

https://dspace.org/

The development of an inventory management tool using machine learning techniques, for an SME in the foodservice industry

Naidoo, Davina

2021
https://hdl.handle.net/10539/33363

Downloaded from a DSpace repository. https://dspace.org/



The development of an inventory management tool using machine learning techniques, for an SME in the food service industry

Student name: Davina Naidoo

Student number: 2329050

Supervisor: Dr. Joke Buhrmann

A research report submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, in partial fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, South Africa

July 2021

Declaration

I declare that this research report/dissertation is my own unaided work. It is being submitted for the Degree of Master of Science in Engineering to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination to any other University.

Davina N

(Signature of Candidate)

22nd day of July , 20 21 in Morningside

Abstract

This research paper investigated the applicability of a machine learning forecasting model, as a solution to improved inventory management for a local Small to Medium-sized Enterprise (SME). The investigation was conducted through the development of various neural networks using the SMEs Point of Sale data, which spanned two and a half years. Sales data was combined with external data, such as weather patterns and yearly periodicity to test possible correlation with demand. The neural networks developed were compared to two other forecasting methods using statistical performance measures. The methods compared were Support Vector Regression and an Auto-regressive Integrated Moving Average model. The analysis was carried out on one perishable item across models to test the validity of the predictive methods. This product's demand patterns were representative of the volatile demand patterns observed for products sold by the SME. Reliability was then tested using a second perishable product across models. After performing the analysis the Recurring Neural Network (using Long Short Term Memory) performed the best overall with a 20% Mean Absolute Percentage Error in predictions.

Acknowledgements

I'd like to thank my supervisor, Dr Buhrmann for her continuous support and guidance.

I'd also like to thank my Dad and Chetan, for all their help and support.

Contents

1 Introduction **1**
- 1.1 Research background . . . 1
- 1.2 Research motivation . . . 3
- 1.3 Research questions and objectives . . . 4
- 1.4 Research method overview . . . 5
- 1.5 Research limitations . . . 5
- 1.6 Report layout . . . 6

2 Literature Review **7**
- 2.1 Overview . . . 7
- 2.2 Time Series forecasting . . . 7
- 2.3 Significant works in the field . . . 9
- 2.4 Auto-regressive Integrated Moving Average (ARIMA) . . . 11
  - 2.4.1 Theoretical background . . . 11
  - 2.4.2 Software implementation of ARIMA . . . 13
- 2.5 Support Vector Machines (SVM) and Support Vector Regression (SVR) . 14
  - 2.5.1 Theoretical background . . . 14
  - 2.5.2 Software implementation of SVR . . . 17
- 2.6 Neural networks in forecasting . . . 17
  - 2.6.1 Overview . . . 17
  - 2.6.2 Theoretical background . . . 18
    - 2.6.2.1 Overview . . . 18
    - 2.6.2.2 Artificial Neural Networks (ANNs) . . . 19
    - 2.6.2.3 Deep Neural Networks (DNNs) . . . 20
    - 2.6.2.4 Dense Layers . . . 22
    - 2.6.2.5 Convolutional Neural Networks (CNNs) . . . 22
    - 2.6.2.6 Recurrent Neural Networks (RNNs) . . . 23
  - 2.6.3 Software implementation of neural networks . . . 24
    - 2.6.3.1 Multi step and single step neural networks . . . 24
    - 2.6.3.2 Data normalisation and standardisation in neural networks . . . 26
    - 2.6.3.3 Optimisers in neural networks . . . 27
    - 2.6.3.4 Activation functions in neural networks . . . 28
- 2.7 Comparison between traditional and machine learning techniques . . . 30
- 2.8 Methods to measure the performance of models . . . 31
  - 2.8.1 Mean Absolute Error (MAE) . . . 32
  - 2.8.2 Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) 32

2.8.3 Mean Absolute Percentage Error (MAPE) . . . 32

3 Research Method **34**
3.1 Methodology overview . . . 34
3.2 Branch and product selection method . . . 35
3.2.1 Overview of coffee franchise data . . . 36
3.2.2 Analysis on coffee franchise data . . . 38
3.3 Data and data preparation method . . . 42
3.3.1 Weather data set . . . 42
3.3.2 Data preparation . . . 44
3.4 Implementation method of benchmark models . . . 48
3.4.1 Environment Overview . . . 48
3.4.2 ARIMA . . . 48
3.4.2.1 Parameter tuning . . . 50
3.4.3 SVR . . . 52
3.4.3.1 Parameter tuning . . . 53
3.5 Implementation method of neural networks . . . 54
3.5.1 Environment overview . . . 54
3.5.2 Overview of the implementation method . . . 55
3.5.3 Baseline and linear models . . . 56
3.5.4 Dense layers . . . 57
3.5.5 CNN . . . 57
3.5.6 RNN (LSTM) . . . 57
3.5.7 Parameter tuning . . . 58
3.5.7.1 Features . . . 58
3.5.7.2 Batch size . . . 59
3.5.7.3 Epochs . . . 59
3.5.7.4 Optimiser . . . 59
3.5.7.5 Window size . . . 60
3.5.7.6 Activation function . . . 60
3.6 Methods to evaluate performance . . . 60
3.6.1 Measures to test the validity of the models . . . 60
3.6.2 Measures to test the reliability of the models . . . 61

4 Results and Analysis **63**
4.1 Overview . . . 63
4.2 Benchmark models' results and analysis . . . 64
4.2.1 ARIMA results and analysis . . . 64
4.2.2 SVR results and analysis . . . 66
4.3 Neural network results and analysis . . . 68
4.3.1 Parameter tuning results . . . 68
4.3.1.1 Features . . . 69
4.3.1.2 Batch size . . . 72
4.3.1.3 Epochs . . . 74
4.3.1.4 Optimiser . . . 80
4.3.1.5 Window size . . . 83
4.3.1.6 Activation functions . . . 87
4.3.1.7 Data split . . . 90

4.3.2 Baseline results and analysis . . . 91
4.3.2.1 Single step model . . . 91
4.3.2.2 Multi step model . . . 92
4.3.3 Dense layers results and analysis . . . 93
4.3.3.1 Single step model . . . 93
4.3.3.2 Multi step model . . . 95
4.3.4 CNN results and analysis . . . 96
4.3.4.1 Single step model . . . 96
4.3.4.2 Multi step model . . . 97
4.3.5 RNN (LSTM) results and analysis . . . 99
4.3.5.1 Single step model . . . 99
4.3.5.2 Multi step model . . . 100
4.4 Further analysis on the top performing models . . . 102
4.4.1 MAPE evaluation on the top performing models . . . 102
4.4.2 Product 2 results and analysis . . . 104

5 Discussion **108**
5.1 Discussion on the models developed for Product 1 . . . 108
5.1.1 Overview . . . 108
5.1.2 Discussion based on the model architecture . . . 109
5.1.3 Discussion on statistical measures . . . 111
5.2 Discussion on the models developed for Product 2 . . . 112
5.3 Discussion on the business operations . . . 112

6 Conclusion and Recommendations **114**
6.1 Conclusion . . . 114
6.2 Recommendations and future work . . . 116

Bibliography **118**

A Coffee franchise SQL output summary **125**

B Python code: ARIMA **127**

C Python code: SVR **133**

D Python code: Neural networks **145**

E Parameter tuning: Numerical results **176**

List of Figures

2.1 ACF Plots [11] . . . 11
2.2 SVR margin of error [25] . . . 15
2.3 Learning stages in neural networks [30] . . . 18
2.4 The architecture of a neural network [10] . . . 20
2.5 Dense Neural Network Representation on TensorFlow Playground [34] . 22
2.6 A simplified CNN architecture [36] . . . 23
2.7 An LSTM memory cell structure [14] . . . 24
2.8 Data window for a single step model [40] . . . 25
2.9 Data window for a multi step auto-regressive model [40] . . . 26
2.10 Sigmoid and tanh activation functions [44] . . . 29
2.11 ReLU activation function [45] . . . 30

3.1 Example of an SQL query . . . 37
3.2 The top 5 products sold by volume between 2016 and mid 2019 . . . 38
3.3 Daily number of muffins sold for 2016 across branches . . . 40
3.4 Daily number of breakfasts sold for 2016 across branches . . . 41
3.5 Daily number of croissants sold for 2016 across branches . . . 42
3.6 Plot of temperature and rain against number of Product 1 items sold between 2016 and mid 2019 . . . 45
3.7 sine and cosine curves representing various periodicity . . . 46
3.8 Normalised data . . . 47
3.9 ACF plot for 800+ lags . . . 49
3.10 ACF plot for 30 lags . . . 49
3.11 PACF plot for 30 lags . . . 50
3.12 Output from the automatic ARIMA function . . . 51
3.13 ARIMA residual density plot . . . 52
3.14 Python code extract to execute a grid search for RBF . . . 53
3.15 Python code to evaluate performance for grid search in SVR . . . 54

4.1 ARIMA output with normalised data . . . 64
4.2 SVR results from grid search . . . 67
4.3 Comparison of MAE across models that include all features, and those that include only sine and cosine curves of year periodicity . . . 70
4.4 Comparison of MSE across models that include all features, and those that include only sine and cosine curves of year periodicity . . . 71
4.5 Comparison of RMSE across models that include all features, and those that include only sine and cosine curves of year periodicity . . . 72
4.6 MAE across models with a batch size of 64 . . . 73

4.7 MSE across models with a batch size of 64 73
4.8 RMSE across models with a batch size of 64 74
4.9 Effect of number of epochs on MAE across models 75
4.10 Effect of number of epochs on MSE across models 76
4.11 Effect of number of epochs on RMSE across models 77
4.12 MAE across models with epoch = 100 and 50 and patience = 5 78
4.13 MSE across models with epoch = 100 and 50 and patience = 5 79
4.14 RMSE across models with epoch = 100 and 50 and patience = 5 80
4.15 Effects of optimiser on MAE across models 81
4.16 Effects of optimiser on MSE across models 82
4.17 Effects of optimiser on RMSE across models 83
4.18 Effects of window size on MAE across models 85
4.19 Effects of window size on MSE across models 86
4.20 Effects of window size on RMSE across models 87
4.21 Effects of activation functions (tanh and sigmoid) on MAE across models 88
4.22 Effects of activation functions (tanh and sigmoid) on MSE across models 89
4.23 Effects of activation functions (tanh and sigmoid) on RMSE across models 90
4.24 Single step model linear results 91
4.25 Single step linear learning curve 92
4.26 Multi step model linear results 92
4.27 Multi step linear learning curve 93
4.28 Single step dense layer learning curve 94
4.29 Single step multi input step dense layer input signal 94
4.30 Single step multi input step dense layer learning curve 95
4.31 Multi step model dense layer results 95
4.32 Multi step dense learning curve 96
4.33 Single step CNN results 97
4.34 Single step CNN learning curve 97
4.35 Multi step CNN results 98
4.36 Multi step CNN learning curve 98
4.37 Single step LSTM results 99
4.38 Single step LSTM learning curve 100
4.39 Multi step LSTM results 100
4.40 Multi step LSTM learning curve 101
4.41 Multi step AR LSTM results 101
4.42 Multi step AR LSTM learning curve 102
4.43 multi step CNN after inverse scaling - across 10 days of test data 103
4.44 multi step RNN (LSTM) after inverse scaling - across 10 days of test data 103
4.45 multi step dense layers after inverse scaling - across 10 days of test data . 103
4.46 multi step CNN with Product 2 106
4.47 multi step RNN with Product 2 106
4.48 multi step dense layers with Product 2 107

List of Tables

3.1 Coffee franchise data summary 37
3.2 Weather dataset summary 43
3.3 Automatic ARIMA modelling results with AIC values 50
3.4 Optimum parameters for SVR across kernels 54

4.1 Comparative statistical results of the tested methods on Product 1 64
4.2 Key differences in Product 1 data after normalisation 65
4.3 Performance of SVR model across kernels 67
4.4 Current and adjusted window size inputs for parameter tuning 84
4.5 Comparative statistical results for the tested methods using Product 2 data 104
4.6 Key differences in Product 2 data after normalisation 104

5.1 Converted MAE for Product 1 models analysed in Chapter 4 108

A.1 Summary of SQL output from coffee franchise data (colours indicate link variables across tables) 126

E.1 MAE results for parameter tuning numerical analysis 177
E.2 MSE results for parameter tuning numerical analysis 178
E.3 RMSE results for parameter tuning numerical analysis 179

Acronyms

2D Two Dimensional.

4IR The Fourth Industrial Revolution.

ACF Auto-correlation Function.

Adam Adaptive Moment Estimation.

ADF Augmented Dickey-Fuller.

AI Artificial Intelligence.

AIC Akaike Information Criterion.

ANN Artificial Neural Network.

AR Auto-regressive.

ARIMA Auto-regressive Integrated Moving Average.

CNN Convolutional Neural Network.

DNN Deep Neural Network.

FMCG Fast Moving Consumer Goods.

IT Information Technology.

LSTM Long Short-Term Memory.

MA Moving Average.

MAE Mean Absolute Error.

MAPE Mean Average Percentage Error.

MLR Multiple Linear Regression.

MSE Mean Squared Error.

PACF Partial Auto-correlation Function.

POS Point of Sale.

RBF Radial Basis Function.

ReLU Rectified Linear Unit.

RMSE Root Mean Squared Error.

RMSprop Root Mean Square Propagation.

RNN Recurrent Neural Network.

SGD Stochastic Gradient Descent.

SME Small to Medium-sized Enterprise.

SVM Support Vector Machine.

SVR Support Vector Regression.

Chapter 1

Introduction

1.1 Research background

The Fourth Industrial Revolution (4IR) signifies the incorporation of Information Technology (IT) into business processes [1]. More specifically, 4IR has given rise to the ability to analyse and extract valuable insights from big data. This has changed the way products and services are developed and marketed to consumers. These key characteristics of 4IR have created a world where information from intelligent machines is at the forefront of profit driven initiatives.

The food and beverage service industry is often exposed to volatile demand fluctuations. These fluctuations are based on uncontrollable factors that affect consumers' day to day demand. This not only creates a high level of food wastage but also creates instances of unfulfilled demand. Therefore, the ability to predict demand with a high accuracy is crucial in improving profitability within the industry.

The adoption of data analytics by large brands has caused a disruption within the industry as it leverages the knowledge of individual customer needs. An example of this is the recent initiative by McDonald's to reduce costs and improve efficiencies. By using big data, AI and data analytics, they have created a tailored customer experience through the

personalisation of their mobile application service [2]. The insights gathered from the application allow McDonald's to understand demand patterns and significant trends, thus continuously enhancing their business and operational improvements.

The predictive tools and technologies that have been developed to improve efficiency are widely available to large enterprises. From a cost perspective, they become a less viable option for Small to Medium-sized enterprises (SMEs). This highlights the issue that large scale solutions do not often address the needs and constraints of SMEs [3]. Ironically, small enterprises are more likely to operate on a finer line of repercussions from inefficiencies, but are the last to gain from the developments made to reduce this.

A study conducted in 2012 [2] in the UK found SMEs had a 0.2% adoption rate of big data analytics, compared to a rate of 25% in larger companies (1000+ employees). While the number of SME's incorporating data analytics in their strategies has grown since 2012, the gap between small and large companies still remains. Because customer needs are fulfilled in a customised way with large scale companies, small enterprises are under pressure to deliver the same level of service to their consumers.

The realm of data analytics within the food service industry has pushed all boundaries. The building block on which all subsequent analytical technologies has stemmed from is the extraction and analysis of Point of Sale (POS) data [4]. This is an underestimated but a critical step for any business adding an additional layer of understanding to the intricacies of consumer behaviour.

For this project, one such SME (a coffee franchise) was investigated to understand how data analytics could improve their operations. The company has 10 branches around the city of Johannesburg strategically situated in central business areas. The franchise is known as a fast turnover café with shelved food items and coffee product offerings. For the purposes of non-disclosure, the shop was not named.

1.2 Research motivation

The ability to predict and manage stock on an ongoing basis has proven successful in larger enterprises. A study conducted in Nigeria tested the value of an inventory management system on a 7up (a beverage brand created in America) bottling store. The research found that implementation of a flexible inventory management system had a strong relationship to improved organisational performance [5].

SMEs in South Africa rarely have targeted inventory management tools available at a price point that is affordable to them. The motivation for this research project came from the need to equip an SME in the food and beverage service industry with the ability to harness data analytics to improve their inventory management.

As with most small businesses, the coffee franchise under investigation lacked an understanding of the drivers of their customer behaviour. The result of this was product quantities that did not reflect consumer demand. This inevitably led to difficulties in understanding how to increase business profitability through top line growth.

Two previous Honours research projects at the University of Witwatersrand were conducted on the coffee franchise in 2019. The first analysis into the coffee shop [6] focused mainly on descriptive analytics using dashboarding tools to graphically visualise the current operations. The study also looked at an Auto-regressive Integrated Moving Average (ARIMA) forecasting methodology (described in Chapter 2) utilising Holt Winters exponential smoothing forecast to analyse the data from a predictive perspective.

The data for the project was provided by the coffee franchise through their POS system. This data included total sales, quantity and items per branch from 2016 to 2019. Through the analysis of this data, it was concluded that the tools utilised could not provide accurate forecasting due to data volatility and insufficient information on the cost of goods.

The second analysis on the same coffee shop was run in parallel in 2019 [7]. The analysis focused on developing predictive inventory models for each branch. The Newsvendor

mathematical model was utilised (explained in chapter 2) with inventory data from 2018 to 2019. The accuracy of the models could not be confirmed due to limited data at the time the research was conducted. The study made an assumption that the the demand was constant throughout the year. Given this assumption it was likely the predicted demand was largely different to the actual demand, as the demand was known to be volatile.

The two research investigations conducted proved that there was a gap in the coffee franchises ability to accurately predict future demand. Utilising machine learning techniques may provide the accuracy that is currently missing. As the algorithms have been known to deal with volatile data far more accurately than traditional statistical methods [8].

1.3 Research questions and objectives

The primary research question can be framed as: With the help of machine learning techniques, how can the coffee franchise improve their understanding of customer behaviour and future demand, such that inventory management can be correctly aligned to consumers' needs and preferences?

This research seeks to:

- Produce a machine learning tool that is able to predict product demand using past sales data; habits and demands of consumers based on time of year, month and weather patterns.
- Evaluate the coffee franchise holistically for opportunities to further understand customer behaviour and demand.
- Determine how these findings can streamline their inventory management for overall improvement in profitability.

1.4 Research method overview

To implement data analytics into the SMEs business operations, the first step was accessing and extracting the POS data that was stored on a central database. This data was interpreted and analysed to understand what products and branches were the most sensible to use in the development of a predictive tool. Subsequent to this, an intelligent system was created using predictive inventory management methods. The methods utilised encompassed both traditional statistical techniques and machine learning methods. The results of the methods were analysed and compared to determine the success of the predictive methods tested.

The predictive model that was developed can be scaled and practically used as an inventory management tool by the SME. All products would have their own specific demand forecasting that would feed into the businesses resourcing decisions on a weekly or fortnightly basis.

The coffee franchise was also evaluated holistically. This was done to determine whether any key business processes could change, in order to better manage their perishable inventory and better understand their customer segment.

1.5 Research limitations

The coffee franchise has provided POS data from 2016 to June 2019 across their branches. This resulted in 860 data points (weekends and public holidays did not have any associated data). This data indicated all product quantities sold, but lacked information on stock availability for a specific day. Since there was not a way to determine if there was a shortage of a product on a given day, an assumption was made that no excess demand was left unfulfilled. The owner of the franchise has confirmed the validity in this assumption based on his operations. Details on the day-to-day operations of the business from stock fulfilment, stock taking and current demand management techniques were provided.

The results of the machine learning models presented in Chapter 4 have been reported as single values. A limitation of the study exists in that repeatedly running a model without altering any inputs will still result in slight variations in the results, given the different weighting the model assigns based on learning [9]. The nature of this data is chronological, therefore it was not shuffled when input into the model. This reduced the difference in output results significantly which was observed during testing.

1.6 Report layout

This report captured the process of creating a demand forecasting model. The breakdown of chapters was as follows:

- Chapter 2 covered a literature review on the concepts, techniques and past work in the fields that were explored and tested.
- Chapter 3 outlined the research methodology from data preparation to statistical methods used to compare accuracy's of the different models.
- Chapter 4 presented and analysed the results.
- Chapter 5 discussed and compared the outcomes of Chapter 4 to further understand the results.
- Chapter 6 concluded the report and highlighted further research and recommendations.

Chapter 2

Literature Review

2.1 Overview

This literature study was divided into five sections. Section 2.2 introduced the topic of time series forecasting. This was followed by a review of notable works in the field of time series forecasting specific to demand predictions. The focus of the Chapter was on methods that were pursued within this study based on the findings in Section 2.3. In Section 2.4, 2.5 and 2.6 the traditional statistical method of ARIMA and machine learning methods to solving time series forecasts were discussed respectively. These sections covered both the theoretical background of the methods and the respective software implementation theory. Section 2.7 compared the two approaches (machine learning versus statistical methods) and their relative success and shortfalls within the field. Section 2.8 highlighted the statistical performance measures utilised to evaluate the success of various models. These methods were implemented into the study to draw comparisons.

2.2 Time Series forecasting

A time series can be described as a series of data that is observed sequentially over a specific period of time [10]. The aim of time series forecasting is to utilise past data to

predict future outcomes based on historical observations of trends and patterns.

There are three main types of models in time series forecasting, each representing different ways in which inputs (commonly known as predictor variables) may influence the predicted outcome. The first is an explanatory model, where variables are assumed to have a relationship with the output variable along with an "error" variable to account for random variation that may occur [11]. For example, demand can be seen as a function of a number of variables illustrated below [11]:

$$Demand = f(temperature, time\,of\,week, time\,of\,month, time\,of\,year, error) \tag{2.1}$$

The second is a time series model which uses historical data to predict the outcome. This model does not make use of external variables as the explanatory model does. This is represented in the equation below [11]:

$$Demand_{t+1} = f(Demand_t, Demand_{t-1}, ..., error) \tag{2.2}$$

The third type of model is a mixed model, which is a combination of the explanatory and time series model described above.

Forecasting methods rely heavily on the identification of trends, seasonal patterns and or cyclic patterns within the data. There are slight nuances that differentiate these patterns. Trends are clearly observed increases or decreases in the data. Seasonal patterns usually present themselves graphically as an oscillating frequency depending on the time of year or month. Cyclic patterns are observed when there is no fixed frequency but rises and falls are apparent. Data with random fluctuations that do not exhibit the patterns described are often difficult to use to build a predictive model [11].

Historically, time series forecasting have used statistical techniques such as the ARIMA model, which has been one of the more successful methods to compute forecasts [11]. In recent years machine learning has proved to be successful in its ability to predict based

on highly volatile historical data (data with random fluctuations). This paper explored and compared these techniques in their applicability to forecast demand for the coffee franchise.

2.3 Significant works in the field

Time series forecasting is used in problem solving in many fields. Many researchers have tailored forecasting techniques, in some cases using a hybrid of techniques in order to optimise results.

In 2012 Karin Kandananond [10] explored forecasting techniques that dealt specifically with auto-correlated time series. These are time series that when compared to a lagged version of the same series show a visible trend or similarity in the pattern. Auto-correlated time series are not stationary by nature. Methods compared in Kandananond's analysis were Artificial Neural Networks (ANN), Support Vector Machines (SVM) and ARIMA. The findings of the research referenced above showed that for the specific forecasting of a consumer goods company, SVM was the most successful method.

In 2018, Takashi Tanizaki set out to forecast demand in restaurants [12] using machine learning. The following methods were tested: Bayesian Linear, Boosted Decision And Decision Forest Regression, using the Step-wise method (a statistical analysis method). His research reported forecasting success rates of approximately 85% across these methods.

An exploratory research paper reported in 2019 [13] looked into the use of machine learning for predictions in Fast Moving Consumer Goods (FMCG). Their research showed that from 2013 there has been a significant increase in the number of articles that explored machine learning within the field of forecasting. A general consensus within the articles reviewed was that machine learning had proven to be significantly more successful in accurate demand forecasting than statistical methods, more specifically, deep learning.

In 2020 Hyojoo Son and Changwan Kim [14] utilised deep learning to forecast electricity demand, using twenty-two years of data with variables such as economic performance, demographics and weather conditions. These variables were non-linear, making machine learning a better methodology due its ability to handle volatility in data. The paper used a Long Short-Term Memory (LSTM) neural network as the preferred method. The results of the LSTM neural network were compared to several other machine learning and statistical tools (SVR, ANN, ARIMA). The conclusion drawn was that LSTM outperformed all four of the models in comparison. The comparison was carried out through the use of various statistical performance measurement tools such as: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE) which was discussed further in Section 2.8.

The above mentioned papers along with numerous others in the field: [15], [16], [17], [18] had seen great success in the incorporation of machine learning into their forecasting. Within demand predictions two applicable papers, one dealing with electricity demand [14] and the other dealing with sales demand [18] exhibited similar data types and dependant variables. Using this information along with in depth research into powerful machine learning tools to predict demand, various neural networks were applied to the coffee franchise data (specific focus on the LSTM method within Recurring Neural Networks (RNN)). Neural network models were then compared to another successful method in machine learning known as SVM (specifically Support Vector Regression (SVR)) along with the ARIMA statistical technique.

There is a large scope of methodologies that exist within time series forecasting. Therefore, it was necessary to prioritise methodologies covered in this paper given the existing research available. This literature review confined itself to the selected methods and topics explored in this paper.

