multicollinearity between the predictors, an assumption of multiple regression, can be circumvented. This holds since principal component analysis is essentially an orthogonal transformation in which $Cov(PC_i, PC_j) = 0$ by definition.

FIGURE 4.11. Variance of Principal Components

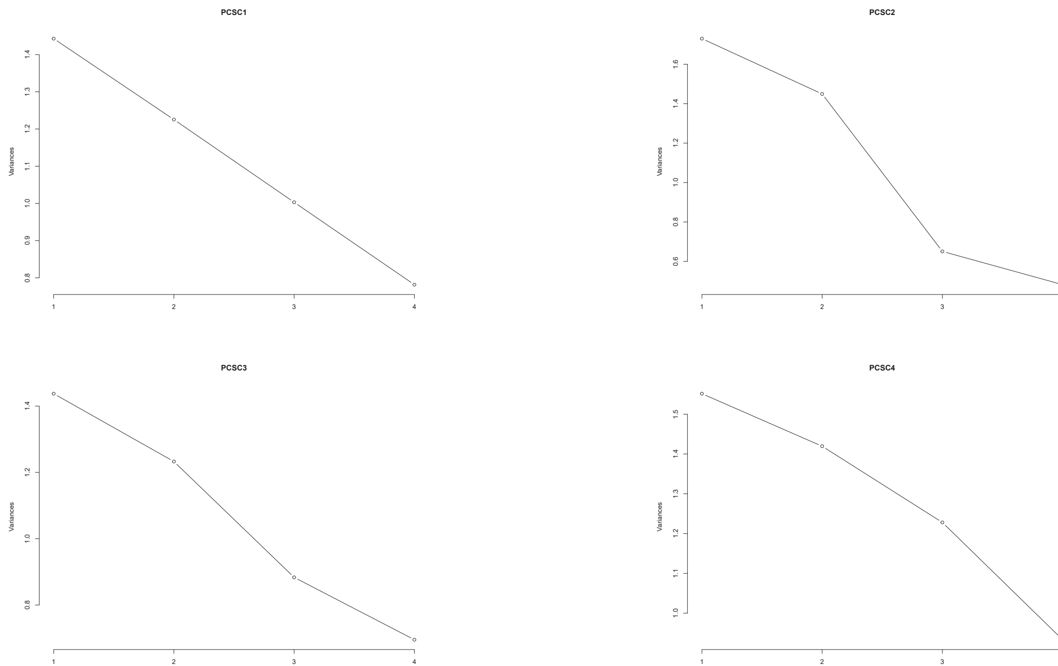


FIGURE 4.12. Projection on first 2 PCs

Chapter 5

ANALYSIS OF SEQUENTIAL DATA

We now proceed with the analysis of time series data for both the Dow Jones Industrial Average Index as well as Apple stock. The Dow Jones is an index that follows 30 large publicly owned companies based in the United States. The value of the Dow is the sum of the component prices divided by a factor which changes whenever one of the component stocks has a stock split or stock dividend. In my analysis, I take 20 years worth of the daily closing price for the Dow index, between 1997-01-20 and 2017-01-20. The daily closing price consists of incorporating the last daily execution price of the constituent stocks. The prices that I use in my analysis for Apple data are the limit order execution prices of Apple stock, again for the trading day of June 21, 2012 between the hours of 9:30 and 16:00. Specifically I take $\alpha_E(t_{B_n}(T))$ $(\beta_E(t_{B_n}(T)))$, the limit order execution price that caused the last update in a market order block, whether the execution was that of a bid or ask limit order. In figure 4.10, I represented the first such 100 prices for the testing set used in that context.

In our analysis of cross-sectional data, it was determined that the optimal dimensions to train a self-organizing map for 3443 vectors was a $10 \cdot 10$ map with a Gaussian neighborhood function. For this reason, in this section, I use a rolling window approach with M=3443 data vectors to train a $10 \cdot 10$ map with a Gaussian neighborhood function at every iteration for both the Dow Jones and Apple data. Although here we analyze a sequential data set pertaining to time series of financial product prices as opposed to snapshots of the state of the limit order book, we again assume that the number of training vectors is the main determining factor as to the optimal number of neurons. In fields outside of finance a driving factor would be an insight into the nature of the data that would give an idea as to the number of clusters that should form. Since we possess no insight as to the number of stationary clusters in a given time series we make use of the

previously determined results.

The value of λ that I use in the training of the X-Y fused SOM is a constant 0.5 and I incorporate a lag of 5 for the independent variable into the model. In other words, my objective here is to determine $\hat{X}(t+1) = E_\omega(X(t+1)|\{X(t), X(t-1), ..., X(t-4)\})$, which for the X-Y fused SOM is based on a fused similarity measure of:

$$D = 0.5 * || < X(t), .., X(t-4) > - < m_{j_{X(t)}}, .., m_{j_{X(t-4)}} > || + 0.5 * ||X(t+1) - m_{j_{X(t+1)}}||.$$

I adopt these values to simplify the model in order to reduce the computation time as the rolling window approach comes at a high computational cost. Further directions that research in this field can be taken in is the determination of the optimal values of γ as well the lag to apply on the independent variables. Another direction for further research is in improving the quality of the local models through the mapping between the $m_{i(x)}$ and $m_{i(y)}$. By building local time series models into the synaptic weights, a greater predictive accuracy can be achieved since this would combine the unsupervised clustering properties of the SOM with the accuracy that a supervised model would bring in reducing the error. It is also to note that the prices for each lag here are scaled by their respective means and variances, as this was shown empirically to produce better results. The remaining parameters of the self-organizing map are set to their default values, as specified in the previous section.

5.1. Dow Jones Industrial Average Index

First we note that the analysis of the Dow Jones Industrial Average Index through the use of a rolling window approach on 3443 training vectors yielded 1586 values of $\hat{X}(t+1)$ with the min/max method producing 1335 reliable predictions and the confidence interval method yielding only 17 reliable predictions. Referring to table 5.1, we note that the MSE shows that the ARIMA model exhibits the lowest testing error. However, the X-Y fused SOM, after application of the confidence interval extraction rule, actually lowers the error to below that of the ARIMA. Although, through the analysis of the MAPE metric, the ARIMA model performs slightly better than the others even through the implementation of the min/max and confidence interval extraction rule methods for the X-Y fused SOM. It is important to remember, however, that my goal in the application of these models was to model non stationary time series data. The ARIMA model actually corrects for non-stationarity through a differencing operator after conducting the KPSS test while the self-organizing map algorithm performs no such correction. My goal is to show that through the implementation of the SOM, localized stationary clusters are produced. Since the error for the ARIMA model is not far off from the errors produced by the SOM algorithms, to a certain extent, my hypothesis is verified. It is also worth noting that the min/max and confidence interval methods lower the forecasting error through both the MAPE and MSE criteria. Although, it may seem impractical that over 20 years of daily data, only 17 daily forecasts were deemed reliable by the confidence interval method. It is for this reason, as robust as this method might be, its practical significance becomes apparent in its implementation on high frequency data, as presented in the following section.