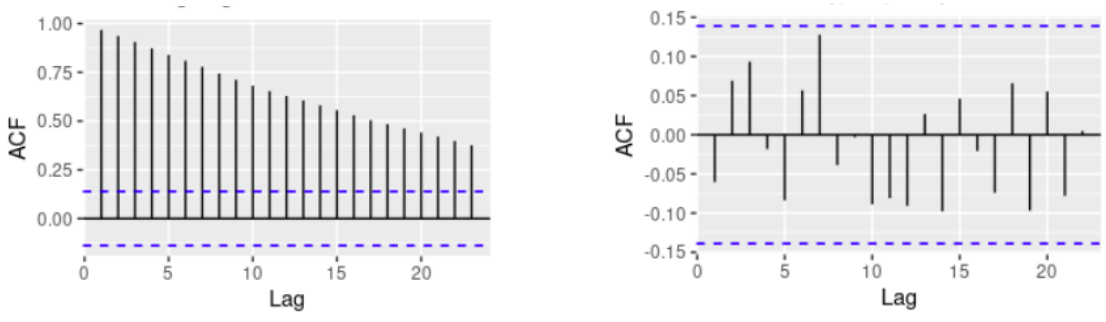
2.4 Auto-regressive Integrated Moving Average (ARIMA)

2.4.1 Theoretical background

The ARIMA method is one of the more common methods utilised in time series forecasting. As implied in its title, it combines the theories behind auto-regressive, integrated and moving average models [11].

An ARIMA model requires a stationary time series. This is a time series that is independent of time i.e. it exhibits no trends or seasonality. A series that is dependent on time can be transformed to a stationary time series via the method of differencing. Differencing is essentially a plot of the difference in subsequent values of the time series in question. Using the plot of the differenced values creates a new time series that should be stationary. If the series is still not stationary the method is repeated. This gives rise to different orders of differencing. The process of creating a stationary time series represents the "integrated" aspect of ARIMA [11].

While it is possible to identify whether a time series is stationary or not from the plot of the series itself, an alternative method is the use of an auto-correlation function (ACF) plot [11]. The ACF plot illustrates any correlation a series may have by evaluating the series against a number of lagged versions of the series. Figure 2.1 illustrates how one may extract an understanding of whether a series is stationary or not.



(a) non-stationary

(b) stationary

Figure 2.1: ACF Plots [11]

Figure 2.1a exhibits a typical non-stationary series where the ACF plot moves to zero

slowly. Figure 2.1b is an example of a stationary series, the ACF plot quickly jumps to zero and there are no correlations that lie outside of the 95% limit (the blue dotted line) [11]. The 95% confidence limit indicates the threshold for which data shows correlation. Points that are above this limit exhibit a strong correlation, whereas the data points of uncorrelated sequences fit within the band [19]. The equation used to calculate a 95% confidence band is as follows [19]:

$$\pm\frac{1.96}{\sqrt{N}} \tag{2.3}$$

where 1.96 is derived from the cumulative distribution function of the standard normal distribution and N is the sample size.

Auto-regression relies on the previous values to output a future value. This differs from linear regression which makes use of both dependent and independent variables. An auto-regressive model of order p can be written as [11]:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + ... + \phi_p y_{t-p} + \varepsilon_t \tag{2.4}$$

where y_t represents the dependant variable at time interval t, c represents a constant, ϕ represents a parameter and ε_t represents white noise.

A Moving Average (MA) model can be represented by the following formula [11]:

$$y_t = c + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + ... + \theta_q \varepsilon_{t-q} + \varepsilon_t \tag{2.5}$$

where q represents the order of the MA model, θ represents a parameter and y_t then becomes a weighted MA of past forecast errors.

Combining the methodologies explained above, results in a non-seasonal ARIMA model, represented by the following [11]:

$$y_t^{'} = c + \phi_1 y_{t-1}^{'} + ... + \phi_p y_{t-p}^{'} + \theta_1 \varepsilon_{t-1} + ... + \varepsilon_t \tag{2.6}$$

where $y_t^{'}$ is the differenced series.

This ARIMA model is better expressed in the form of ARIMA(p,d,q) where p represents the coefficient of the order of auto-regression (also known as the lag order), d is the degree of differencing and q is the order of the MA element [11]. The order of the MA element indicates the size of the MA window from which the residual errors are linearly combined [20].

To gauge the order of a model, a number of methods can be used. Using the ACF and the partial ACF (PACF) plot one can deduce the p coefficient through inspection. In more recent years the use of an automatic computing function (available as a package on various programming languages) allows the statistical analysis to be performed automatically on an input data set. This function calculates the most optimum model [11].

2.4.2 Software implementation of ARIMA

As mentioned above, there are statistical packages available that automatically perform the statistical analysis on the input data to generate an ideal model fit. The automatic function (known as *auto.arima* in Python, a computer programming language) utilises a variation of the Hyndman-Khandakar algorithm which makes use of unit root tests and the Akaike Information Criterion (commonly known as AIC) to automate the ARIMA function [11]. Both of these concepts are elaborated on in subsequent paragraphs.

A unit root test is able to test for stationarity, which indicates whether differencing is required. Unit root tests make strong assumptions regarding the input data which is used as a hypothesis to either "reject" or "fail to reject" within the algorithm [11]. For this reason the unit root test is rather a tool to assist in confirmation, but should not be used as the only tool in determining the characteristics of a data set.

One commonly used unit root test is the Augmented Dickey-Fuller (ADF) test. The ADF test begins with a null hypothesis that there is time dependency within the data. This is either confirmed or rejected [21]. The output of the ADF test is a $p-value$ that sets

the threshold for stationary data versus non-stationary data. If the $p-value$ is less than 0.05 the data is considered to be stationary and if it is greater than 0.05 the data is not stationary, indicating that the method of differencing will be required.

The AIC is given by the following equation [11]:

$$AIC = -2log(L) + 2(p+q+k+1) \tag{2.7}$$

where L is the likelihood of the data (this relates to a statistical output on the fit of the data to the model), $k = 1$ if $c \neq 0$ (c representing the constant in Equation 2.6) and $k = 0$ if $c = 0$ (the parenthesis in the equation refers to the number of parameters in the model). Minimising the AIC equation will result in optimised parameters for the ARIMA model [11]. As evident from Equation 2.7 AIC is only useful in selecting the values for p and q but not d. The parameter d relies on unit root tests to inform its value.

The automatic fitting of the ARIMA model uses the theory described above. This automated function also allows the user to modify internal settings for customised use. By switching the step-wise or approximation options off within the function, the algorithm's built in functionality to speed up the process is stopped. This allows the algorithm to rather search through a wider model space, which can produce better results [11].

2.5 Support Vector Machines (SVM) and Support Vector Regression (SVR)

2.5.1 Theoretical background

SVM was developed in 1995 by Vladimir Vapnik. It is commonly used as a classification tool within machine learning. The concept behind SVM can also be applied to regression problems, this is known as SVR [22].

SVM methodology makes use of a hyper-plane that acts as a decision boundary. The

objective of the model is to maximise the margin, which is defined as the distance between the hyper-plane and specific training samples known as support vectors. By maximising the margin, the model is able to avoid over fitting and have a lower generalisation error [23].

In nonlinear problems a kernel SVM is utilised. Kernels effectively transform a two dimensional (2D) plane into a multidimensional space. This allows a linear hyper-plane that is created in a multidimensional space to project back as a nonlinear hyper-plane in the original 2D space [24].

In 1997 Drucker et al. [22] created the SVR extension to SVM. Their technique involved the use of a "tube" of margin ε (epsilon) used in approximating a regression function. Similar to SVM, the margin is created through the use of support vectors. Figure 2.2 below illustrates a simplistic view of the margin on a 2D plane [25].

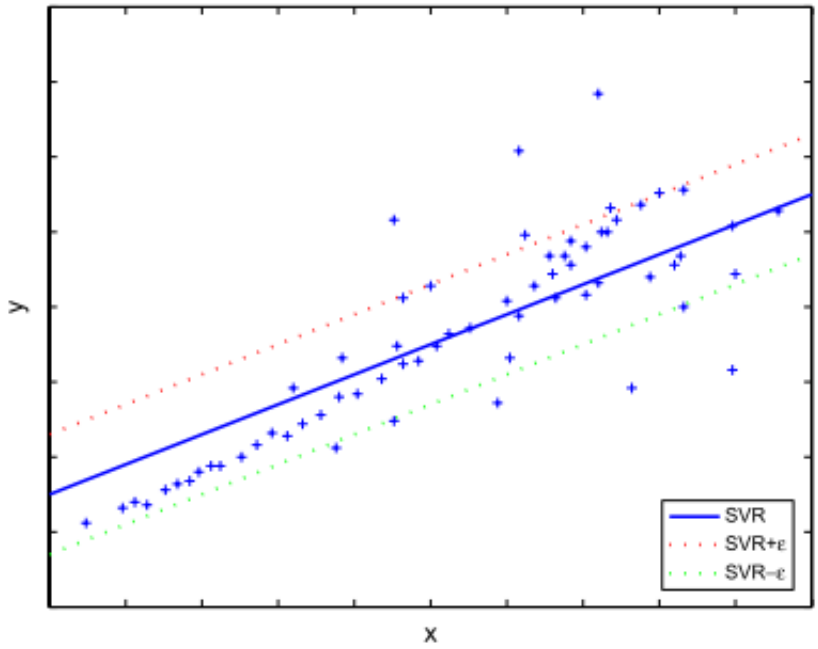


Figure 2.2: SVR margin of error [25]

The utilisation of different kernels can lead to different regression estimates within SVR. Three of the most common kernel types are linear, polynomial and a Gaussian Radial

Based Function (RBF) that are represented by the following equations respectively [26]:

$$K(x,y) = a_1 x^* y + a_2 \tag{2.8}$$

$$K(x,y) = (a_1 x^* y + a_2)^h \tag{2.9}$$

$$K(x,y) = exp(\frac{1}{\delta^2}(x-y)^2) \tag{2.10}$$

where x and y represent feature vectors, a_1 and a_2 are constants, $*$ represents the dot product, h is the polynomial degree and δ represents the bandwidth of the RBF kernel. These equations are used to map points to a feature space.

To approximate the regression, training data is represented in the following form [26]:

$$G = \{(x_t, d_t)\}_{t=1}^{n} \tag{2.11}$$

The input x_t at time t relates to an output d_t. A regression function $f(x_t)$ needs to achieve an output close enough to d_t given input x_t. In cases of non-linearity there is a nonlinear transformation that maps the input space into a different dimension, briefly discussed above. This is represented in the following way [26]:

$$f(x_t) = \omega\varphi(x_t) + b \quad \varphi : R^n, F, \omega \in F, \tag{2.12}$$

where ω represents a weighted vector and b is constant which offsets the function. Further detail on the mathematical derivation presented above was discussed by Oscar Calveria, Enric Monte and Salvador Torra in their research paper on SVR in forecasting [26].

2.5.2 Software implementation of SVR

As described above, the three most common kernels within SVR are the linear kernel, polynomial and RBF kernel. In order to transform the input data into the required dimension, the kernels use a number of specified parameters that aid in carrying out the learning process. These parameters are defined in the Python environment using various libraries and packages (these libraries and packages hold various functions and algorithms that can be applied in the programming environment). This specific library, known as Scikit-learn, assists with the modelling of an SVR model. It defines the following parameters as follows [27]:

- Degree: this refers to the degree of the polynomial kernel function (represented by h in Equation 2.9), this parameter is not utilised by the other kernels.

- Gamma: this represents a coefficient for the chosen kernel, it determines the extent of the effect of outlying points on the model [28]. Gamma is represented in Equation 2.8 to 2.10 by the terms a_1 and $\frac{1}{\delta^2}$ respectively.

- C: this represents a regularisation parameter, the strength of the regulation is inversely proportional to C. This term indicates to the optimisation model how much error can be tolerated, therefore stipulating the balance between over fitting and under fitting a curve [28].

- Epsilon: this specifies the epsilon tube distance discussed previously in this section.

2.6 Neural networks in forecasting

2.6.1 Overview

Machine learning models were introduced as forecasting techniques in FMCG due to the insufficient predictive capability of linear models that were historically used. The methods used in machine learning are able to present a realistic predictive model that can provide a more cognitive representation of forecasting [13].

Artificial Neural Networks (ANNs) are a subset of AI and can be defined as a broad terminology that encompasses technology that allows computers to mimic the human cognitive function [29]. ANNs form the basis of deep learning which incorporate multiple neural network layers [29]. Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) are types of neural networks. The theory and implementation of these methods are described in detail within this section.

2.6.2 Theoretical background

2.6.2.1 Overview

Figure 2.3 below illustrates an overview of the stages found in a neural network learning process in machine learning [30]. Labelled data is defined as the historical data of the metric being predicted, for example: number of sales, cost of product etc. Labelled data is split into three categories, one for training the algorithm, one for validating the data in order to tune parameters, and lastly data to act as a test sample and evaluate the performance of the algorithm. Similar to labelled data, feature data is input data provided by the user. But unlike labelled data, feature data acts as a guide for the algorithm by providing a level of correlation or pattern to the labelled data.

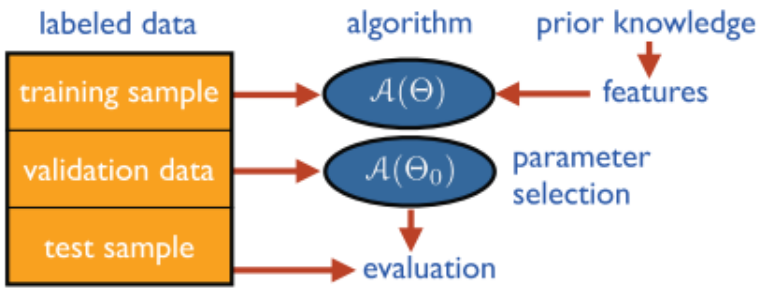


Figure 2.3: Learning stages in neural networks [30]

Hyper parameters defined as Θ in Figure 2.3 are not determined by the algorithm but rather act as inputs into the learning algorithm [30]. Hyper parameters are tuned through testing various values, the success of this tuning is measured on the validation data. The most successful hyper parameter values result in an algorithm with the most successful

hypothesis. This hypothesis proceeds into the testing phase. Depending on the type of training data available different learning scenarios are possible within machine learning [30]:

- Supervised learning: labelled data is available and used for training.
- Unsupervised learning: only unlabelled data is available, meaning testing of the accuracy of the algorithm is difficult.
- Semi-supervised learning: labelled and unlabelled data is available for training.
- Transductive inference: Similar to semi-supervised learning except the objective is to predict the points of the unlabelled test data only.
- On-line learning: dynamically training and testing continuously.
- Reinforcement learning: the learning algorithm interacts with the environment in order to predict outcomes.
- Active learning: the learning algorithm adaptively collects training samples, which is useful in cases where labels are expensive to obtain.

2.6.2.2 Artificial Neural Networks (ANNs)

ANNs consist of numerous non-linear computing elements that form part of a hidden layer network in-between input and output layers [15]. Because of its structure, ANNs are known to handle outliers and non-linear data patterns well.

A study based on developing a sales forecasting system [15] customised an ANN to the specific problem to produce a successful model. This indicated how crucial customisation is to ensure success in accurate forecasting. Customising an ANN involves configuration of the architecture, such as the number of layers, the number of nodes in each layer and the subsequent connections among nodes to bring the network together, as seen in Figure 2.4 below [10]. Ensuring the model's applicability to the problem is therefore strongly

dependant on the selection and development of the correct architecture.

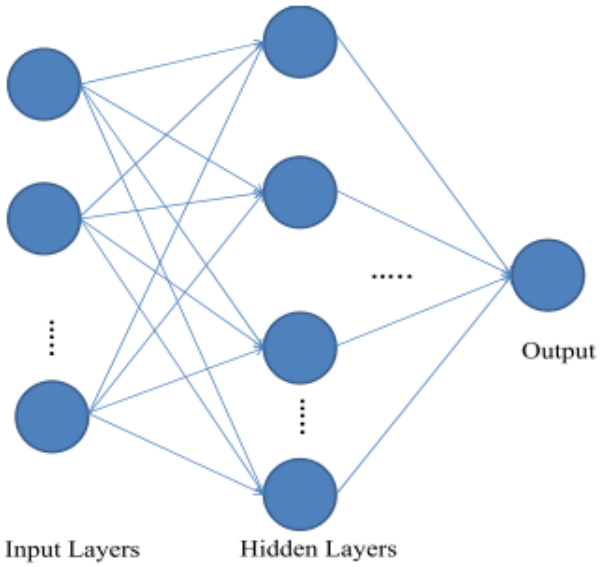


Figure 2.4: The architecture of a neural network [10]

2.6.2.3 Deep Neural Networks (DNNs)

A DNN utilises deep learning by using more than one hidden layer [8]. The concept of a hidden layer is illustrated in Figure 2.4. By this definition ANNs are essentially the building blocks to deep learning and DNNs.

The inner workings of a DNN are similar to that of an ANN. The computational flow starts with data inputs being weighted and passed through the nodes in the network. Each node has an activation function attached to it [31]. The activation function acts as a trigger which either allows the node to send its output to the next node or not (mimicking the brain's neural network to an extent). In recent years a commonly used activation function is the sigmoid s-shaped function (explained in Section 2.6.3). The data moves through the network via the nodes in one direction (feed-forward) until the output is reached. This output is then passed through a loss function. The purpose of the loss function is to compute the difference between the output and the historical training data. The neural network is programmed to minimise this difference or "loss" through the process of back propagation, which entails the models' weights readjusting iteratively through the use of an optimisation function (discussed in Section 2.6.3) [31]. Programming tools such as

TensorFlow and its machine learning libraries allow a large part of the computing to be done in the background.

A study conducted by Lawrence Snyder [16] provided a detailed example into the customisation of a DNN with parameters from The Newsvendor Problem. The concept of the Newsvendor Problem is based on determining the optimum amount of newspaper a newsvendor should buy each morning to optimise profit, taking into account the underage and overage costs. These costs are associated with potentially overstocking and not selling all newspapers, versus understocking and the subsequent loss of sales [32]. The mathematical equation used to solve the Newsvendor Problem, known as the Newsvendor Model, is based on retrofitting past sales records to a cumulative distribution function. The weaknesses of such methods has been highlighted in Chapter 1, as it was the chosen method for a previous study conducted on the coffee franchise. In summary this method cannot handle the volatility that data may present.

Lawrence Snyder's study used the relevant mathematical equations from the Newsvendor Model as inputs into the loss function utilised in DNN. The loss function used was both a Newsvendor loss function and a revised Euclidian loss function [16]. This was not only able to predict the demand but also minimise the costs associated, thus taking the network a step further. This paper compared the results to other attempts at solving a multi-feature Newsvendor problem (MFNV) i.e. the Newsvendor Problem where demand is dependent on a multitude of variables, known as features. These features could be things such as current weather conditions, the day of the week, time of the year etc. The comparison of the results showed that DNNs outperformed other methods in their ability to deal with volatile and sparse historical data. The ability to incorporate the concepts of the Newsvendor Problem into the algorithm showed that DNNs do not replace other previously successful inventory management solutions, but rather improved on them.

2.6.2.4 Dense Layers

Dense layers are fully connected layers within a network, each neuron within a layer receives input from all the neurons present in the layer before [33]. Figure 2.5 below illustrates this using the TensorFlow Playground which allows one to visualise hypothetically structured networks [34].

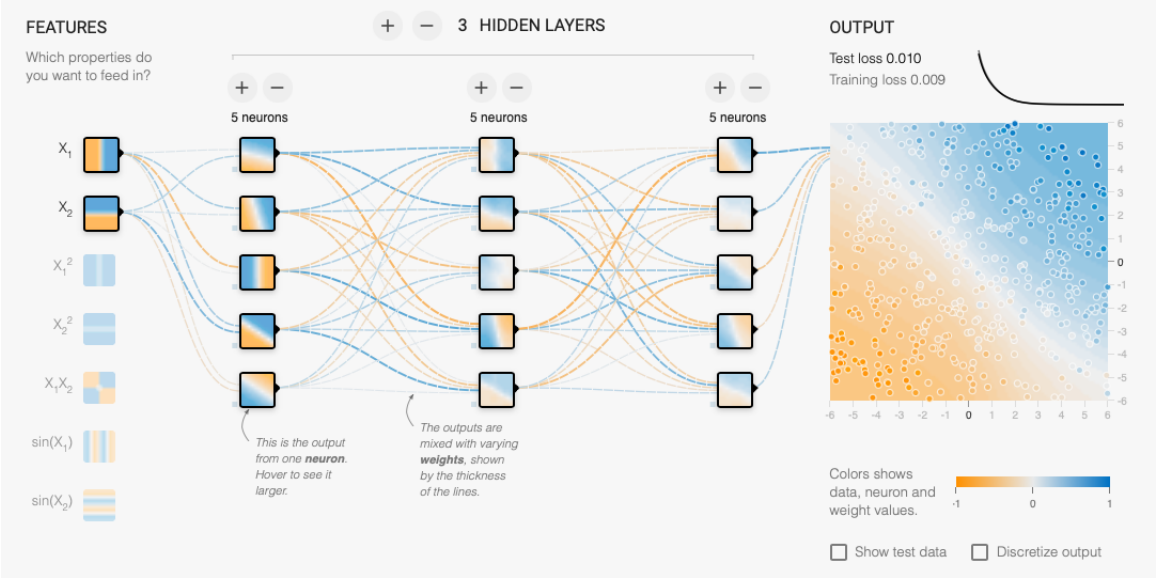


Figure 2.5: Dense Neural Network Representation on TensorFlow Playground [34]

The use of an activation function is what differentiates a dense layer from a linear network. If a linear activation function was utilised in a dense network the results achieved would be the same as a linear network. By using non-linear activation layers, dense layers stacked together create a level of complexity which classifies them as deep learning [35].

Dense layers are considered to be an elementary type of neural network due to the inability to process repetition in time. This means for a single input the output will always be the same ($f(2) = 9$ will always be the same despite whether $f(2)$ appears as the first input, a middle input, or as the last input) [35].

2.6.2.5 Convolutional Neural Networks (CNNs)

A Convolutional Neural Network (CNN) uses smoothing parameters as part of its data processing [36]. The network works similarly to a DNN described in the previous sec-

tion, but with a few differences. Figure 2.6 below shows a simplified view of the CNN architecture [36]. The input data is passed through both a convolution and pooling layer, the main purpose of these layers is to smooth the input data. The convolution layer performs a similar function as a weighted MA, it assigns specific weights to all inputs and sums them before output. The pooling layer groups vectors into set sizes and then assigns a generalised summary statistic for each group. Some common pooling types are max pooling, average or mean pooling [36]. Thereafter the output from convolution and pooling layers goes through the Rectified Linear Unit (ReLu) non-linear transformation. Once non-linearity has been applied the output layer applies an activation function to the output data.

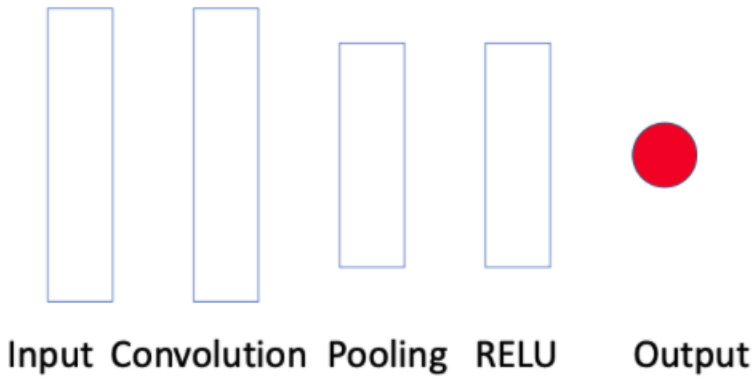


Figure 2.6: A simplified CNN architecture [36]

CNNs are most commonly applied in computer vision, where inputs are matrices of pixel values. The network is able to learn representations of large inputs better than other networks [37] .

2.6.2.6 Recurrent Neural Networks (RNNs)

An RNN differs from a feed-forward neural network in that it uses feedback loops with a time variable, in this way the neurons form a cycle. The recurrent in RNN refers to the trait that neurons only fire for a specific period before they get temporarily deactivated. These neurons can get reactivated at a later point in time. In this way the inputs from much earlier are still able to be accounted for [23].

A weakness of the RNN is the vanishing gradient problem. This problem arises when the partial gradients used in the network approach zero. The gradient determines the level of training that can occur, hence when it approaches close to zero, very little to no training can happen [38]. A solution to the problem was developed and named Long Short-Term Memory (LSTM) which is a branch of RNN. An LSTM network is able to handle time series data with long time steps. The neuron in an RNN is replaced with three sigmoid layers in an LSTM that allows for a "forget function".

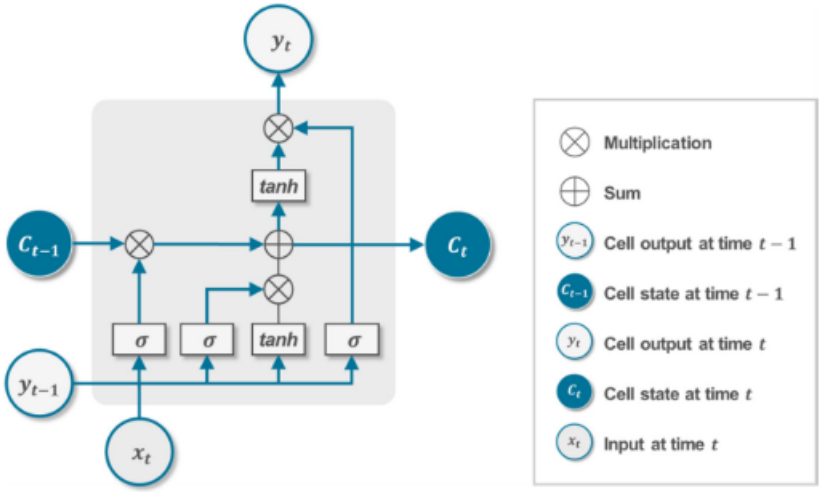


Figure 2.7: An LSTM memory cell structure [14]

Figure 2.7 illustrates the LSTM memory cell [14]. The structure of the cell enables the extraction and memory of information for a long period of time. This makes the LSTM method a popular choice in deep learning in time series forecasting [14].

2.6.3 Software implementation of neural networks

2.6.3.1 Multi step and single step neural networks

A single step model predicts one value into the future by defining a set input, known as a data window. This is done without the use of feedback [39]. This is the most simplistic structure of a neural network model. Figure 2.8 below illustrates the data window input for a single step model [40]. This comprises of three parts: an input, offset and output (equal to one for a single step model). The offset can also be accepted as zero if there is no delay in the time step predicted versus the time steps inputted. Another aspect of

data windowing is the definition of features (independent variables) and labels (dependant variables) in the data set.