Table 5.1. Error Criteria-Dow Jones

	MSE	MAPE
XYF SOM	0.003576579	9.738847
Min/Max	0.003170116	9.010667
CI	0.001343007	8.511181
Double VQTAM	0.002950391	9.986324
ARIMA	0.001567704	6.392019

Referring to figures 5.1, 5.2 and 5.3, we also can visually inspect the one step ahead forecast produced by the various models in relation to their actual values. The points represent the realized standardized prices, and the solid lines, their forecast. In order to implement these forecasting methods in trading strategies,

it would simply be a matter of applying the mean and variance of $X(t+1)$ in unstandardizing both $X(t+1)$ and $\hat{X}(t+1)$.

FIGURE 5.1. X-Y Fused SOM Model-Dow Jones

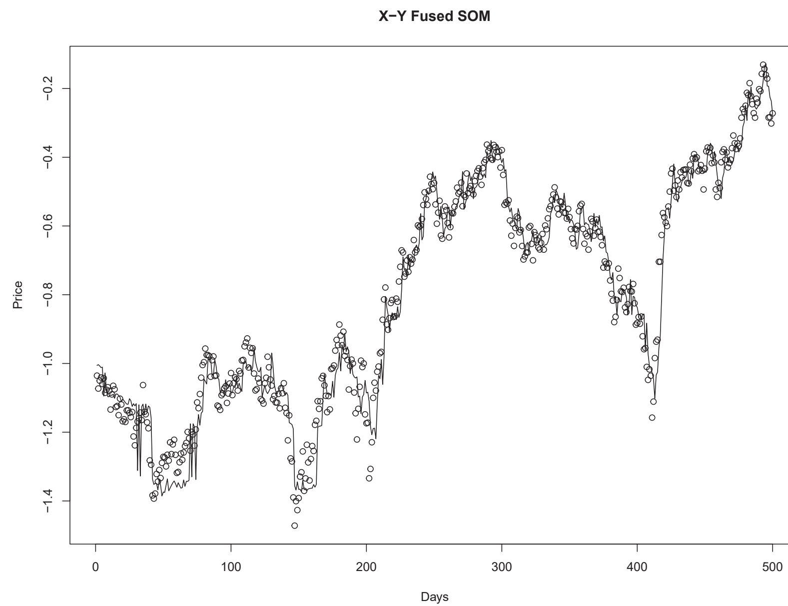


FIGURE 5.2. Double VQTAM Model-Dow Jones

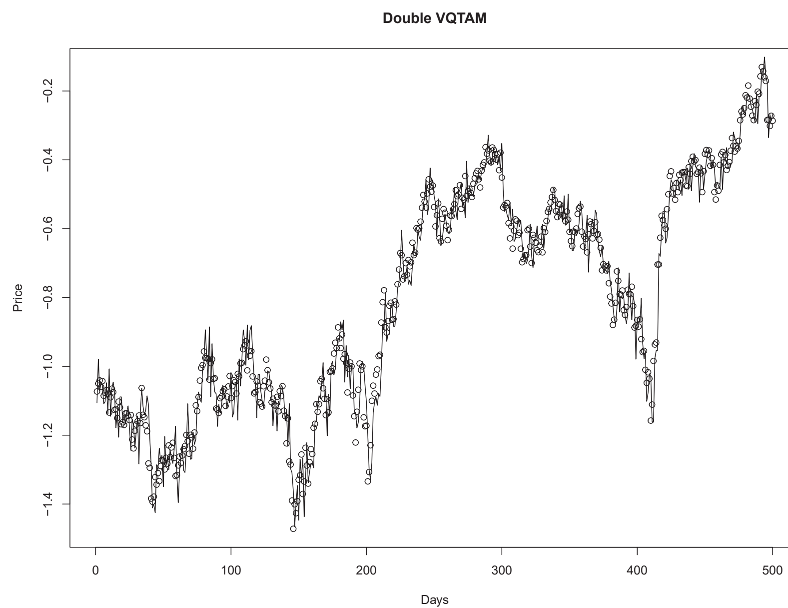
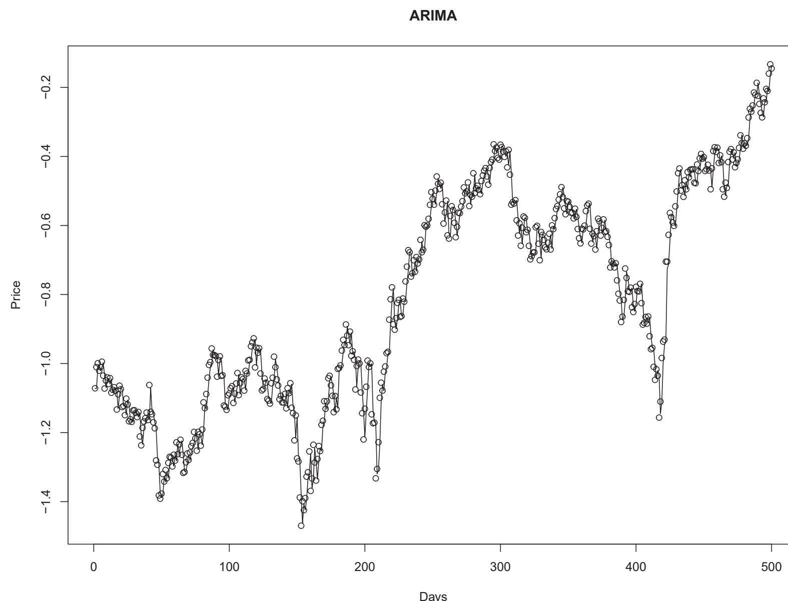


Figure 5.3. ARIMA Model-Dow Jones



5.2. Apple

In the analysis of execution prices extracted from the limit order book of Apple stock, the number of values of $\hat{X}(t+1)$ when the rolling window was applied on 3443 vectors was 6880, the number of reliable predictions was 5259 as established by the min/max method and 95 as established by the confidence interval method. We also immediately notice that in percentage terms the ARIMA yields the worst forecasting accuracy out of all the applied models. While analysis of the MSE shows that the ARIMA performs better, the errors are so small that the differences are negligible. We additionally note that the confidence interval extraction rule lowers the prediction error in both instances, for both the Dow Jones Index and Apple data. Additionally, although there were only 95 reliable prediction determined by this method, since these are high frequency execution prices of one trading day of Apple stock, it is certainly more practical to implement a trading strategy with these results than with those obtained with the Dow Jones index. For this reason, I can state with certainty that this is an improvement over Barreto's (2007) previously proposed min/max method. We note, though, that for both the Dow Jones Index and Apple stock the min/max and the confidence interval prediction rules lower the forecasting error as measured by the MSE and MAPE, making them both efficient methods. All in all, the results obtained here show that on a micro structure level, the SOM algorithms perform an excellent

job of modelling non stationary time series without the application of a correction as is done in the ARIMA model.

TABLE 5.2. Error Criteria-Apple

	MSE	MAPE
XYF SOM	0.00189613	27.49212
Min/Max	0.001239807	26.57013
CI	0.000682054	15.65443
Double VQTAM	0.000970927	19.23865
ARIMA	0.000618478	53.3081

Just as before, we refer to figures 5.4, 5.5 and 5.6 to visually inspect the one-step ahead forecast produced by the various models in relation to their actual values. The points represent the realized standardized prices, and the solid lines, their forecast.

FIGURE 5.4. *X-Y* Fused SOM Model-Apple

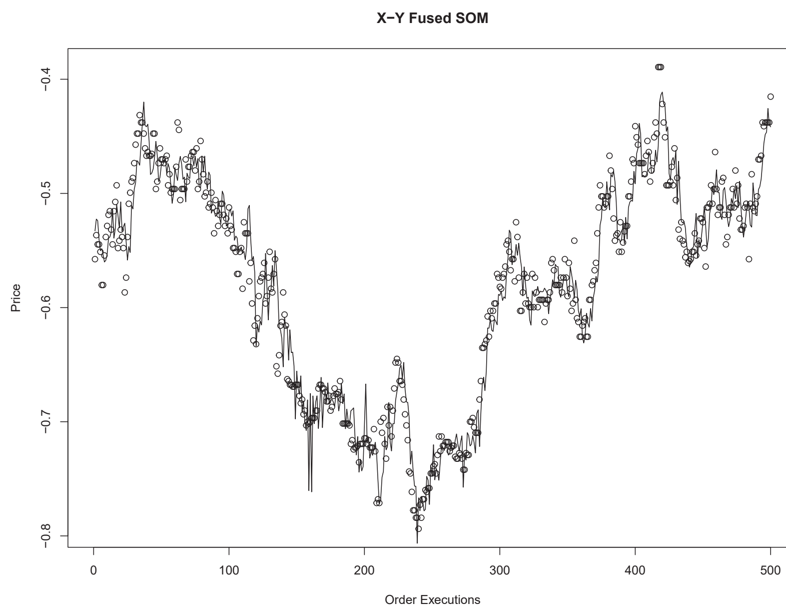


FIGURE 5.5. Double VQTAM Model-Apple

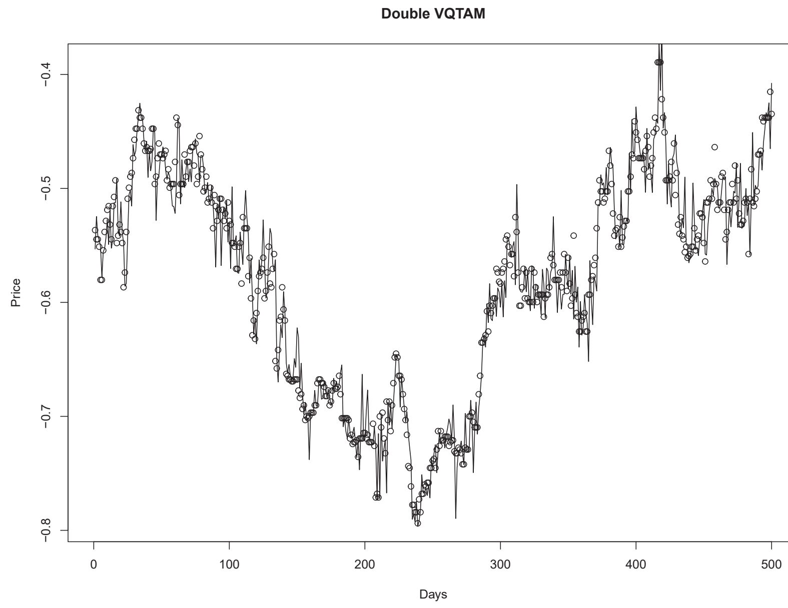
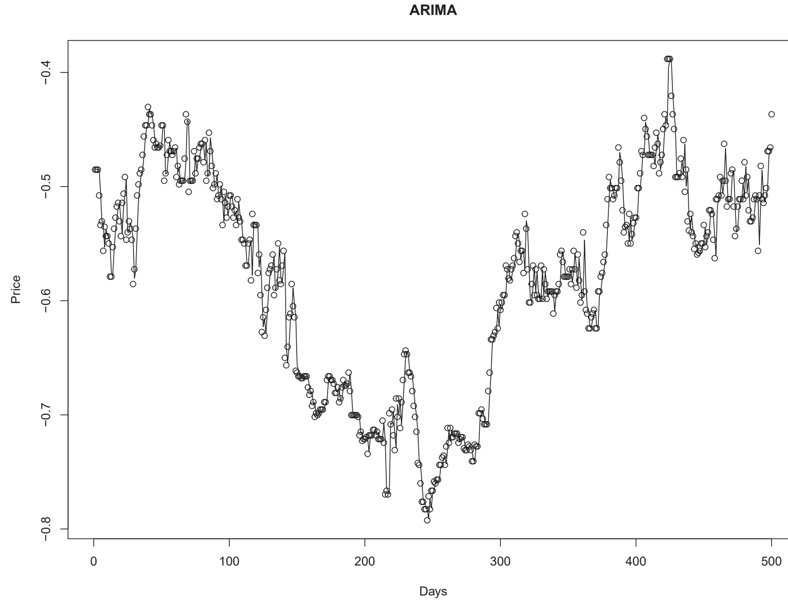


FIGURE 5.6. ARIMA Model-Apple



Chapter 6

CONCLUSION

In conclusion, the field of neural networks has vast applications in all data science driven fields. While error correcting networks such as feed forward and recurrent neural networks are being extensively studied for forecasting purposes, my goal in this Master's thesis was to assess the self-organizing map's ability in forecasting specifically non-stationary time series data. The intuition behind using such an approach was that through its topological preserving and density matching properties, local stationary clusters can be determined through the application of its iterative algorithm.

It is important to note, though, that the self-organizing map falls in the realm of evolutionary computation, in other words algorithms that offer only a heuristic approach to optimization. As such, the iterative SOM algorithm is not based on a solid mathematical framework and much of its results and practical significance are based on empirical results. Nonetheless, I follow Kohonen et al. (1991) in his derivation of the self-organizing map algorithm based on an error functional as well as attempt to provide framework for the algorithm's implementation in time series forecasting.