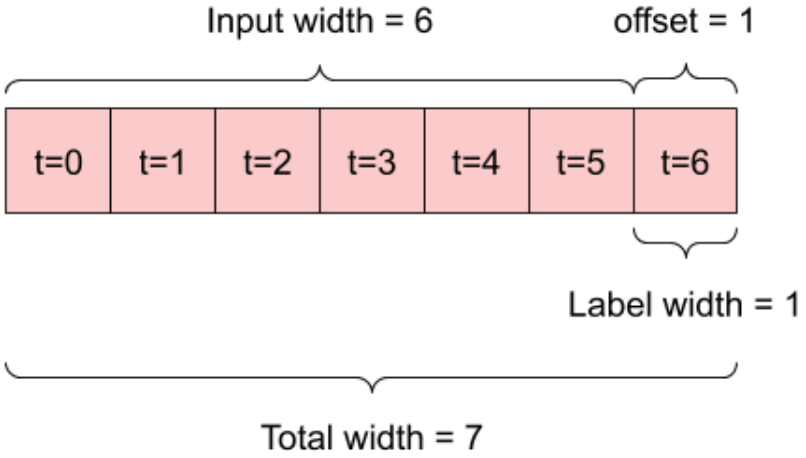


Figure 2.8: Data window for a single step model [40]

The multi step model is able to predict more than one time step into the future given a set window of data. There are two types of multi step models. The first behaves similarly to the single step model while the other is auto-regressive in nature i.e. it incorporates feedback from previous predictions into the next prediction [39]. These two different methods are referred to as single shot and auto-regressive respectively [40]. Figure 2.9 below shows an auto-regressive data window. This illustrates the feedback loop the model incorporates as it makes a series of predictions [40].

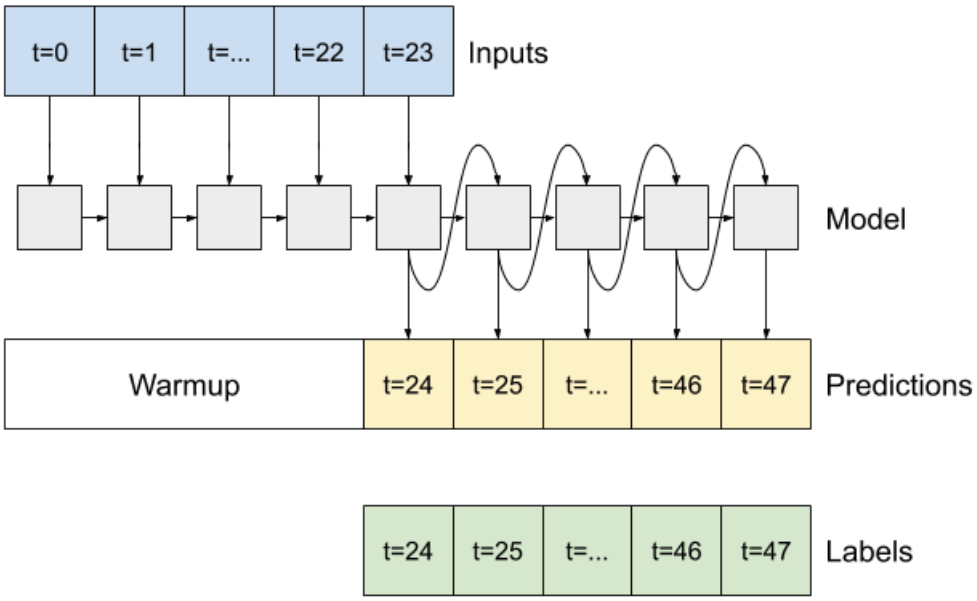


Figure 2.9: Data window for a multi step auto-regressive model [40]

2.6.3.2 Data normalisation and standardisation in neural networks

A key aspect of neural networks is the form of input data entering the model. Data normalisation is the task of scaling the data, usually performed before the data is inputted into a neural network. If data is not pre-processed in this manner, an unstable and unsuccessful model usually results [41]. This is because of the weights and error metrics calculated between predicted and actual values. When working with unscaled data that has a large range between the minimum an maximum values the computational requirements become difficult.

The scaling of data is achieved in two ways, either through normalising or standardising the real value inputs. Data normalising is achieved by rescaling the data to fit between the range of zero and one. Equation 2.13 illustrates the formula utilised to perform this scaling:

$$x_{normalised} = \frac{x - min}{max - min} \tag{2.13}$$

where x is the real value input, max and min represent the largest and smallest values within the data range respectively, and $x_{normalised}$ is the scaled input ready to be entered

into the model [21].

Data standardisation differs from data normalisation in that it re-scales the distribution of values such that the mean of the observed value is zero, and the standard deviation is one. This method works the best when the data fits a Gaussian distribution, indicating that the data is already well centred [21]. Equation 2.14 illustrates the formula utilised to perform this scaling:

$$x_{normalised} = \frac{x - mean}{\sigma} \tag{2.14}$$

where the mean is calculated as the average value of the data and σ represents the standard deviation, which is calculated as follows:

$$\sigma = \sqrt{\frac{\sum(x - mean)^2}{count(x)}} \tag{2.15}$$

Both methods of scaling data have to be tested on individual data sets to determine which performs the best [21].

2.6.3.3 Optimisers in neural networks

In DNNs in order to assess the performance of a model (how close a predicted outcome was to the expected) a loss function is utilised during training and validation. The objective of the model is to minimise the loss function through the use of an optimiser [42]. Optimisers use a specific algorithm to change the weights and learning rate of the model (among other attributes) in order to minimise the loss.

In time series analysis, some commonly used optimisers are Stochastic Gradient Descent (SGD), Root Mean Squared Propagation (RMSprop) and Adaptive Moment Estimation (Adam).

SGD is based on the gradient descent method of optimisation covered in detail by Goodfellow et al. [8]. Given that a gradient indicates the direction of increase, the method uses the negative gradient direction to minimise the loss [43]. It is a commonly used

optimisation function that has been faulted only on its slowness to produce results.

The RMSprop algorithm adapts its learning to reach convergence. This algorithm is a derivative of the Adaptive Gradient Algorithm, both algorithms are based on the squaring of gradients [8]. RMSprop differs from the Adaptive Gradient Algorithm in that it utilises a MA of the squared gradient [43]. This algorithm is widely used in DNN.

The Adam optimiser makes use of an adaptive learning rate which makes it computationally efficient. "Adam" stems from the term adaptive moments. It is similar to the algorithm of RMSprop combined with momentum. It uses the exponential MA of the gradients to scale the learning rate [43]. Further detail into the mathematical properties of the algorithm is covered by Goodfellow et al. [8].

2.6.3.4 Activation functions in neural networks

The activation function in a neural network is attached to each neuron within the network to determine whether the neuron is fired or not. This indicates that the activation function determines the output of the deep learning model, thereby acting as a key differentiator between linear regression and neural networks [44].

Figure 2.10 below illustrates two common activation functions, sigmoid and hyperbolic tangent (tanh). The difference as illustrated is that sigmoid runs from zero to one and tanh runs from negative one to one [44].

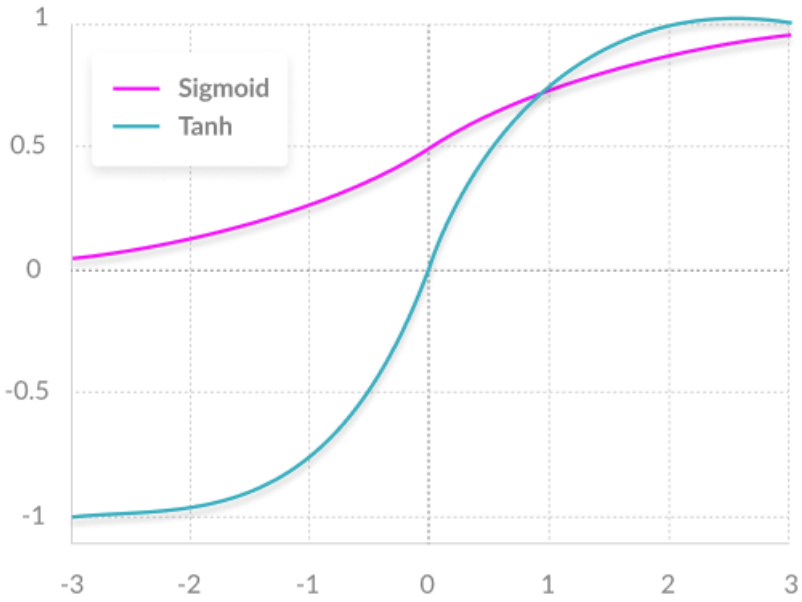


Figure 2.10: Sigmoid and tanh activation functions [44]

The sigmoid function is represented by the following equation:

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.16}$$

and the tanh function is represented by the equation below:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.17}$$

The magnitude of neurons that can be found in a network have made it necessary for activation functions to be computationally efficient. Therefore, in recent years non linear activation functions have been introduced as they have proven to be more efficient than linear activation functions. A common type of non linear activation is known as the ReLU function. Non linear functions are able to handle back propagation through the use of their derivative functions. They also perform well in the stacking of layers in more complex neural networks. Figure 2.11 below illustrates the ReLU function, while it may look linear, it behaves as a nonlinear function (it gives an output if x is positive and zero

otherwise) [45].

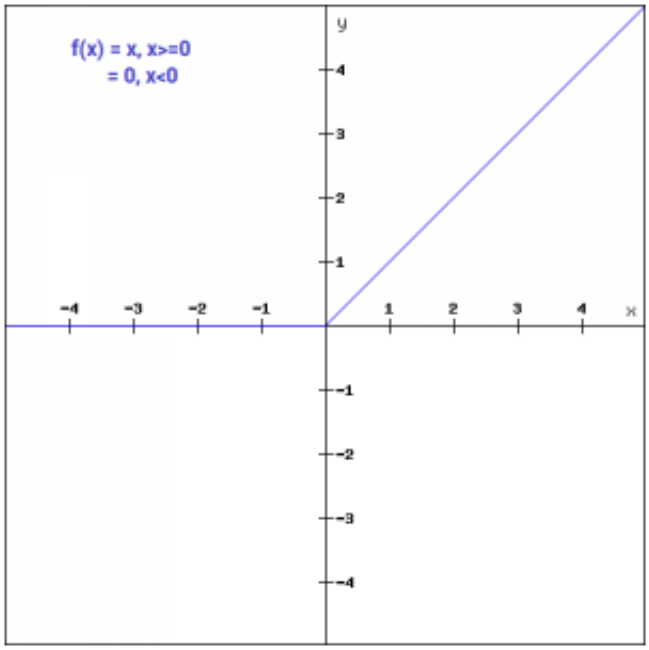


Figure 2.11: ReLU activation function [45]

2.7 Comparison between traditional and machine learning techniques

Machine learning algorithms can be used to predict outcomes based on previous events. The difference in these algorithms to other statistical analyses is that the sample complexity i.e. the amount of learning data provided significantly affects the potential accuracy of its output [30].

The drawbacks of traditional statistical models is the underlying linear method that supports the model. The linear methodology struggles to handle volatile data points that fluctuate drastically.

A case study was conducted on both machine learning and statistical analysis tools applied to the prediction of customer demand in a restaurant. The study utilised POS data and other external factors such as the weather to predict the future number of customers

[12]. The comparison assessed three machine learning regression techniques; Bayesian Linear Regression, Boosted Decision Tree and Decision Forest Regression and the Stepwise Statistical Method. All four technologies yielded accuracy's greater than 85% compared to actual outcomes. There was also no large difference between the results of the regression techniques and statistical method. This could have be due to the simplicity of the predicted metric. The study illustrated that as the complexity and dependants of the output increase so does the gap between the results of machine learning and traditional techniques.

This deduction is further shown by a study conducted in 2012 that compared two machine learning techniques with a traditional forecasting technique [10]. The application of these methodologies was tested in the field of supply chain forecasting. The methods utilised were ANN, SVM and ARIMA. The study aimed to predict the demand for various products over time using MAPE, a statistical tool to calculate the accuracy of the forecasting. The outcomes found that the machine learning techniques, specifically SVM performed the best.

In this study, the auto-correlation (ACF) structure of the products being tested varied, but had no visible effect on the results. This signified that similar trends over a time period did not affect the performance of the machine learning tools. This differed to the ARIMA model that did not perform as well, the model is known to be successful in situations with higher auto-correlation in data points [12].

2.8 Methods to measure the performance of models

In order to evaluate the models in this paper a unified performance test was needed to ensure the most successful method was accurately represented. For times series prediction MAE, Mean Squared Error (MSE) and MAPE are some of the most popular and recommended techniques across research papers referenced in this literature review. Both MAE and MSE are scale dependant errors, therefore they are only comparable in series with the same units, while MAPE is unit free [11]. The three measures can be explained as

follows.

2.8.1 Mean Absolute Error (MAE)

The MAE measure takes the mean of the error between predicted and actual test data as seen in the equation below [46]:

$$MAE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n} \tag{2.18}$$

where y_i and $\hat{y}_i$ represent the actual value and the predicted value respectively and n represents the sample size.

2.8.2 Mean Squared Error (MSE) and Root Mean Squared Error (RMSE)

The MSE and RMSE metrics are linked together as they both stem from the same base calculation provided below [46]:

$$MSE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n} \tag{2.19}$$

The RMSE is a more popular version of the MSE as it takes the square root of equation 2.19. While the RMSE may be more difficult to interpret it does allow the output to be in the same unit scale as the data. This proves to be an advantage when reporting the metric [11].

2.8.3 Mean Absolute Percentage Error (MAPE)

The MAPE is calculated using the following formula [46]:

$$MAPE = \frac{100\%}{n} \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{y_i} \tag{2.20}$$

A shortfall of MAPE is that even though it is unit free it places a large penalty on negative errors, therefore it will favour a method that forecasts lower than the actual outcome [46].

Chapter 3

Research Method

3.1 Methodology overview

To investigate whether a model could successfully forecast demand for the coffee franchise, various machine learning and statistical models were developed as part of the research method. A main objective was to determine whether a forecasting model would be able to reduce wastage and improve profitability through accurate predictions of demand for perishable food.

The methodology overview was as follows:

- An analysis on available branch and product data was carried out. This determined the most suitable branch and product/s to use in the investigation.
- The specific branch and perishable product formed the input data used to evaluate the following benchmark models:
  - ARIMA
  - SVR
- The same input data was used to evaluate the following neural networks:

  - Linear network (used as a baseline neural network)
  - Dense layers
  - CNN
  - RNN (LSTM)

- The results were compared via statistical measures (MSE, RMSE, MAE).
- The MAPE of the top performing models was calculated and compared.
- An additional product was tested on the top performing models to gauge the reliability of the models.
- The reliability and validity of the models in relation to its ability to successfully forecast demand was discussed.
- The forecasting models' ability to improve profitability and the overall business model was investigated and discussed.
- Other areas of improvement over and above demand forecasting that may impact profitability and or waste reductions were identified and discussed.

3.2 Branch and product selection method

To test the validity of the forecasting models, one product from a branch was selected to test across the models, in order to draw comparisons on the accuracy of predictions. The parameter selection and tuning of the various models was based on this product. To test the reliability of the models, a second product from a branch was tested. The same tuning parameters were utilised during the second testing. This was done to confirm whether the model's results could be replicated and carried out to all perishable products across the coffee franchise. This Section outlined the process of selecting the specific branch and products used in this investigation.

3.2.1 Overview of coffee franchise data

The POS data from the coffee franchise was stored in a SQL (a database management language) database. The database contained 42 tables of data (see Appendix A for a breakdown). The tables were connected by the variable "sumid" with which other variables such as: items, price, branch, date etc. were linked together. Access to the data was granted by the coffee franchise owner as part of the ethical clearance for this research report.

Table 3.1 below captured the main information extracted from the database. Nine out of 150 available variables were extracted from the database as they proved to be the most useful information needed to build a forecasting tool. Table 3.1a indicated the number of stores tagged in the data, of which only nine store ID's were linked to current operating branches of the coffee franchise. Of these nine stores only five had data from their POS system that could be extracted for the available years successfully, these are circled in Table 3.1a.

STRID
237
97
116
222
141
208
191
298
326
333
349
352
357
396

(a) Store ID list

Prioritised Variables	Unit
Sumid	Number
Sumdate	Date
Sumgrossales	Currency
Strid	Number
Ibritemname	Text String
Ibritemplu	Number
Ibrcaegoryname	Text String
Ibrnumsold	Quantity
Ibrpricesold	Currency

(b) Prioritised variables from POS data

Table 3.1: Coffee franchise data summary

Figure 3.1 below shows an example of the SQL query utilised to extract the main data required. This code was used to extract data from the store number (STRID) 237 for 2019. A similar code was run for the other four branches from 2016 to June 2019.

```
SELECT summary.sumid, summary.sumdate, summary.strid, itemsbreakdown.sumid,
itemsbreakdown.ibritemname, itemsbreakdown.ibritenplu,
itemsbreakdown.ibrcategoryname, itemsbreakdown.ibrnumsold,
itemsbreakdown.ibrpricesold
FROM summary
INNER JOIN itemsbreakdown
ON summary.sumid = itemsbreakdown.sumid
WHERE summary.sumdate > '2019/01/01'
AND summary.sumdate < '2019/04/01'
AND summary.strid = 237
```

Figure 3.1: Example of an SQL query

3.2.2 Analysis on coffee franchise data

Each of the fives branches data was analysed separately to determine the highest volume of perishable products sold from 2016 to 2019. The top five perishable products from each branch were then collated. This was used to determine the top five perishable products sold overall (this was done on a volume basis and not on a cost or profit margin basis). Figure 3.2 shows the top five products across branches. Muffins, breakfast and croissants were the top three products sold.

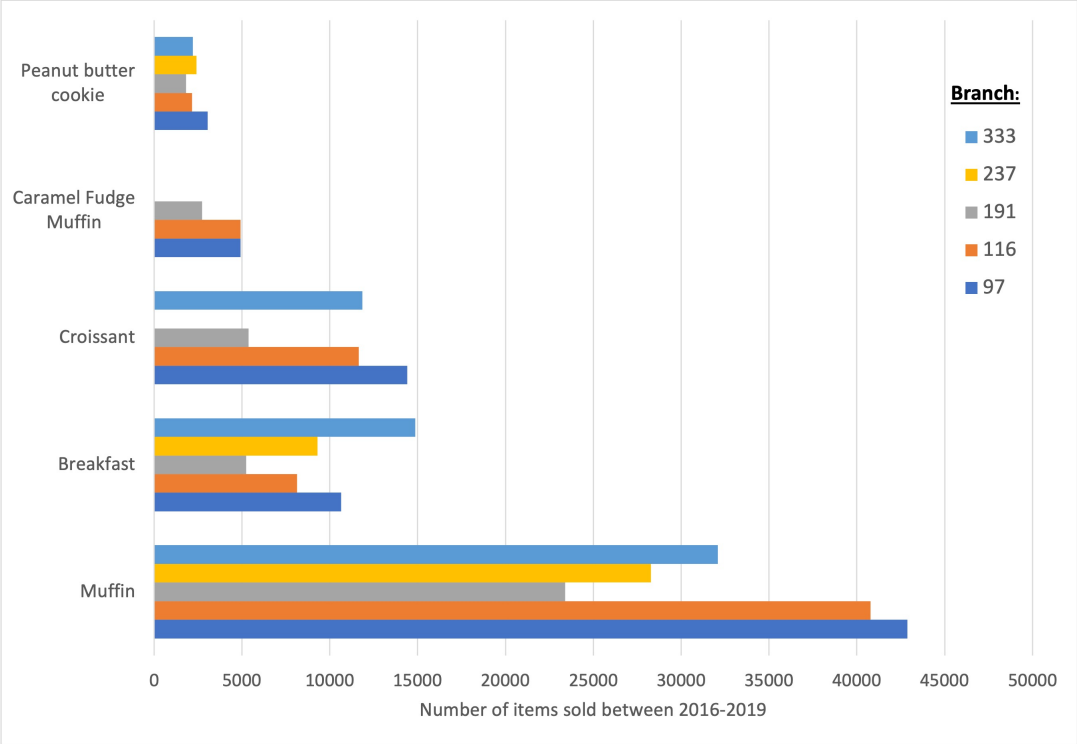


Figure 3.2: The top 5 products sold by volume between 2016 and mid 2019

The graphical plots of the top three products sold across the years were then examined to understand the quality of data available. Figure 3.3 to 3.5 show the daily volumes sold across branches per product in 2016. Similar graphs were examined for the years 2017, 2018 and 2019. The patterns seen in Figure 3.5 indicated that the croissant data was unreliable. This could be due to the various product categories that include the field "croissant" (plain croissant, chocolate croissant etc.). This may mean that the cashier randomly assigned a croissant purchase to any one of those options.

Both the muffin and breakfast data in branches 237 and 333 in Figures 3.3 and 3.4 exhibited no extreme or inexplicable outlying data points. They also displayed similar volatility patterns observed across perishable products. Therefore, they proved to be useful as indicative products for testing.

Branch 237 was the selected branch as it was open for the most days over the period from 2016 to mid 2019. Hence, it offered more time steps for the model to be trained on. Breakfast was an item that was more problematic to the coffee franchise owner, in terms of wastage from inaccurate stock in comparison to muffins. For this reason it was the chosen Product 1 going forward. Muffins were selected to be Product 2 (used to test reliability) going forward. The naming convention of these products was Product 1 and Product 2 for the remainder of this paper.