Despite its lack of theoretical background, the results obtained in this thesis show that the SOM is a powerful tool in time series forecasting of non-stationary data as the errors it produced either closely matched or improved upon the errors produced by the ARIMA model, which applied an approach based on the mathematical framework of unit roots for correcting non-stationarity. We also noted of its particular efficiency in forecasting high-frequency time series data, extracted from the limit order book of Apple stock.

Additionally, we also determined that the original, unsupervised variant of the SOM was also extremely efficient in clustering bid-ask volume imbalances in the same limit order book of Apple stock. Through an unsupervised study of the relationship between volume imbalance and volume submission and deletion in the order book, I presented a model that can be used to determine optimal times to submit market orders. The output of the self-organizing map was also used in conjunction with hierarchical clustering and principal component analysis to demonstrate further techniques that can be used for both visualization purposes as well as starting points for further research. All in all, I can conclude that as an unsupervised machine learning algorithm the SOM is certainly not the ideal solution for all predictive models nor is it the ideal robust visualization tool. However, its simplicity of use, the intuitive nature of its iterative algorithm as well as its ability to produce clear and concise visual representations of its output make it a powerful algorithm that all data scientists should have knowledge of.

Bibliography

[1] Osama Abu Abbas. Comparisons between data clustering algorithms. *International Arab Journal of Information Technology*, 5(3), 2008.

[2] Gennady Andrienko, Natalia Andrienko, Sebastian Bremm, Tobias Schreck, Tatiana Von Landesberger, Peter Bak, and Daniel Keim. Space in time and time in space self-organizing maps for exploring spatiotemporal patterns. In *Computer Graphics Forum*, 2010.

[3] Suwardi Annas, Takenori Kanai, and Shuhei Koyama. Principal component analysis and self-organizing map for visualizing and classifying fire risks in forest regions. *Agricultural Information Research*, 16(2):44–51, 2007.

[4] Marco Avellaneda and Jeong-Hyun Lee. Statistical arbitrage in the us equities market. *Quantitative Finance*, 10(7):761–782, 2010.

[5] Guilherme Barreto. Time series prediction with the self-organizing map. *Perspectives of neural-symbolic integration*, pages 135–158, 2007.

[6] Guilherme Barreto, João Mota, Luis Souza, and Rewbenio Frota. Nonstationary time series prediction using local models based on competitive neural networks. *Innovations in Applied Artificial Intelligence*, pages 1146–1155, 2004.

[7] Christopher M Bishop. *Neural Networks for Pattern Recognition*. Oxford university press, 1995.

[8] Adam Blazejewski and Richard Coggins. Application of self-organizing maps to clustering of high-frequency financial data. In *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, 2004.

[9] Nicolas Chapados. Sequential machine learning approaches for portfolio management, Université de Montréal Doctoral Thesis Repository. 2010.

[10] Marie Cottrell, Madalina Olteanu, Fabrice Rossi, and Nathalie Villa-Vialaneix. Theoretical and applied aspects of self-organizing maps. In *Advances in Self-Organizing Maps and Learning Vector Quantization*. Springer, 2016.

[11] Simon Dablemont, Geoffroy Simon, Amaury Lendasse, Alain Ruttiens, François Blayo, and Michel Verleysen. Time series forecasting with som and local non-linear models. In *Proceedings of the workshop on self-organizing maps*, 2003.

[12] Guido Deboeck and Teuvo Kohonen. *Visual Explorations in Finance: with Self-Organizing Maps*. Springer Science & Business Media, 2013.

[13] Pablo A Estévez, JC Prıncipe, and Pablo Zegers. *Advances in Self-Organizing Maps*. Springer, 2013.

[14] Simon Haykin. Neural networks: a comprehensive foundation. *The Knowledge Engineering Review*, 13(4):409–412, 1999.

[15] Rob J Hyndman et al. Another look at forecast-accuracy metrics for intermittent demand. *Foresight: The International Journal of Applied Forecasting*, 4(4):43–46, 2006.

[16] Ian T Jolliffe. Principal component analysis and factor analysis. In *Principal component analysis*. Springer, 1986.

[17] Michael Kearns and Yuriy Nevmyvaka. Machine learning for market microstructure and high frequency trading. *High frequency trading: New realities for traders, markets and regulators*, 2013.

[18] Teuvo Kohonen. Self-organizing maps: Optimization approaches. 1991.

[19] Teuvo Kohonen. Essentials of the self-organizing map. *Neural networks*, 37:52–65, 2013.

[20] Timo Koskela, Markus Varsta, Jukka Heikkonen, and Kimmo Kaski. Time series prediction using recurrent som with local linear models. *Int. J. of Knowledge-Based Intelligent Engineering Systems*, 2(1):60–68, 1998.

[21] Denis Kwiatkowski, Peter CB Phillips, Peter Schmidt, and Yongcheol Shin. Testing the null hypothesis of stationarity against the alternative of a unit root. *Journal of econometrics*, 54(1-3):159–178, 1992.

[22] Willem Melssen, Ron Wehrens, and Lutgarde Buydens. Supervised Kohonen networks for classification problems. *Chemometrics and Intelligent Laboratory Systems*, 83(2):99–113, 2006.

[23] Andrew Pole. *Statistical Arbitrage: Algorithmic Trading Insights and Techniques*, volume 411. John Wiley & Sons, 2011.

[24] Noelia Sánchez-Marono, Oscar Fontenla-Romero, Amparo Alonso-Betanzos, and Bertha Guijarro-Berdinas. Self-organizing maps and functional networks for local dynamic modeling. In *ESANN*, 2003.

[25] Bruno Silva and Nuno Cavalheiro Marques. Feature clustering with self-organizing maps and an application to financial time-series for portfolio selection. In *IJCCI*, 2010.

[26] Leonard J Tashman. Out of sample tests of forecasting accuracy. *International journal of forecasting*, 16(4):437–450, 2000.

[27] Juha Vesanto and Esa Alhoniemi. Clustering of the self-organizing map. *Transactions on neural networks*, 11(3):586–600, 2000.

[28] Xiaozhe Wang, Kate A Smith, Rob Hyndman, and Damminda Alahakoon. A scalable method for time series clustering. *Technical Report*, 2004.

[29] Ron Wehrens and Lutgarde MC Buydens. Self and super-organizing maps in r: the kohonen package. *J Stat Software*, 21(5):1–19, 2007.

[30] William WS Wei et al. *Time Series Analysis: Univariate and Multivariate Methods*. Pearson Addison Wesley, 2006.

[31] Hujun Yin. The self-organizing maps: Background, theories, extensions and applications. In *Computational intelligence: A compendium*, pages 715–762. Springer, 2008.

[32] Ban Zheng, Eric Moulines, and Frédéric Abergel. Price jump prediction in limit order book. 2012.

Appendix A

R CODE

```
## VISUALIZATION ##

# XYf-SOM #

TestingData<-apply(Prices[1:(nrow(Prices)-3443),],2,rev)
plot(TestingData[1:500,1],xlab="Order Executions",ylab="Price",main="X-Y Fused SOM")
lines(VectorofPredictionsXyf[1:500])

# Double VQTAM #

TestingData<-apply(Prices[1:(nrow(Prices)-3444),],2,rev)
plot(TestingData[1:500,1],xlab="Days",ylab="Price",main="Double VQTAM")
lines(VectorofPredictionsDoubleVQTAM[1:500])

# ARIMA #

TestingDataARIMA<-DataforARIMA[3444:length(DataforARIMA)]
plot(TestingDataARIMA[1:500],xlab="Days",ylab="Price",main="ARIMA")
lines(VectorofPredictionsARIMA[1:500])

# SIMPLE SOM #

set.seed(9999)

library(readr)

library("caret", lib.loc="/Library/Frameworks/R.framework/Versions/3.3/Resources/library")
library("kohonen", lib.loc="/Library/Frameworks/R.framework/Versions/3.3/Resources/library")
library(pastecs)

Limit_Order_Book_Data_Depth_2 <-
read_csv("~/Dropbox/Thesis/LOBSTER/Limit Order Book Data Depth 2 with time.csv")

View(Limit_Order_Book_Data_Depth_2)
```

```
LimitOrder<-as.matrix(scale(Limit_Order_Book_Data_Depth_2[,c(22,24,26,28)]))

View(LimitOrder)

## Training/Validating ##

TrainingData<-head(LimitOrder,round((2/3)*nrow(LimitOrder)))

nrow(TrainingData)

Folds<-createFolds(TrainingData[,1], k=2)

## K-Means ##

SSETrainingKmeans<-c(rep(NA,11))

SSEValidationKmeans<-c(rep(NA,11))

for (i in 1:11) {

  parameters<-somgrid(xdim = (i+4), ydim = (i+4), topo = "hexagonal",neighbourhood = "bubble")

  TrainingKmeans<-som(TrainingData[Folds$Fold1,], grid=parameters,radius =c(0.99,0.99))

  TrainingErrorsFold1<-sum(TrainingKmeans$distances)

  ValidationKmeans<-predict(TrainingKmeans,TrainingData[Folds$Fold2,])

  ValidationErrorsFold2<-c(rep(NA,length(Folds$Fold2)))

  Distances<-c(rep(NA,length(Folds$Fold2)))

  for (j in 1:length(Folds$Fold2)) {

    Distances[j]<-dist(rbind(TrainingData[Folds$Fold2,][j,],ValidationKmeans$predictions[[1]][j,]))
  }

  ValidationErrorsFold2<-sum(Distances^2)

  TrainingKmeans<-som(TrainingData[Folds$Fold2,], grid=parameters, radius=c(0.99,0.99))

  TrainingErrorsFold2<-sum(TrainingKmeans$distances)

  ValidationKmeans<-predict(TrainingKmeans,TrainingData[Folds$Fold1,])

  ValidationErrorsFold1<-c(rep(NA,length(Folds$Fold1)))

  Distances<-c(rep(NA,length(Folds$Fold1)))
```

```
  for (j in 1:length(Folds$Fold1)) {

    Distances[j]<-dist(rbind(TrainingData[Folds$Fold1,][j,],ValidationKmeans$predictions[[1]][j,]))
  }

  ValidationErrorsFold1<-sum(Distances^2)

  SSETrainingKmeans[i]<-(TrainingErrorsFold1+TrainingErrorsFold2)/length(TrainingData)

  SSEValidationKmeans[i]<-(ValidationErrorsFold1+ValidationErrorsFold2)/length(TrainingData)

}

plot(5:(length(SSEValidationKmeans)+4),SSETrainingKmeans,ylim=c(min(SSETrainingKmeans),
max(SSEValidationKmeans)),type="b",xlab="sqrt(#Neurons)",ylab="MSE",
main="Validation Curve",sub="K-means")
points(5:(length(SSEValidationKmeans)+4),SSEValidationKmeans, type="b",col=2)
legend(11,0.32, box.lty=0, legend=c("Validation Error", "Training Error"),
lty=1, col=c("red", "black"), cex=0.8)

## Bubble Neighborhood ##

SSETrainingBubble<-c(rep(NA,11))

SSEValidationBubble<-c(rep(NA,11))

for (i in 1:11) {

parameters<-somgrid(xdim = (i+4), ydim = (i+4), topo = "hexagonal", neighbourhood = "bubble")

TrainingSOM<-som(TrainingData[Folds$Fold1,], grid=parameters)

TrainingErrorsFold1<-sum(TrainingSOM$distances)

ValidationSOM<-predict(TrainingSOM,TrainingData[Folds$Fold2,])

ValidationErrorsFold2<-c(rep(NA,length(Folds$Fold2)))

Distances<-c(rep(NA,length(Folds$Fold2)))

for (j in 1:length(Folds$Fold2)) {

  Distances[j]<-dist(rbind(TrainingData[Folds$Fold2,][j,],ValidationSOM$predictions[[1]][j,]))
}

ValidationErrorsFold2<-sum(Distances^2)

TrainingSOM<-som(TrainingData[Folds$Fold2,], grid=parameters)
```

```
TrainingErrorsFold2<-sum(TrainingSOM$distances)

ValidationSOM<-predict(TrainingSOM, TrainingData[Folds$Fold1,])

ValidationErrorsFold1<-c(rep(NA, length(Folds$Fold1)))

Distances<-c(rep(NA, length(Folds$Fold1)))

for (j in 1:length(Folds$Fold1)) {

  Distances[j]<-dist(rbind(TrainingData[Folds$Fold1,][j,], ValidationSOM$predictions[[1]][j,]))
}

ValidationErrorsFold1<-sum(Distances^2)

SSETrainingBubble[i]<-(TrainingErrorsFold1+TrainingErrorsFold2)/length(TrainingData)

SSEValidationBubble[i]<-(ValidationErrorsFold1+ValidationErrorsFold2)/length(TrainingData)

}

plot(5:(length(SSEValidationBubble)+4),SSETrainingBubble,ylim=c(min(SSETrainingBubble),
max(SSEValidationBubble)),type="b",xlab="sqrt(#Neurons)",ylab="MSE",