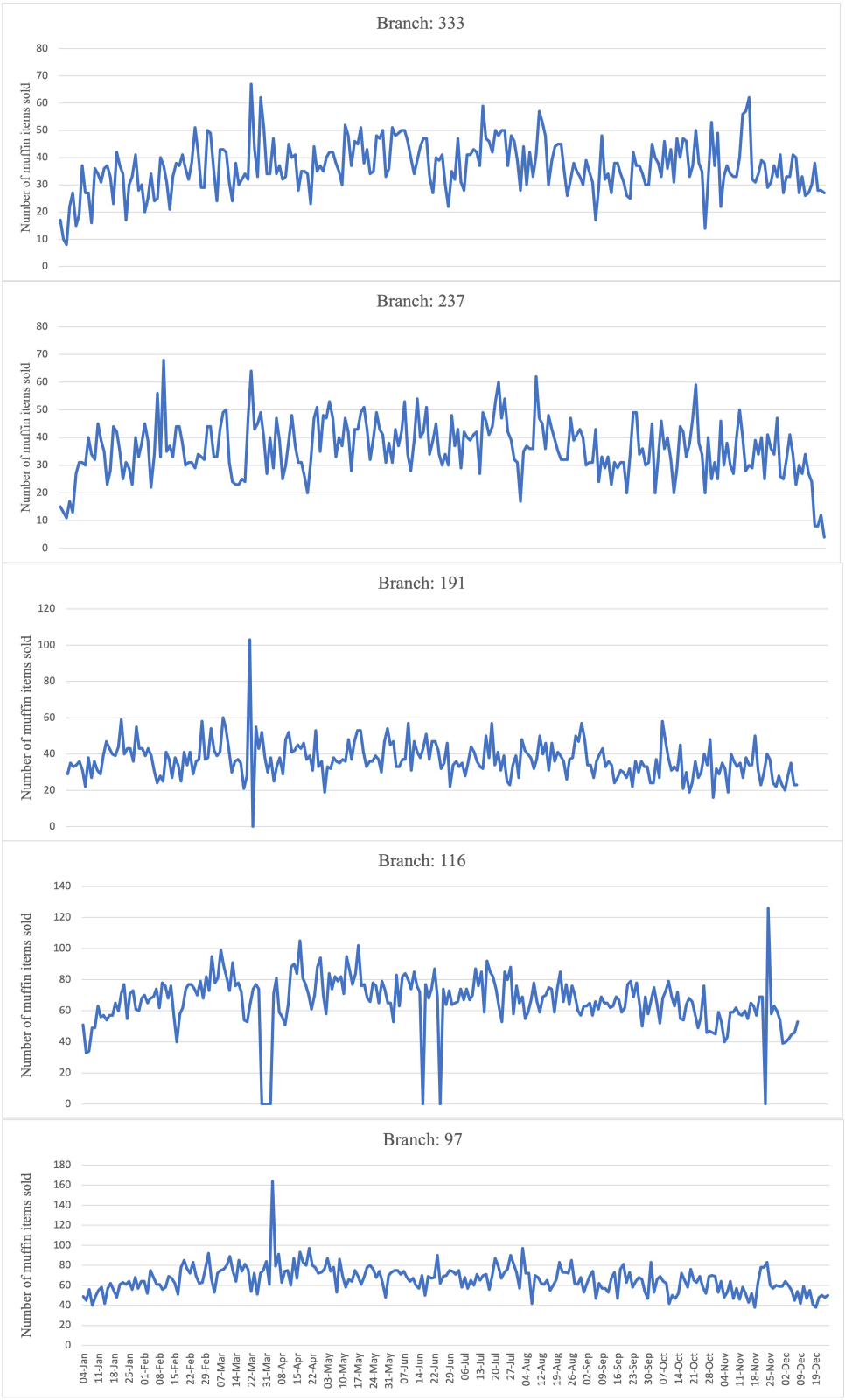


Figure 3.3: Daily number of muffins sold for 2016 across branches

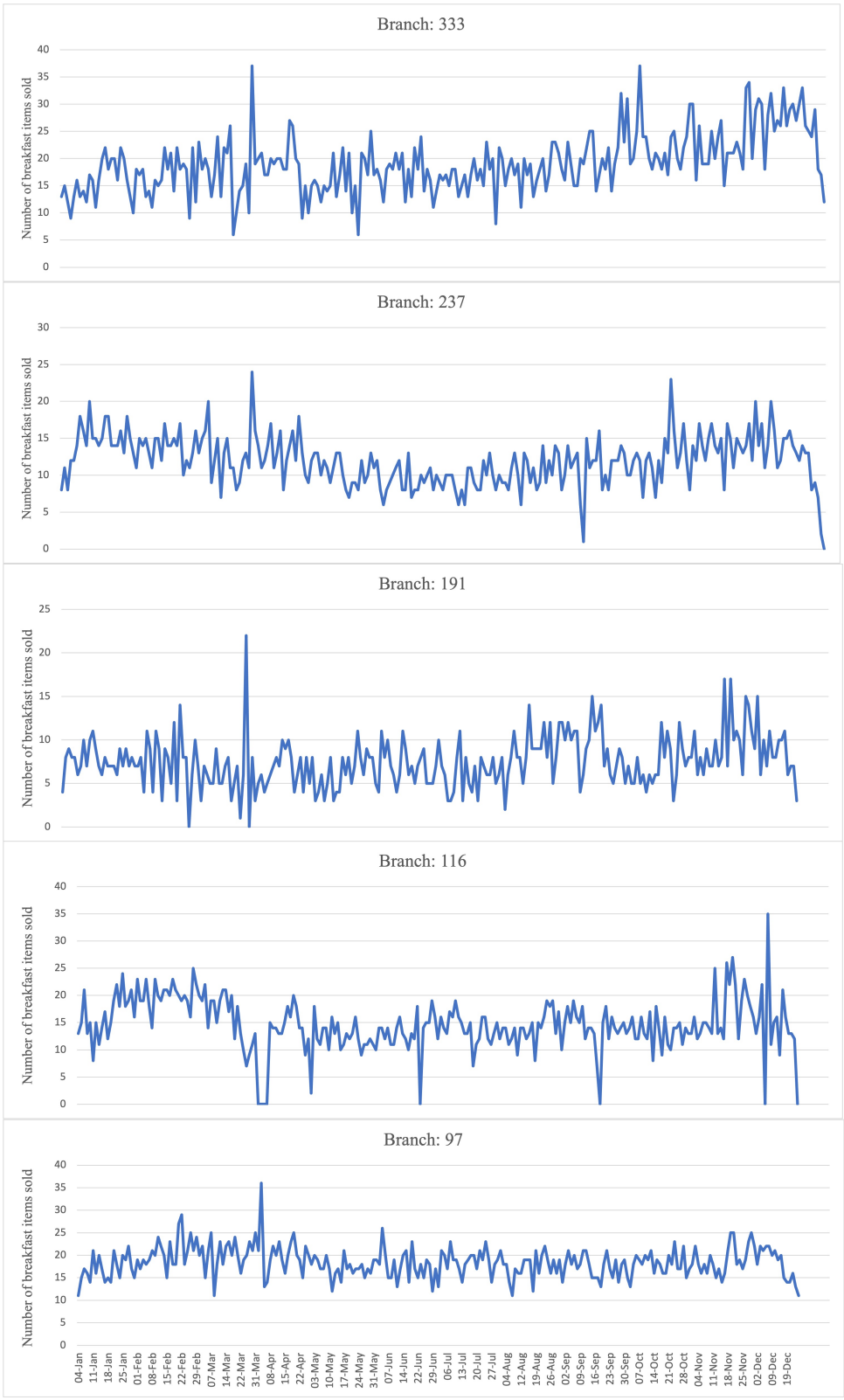


Figure 3.4: Daily number of breakfasts sold for 2016 across branches

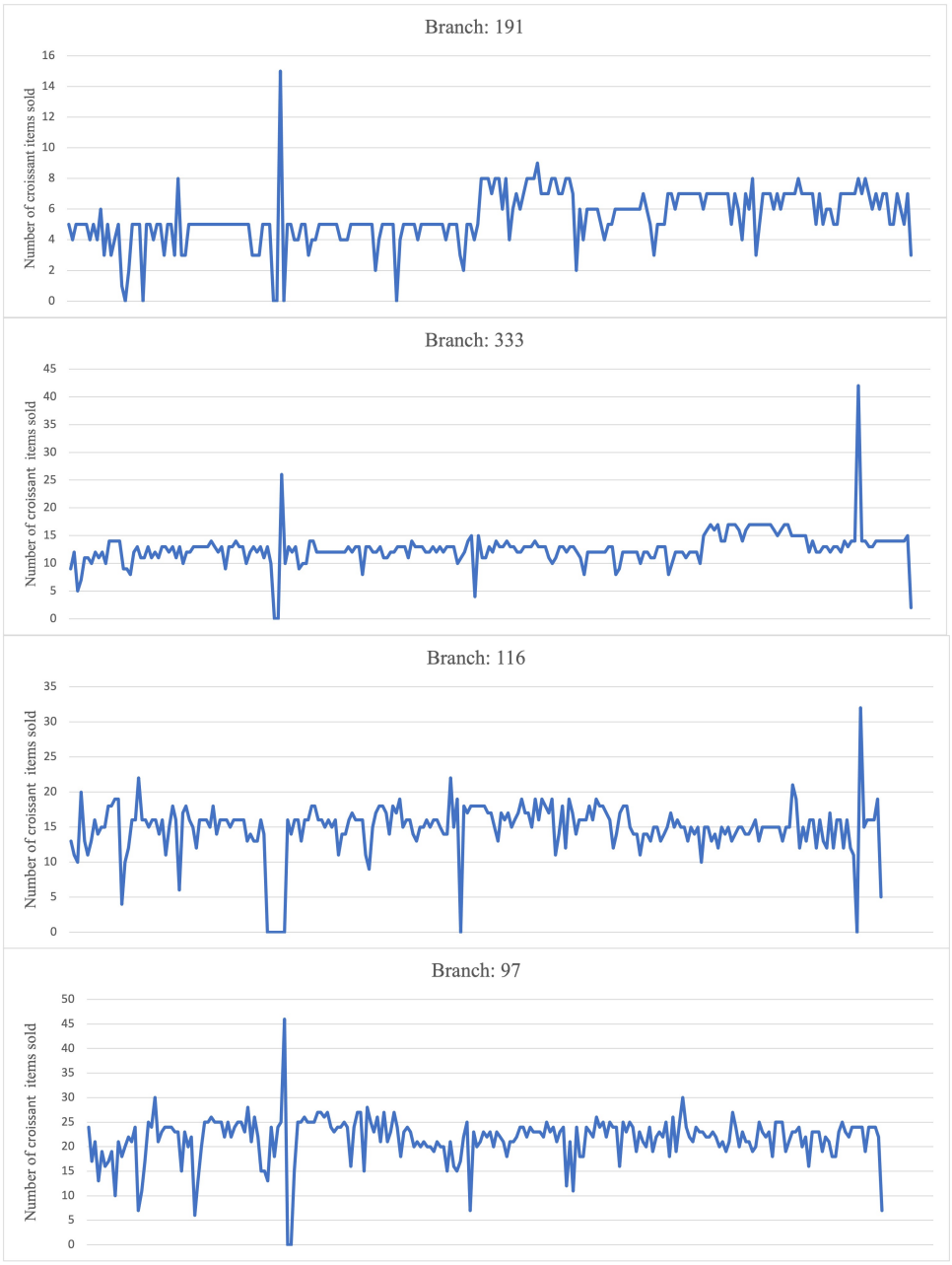


Figure 3.5: Daily number of croissants sold for 2016 across branches

3.3 Data and data preparation method

3.3.1 Weather data set

A number of external factors were considered specifically when building a neural network as the architecture allowed for multiple features to be examined in relation to one desired output.

To add external factors to the data set, Pretoria airport weather data was utilised [47]. The data included 13 columns of metrics for weather measures taken at hourly intervals throughout the day. Table 3.2 below shows a summary of the measurement variables along with the format and units. The addition of weather into the data set was based on the hypothesis that weather may influence the buying patterns of consumers, a similar study conducted on electricity demand found positive results in incorporating external environmental factors [14].

Weather Measurements	Unit
Local time in Wonderboom / Pretoria (airport)	Time
Temperature	Degree Celsius
Atmospheric pressure at station	mm of mercury
Atmospheric pressure reduced to mean sea level	mm of mercury
Relative humidity	%
Mean wind direction	m/s
Mean wind speed 10m above ground	m/s
Maximum gust value	m/s
Stand out issues	Text string
Stand out issues 2	Text string
Total Cloud Cover	Number/description
Horizontal visibility	km
Dewpoint Temp 2m from surface (C)	Degree Celsius

Table 3.2: Weather dataset summary

Pretoria's weather was used even though the location of the coffee franchise was central Johannesburg. This was due to the lack of availability of historic weather data for Johannesburg. The Pretoria weather data still allowed for a relative relationship to be built given the close proximity of the two areas.

3.3.2 Data preparation

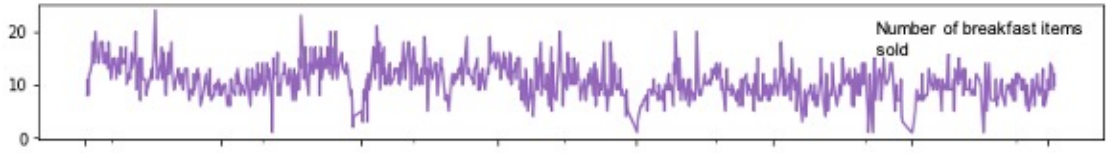
In order to prepare the data, both the coffee franchise data and the weather dataset had to be combined.

The weather dataset was prepared in the following way:

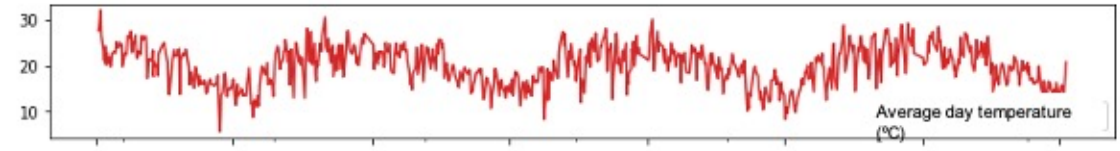
- All time intervals before 8:00 and after 17:00 were excluded based on the operational hours of the coffee shops.
- Only the following metrics were considered:
  - Date and time.
  - Temperature (Celsius).
  - Mean wind speed 10m above ground (m/s).
  - Rain (0-3).
  - Stand out issues (text describing rain conditions).
  - Cloud cover (0-3).
  - Total cloud cover (text describing cloud conditions).
- Further data cleansing was carried out:
  - Rain was ranked from 0 – 3 based on none, light, heavy or thunderstorm criteria.
  - Cloud cover was ranked from 0 – 3 based on clear, few clouds, scattered clouds or overcast criteria.
  - The average wind speed of 1- 1.5 m/s was considered not windy.
  - All metrics were reported as a daily average of the hourly measurements. This aggregation was useful as it indicated the weather pattern for the day. This turned out to be similar for the two cities, therefore making the weather data compatible for predictions in Johannesburg.

Once the weather data set was prepared, it was linked to the POS data set via the date column. In addition, a column for the day of the week was added as a feature to account for any patterns presented across Monday to Friday.

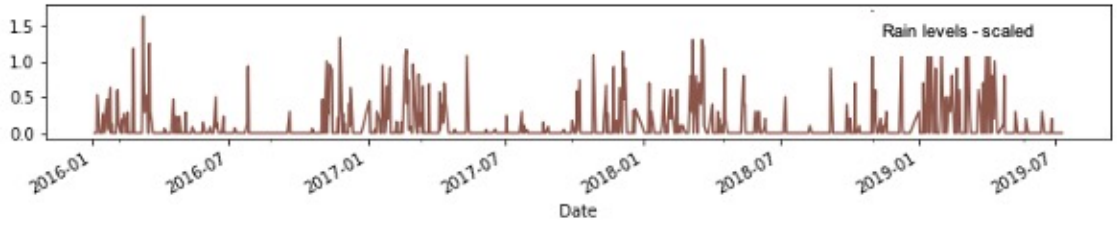
Figure 3.6 below shows graphical plots of the data for Product 1. The quantity sold was plotted against the average daily temperature in Figure 3.6b and the average rain in Figure 3.6c (scaled rain data followed the process highlighted in data preparation). Wind speed and cloud cover were not included in the initial analysis to avoid unnecessary noise to the data set before tuning took place.



(a) Number of Product 1 items sold 2016/01-2019/07



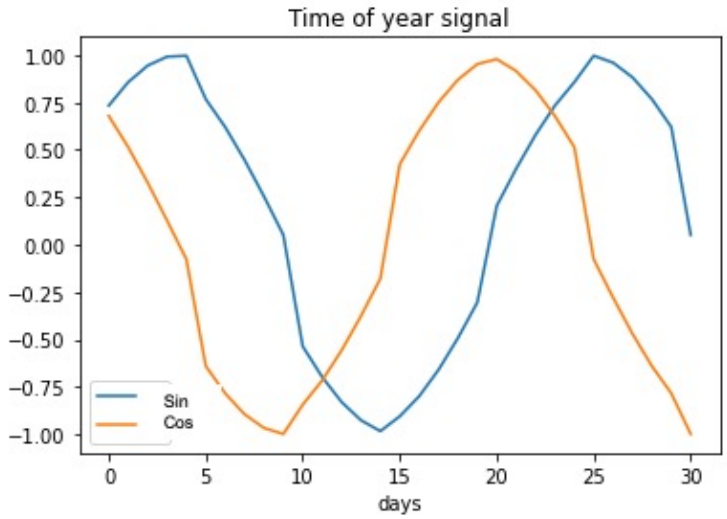
(b) Average daily temperature in Gauteng 2016/01-2019/07



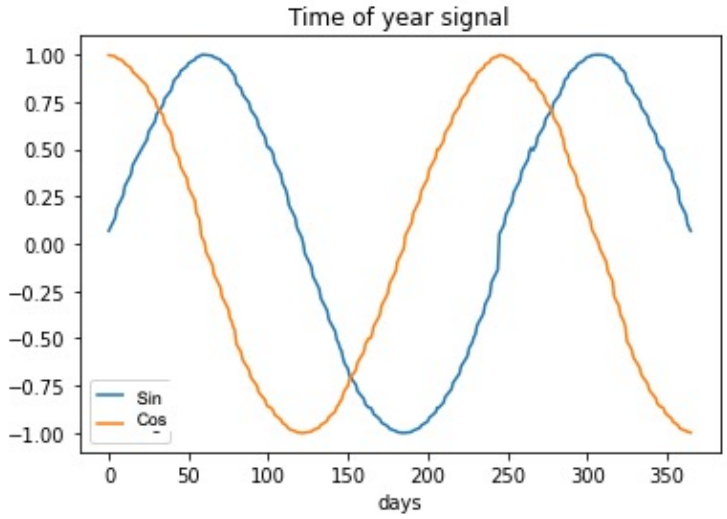
(c) Rainfall (scaled) 2016/01-2019/07

Figure 3.6: Plot of temperature and rain against number of Product 1 items sold between 2016 and mid 2019

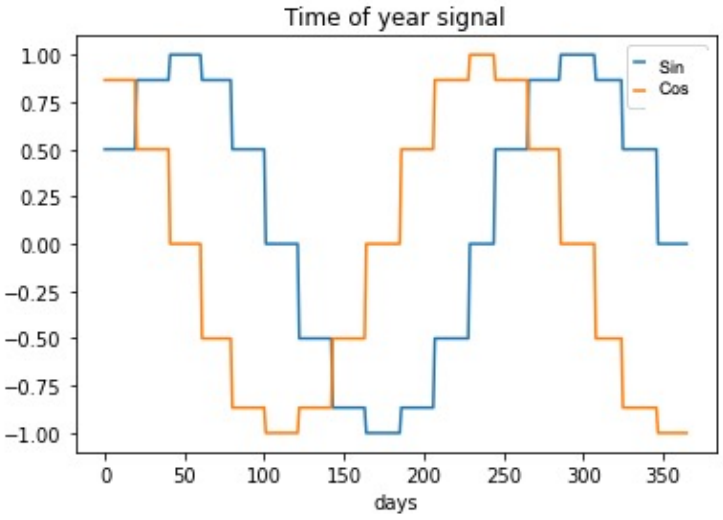
A major aspect of time series forecasting is the inclusion of periodicity in the data, especially in neural networks where patterns and relations are built between features. To introduce periodicity, adding the weekday (represented by numbers one to five) and month of the year (represented by numbers one to twelve) was not sufficient as there was a disjoint between day/month one and day/month twelve. The disjoint occurs because of the jump from twelve to one graphically, even though December is a month before January (a similar issue was noted with the days of the week). A recommended method was to add sinusoidal (sine and cosine) curves to data set [40]. Figure 3.7 below illustrates the incorporation of sine and cosine curves to represent monthly and yearly patterns.



(a) sine and cosine curves of days in a month



(b) sine and cosine curves of days in a year



(c) sine and cosine curves of months in a year

Figure 3.7: sine and cosine curves representing various periodicity

The final step in data preparation was the scaling of feature and label inputs through both data normalisation and data standardisation as discussed in Section 2.6.3. Both methods were performed to test which resulted in the most successful model. For the coffee franchise's POS data, data normalisation using a scale from zero to one proved to output better predictive results when the subsequent neural network models were run. Figure 3.8 illustrates the normalised data. It was evident that the rain measurement had a longer tail than the other data points. This was not considered an issue as there were no obvious or large differences within the data that were potential errors. A longer tail could represent extreme weather such as thunderstorms observed on specific days, an occurrence that was not unusual in Gauteng.

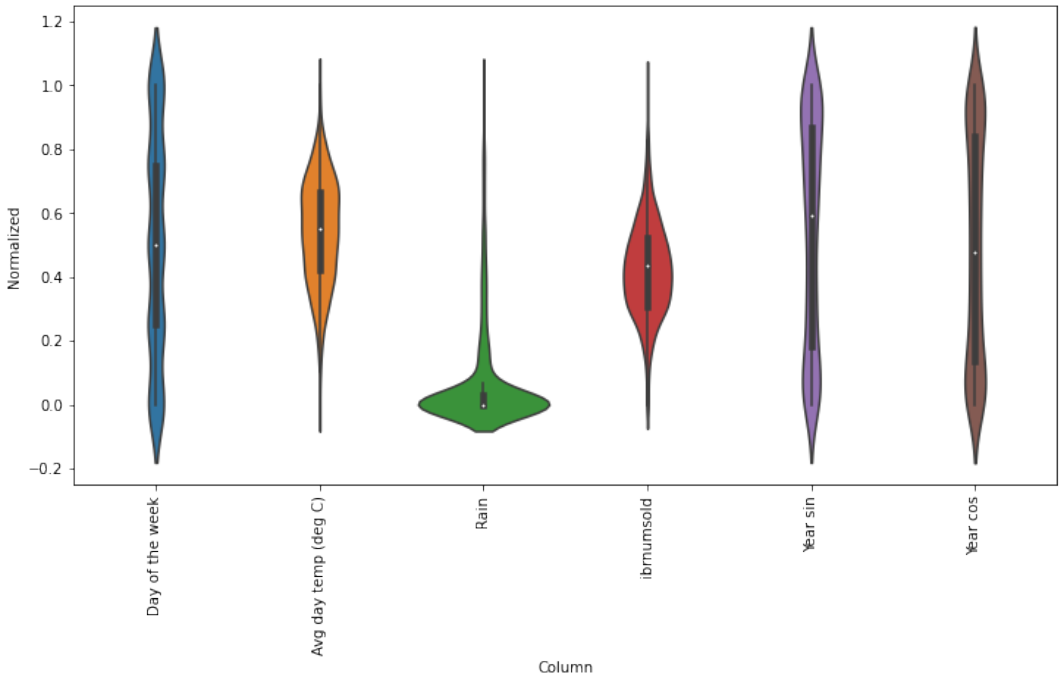


Figure 3.8: Normalised data

As seen in Figure 3.8 many of the input features discussed earlier were not accounted for. It was evident some features were adding noise to the data rather than offering a correlation to the number of items sold. The exclusion of these features from the final input data was done iteratively, through model testing and through observation of the data plots. Analysing the data plots assisted in identifying any visible correlation between the data. If it was apparent none existed it was excluded from the final input.

3.4 Implementation method of benchmark models

3.4.1 Environment Overview

The following environment settings were used to develop both the ARIMA and SVR models for the coffee franchise data set (See Appendix B and C for the code extract):

- Python notebook hosted in Google Collaboratory
- Scikit-learn package
- Statsmodel package
- Pmdarima package
- Numpy package
- pandas package
- Matplotlib.pyplot

3.4.2 ARIMA

The ACF plots seen in Figure 3.9 and Figure 3.10 were created to determine whether the data series was stationary or not. The plot showed a number of lagged versions of the series of data, which indicated whether or not the data set was correlated (this was discussed in detail in Section 2.4). Figure 3.9 shows an ACF plot of the entire data set, whereas Figure 3.10 shows an ACF plot of the data for the first 30 lags. Examining the patterns of both plots, it was clear that the data was not stationary and would require a first order level of differencing. In the $ARIMA(p,d,q)$ model, this would affect the d value, which represents the level of differencing required to transform the data to a stationary data series.

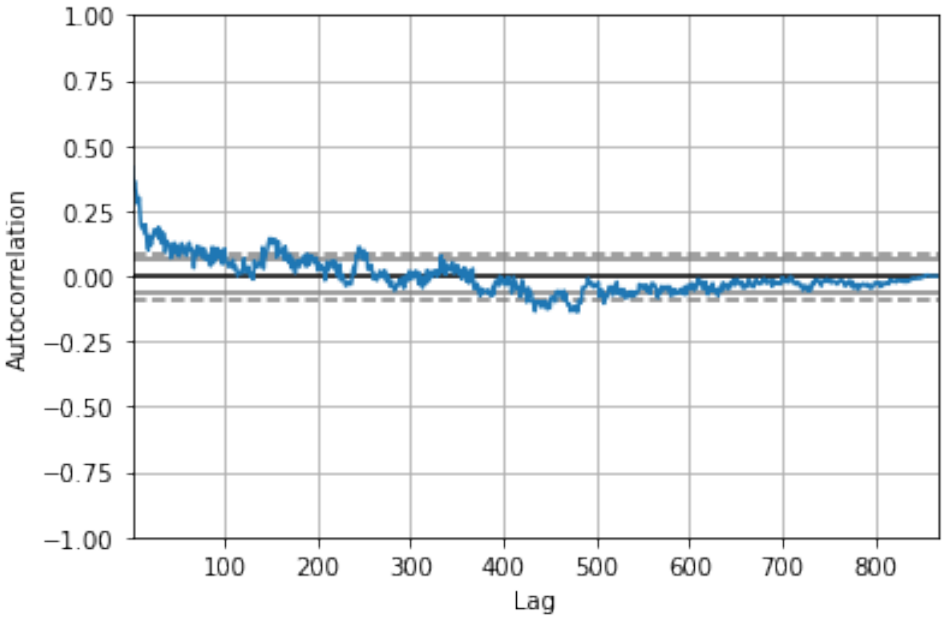


Figure 3.9: ACF plot for 800+ lags

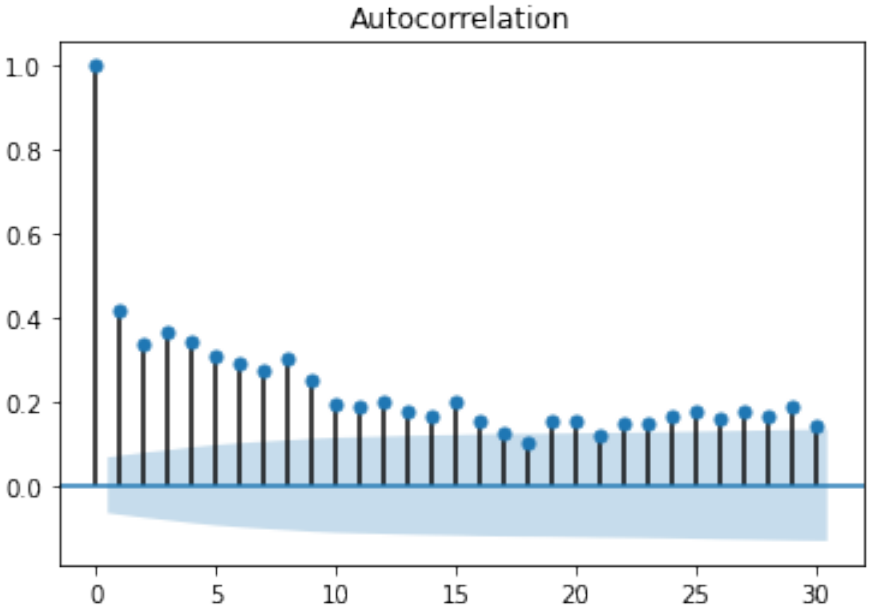


Figure 3.10: ACF plot for 30 lags

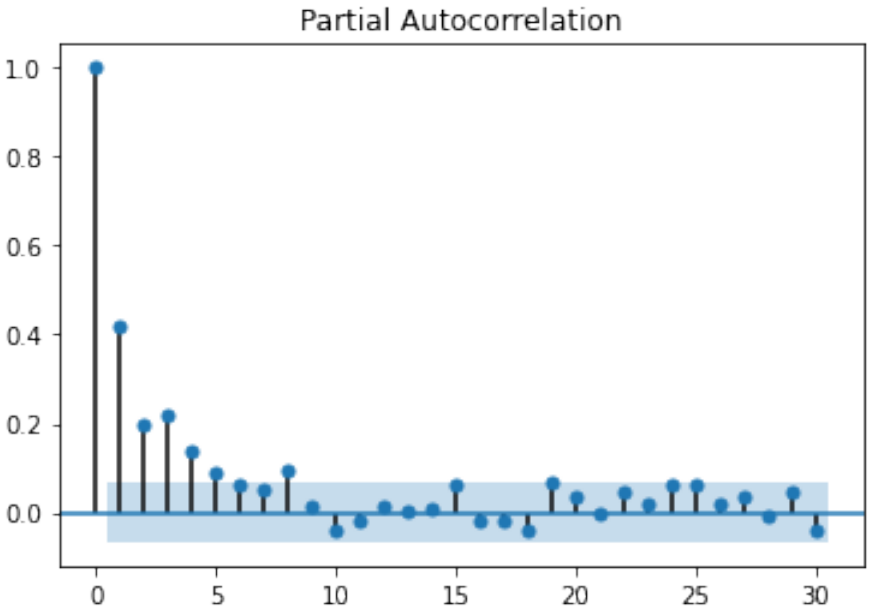


Figure 3.11: PACF plot for 30 lags

Figure 3.11 shows the PACF plot for the data set. Analysis of the data reflected in Figure 3.11 revealed a correlation between residuals. Confirmation of the first order differencing requirement for the data was indicated by the differences in the level of significance between the first and succeeding lags.

3.4.2.1 Parameter tuning

The automatic modelling of the ARIMA model described in Section 2.4.2 was carried out to define the parameters for the model using the coffee franchise data. The automatic function was run with and without editing the step wise and approximation criteria. The outputs in Table 3.3 were given by the automatic function, suggesting an ARIMA model with an order of $(4, 1, 1)$ to be the optimum given it has the lowest AIC score.

Parameters			AIC
p	d	q	
4	1	1	-1135.87
0	1	2	-1126.38

Table 3.3: Automatic ARIMA modelling results with AIC values

Figure 3.12 below shows the output after running the automated modelling function for ARIMA. The results captured a significant amount of statistical analysis performed to reach the suggested modelling parameters.

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	867
Model:	SARIMAX(4, 1, 1)	**Log Likelihood**	574.933
Date:	Sun, 07 Feb 2021	**AIC**	-1135.867
Time:	18:26:43	**BIC**	-1102.520
Sample:	0	**HQIC**	-1123.104
	- 867		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	-9.151e-05	8.29e-05	-1.104	0.270	-0.000	7.09e-05
ar.L1	0.2291	0.032	7.199	0.000	0.167	0.291
ar.L2	0.0737	0.035	2.084	0.037	0.004	0.143
ar.L3	0.1385	0.035	3.935	0.000	0.070	0.207
ar.L4	0.0962	0.033	2.874	0.004	0.031	0.162
ma.L1	-0.9824	0.009	-115.306	0.000	-0.999	-0.966
sigma2	0.0155	0.001	24.432	0.000	0.014	0.017

Ljung-Box (L1) (Q):	0.06	**Jarque-Bera (JB):**	54.63
Prob(Q):	0.81	**Prob(JB):**	0.00
Heteroskedasticity (H):	0.83	**Skew:**	0.28
Prob(H) (two-sided):	0.12	**Kurtosis:**	4.10

Figure 3.12: Output from the automatic ARIMA function

From the libraries installed in the Python environment, functions (such as $model.fit$) could be called to build an ARIMA model given the set input parameters. From these functions a density plot of the residual errors was extracted. The density plot in Figure 3.13 showed the mean centred around zero, indicating there was no bias in the prediction [20].

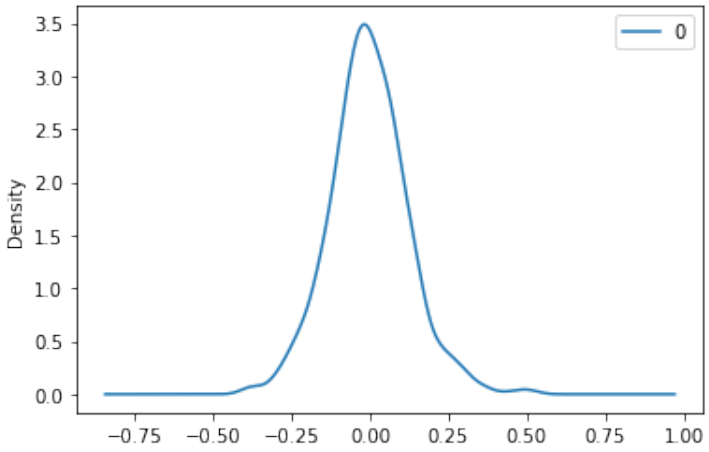


Figure 3.13: ARIMA residual density plot

Once the model fitting proved to be successful, the model was utilised to extract predictions using a portion of input data. This portion of data was allocated as testing data, it represented 33% of the total data set based on theoretical recommendations [20]. The success and performance of the model was tested via the statistical measures MAE, MSE and RMSE (these were also used to evaluate the neural networks) and reported in Chapter 4. These measures were discussed in detail in the upcoming Section 3.6.

3.4.3 SVR

SVR as described in Chapter 2 is a supervised machine learning model specifically designed for regression analysis.

The data pre-processing followed the same methodology used for neural networks. The SVR algorithm can take in a number of feature inputs to use in the prediction of one output. Different feature inputs were tested in a similar manner to the tuning of the neural networks to determine whether an increased feature number improved the SVR model.

To test the SVR algorithm against the coffee franchise data set the RBF, polynomial and linear kernels were used. The kernels used a number of specified parameters in order to carry out the learning process in the Python environment, as described in Section

2.5.

3.4.3.1 Parameter tuning

In order to tune these parameters across kernels, a built in grid search function (from the Scikit-learn library) was used with K-fold cross validation. The function was developed to automate the hyper parameter tuning process that was otherwise a time-consuming task. To determine the best performing parameters, the process used cross validation. K-fold cross validation divided the data set into folds, each fold was involved in the testing and training process in order to extract the most accurate model parameters [48].

The parameter tuning was an iterative process to define the ranges in which the model was optimised. This began with four to five values ranging from much smaller than one to much greater than one. The range was narrowed down depending on the best parameters selected by the grid search method. Figure 3.14 below showed a Python code extract of the implementation of grid search and the range input for the parameters C, epsilon, and gamma. This process was repeated for both the polynomial and linear kernels.

```
gsc = GridSearchCV(
        estimator=SVR(kernel='rbf'),
        param_grid={
            'C': [1,5,10],
            'epsilon': [0.1, 0.001,0.0001],
            'gamma': [0.01,0.001,0.1,1]
        },
        cv=5, scoring='neg_mean_squared_error', verbose=0, n_jobs=-1)
```

Figure 3.14: Python code extract to execute a grid search for RBF

Figure 3.15 shows the Python code extract of the performance scoring used in conjunction with the cross-validation technique to define the best performing parameters for the model.

```
scoring = {
                'abs_error': 'neg_mean_absolute_error',
                'squared_error': 'neg_mean_squared_error'}

scores = cross_validate(best_svr, X, y, cv=10, scoring=scoring,
                        return_train_score=True)
print("MAE :", abs(scores['test_abs_error'].mean()), "| RMSE :",
      math.sqrt(abs(scores['test_squared_error'].mean())))
```

Figure 3.15: Python code to evaluate performance for grid search in SVR

The parameters suggested by the grid search method were captured in Table 3.4. To ensure the grid search outputs were correct, the outputs were inputted into a separate model with a smaller data set. This model was fitted and tested to extract the performance metrics which were then compared to the grid search metrics. Some of the parameters in Table 3.4 had default as their selection. This was because the SVR default calculation for epsilon and gamma was more suitable than the grid search for these parameters. It was found that utilising the grid search for two parameters at a time outputted better results. The brackets next to default indicated the Scikit-learn default parameter calculation.

	C	Gamma	Epsilon	degree
RBF	0.1	100	Default $(0,1)$	-
Polynomial	250	Default $(\frac{1}{n_features * X.var()})$	Default $(0,1)$	3
Linear	10	10	Default $(0,1)$	-

Table 3.4: Optimum parameters for SVR across kernels

3.5 Implementation method of neural networks

3.5.1 Environment overview

The environment was set up with a Python notebook hosted on Google Collaboratory (see Appendix D for extract) with the following libraries installed:

- TensorFlow
- Keras

- Numpy
- Scikit-learn

3.5.2 Overview of the implementation method

To develop the neural networks Google Collaboratory was utilised. This is an online cloud hosted notebook that runs Python (extension .ipynb). Google Collaboratory has made the sharing and running of Python scripts accessible via web browsers, in addition it also provides accessibility to the machine learning libraries mentioned above.

A tutorial on time series forecasting created by TensorFlow [40] outlined the recommended process to explore neural networks on a time series data set. A simple single step neural network was created first, which was able to build a prediction for one time step into the future. This was done for a number of networks, starting with the baseline, a linear model, a dense model and lastly two more complex networks: CNN and RNN. Once the performance of these models had been evaluated statistically, a multi step model for the aforementioned networks was developed. This was done in two parts, first as a single shot model, where the steps were predicted at once and then as an auto-regressive model where a single output was fed back into the loop to further improve the next generated prediction (described in Section 2.6).

Before the neural networks were generated, a data-window was created. The purpose of the window was to create a set input length (number of time steps fed into the input), offset (number of time steps that the future prediction was offset by) and the output length which for the single step model was one. This was increased to a desired output for the multi-step model.

For the coffee franchise, demand forecasting for a week in advance would be beneficial based on the current business operations. Meal preparations were done in a central kitchen and then distributed in the mornings; most packaged meals had a maximum shelf life of two days. Based on this an ideal data window would take in an optimum input based on

the most effective training results, and the output would ideally be five days ahead.

The initial data set was divided into three sets: training, evaluating and testing using 60%, 20% and 20% of the data respectively. These ratios were based on recommended ratios, where the biggest portion of the data set was advised to remain as training data [34]. While developing the time series model it was crucial to ensure the data input was kept in chronological order instead of shuffled. Therefore the first 60% of the time series data was used to train the model and the remaining 40% split between evaluating and testing, where testing used the most recently dated data.

A brief outline of the neural network models developed was described in the subsequent sections below, along with an overview of the methodology used in tuning the parameters of the networks.

3.5.3 Baseline and linear models

The baseline and linear models were developed to create a baseline performance measurement from which the performance of the more complex neural networks could be comparatively assessed against. This helped in ascertaining whether the specific parameters and tuning that was applied to the complex networks improved the accuracy significantly more than the baseline.

For the single step models developed (these are models that predict one step into the future, as described in Section 2.6.3) two baseline models were used. The first base model utilised resulted in a flat line for predictions. The second was a linear model that was built with slightly more complexity than the baseline model, whilst still quite simplistic in configuration. Essentially the linear model is a dense layered network without an activation function [40].

For multi step models (models that predict more than one time step ahead, either as a single-shot or by means of auto-regression, described in Section 2.6.3) two additional baseline models were developed. The baseline model configuration termed “Last” in all

graphical results in Chapter 4, used the previous input as the predicted output. Another baseline model termed "Repeat" repeated the entire previous input window as a prediction output.

3.5.4 Dense layers

A dense layered network as described in Section 2.6.2.4 is similar to a linear model. The key difference in the dense layer model was the number of layers stacked between input and output layers. The network also had an activation function, which created a more complex regression algorithm.

A multi step input dense model built with a single step output was also created to understand the impact of using more than one time step as an input to predict one output.

3.5.5 CNN

The CNN architecture utilised was a convolutional and pooling layer that aimed to smooth input data before weights were assigned. The theory behind CNNs was discussed in Section 2.6.2.5. This architecture was implemented through the use of the Keras library on TensorFlow. The development of a CNN was similar to that of the multi dense layered network. The different nuances between these two models was captured in the specific CNN layered architecture that was extracted when developing the model.

3.5.6 RNN (LSTM)

The RNN architecture used feedback loops in order to retain memory of training instances. In addition, the network made use of an LSTM layer to enhance its performance. This theory was described in Section 2.6.2.6. As described above for CNNs, the utilisation of both TensorFlow and Keras made the development of the network straightforward. The specific architecture for the LSTM layer could be called directly from the Keras library on TensorFlow.

In the development of multi step models, the RNN (LSTM) was also developed using auto-regression (referred to as AR LSTM) in addition to the single-shot prediction methodology. The auto-regressive model differed from the single shot approach based on the prediction process. The single-shot method made all predictions at once. The auto-regressive model used the first predicted output to further improve the next time step output through the use of a loop configuration [40].

3.5.7 Parameter tuning

As described in Section 2.6 parameter tuning forms a large part of the software development for neural networks. The process of parameter tuning was implemented across the various single step and multi step (single shot and auto regressive) models developed (described in Sections 3.5.3 to 3.5.6) .

The process of parameter tuning for the neural networks was carried out manually. Previous research papers indicated successful starting point values or methods. These were used as a benchmark to determine the best suited parameters for the model via the process of elimination. The following subsections detail the most crucial parameters in time series forecasting [49].

3.5.7.1 Features

In Section 3.3, the preparation of the input data to include features was discussed. These features allowed the network to build complex patterns around the relationship between the feature and the desired output variable (referred to as a label in neural networks). The neural networks were tested with and without the inclusion of features (external factors such as rain and average daily temperature) to understand whether this brought more accuracy to the predictions. The risk of adding too many features to the data set was noise creation resulting in a difficult learning process for the model.

3.5.7.2 Batch size

The effects of a batch size of 32 and 64 were tested. Experimental testing carried out on time series predictions in the Indian stock market [49] showed that a batch size of 32 yielded the best results. This led to an initial hypothesis before tuning had taken place that the optimum batch size for the neural networks developed would also be 32. This was tested and the results were captured in Chapter 4.

3.5.7.3 Epochs

One epoch refers to a single iteration of all training instances. There was a fine balance between choosing an epoch number that was too small or too large. The first would result in a network unable to successfully find patterns, the second would lead to a case of over-fitting. This was where the model followed the patterns of the training data too closely, and failed in the evaluation and testing data [18]. The number of epochs tested were 20, 50 and 100.

In addition, a metric known as patience level was also altered between five and ten. The patience level refers to the number of epochs that the network would run through before it performs an early stop if there was no progress in validation. The metric patience was set at ten as recommended by TensorFlow [40]. To test the effect patience had on the performance of the model the level was also reduced and the results were captured.

3.5.7.4 Optimiser

In Section 2.6.3 three commonly used optimising algorithms for time series were discussed. The purpose of the optimiser was to minimise the loss function. In this case the MSE was utilised as a loss function based on recommended times series settings from TensorFlow [40]. These three optimisers: SGD, RMSprop and Adam were tested on the neural networks to evaluate their relative performance. In order to account for the reported slowness of the SGD optimiser, the epochs were also increased to observe if this had an effect on the performance of the model.

3.5.7.5 Window size

Data windowing allows input data to be broken down into input windows during the training process as discussed in Section 2.6.3 and 3.5.2.

According to research papers on DNNs, it has been commonly observed that a smaller window size resulted in a more accurate model [49]. By reducing the window size, the network was not overwhelmed by pattern finding and memorising within each window, allowing it to perform optimally. Variations in window sizes were tested on the coffee franchise data to determine whether a smaller window size had a positive effect on training and validation.

3.5.7.6 Activation function

As discussed in Section 2.6.3 the three most relevant activation functions were tested on the model, these were the ReLU, sigmoid and tanh functions. All neurons within the neural network have the same activation function, this was automatically built into TensorFlow. Therefore for testing, each neural network was tested with one activation function at a time. The ReLU function is a complex activation function, as it has the ability to handle back-propagation. Thus it was expected to be the better performing activation function. This was tested for and the results were presented in Chapter 4.

The parameters were tuned according to the methodologies and starting points indicated above. The results of the tuning process were presented in Chapter 4 along with the statistical measures used to evaluate the performance (discussed further in Section 3.6).

3.6 Methods to evaluate performance

3.6.1 Measures to test the validity of the models

To evaluate the performance of the models during the process of tuning and overall in comparing neural networks to ARIMA and SVR, all the measures outlined in Section 2.8 were used except MAPE. MAPE was excluded from initial evaluations because the data

was in a normalised state i.e. it ranged from zero to one during model development. The formula for MAPE becomes unstable and inaccurate when using small numbers due to the denominator [50]. In data sets such as the normalised coffee franchise POS data, the MAPE calculation would be extremely large and inaccurate. Therefore the MAPE could only be considered when the data was returned back to its normal state.

The calculation methodology of MAE, MSE and RMSE is different, and therefore in some instances it was expected that the measures would differ in their result of the best performing model for a given situation. RMSE does not treat each error the same. If there is a significant error the RMSE has a high negative impact compared to the MAE measurement [51]. A low MAE score is indicative of a forecast that prioritises the median (an equal split above and below the forecast). This is different to when the RMSE (and MSE) score is low, which indicates the forecast has prioritised the mean instead (the metrics looks at the average forecasting accuracy). Both nuances were considered during the parameter tuning and final model comparisons. To evaluate these metrics the TensorFlow and Scikit-learn metric libraries were utilised.

After the models were built and tested, the top three performing models were transformed back into their original state to test the MAPE and draw comparisons on their performance.

3.6.2 Measures to test the reliability of the models

As mentioned in Section 3.2 a component of the research methodology was to test the method on more than one product to gain an understanding of the reliability of the model. The model training, evaluating and testing on Product 1 was repeated numerous times to ensure reliability of results. In addition the model was also trained, evaluated and tested with Product 2 data to understand the models' performance with completely new product data.

Give the proximity and similarity of the branches within the coffee franchise, it was not necessary to test across branches. Even though Product 1 did not respond well to the

addition of weather patterns, Product 2 was modelled with the inclusion of these external patterns initially to test whether weather data improved the model.

The tuning performed on the Product 1 was kept constant for the Product 2. The graphical and statistical results were captured in Chapter 4.

Chapter 4

Results and Analysis

4.1 Overview

Table 4.1 shows a summary of the performance metrics for each of the models discussed in Chapter 3. These results were discussed in further detail throughout the Chapter.

This summary indicated that neural networks on average outperformed both ARIMA and SVR in their statistical scoring. The multi step CNN model performed the best overall across measurements, followed by the multi step RNN and dense model, both of which exhibited similar performances. This was followed in performance efficacy by the auto-regressive RNN and SVR models.

SVR exhibited better results than the single step neural networks and the multi step linear network model. This indicated that as a machine learning algorithm SVR was competitive with some of the more simplistic machine learning neural network models. The results from SVR also indicated that the model outperformed ARIMA.

Model:	MAE	MSE	RMSE
ARIMA	0.090	0.014	0.12
SVR	0.069	0.0080	0.088
Single step: linear	0.097	0.015	0.12
Single step: Dense layers	0.084	0.012	0.11
Single step: CNN	0.093	0.017	0.13
Single step: RNN (LSTM)	0.096	0.013	0.11
Multi step: Linear	0.040	0.059	0.024
Multi step: Dense layers	0.053	0.0086	0.093
Multi step: CNN	0.050	0.0075	0.087
Multi step: RNN (LSTM)	0.054	0.0065	0.081
Multi step: AR RNN (LSTM)	0.060	0.0071	0.084

Table 4.1: Comparative statistical results of the tested methods on Product 1

4.2 Benchmark models' results and analysis

4.2.1 ARIMA results and analysis

Figure 4.1 below showed the ARIMA model output on using Product 1 data from the coffee franchise. The results were based on the normalised input.

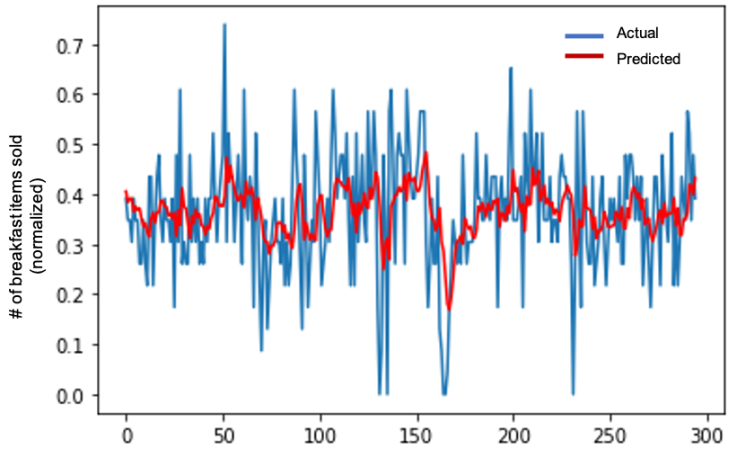


Figure 4.1: ARIMA output with normalised data

Similar to the single step neural network models, the ARIMA model predicted one time step into the future. The rolling forecast method was used to create the graphical plot seen in Figure 4.1 by rerunning the model at each time input.

The statistical results captured in Table 4.1 showed that the ARIMA model had some of the higher MAE, MSE and RMSE values. This indicated that the model was one of the lower performing models. The MAE, MSE and RMSE measures are all scale and unit dependant measures. Therefore, the measures are best used as a comparative measurement across the models.

It is difficult to evaluate the meaning of a MAE value of 0.090 for the ARIMA model, given the data underwent a process of normalisation. This meant that the data input into the ARIMA model was not the actual number of units sold but rather a scaled version of the value. Table 4.2 below states the differences between the actual data and the normalised version of this data.

Data type:	Count	Mean	Minimum value	Maximum value
Product 1	867	11	1.0	24
Product 1 normalised	867	0.42	0.0	1.0

Table 4.2: Key differences in Product 1 data after normalisation

In order to interpret a meaning for the MAE value, Equation 2.13 is utilised. This equation stated that in order to normalise the data the minimum value was subtracted from the data point and then divided by the difference between the minimum and maximum values. This equation was reordered to produce the equation used to translate a normalised data point to its original value. Using the data in Table 4.2 this conversion would result in following equation:

$$x = (x_{normalised} \times 23) + 1 \tag{4.1}$$

Which indicated the method used to transform the mean, minimum value and maximum values back into its original form in Table 4.2.

The conversion stated in Equation 4.1 is substituted into the equation used to calculate MAE (Equation 2.18) for each y_i and $\hat{y}_i$. The substitution exercise resulted in a formula to convert the $MAE_{normalised}$ to the actual MAE value based on the original data. This is shown below:

$$MAE = 23 \times MAE_{normalised} \tag{4.2}$$

Using Equation 4.2 the MAE for the ARIMA model equated to 2.1. This valued represented the mean of the absolute error in ARIMA forecasting. If the average units of Product 1 sold is ±11 this indicated the average error in predicting would either be two units above or two units below the actual number sold.

The values for RMSE and MSE can also be converted using a similar methodology. But this did not add any value to the interpretation of the individual model performance. The MSE and RMSE values were better utilised for comparative purposes between models. The different calculation methodologies for these metrics meant the results offered different views on the performance of models, in terms of how the errors were spread around the actual value. This was discussed further in Chapter 5.

4.2.2 SVR results and analysis

Figure 4.2 below shows the SVR model fit output utilising the grid search recommended parameters. The results captured in Table 4.1 were for the SVR model with a polynomial kernel. This model proved to be the best performing amongst the SVR models.

The results of all three kernels tested are captured below in Table 4.3.

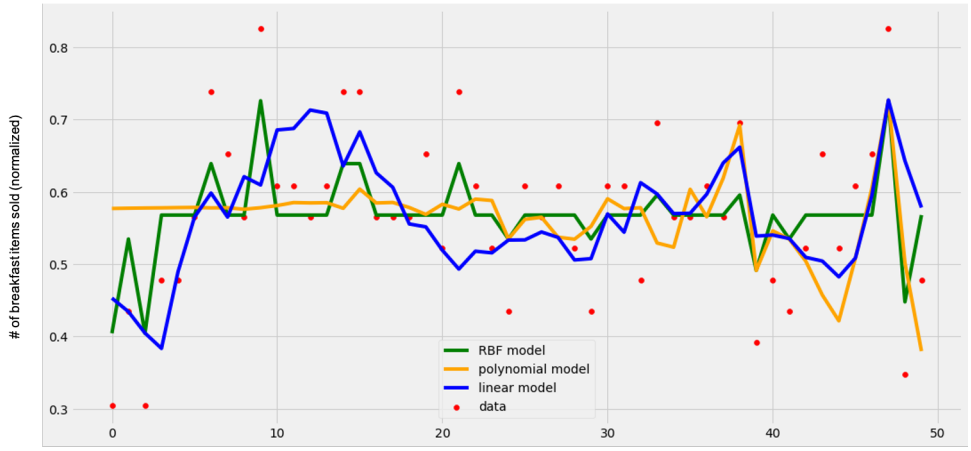


Figure 4.2: SVR results from grid search

Kernel type:	MAE	MSE	RMSE
RBF	0.13	0.026	0.16
Polynomial	0.069	0.0080	0.088
Linear	0.16	0.039	0.20

Table 4.3: Performance of SVR model across kernels

From the results in Figure 4.2 the RBF model exhibited the closest trend line to the actual data points (marked as red dots). However, the results captured in Table 4.3 reflect that it was the worst performing kernel amongst the three kernels tested in SVR. The test data was separated from the training data. Therefore, it was likely that the RBF over-fitted to the training data which resulted in inaccurate predictions of unseen data.

From Table 4.3 it is evident that the simplicity of the linear kernel would not allow the model to accurately predict the volatile data presented.

The success of the polynomial kernel likely came from its ability to avoid over fitting to the training data pattern. This allowed it to predict unseen data better than the RBF kernel. The MAE stated in Table 4.3 and 4.1 for the polynomial kernel was converted using Equation 4.2. This resulted in an actual MAE of 1.6. This implied that the average absolute error is 1.6 units of Product 1.

The SVR model performed better when more features were inputted into the model. The optimum feature list was determined through trial and error and resulted in the following final features:

- Rain
- Temperature
- Day of the week
- All sine and cosine curves described in Section 3.3.2

Comparing the MAE of the two benchmark models showed that the average absolute unit difference between these two models is ±0.5. Meaning the ARIMA model will on average have a unit error of 0.5 higher than the SVR model when predicting the estimated demand for Product 1.

4.3 Neural network results and analysis

4.3.1 Parameter tuning results

The following comparison of results across tuning parameters looked at both test and validation results separately. Separating the test and validation results was necessary to ensure that the model was not over or under fitted. In Section 3.5 the different neural networks under single step and multi step models were listed. For the purpose of comparison and tuning; the baseline, first and linear models served as a benchmark for the neural network results. For this reason they were not examined as closely as the complex networks.

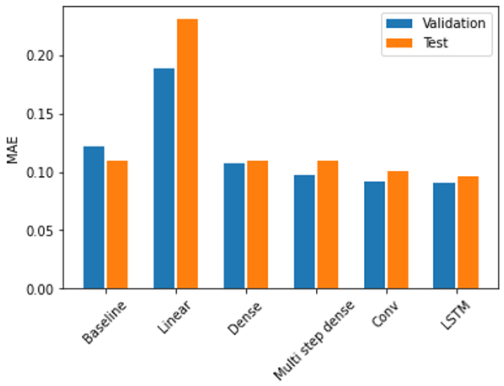
The graphical results displayed in this section are captured as numerical values in Appendix E. These values were used for detailed analysis during the process of parameter tuning.