main="Validation Curve",sub="Bubble Neighborhood")
points(5:(length(SSEValidationBubble)+4),SSEValidationBubble, type="b",col=2)
legend(11,0.32, box.lty=0, legend=c("Validation Error", "Training Error"),
lty=1, col=c("red", "black"), cex=0.8)

## Gaussian Neighborhood ##

SSETrainingGaussian<-c(rep(NA,11))

SSEValidationGaussian<-c(rep(NA,11))

for (i in 1:11) {

  parameters<-somgrid(xdim = (i+4), ydim = (i+4), topo = "hexagonal", neighbourhood = "gaussian")

  TrainingSOM<-som(TrainingData[Folds$Fold1,], grid=parameters)

  TrainingErrorsFold1<-sum(TrainingSOM$distances)

  ValidationSOM<-predict(TrainingSOM, TrainingData[Folds$Fold2,])

  ValidationErrorsFold2<-c(rep(NA, length(Folds$Fold2)))

  Distances<-c(rep(NA, length(Folds$Fold2)))
```

```
  for (j in 1:length(Folds$Fold2)) {

    Distances[j]<-dist(rbind(TrainingData[Folds$Fold2,][j,],ValidationSOM$predictions[[1]][j,]))
  }

  ValidationErrorsFold2<-sum(Distances^2)

  TrainingSOM<-som(TrainingData[Folds$Fold2,], grid=parameters)

  TrainingErrorsFold2<-sum(TrainingSOM$distances)

  ValidationSOM<-predict(TrainingSOM,TrainingData[Folds$Fold1,])

  ValidationErrorsFold1<-c(rep(NA,length(Folds$Fold1)))

  Distances<-c(rep(NA,length(Folds$Fold1)))

  for (j in 1:length(Folds$Fold1)) {

    Distances[j]<-dist(rbind(TrainingData[Folds$Fold1,][j,],ValidationSOM$predictions[[1]][j,]))
  }

  ValidationErrorsFold1<-sum(Distances^2)

  SSETrainingGaussian[i]<-(TrainingErrorsFold1+TrainingErrorsFold2)/length(TrainingData)

  SSEValidationGaussian[i]<-(ValidationErrorsFold1+ValidationErrorsFold2)/length(TrainingData)

}

plot(5:(length(SSEValidationGaussian)+4),SSETrainingGaussian,ylim=c(min(SSETrainingGaussian),
max(SSETrainingGaussian)),type="b",xlab="sqrt(#Neurons)",ylab="MSE",
main="Validation Curve",sub="Gaussian Neighborhood")
points(5:(length(SSEValidationGaussian)+4),SSEValidationGaussian, type="b",col=2)
legend(11,0.35, box.lty=0, legend=c("Validation Error", "Training Error"),
lty=1, col=c("red", "black"), cex=0.8)

## Testing ##

set.seed(9999)

TestingData<-LimitOrder[round((2/3)*nrow(LimitOrder)+1):nrow(LimitOrder),]

nrow(TestingData)

parameters<-somgrid(xdim = 10, ydim = 10, topo = "hexagonal",neighbourhood = "gaussian" )

TestingSOM<-som(TestingData, grid=parameters)
```

```
# Training Progress #

plot(TestingSOM, type="changes")

# Node Count #

plot(TestingSOM, type="count")

# Codes / Weight vectors #

plot(TestingSOM, type="codes")

## SUPERCLUSTERS ##

HierClust<-hclust(dist(unlist(TestingSOM$codes)))

KohonenCluster<-cutree(HierClust,4)

plot(HierClust,labels = FALSE,main = "Cluster Dendrogram",
xlab="Synaptic Weights", ylab = "Height",sub="")

Pal <- colorRampPalette(c('blue','yellow','green','red'))
plot(TestingSOM, type="mapping", main = "Superclusters", bgcol = Pal(4)[KohonenCluster])

SuperClusters<-c(rep(1,nrow(TestingData)))
for (i in 1:nrow(TestingData))
{SuperClusters[i]=KohonenCluster[TestingSOM$unit.classif[i]]}

## DESCRIPTIVE STATS ##

DataForStats<-Limit_Order_Book_Data_Depth_2[round((2/3)*nrow(LimitOrder)+1):nrow(LimitOrder),]

DataForStats<-cbind(DataForStats,SuperClusters)

View(DataForStats)

write.csv(DataForStats, "/Users/maxim/Dropbox/DataForStatsEvenSplit.csv")

## SUPERCLUSTER 1 ##

SC1Vectors<-subset(DataForStats,SuperClusters==1)

Averages<-c(mean(SC1Vectors[,22]),mean(SC1Vectors[,24]),mean(SC1Vectors[,26]),mean(SC1Vectors[,28]))

BarPlotSC1<-barplot(Averages, main="SC1 Bid Ask Volume Imbalance",
names.arg=c("BAVI1","BAVI2","BAVI3","BAVI4"))

StatsForBuyOrdersSC1<-subset(SC1Vectors,SC1Vectors[,33] == 1)
```

```
StatsForSellOrdersSC1<-subset(SC1Vectors,SC1Vectors[,33] == -1)

TotalNumberofBuyUpMovementsSC1<-table(StatsForBuyOrdersSC1[,32])/nrow(StatsForBuyOrdersSC1)
TotalNumberofSelUpMovementsSC1<-table(StatsForSellOrdersSC1[,32])/nrow(StatsForSellOrdersSC1)

## SUPERCLUSTER 2 ##

SC2Vectors<-subset(DataForStats,SuperClusters==2)

Averages<-c(mean(SC2Vectors[,22]),mean(SC2Vectors[,24]),mean(SC2Vectors[,26]),mean(SC2Vectors[,28]))

BarPlotSC2<-barplot(Averages, main="SC2 Bid Ask Volume Imbalance",
names.arg=c("BAVI1","BAVI2","BAVI3","BAVI4"))

StatsForBuyOrdersSC2<-subset(SC2Vectors,SC2Vectors[,33] == 1)

StatsForSellOrdersSC2<-subset(SC2Vectors,SC2Vectors[,33] == -1)

TotalNumberofBuyUpMovementsSC2<-table(StatsForBuyOrdersSC2[,32])/nrow(StatsForBuyOrdersSC2)
TotalNumberofSelUpMovementsSC2<-table(StatsForSellOrdersSC2[,32])/nrow(StatsForSellOrdersSC2)

## SUPERCLUSTER 3 ##

SC3Vectors<-subset(DataForStats,SuperClusters==3)

Averages<-c(mean(SC3Vectors[,22]),mean(SC3Vectors[,24]),mean(SC3Vectors[,26]),mean(SC3Vectors[,28]))