4.3.1.1 Features

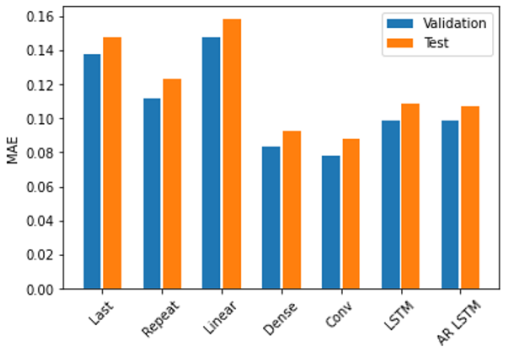
In Section 3.3.2 the complete feature list was filtered down to include: day of the week, daily temperature, rain and sinusoidal curves spanning the year. This was done to exclude any features that were contributing more noise to the data set (such as the other sinusoidal curves and weather patterns). To further test whether the filtered list of features improved the results, the model was run solely with the sinusoidal curves in Figure 3.7b.

The results below in Figure 4.3 to 4.5 showed that across all metrics, there was a visible improvement in prediction accuracy when day of the week and daily temperature were excluded. For Product 1 this was understandable, as weather might not effect the demand for the product as much as it would for temperature dependant food (such as juices or salads in summer and hot drinks in winter).

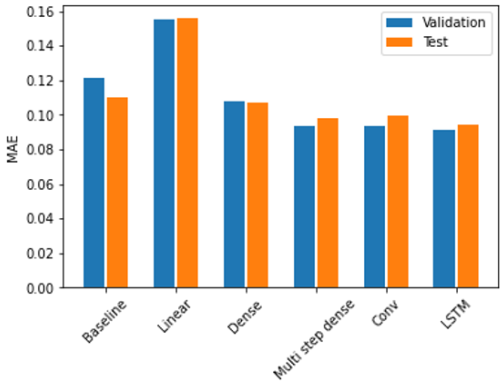
Figure 4.3 to 4.5 also illustrated that the multi step model had better prediction accuracy, which was expected given its ability to handle more complex analysis and training compared to a single step model.



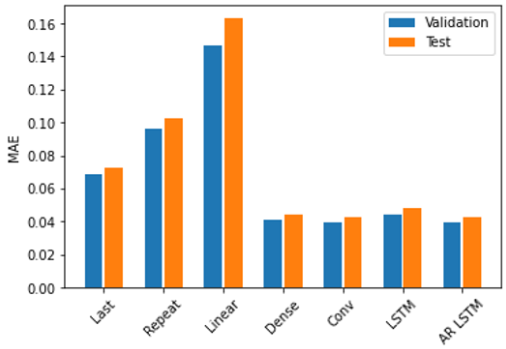
(a) Single step models with all features included



(b) Multi step models with all features included



(c) Single step models with only sine and cosine curve features included

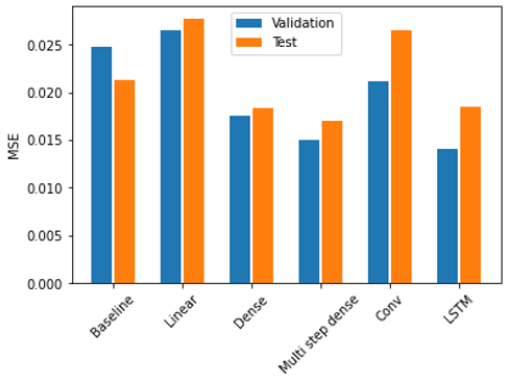


(d) Multi step models with only sine and cosine curve features included

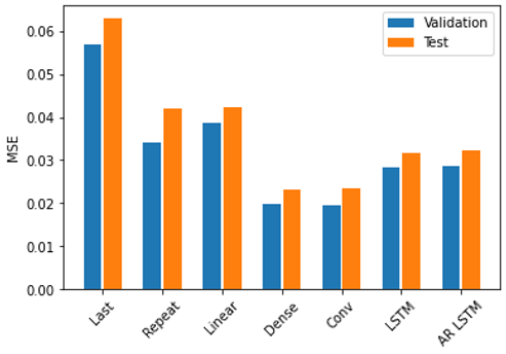
Figure 4.3: Comparison of MAE across models that include all features, and those that include only sine and cosine curves of year periodicity [1]

[1]For all subsequent graphs within parameter tuning the y-axis references the following models:

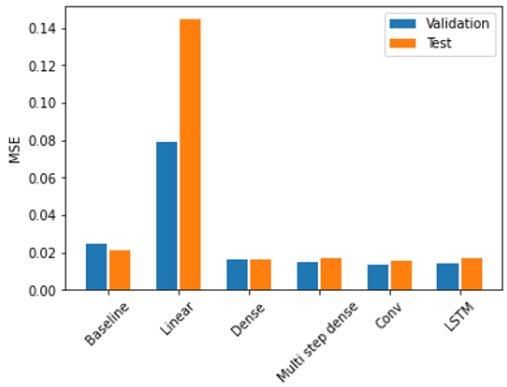
- Baseline: Baseline network
- Last: Baseline model - repeats the last window step
- Repeat: Baseline model - repeats the last input window
- Linear: Linear neural network
- Dense: Dense layers neural network
- Conv: CNN
- LSTM: RNN (LSTM)
- AR LSTM: Auto-regressive RNN (LSTM)



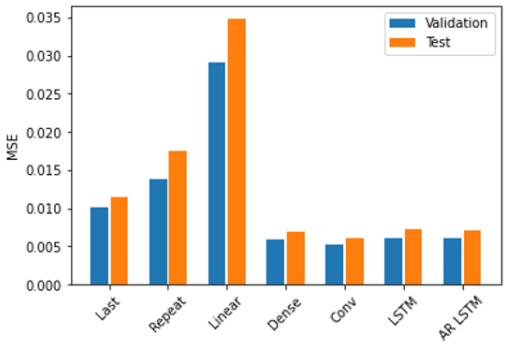
(a) Single step models with all features included



(b) Multi step models with all features included

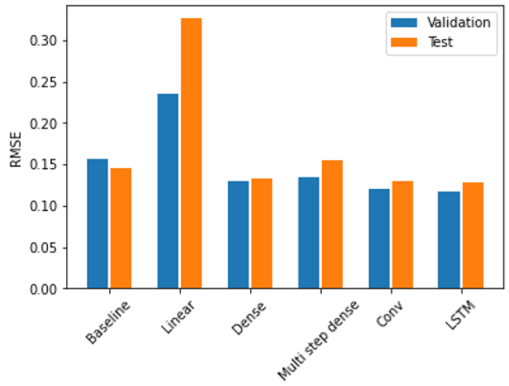


(c) Single step models with only sine and cosine curve features included

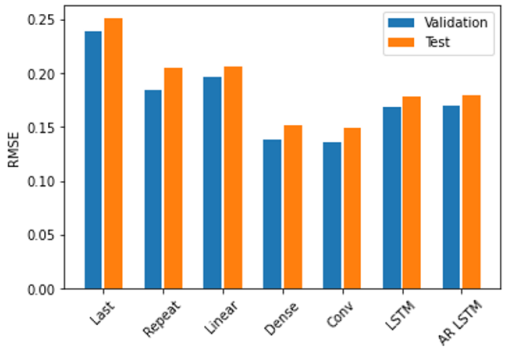


(d) Multi step models with only sine and cosine curve features included

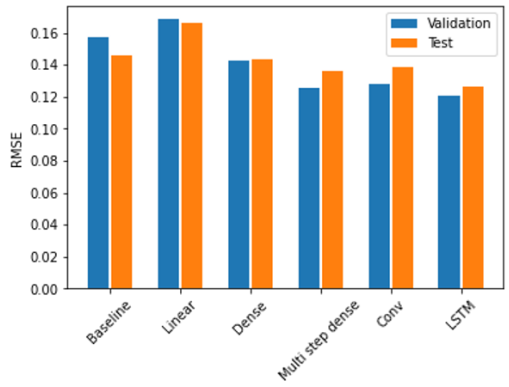
Figure 4.4: Comparison of MSE across models that include all features, and those that include only sine and cosine curves of year periodicity



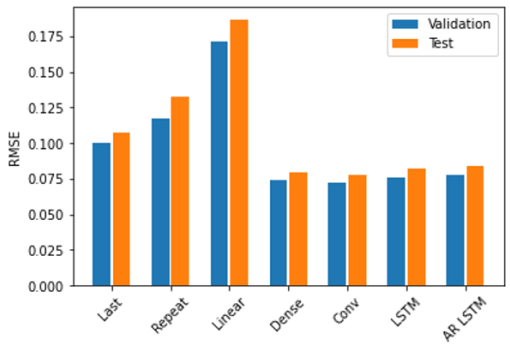
(a) Single step models with all features included



(b) Multi step models with all features included



(c) Single step models with only sine and cosine curve features included



(d) Multi step models with only sine and cosine curve features included

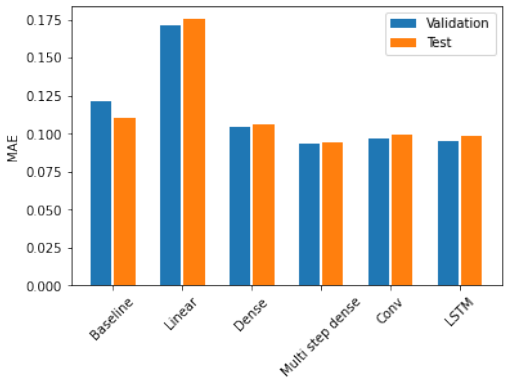
Figure 4.5: Comparison of RMSE across models that include all features, and those that include only sine and cosine curves of year periodicity

4.3.1.2 Batch size

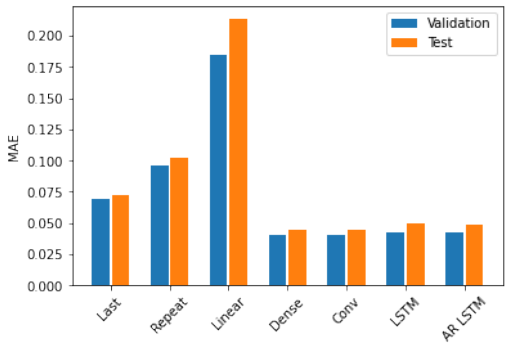
In Figure 4.3c, 4.3d, 4.4c, 4.4d, 4.5c and 4.5d the results shown used a batch size of 32. Figure 4.6 to Figure 4.8 below show the model with a batch size of 64. There was minimal improvement in the forecasting accuracy with a larger batch size and in some cases it negatively impacted the forecasting accuracy.

A batch size of 32 yielded marginally better results. This was confirmed by experimental testing carried out on time series predictions in the Indian stock market [49] that showed that a batch size of 32 yielded the best results. A possible reason that a batch size of 64 did not perform as well as a batch size of 32, could be due to larger the amount of samples shown to the network when the batch size was set to 64. These samples were used to

update the networks weights, therefore overloading the network with samples may have led to inaccurate forecasting.

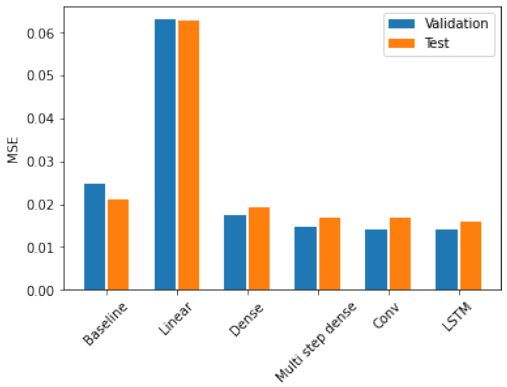


(a) Single step models with a batch size of 64

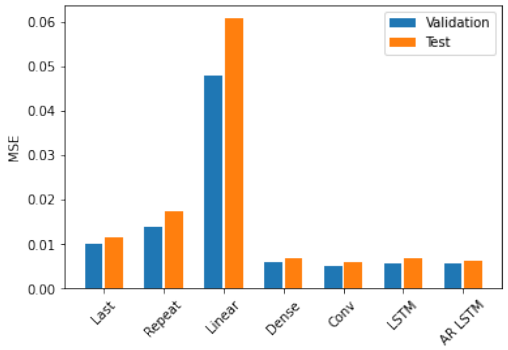


(b) Multi step models with a batch size of 64

Figure 4.6: MAE across models with a batch size of 64

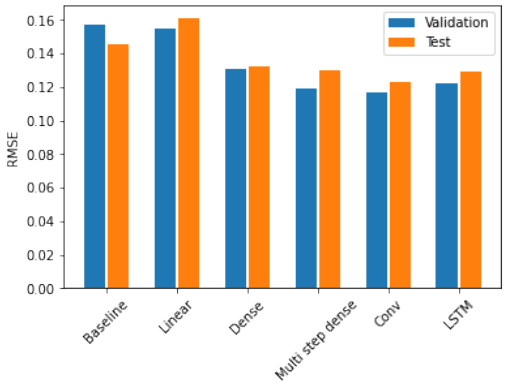


(a) Single step models with a batch size of 64

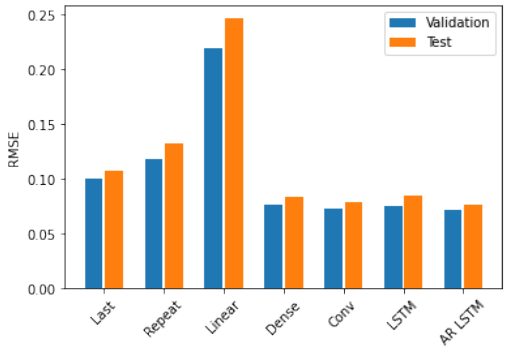


(b) Multi step models with a batch size of 64

Figure 4.7: MSE across models with a batch size of 64



(a) Single step models with a batch size of 64

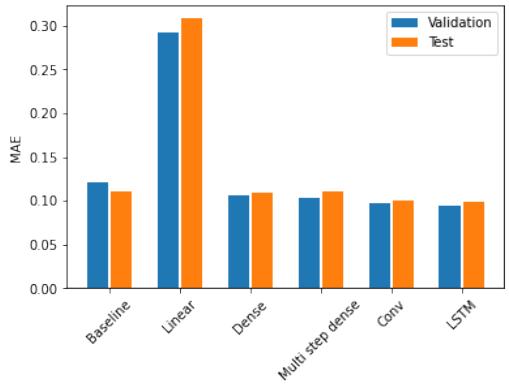


(b) Multi step models with a batch size of 64

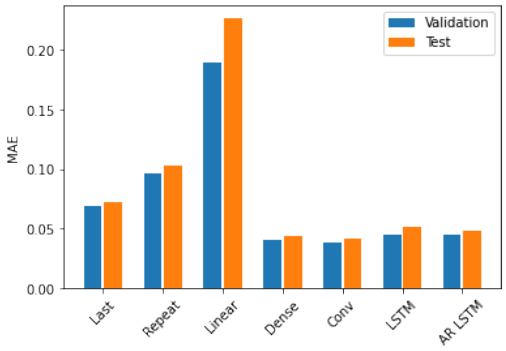
Figure 4.8: RMSE across models with a batch size of 64

4.3.1.3 Epochs

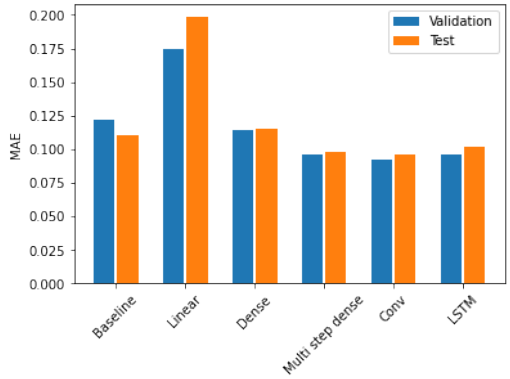
In Figure 4.3c, 4.3d, 4.4c, 4.4d, 4.5c and 4.5d 50 epochs were used. Figure 4.9 to 4.11 below shows the model results for 20 and 100 epochs respectively. From the figures it was evident that the optimum number of epochs was between the range of 50 to 100. This was confirmed by testing the model with 150 epochs, which resulted in a decrease in performance.



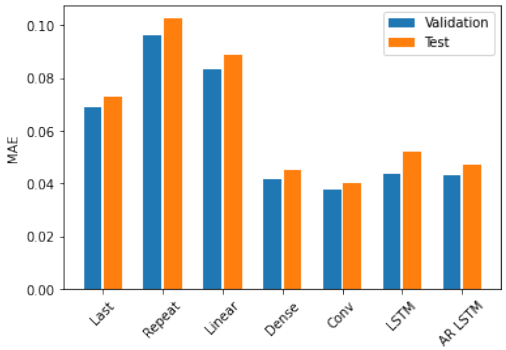
(a) Single step models with epochs set to 20



(b) Multi step models with epochs set to 20

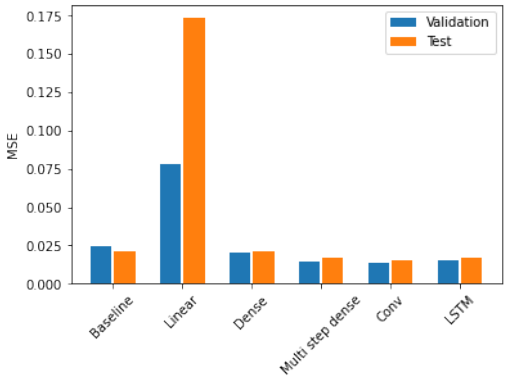


(c) Single step models with epochs set to 100

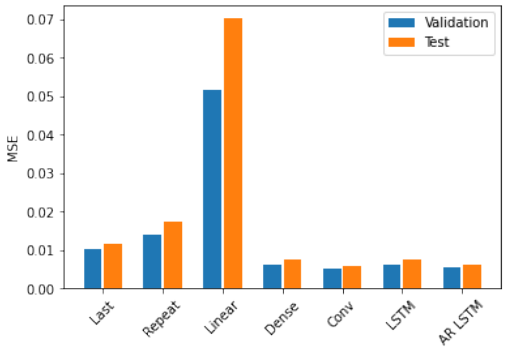


(d) Multi step models with epochs set to 100

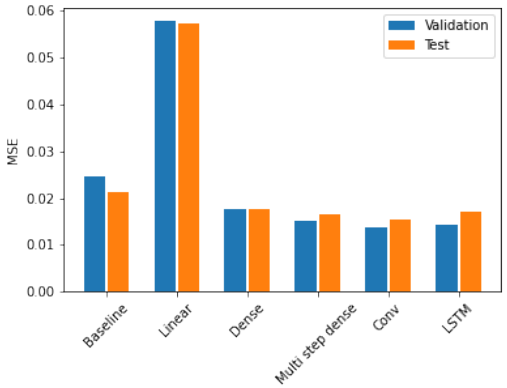
Figure 4.9: Effect of number of epochs on MAE across models



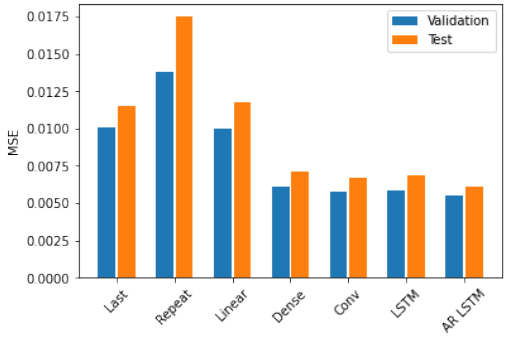
(a) Single step models with epochs set to 20



(b) Multi step models with epochs set to 20

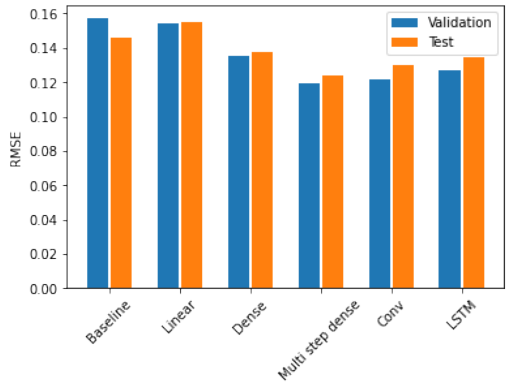


(c) Single step models with epochs set to 100

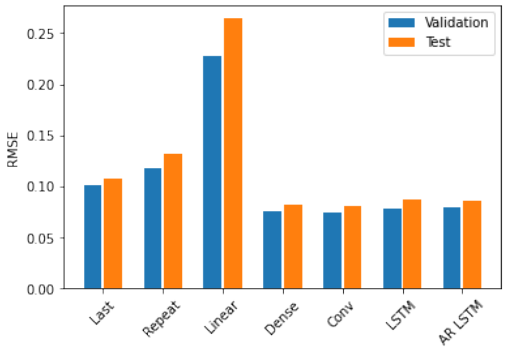


(d) Multi step models with epochs set to 100

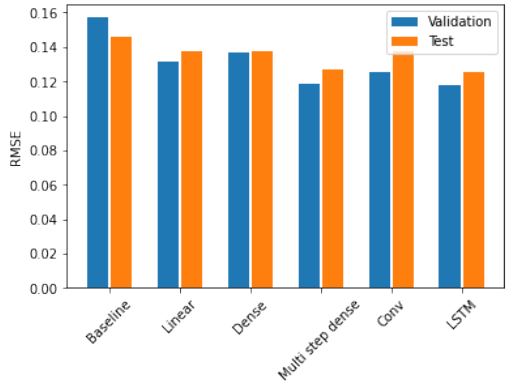
Figure 4.10: Effect of number of epochs on MSE across models



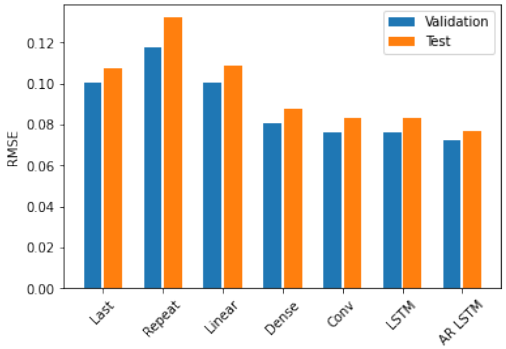
(a) Single step models with epochs set to 20



(b) Multi step models with epochs set to 20



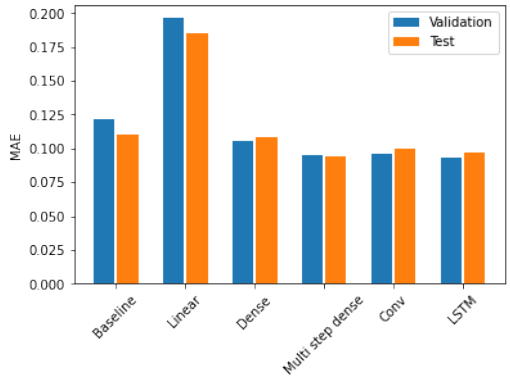
(c) Single step models with epochs set to 100



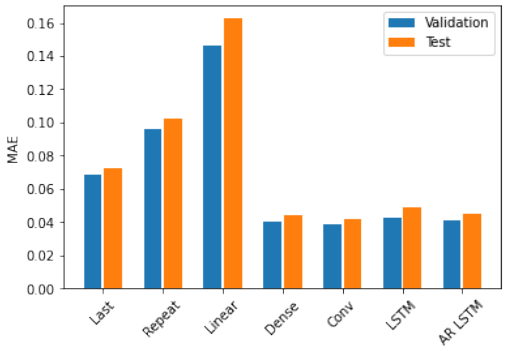
(d) Multi step models with epochs set to 100

Figure 4.11: Effect of number of epochs on RMSE across models

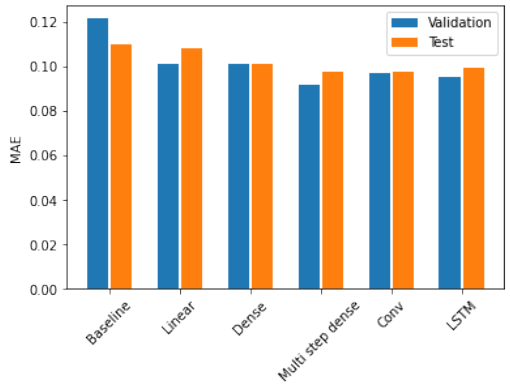
TensorFlow in [34] recommended a patience setting of ten. Therefore in all figures up till this point this was the patience setting used. To further test the epoch settings the patience was reduced to five. The results are seen in Figure 4.12 to 4.14 below. In general a patience of ten yielded better results. A lower patience can improve the oscillation in graphs through an early stop of the learning stage, but this can also hinder the learning ability of the model, hence in some cases a larger patience is ideal [34]. In the case of the coffee franchise data set, a larger patience did result in a better performing model. The model was also run with a patience larger than ten to confirm this and the results were conclusive. In order to ensure the model ran optimally the patience was capped at ten to avoid a slow training, evaluating and testing process.



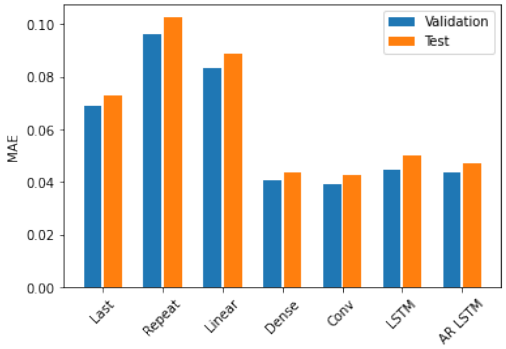
(a) Single step models with epochs = 50 and patience = 5



(b) Multi step models with epochs = 50 and patience = 5

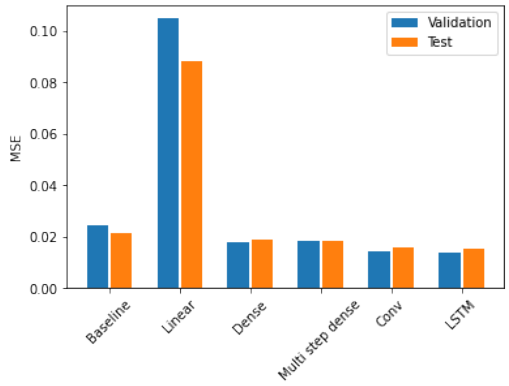


(c) Single step models with epochs = 100 and patience = 5

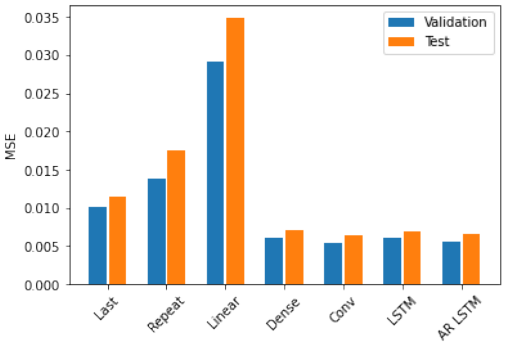


(d) Multi step models with epochs = 100 and patience = 5

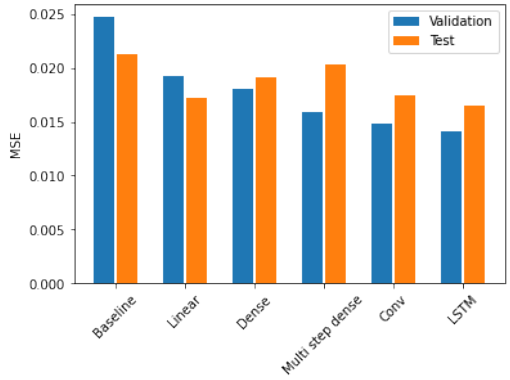
Figure 4.12: MAE across models with epoch = 100 and 50 and patience = 5



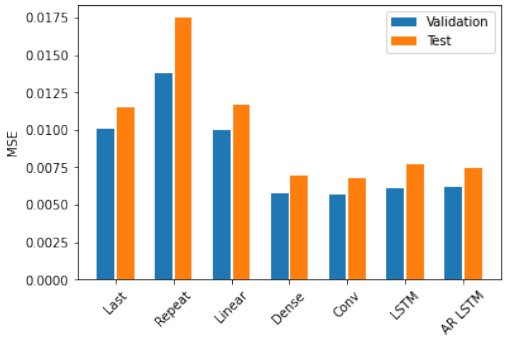
(a) Single step models with epochs = 50 and patience = 5



(b) Multi step models with epochs = 50 and patience = 5

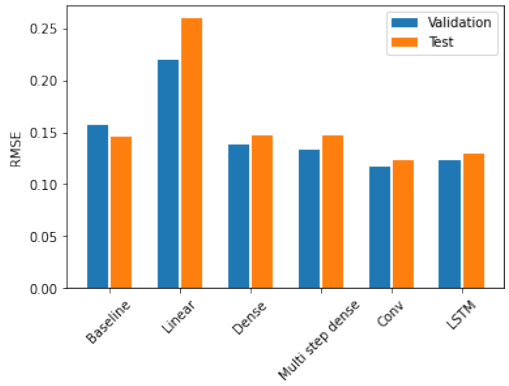


(c) Single step models with epochs = 100 and patience = 5

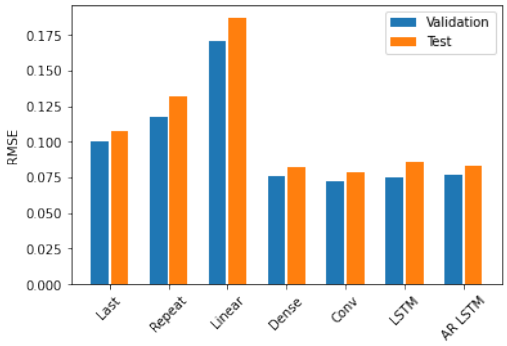


(d) Multi step models with epochs = 100 and patience = 5

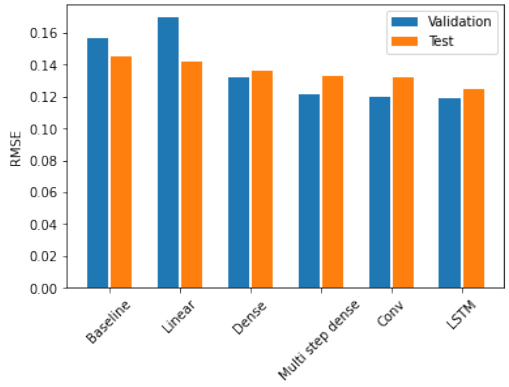
Figure 4.13: MSE across models with epoch = 100 and 50 and patience = 5



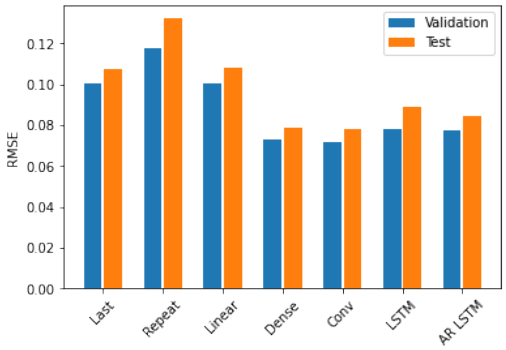
(a) Single step models with epochs = 50 and patience = 5



(b) Multi step models with epochs= 50 and patience = 5



(c) Single step models with epochs = 100 and patience = 5



(d) Multi step models with epochs = 100 and patience = 5

Figure 4.14: RMSE across models with epoch = 100 and 50 and patience = 5

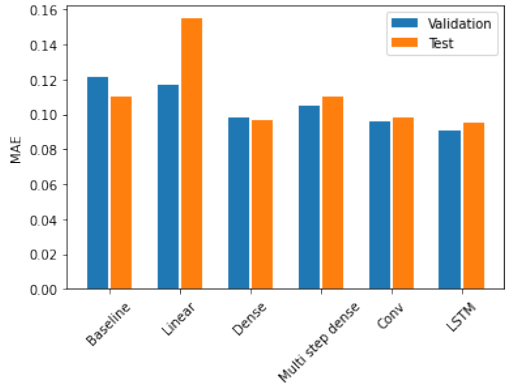
The figures above illustrated that the optimum number of epochs was between 50 and 100. Therefore the number of epochs was set to 75 with a patience of ten going forward.

4.3.1.4 Optimiser

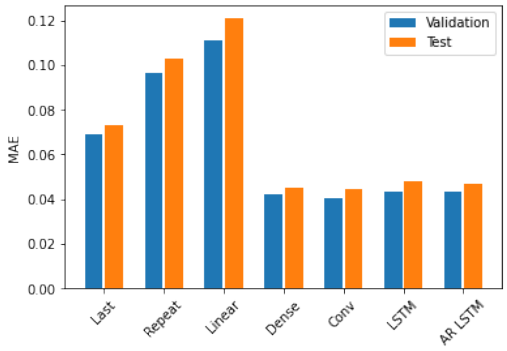
Figure 4.15 to 4.17 show the performance results for the models tested with an Adam, RMSprop and SGD optimiser, as described in Section 2.6.3. The role of the optimiser is to minimise the loss between actual versus predicted values through the alteration of model weights and learning rates.

It was clear from the results that the figures presented, that the SGD optimiser was the worst performing optimiser. The Adam and RMSprop optimisers were equally successful as they exhibited similar results. The detailed analysis on the results from both these

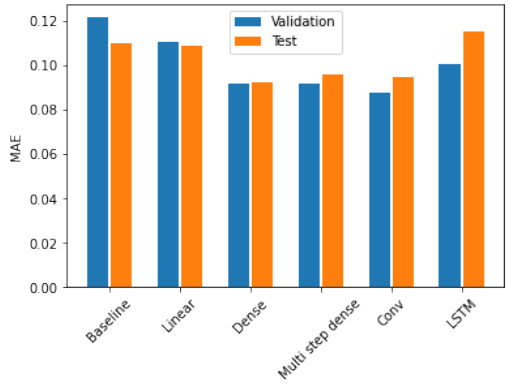
tests showed that the RMSprop optimiser had an advantage over the Adam optimiser. Therefore the RMSprop was the selected optimiser utilised in the neural network models going forward.



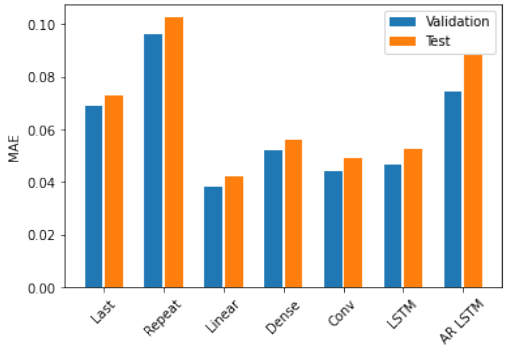
(a) Single step models with an Adam optimiser



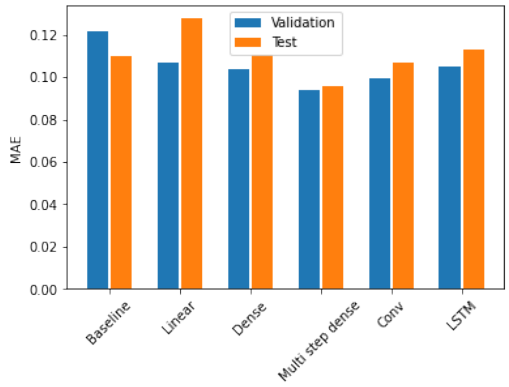
(b) Multi step models with an Adam optimiser



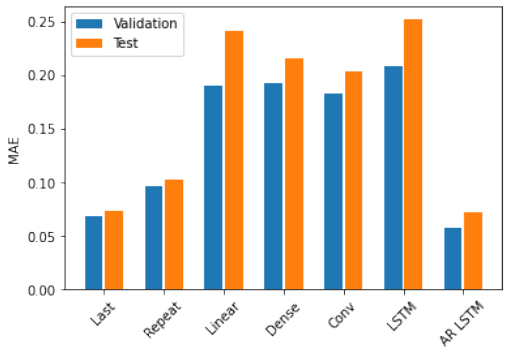
(c) Single step models with an RMSprop optimiser



(d) Multi step models with an RMSprop optimiser

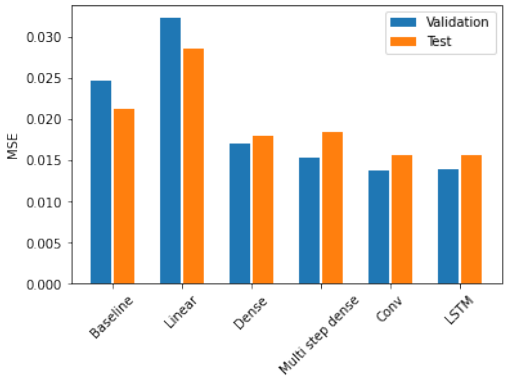


(e) Single step models with an SGD optimiser

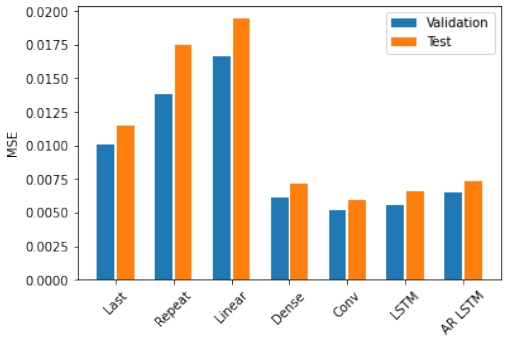


(f) Multi step models with an SGD optimiser

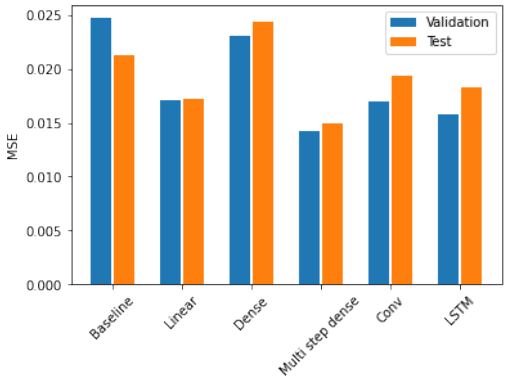
Figure 4.15: Effects of optimiser on MAE across models



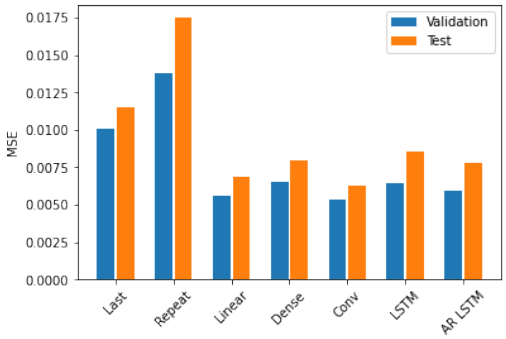
(a) Single step models with an Adam optimiser



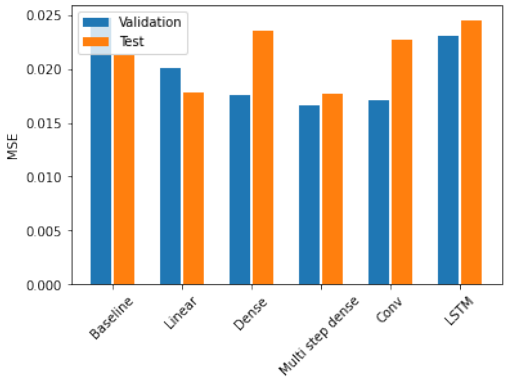
(b) Multi step models with an Adam optimiser



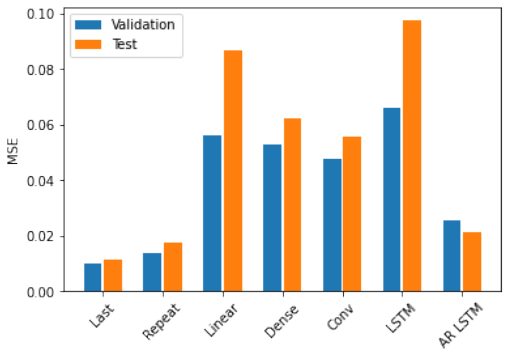
(c) Single step models with an RMSprop optimiser



(d) Multi step models with an RMSprop optimiser

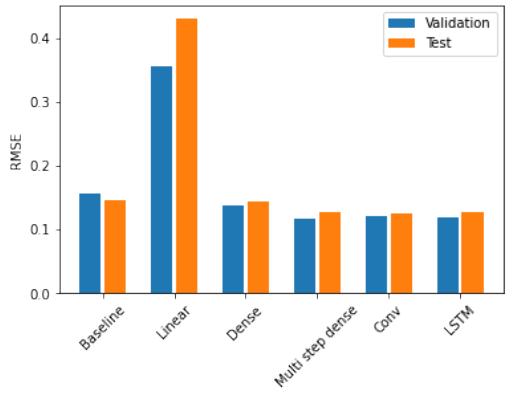


(e) Single step models with an SGD optimiser

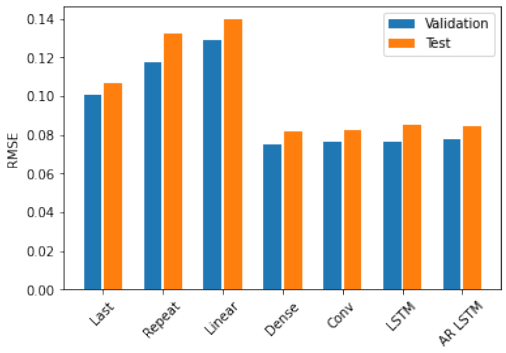


(f) Multi step models with an SGD optimiser

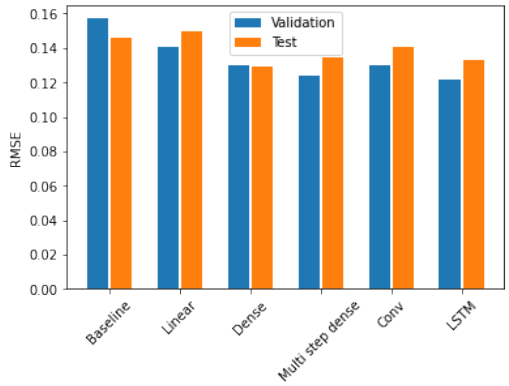
Figure 4.16: Effects of optimiser on MSE across models



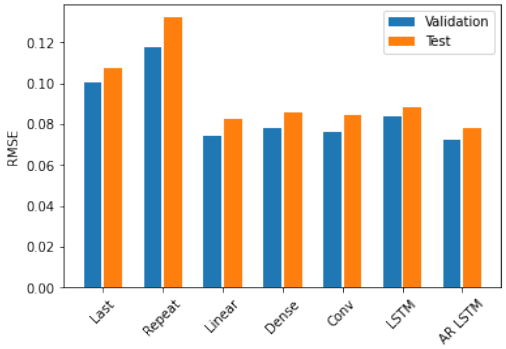
(a) Single step models with an Adam optimiser



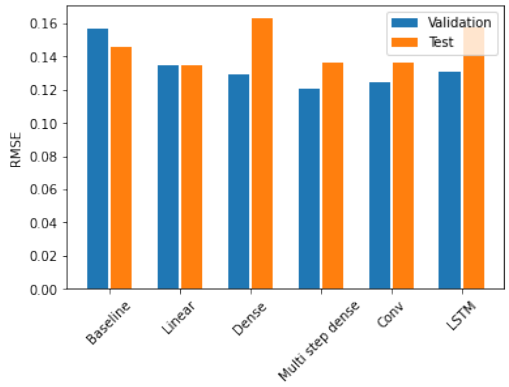
(b) Multi step models with an Adam optimiser



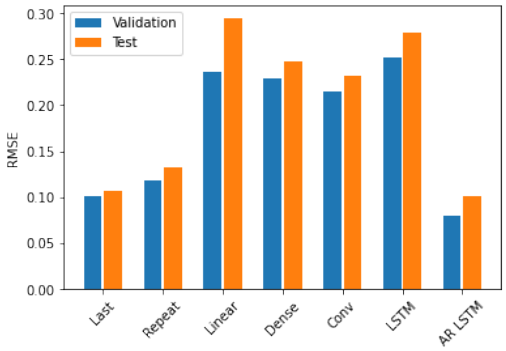
(c) Single step models with an RMSprop optimiser



(d) Multi step models with an RMSprop optimiser



(e) Single step models with an SGD optimiser



(f) Multi step models with an SGD optimiser

Figure 4.17: Effects of optimiser on RMSE across models

4.3.1.5 Window size

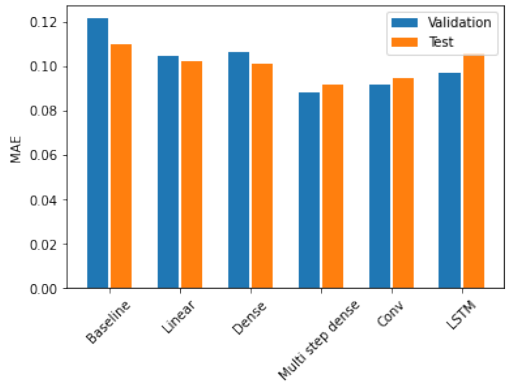
The initial input width window size, along with the values used to test the effects of an increased and decreased window size are captured in Table 4.4 below. The models marked with a dash did not have a configuration that allowed the window size to be altered.

Model:	Initial input width:	Input width increased to:	Input width decreased to:
Single step models:			
Baseline	N/A	-	-
Linear	1	-	-
Dense	1	-	-
Multi step dense	10	15	5
CNN	10	15	5
RNN (LSTM)	1	-	-
Multi step models:			
Baseline	N/A	-	-
Linear	1	-	-
Dense	1	-	-
CNN	3	6	2
RNN (LSTM)	10	15	5
AR LSTM	10	15	5

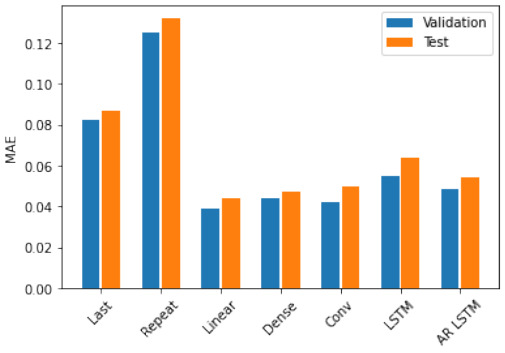
Table 4.4: Current and adjusted window size inputs for parameter tuning

The results from the initial input width window size are reflected in Figures 4.15 to 4.17 c.and d. (RMSprop optimiser). These results were compared to the results of the altered window sizes, shown in Figures 4.18 to 4.20.

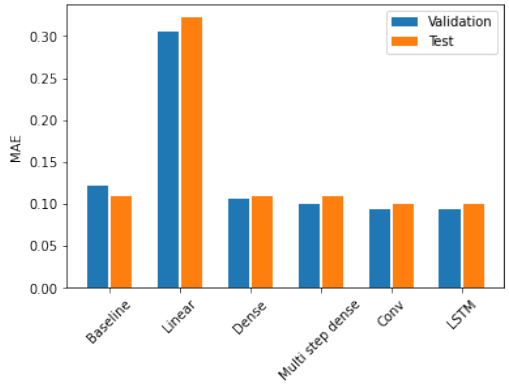
From analysis of the figures below, an increased window size showed improvement in the MAE and MSE values, but a decrease in performance according to RMSE values. The overall performance of the models using an increased window size, proved to be similar to the models' performance using the initial window size settings. The detailed analysis across all performance measures showed that the initial window input width performed marginally better. These initial settings were also recommended values by TensorFlow [40]. Therefore, these settings were kept the same.



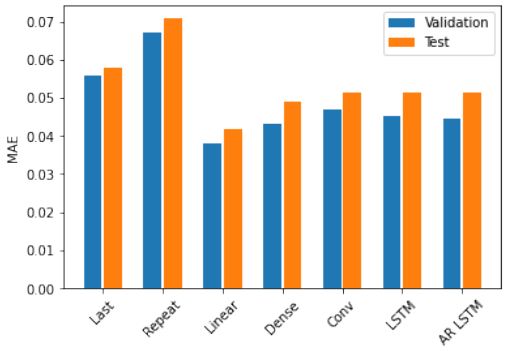
(a) Single step models with an increased window



(b) Multi step models with an increased window

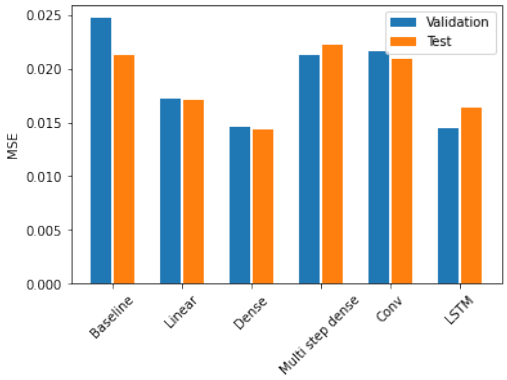


(c) Single step models with a decreased window

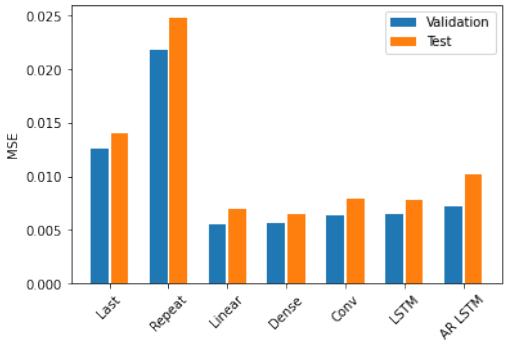


(d) Multi step models with a decreased window

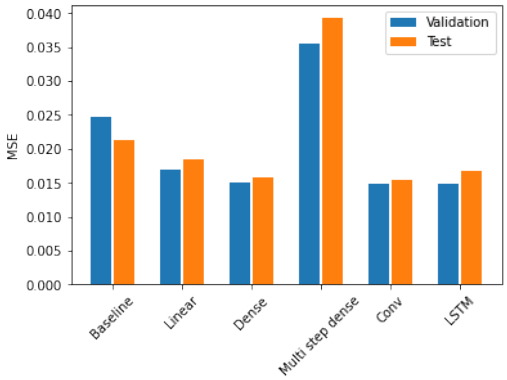
Figure 4.18: Effects of window size on MAE across models



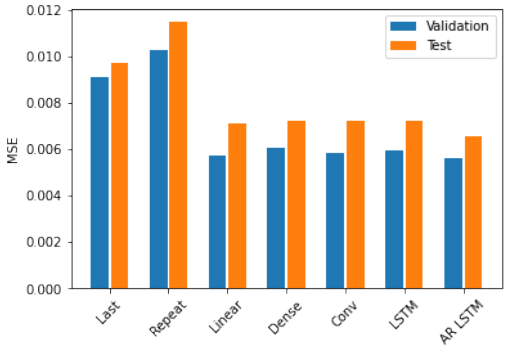
(a) Single step models with an increased window



(b) Multi step models with an increased window

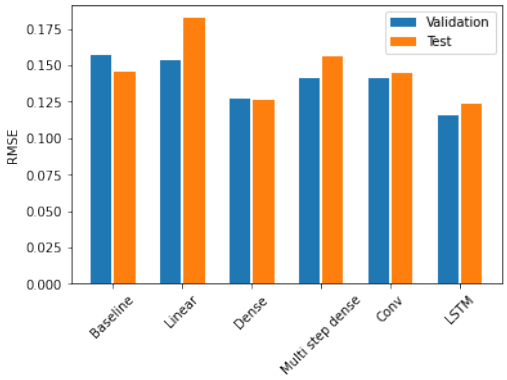


(c) Single step models with a decreased window

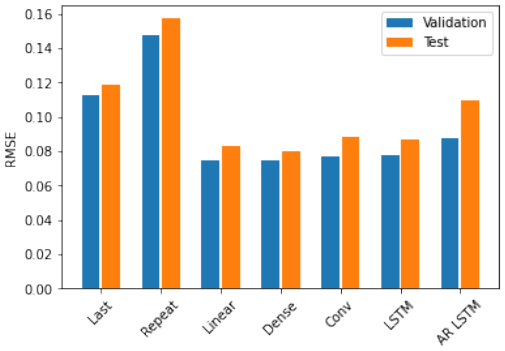


(d) Multi step models with a decreased window

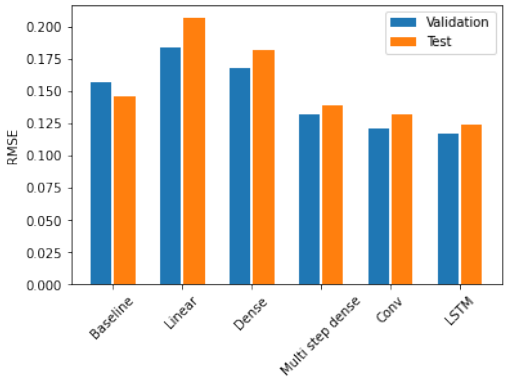
Figure 4.19: Effects of window size on MSE across models



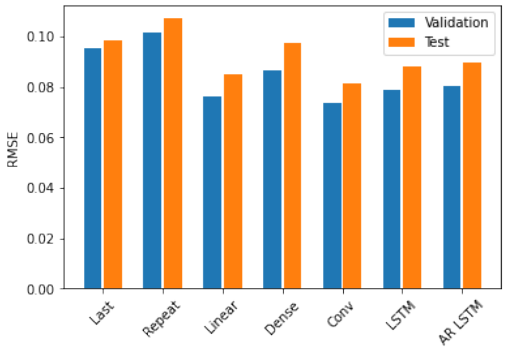
(a) Single step models with an increased window



(b) Multi step models with an increased window



(c) Single step models with a decreased window



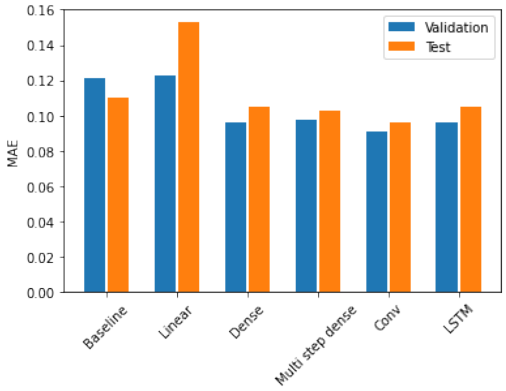
(d) Multi step models with a decreased window

Figure 4.20: Effects of window size on RMSE across models

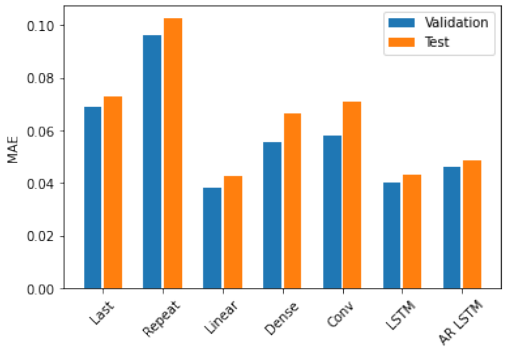
4.3.1.6 Activation functions

Figures 4.15 - 4.17 c. and d. showed the results for the ReLU activation function which was also used till this point. Figure 4.21 to 4.23 below shows the results for the sigmoid and tanh activation functions.