BarPlotSC3<-barplot(Averages, main="SC3 Bid Ask Volume Imbalance",
names.arg=c("BAVI1","BAVI2","BAVI3","BAVI4"))

StatsForBuyOrdersSC3<-subset(SC3Vectors, SC3Vectors[,33] == 1)

StatsForSellOrdersSC3<-subset(SC3Vectors, SC3Vectors[,33] == -1)

TotalNumberofBuyUpMovementsSC3<-table(StatsForBuyOrdersSC3[,32])/nrow(StatsForBuyOrdersSC3)
TotalNumberofSelUpMovementsSC3<-table(StatsForSellOrdersSC3[,32])/nrow(StatsForSellOrdersSC3)

## SUPERCLUSTER 4 ##

SC4Vectors<-subset(DataForStats,SuperClusters==4)

Averages<-c(mean(SC4Vectors[,22]),mean(SC4Vectors[,24]),mean(SC4Vectors[,26]),mean(SC4Vectors[,28]))

BarPlotSC4<-barplot(Averages, main="SC4 Bid Ask Volume Imbalance",
names.arg=c("BAVI1","BAVI2","BAVI3","BAVI4"))

StatsForBuyOrdersSC4<-subset(SC4Vectors,SC4Vectors[,33] == 1)

StatsForSellOrdersSC4<-subset(SC4Vectors,SC4Vectors[,33] == -1)
```

```
TotalNumberofBuyUpMovementsSC4<-table(StatsForBuyOrdersSC4[,32])/nrow(StatsForBuyOrdersSC4)
TotalNumberofSelUpMovementsSC4<-table(StatsForSellOrdersSC4[,32])/nrow(StatsForSellOrdersSC4)

## GRAPH OF OBSERVATIONS BASED ON SUPERCLUSTERS ##

Price<-unlist(DataForStats[1:100,18])

Colors<-Pal(4)[SuperClusters][1:100]
Time<-c(seq(1:length(Price)))
plot(Time,Price,pch=20,col=Colors,xlab="Trades",ylab="Price",main="Apple")
lines(Price)
legend("topright",box.lty=0, legend=c("SC1","SC2","SC3","SC4"), lty=1,
col=c('blue','yellow','green','red'), cex=0.6)

## PRINCIPAL COMPONENT ANALYSIS ##

library(devtools)
library(ggbiplot)

colnames(LimitOrder)<-c("BAVI1","BAVI2","BAVI3","BAVI4")

# Super Cluster 1 #

SC1_BAVA<-subset(LimitOrder,SuperClusters==1)

PCSC1<-prcomp(SC1_BAVA[1:100,])

ggbiplot(PCSC1, obs.scale = 1, var.scale = 1, main="SC1 Principal Components")

print(plot)

plot(PCSC1,type="lines")
summary(PCSC1)
print(PCSC1)

# Super Cluster 2 #

SC2_BAVA<-subset(LimitOrder,SuperClusters==2)

PCSC2<-prcomp(SC2_BAVA[1:100,])

ggbiplot(PCSC2, obs.scale = 1, var.scale = 1)

plot(PCSC2,type="lines")
summary(PCSC2)
print(PCSC2)

# Super Cluster 3 #
```

```
SC3_BAVA<-subset(LimitOrder,SuperClusters==3)

PCSC3<-prcomp(SC3_BAVA[1:100,])

ggbiplot(PCSC3, obs.scale = 1, var.scale = 1)

plot(PCSC3,type="lines")
summary(PCSC3)
print(PCSC3)

# Super Cluster 4 #

SC4_BAVA<-subset(LimitOrder,SuperClusters==4)

PCSC4<-prcomp(SC4_BAVA[1:100,])

ggbiplot(PCSC4, obs.scale = 1, var.scale = 1)

plot(PCSC4,type="lines")
summary(PCSC4)
print(PCSC4)

## IMPORT DATA ##

library(readr)
library(kohonen)

# DowJones20Years <- read_csv("~/Dropbox/Thesis/Data/DowJones20Years.csv")

AppleData <- read_csv("~/Dropbox/Thesis/LOBSTER/Limit Order Book Data Depth 2 with time.csv")

## DEFINE VARIABLES ##

set.seed(9999)

# Data <- DowJones20Years$`Adj Close`

Data <- AppleData$Price

length(Data)

NumberofLags<-5

PricewithLags<- function(NumberofLagsforPrice, NumberofValues, VectorofPrices) {

  Price<-matrix(NA, nrow =NumberofValues-(NumberofLagsforPrice+1), ncol =NumberofLagsforPrice+2)

  for (i in 1:(NumberofLagsforPrice+2)) {
```

```
    Price[,(NumberofLagsforPrice+2)-(i-1)]<-VectorofPrices[(NumberofLagsforPrice+3-i):(NumberofValues+1-i)]
  }

  return(Price[,1:(NumberofLagsforPrice+1)])
}

Prices<-scale(PricewithLags(NumberofLags,length(Data),Data))

## VQTAM MODEL ##

UnitNumbers<-c(rep(NA,(nrow(Prices)-3443)))

ReliablePredictionMinMax<-c(rep(0,(nrow(Prices)-3443)))
ReliablePredictionCI<-c(rep(0,(nrow(Prices)-3443)))

## XYF-SOM ##

start.time <- Sys.time()

parameters<-somgrid(xdim = 10, ydim = 10, topo = "hexagonal", neighbourhood = "gaussian" )

VectorofPredictionsXyf<-c(rep(NA,(nrow(Prices)-3443)))

for (i in 1:length(VectorofPredictionsXyf)) {

TrainingData<-Prices[(nrow(Prices)-3441-i):(nrow(Prices)+1-i),]

XyfSOM<-xyf(Y=TrainingData[,1,drop=F],X=TrainingData[,2:ncol(TrainingData),drop=F],grid=parameters)
SOM<-som(TrainingData[,2:ncol(TrainingData),drop=F],grid=parameters)


TestingData<-t(as.matrix(Prices[(nrow(Prices)-3442-i),2:ncol(Prices)]))
PredictionXyfSOM<-predict(XyfSOM,newdata=TestingData,whatmap = 1)

VectorofPredictionsXyf[i]<-PredictionXyfSOM$prediction[[2]]
UnitNumbers[i]<-PredictionXyfSOM$unit.classif

VectorsinEachCluster<-subset(TrainingData[,2:ncol(TrainingData)],XyfSOM$unit.classif==UnitNumbers[i])

if (nrow(VectorsinEachCluster)>=2) {
MinValue<-apply(VectorsinEachCluster,2,min)
MaxValue<-apply(VectorsinEachCluster,2,max)
CILowerBound<-apply(VectorsinEachCluster,2, function(x) mean(x)
-qt(0.975,df=(length(x)-1))*sd(x)/sqrt(length(x)))
CIUpperBound<-apply(VectorsinEachCluster,2, function(x) mean(x)
+qt(0.975,df=(length(x)-1))*sd(x)/sqrt(length(x)))
}

if (nrow(VectorsinEachCluster)<2) {
```

```
  ReliablePredictionMinMax[i]=0
}
else if (all(MinValue<as.vector(TestingData) & as.vector(TestingData)<MaxValue)) {
  ReliablePredictionMinMax[i]=1
}
else
  ReliablePredictionMinMax[i]=0

if (nrow(VectorsinEachCluster)<2) {
  ReliablePredictionCI[i]=0
}
else if (all(CILowerBound<as.vector(TestingData) & as.vector(TestingData)<CIUpperBound)) {
  ReliablePredictionCI[i]=1
}
else
  ReliablePredictionCI[i]=0
}

TestingData<-apply(Prices[1:(nrow(Prices)-3443),],2,rev)
nrow(TestingData)
TestingData<-TestingData[-14805,]
VectorofPredictionsXyf<-VectorofPredictionsXyf[-14805]
length(VectorofPredictionsXyf)

end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken

# Overall Error #

NaiveForecast<-rowMeans(TestingData[,2:6])

MAPEXyf<-sum(abs((TestingData[,1]-VectorofPredictionsXyf)/TestingData[,1]))/nrow(TestingData)*100
MASEXyf<-(sum(abs(TestingData[,1]-VectorofPredictionsXyf)))/(sum(abs(TestingData[,1]-NaiveForecast)))
MSEXyf<-(sum((TestingData[,1]-VectorofPredictionsXyf)^2))/nrow(TestingData)

# Min/Max Error #

PredictionsMinMax<-subset(VectorofPredictionsXyf,ReliablePredictionMinMax[-14805]==1)
TestVectorsMinMax<-subset(TestingData,ReliablePredictionMinMax[-14805]==1)

NaiveForecast<-rowMeans(TestVectorsMinMax[,2:6])

MAPEXyfMinMax<-sum(abs((TestVectorsMinMax[,1]-PredictionsMinMax)/
TestVectorsMinMax[,1]))/nrow(TestVectorsMinMax)*100
MASEXyfMinMax<-(sum(abs(TestVectorsMinMax[,1]-PredictionsMinMax)))/
(sum(abs(TestVectorsMinMax[,1]-NaiveForecast)))
MSEXyfMinMax<-(sum((TestVectorsMinMax[,1]-PredictionsMinMax)^2))/
nrow(TestVectorsMinMax)
```

```
# CI Error #

PredictionsCI<-subset(VectorofPredictionsXyf,ReliablePredictionCI[-14805]==1)
TestVectorsCI<-subset(TestingData,ReliablePredictionCI[-14805]==1)

NaiveForecast<-rowMeans(TestVectorsCI[,2:6])

MAPEXyfCI<-sum(abs((TestVectorsCI[,1]-PredictionsCI)/TestVectorsCI[,1]))/nrow(TestVectorsCI)*100
MASEXyfCI<-(sum(abs(TestVectorsCI[,1]-PredictionsCI)))/(sum(abs(TestVectorsCI[,1]-NaiveForecast)))
MSEXyfCI<-(sum((TestVectorsCI[,1]-PredictionsCI)^2))/nrow(TestVectorsCI)

## DOUBLE VQTAM MODEL ##

set.seed(9999)

start.time <- Sys.time()

SizeofMap<-10

VectorofPredictionsDoubleVQTAM<-c(rep(NA,(nrow(Prices)-3444)))

NaiveForecast<-c(rep(NA,(nrow(Prices)-3444)))

for (i in 23921:length(VectorofPredictionsDoubleVQTAM)) {

TrainingData<-Prices[(nrow(Prices)-3441-i):(nrow(Prices)+1-i),]

SOM<-som(X=TrainingData,grid=somgrid(SizeofMap, SizeofMap, "hexagonal"))

MatrixofShocks<-matrix(nrow=(nrow(TrainingData)-1),ncol=ncol(TrainingData))

for (j in 1:nrow(MatrixofShocks)) {
  MatrixofShocks[j,]<-(TrainingData[j,]-TrainingData[j+1,])
}

SOMShocks<-som(X=MatrixofShocks, grid=somgrid(SizeofMap, SizeofMap, "hexagonal"))

TestingData<-t(as.matrix(Prices[(nrow(Prices)-3442-i),]))

NeuronsMappedtoinTestingSet<-map(SOM,TestingData)$unit.classif

NeuronandCorrespondingShocks<-cbind(SOM$unit.classif[2:nrow(TrainingData)],SOMShocks$unit.classif)

MatrixofCorrespondingShocks<-subset(NeuronandCorrespondingShocks,
NeuronandCorrespondingShocks[,1]==NeuronsMappedtoinTestingSet)[,2]

RandomShock<-SOMShocks$codes[[1]][sample(MatrixofCorrespondingShocks,1),1]
```

```
VectorofPredictionsDoubleVQTAM[i]<-TestingData[1]+RandomShock

NaiveForecast[i]<-mean(TestingData)

}

TestingData<-apply(Prices[1:(nrow(Prices)-3444),],2,rev)

end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken

# Overall Error #

MAPEDoubleVQTAM<-sum(abs((TestingData[,1]-VectorofPredictionsDoubleVQTAM)/
TestingData[,1]))/nrow(TestingData)*100
MASEDoubleVQTAM<-(sum(abs(TestingData[,1]-VectorofPredictionsDoubleVQTAM)))/
(sum(abs(TestingData[,1]-NaiveForecast)))
MSEDoubleVQTAM<-(sum((TestingData[,1]-VectorofPredictionsDoubleVQTAM)^2))/
nrow(TestingData)

library("forecast", lib.loc="/Library/Frameworks/R.framework/Versions/3.3/Resources/library")

DataforARIMA<-scale(rev(Data))

VectorofPredictionsARIMA<-c(rep(NA,length(DataforARIMA)-3443))

set.seed(9999)

start.time <- Sys.time()

for (i in 1:length(VectorofPredictionsARIMA)) {

  TrainingData <- DataforARIMA[i:(3443+i-1)]
  ARIMA<-auto.arima(ts(TrainingData))
  VectorofPredictionsARIMA[i]<-as.numeric(forecast(ARIMA,h=1)$mean)
}

VectorofPredictionsARIMA<-unlist(VectorofPredictionsARIMA)

TestingDataARIMA<-DataforARIMA[3444:length(DataforARIMA)]

end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken

MAPE_ARIMA<-sum(abs((TestingDataARIMA-VectorofPredictionsARIMA)/
TestingDataARIMA)/(length(TestingDataARIMA)))*100
MASE_ARIMA<-(sum(abs(TestingDataARIMA-VectorofPredictionsARIMA)))/
```

```
(sum(abs(tail(TestingDataARIMA,-1)-head(TestingDataARIMA,-1))))
MSE_ARIMA<-(sum((TestingDataARIMA-VectorofPredictionsARIMA)^2))/
length(TestingDataARIMA)$
```