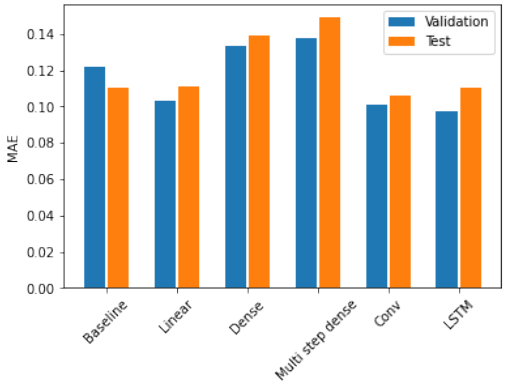
From the results it was evident that the sigmoid activation function was the weakest performer. The results of the tanh and ReLU activation functions exhibited a similar performance. Theoretically ReLU is an activation function designed to handle more complex networks. Therefore, it was kept as the activation function going forward.



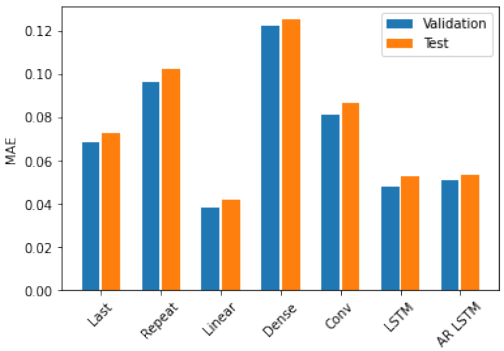
(a) Single step models with a tanh activation function



(b) Multi step models with a tanh activation function

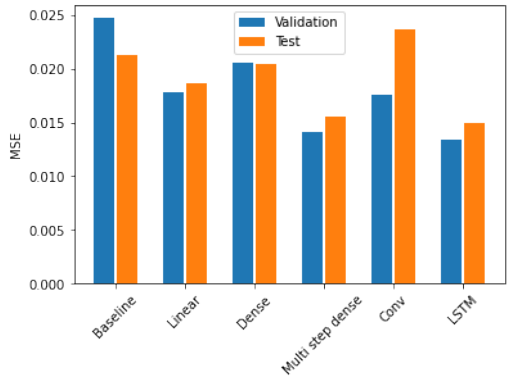


(c) Single step models with a sigmoid activation function

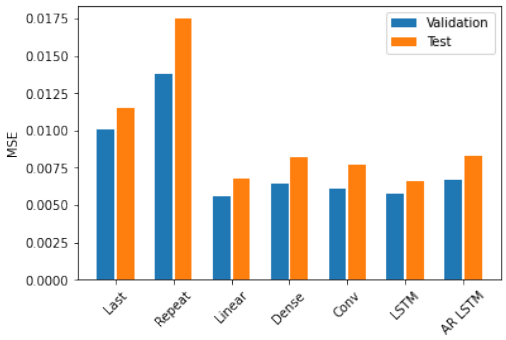


(d) Multi step models with a sigmoid activation function

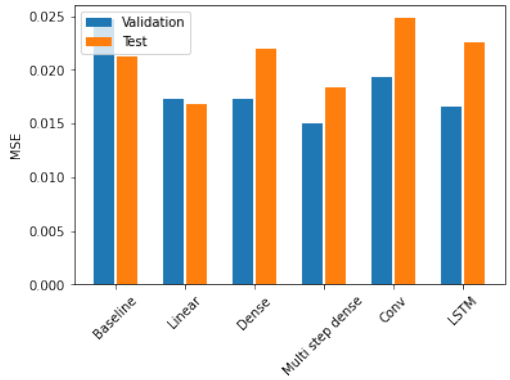
Figure 4.21: Effects of activation functions (tanh and sigmoid) on MAE across models



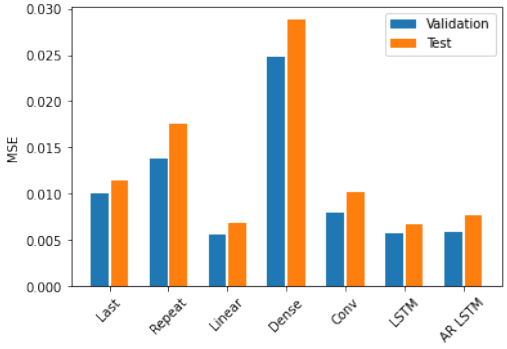
(a) Single step models with a tanh activation function



(b) Multi step models with a tanh activation function

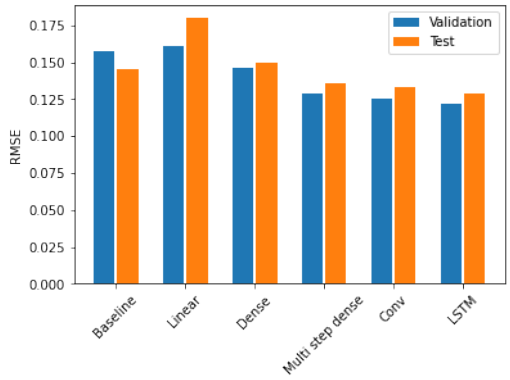


(c) Single step models with a sigmoid activation function

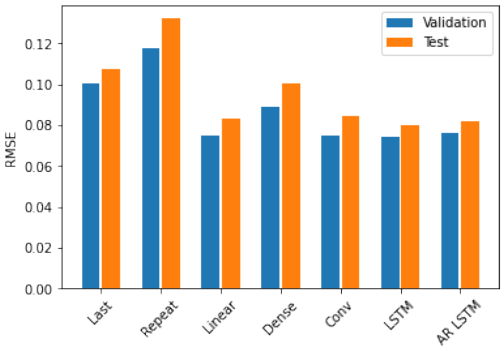


(d) Multi step models with a sigmoid activation function

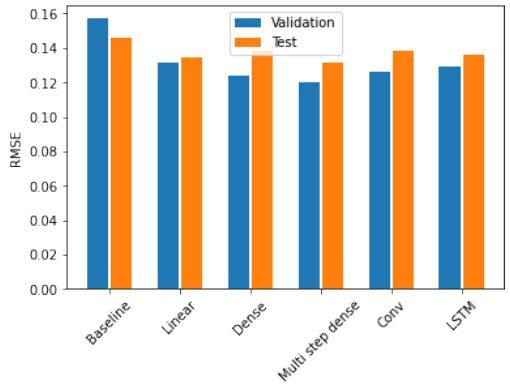
Figure 4.22: Effects of activation functions (tanh and sigmoid) on MSE across models



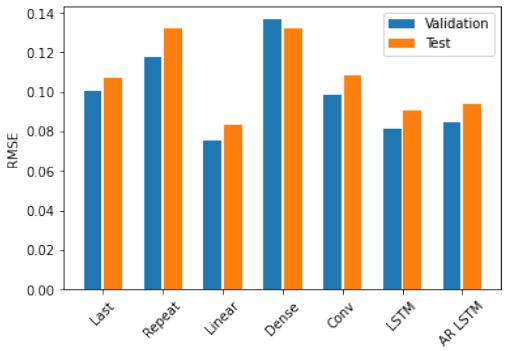
(a) Single step models with a tanh activation function



(b) Multi step models with a tanh activation function



(c) Single step models with a sigmoid activation function



(d) Multi step models with a sigmoid activation function

Figure 4.23: Effects of activation functions (tanh and sigmoid) on RMSE across models

4.3.1.7 Data split

As stated in Section 3.5.2 the training data set constituted 60% of the total data set. The remaining data was split evenly between evaluation and testing (20% each). Learning curves (plots of the loss per iteration during training and evaluation) of the single step models which are shown in the forthcoming sections (e.g. Figure 4.28) displayed patterns that indicated the evaluation data set was too small in comparison to the training data set [52]. The learning curves did not show 75 epochs due to the machine learning error handling that stopped the model from learning when it picked up repetitive patterns in the results. While this did not largely affect the results of the model, it was ideal to have a smooth learning curve, therefore the data split was adjusted to 60%, 30% and 10% respectively.

4.3.2 Baseline results and analysis

The results presented in subsequent subsections are based on the testing portion of the input data set. These results have different output lengths as seen on the x-axis of Figures 4.24, 4.26 etc. This is due to differences in the models' set-up which varies in terms of window size and output length. The models' results display three graphs which represent iterations of testing on the testing data set in order to ensure the stability of the results.

4.3.2.1 Single step model

A linear transformation was the simplest model that could be created for the data set. While the data shown in Figure 4.24 had promising results for predicted versus actual numbers, the model was relying solely on one input to predict the next output. Figure 4.25 showed the learning curve for the training, at each iteration the loss decreased smoothly as expected given the simplicity of the model.

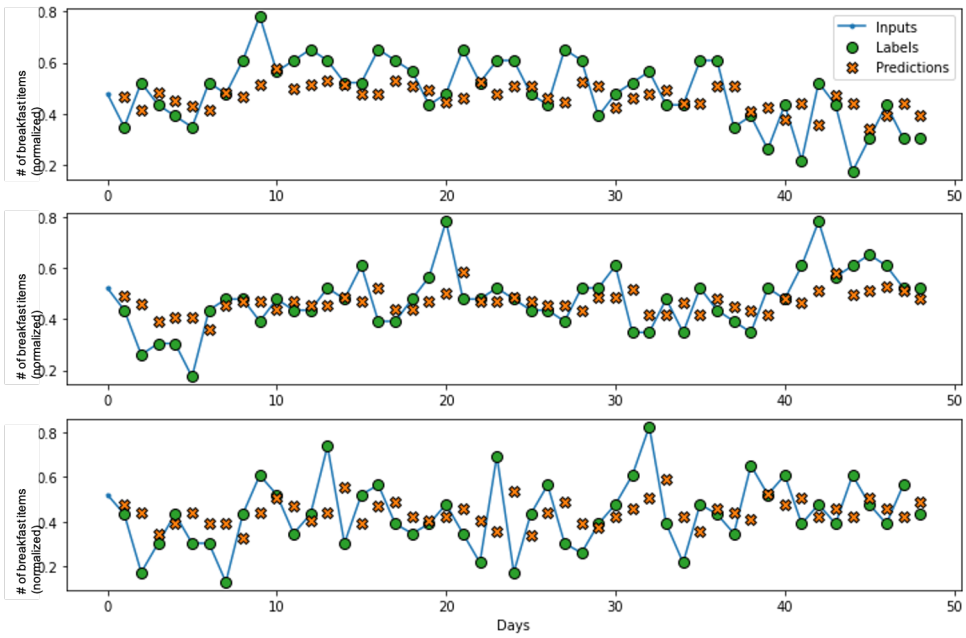


Figure 4.24: Single step model linear results

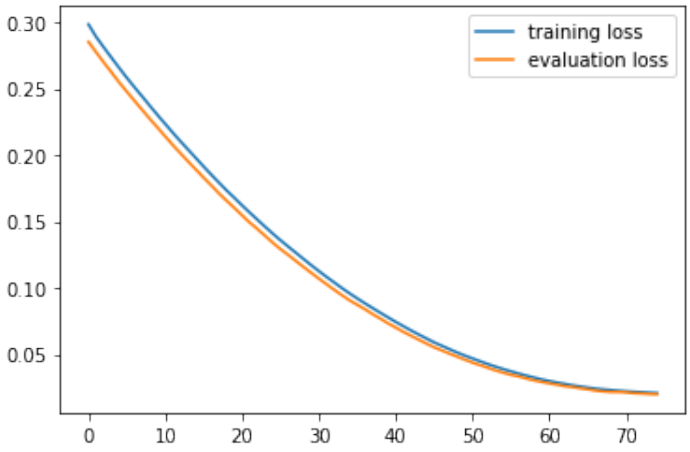


Figure 4.25: Single step linear learning curve

4.3.2.2 Multi step model

The results in Figure 4.26 were not as good as the single step linear model results. The simplicity of the model did not allow for the behaviour to be captured correctly, resulting in a stagnant output. Figure 4.27 showed the learning curve of the model, which exhibited a similar pattern to that of the single step model.

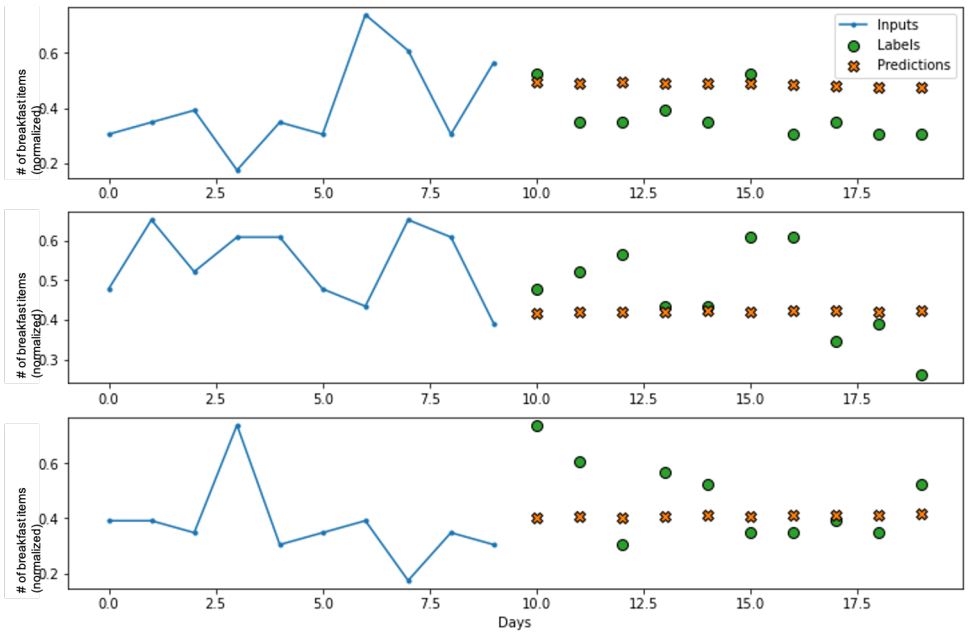


Figure 4.26: Multi step model linear results

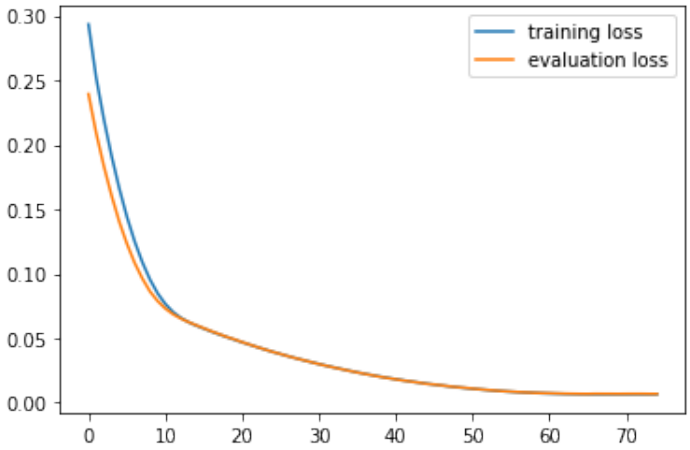


Figure 4.27: Multi step linear learning curve

The graphical results of the linear models' indicated that it might be a successful prediction method. But it was not the best neural network for the data set purely because of its simplicity. The average prediction it made ran very close to the mean average line. This resulted in a MAE for the multi step linear model to be one of the lowest values across models. But upon observation, these would not fluctuate enough with the volatility of any of the input features in a future scenario.

The single step linear model had an MAE of 0.097 and the multi step linear model had an MAE of 0.040, shown in Table 4.1. When these values were converted using Equation 4.2 the results were 2.2 and 0.92 respectively. This indicated an average absolute error of ±2 and ±1 units respectively.

4.3.3 Dense layers results and analysis

4.3.3.1 Single step model

Figure 4.28 below showed the learning curve of a single step dense model. The learning curve indicated that the validation data was not sufficient for this model [52]. This was most likely because the dense layer used one input to determine an output, making training and predicting fairly inaccurate. In Figure 4.29 and 4.30, the input signal and learning curve for a single step multi input step dense model is shown. This model utilised more

than one input to predict an outcome and thus exhibited better results and learning patterns.

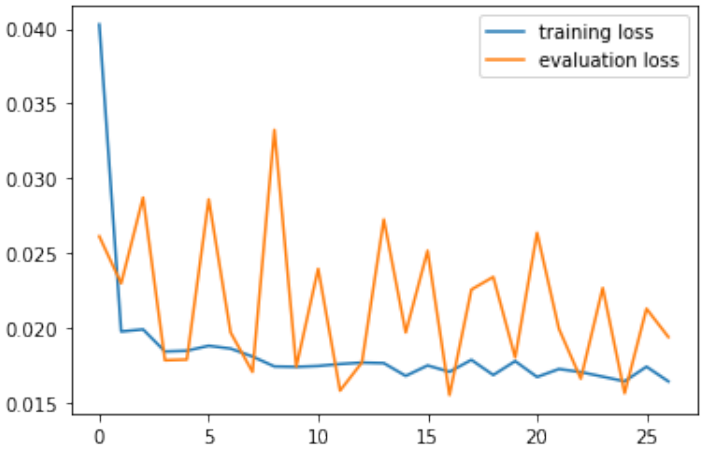


Figure 4.28: Single step dense layer learning curve

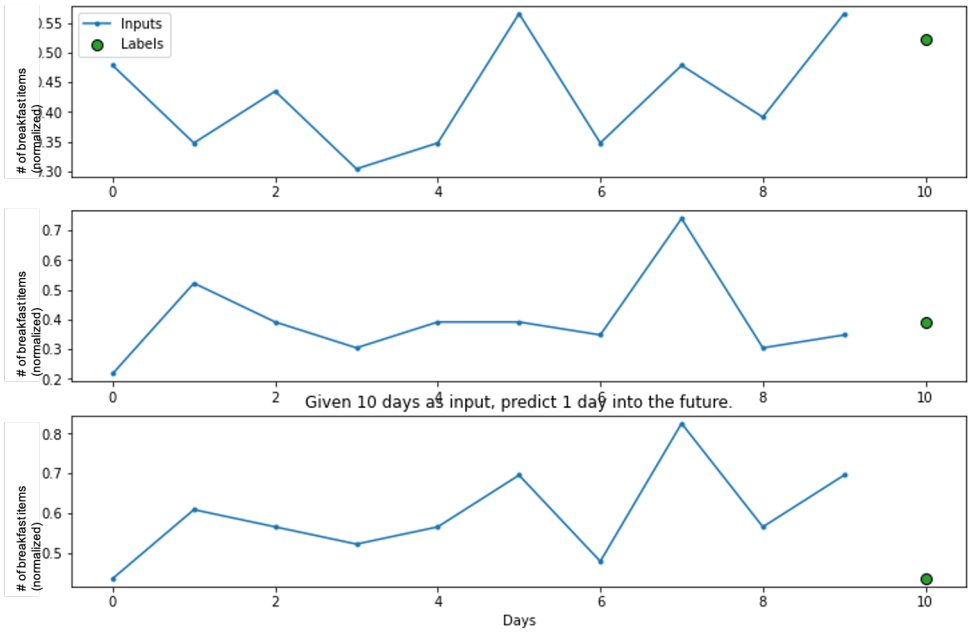


Figure 4.29: Single step multi input step dense layer input signal

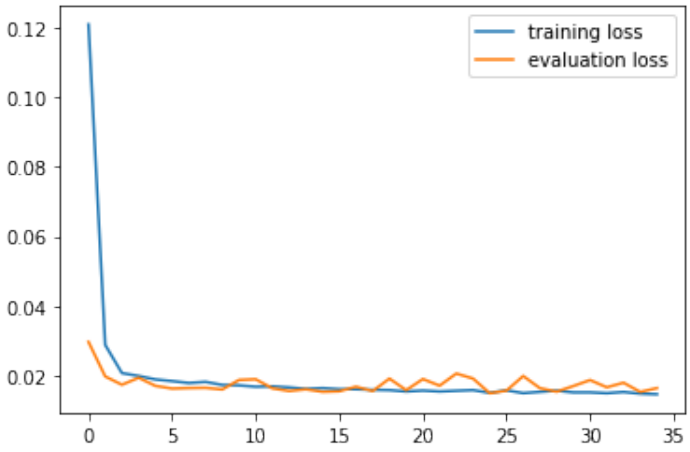


Figure 4.30: Single step multi input step dense layer learning curve

4.3.3.2 Multi step model

Similar to the linear model, the dense layer multi step model in Figure 4.31 was based on a single input time step. This was not ideal for the coffee franchise data set even though it proved to be more powerful than the linear model. Figure 4.32 exhibited a "good fit" learning curve as defined by Brownlee in his article on model diagnosis using learning curves [52].

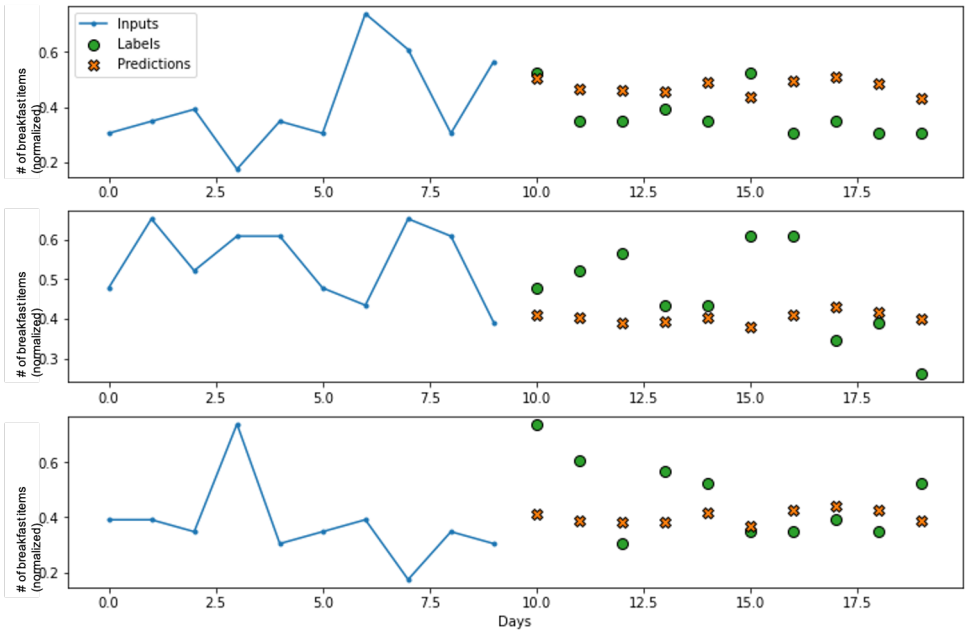


Figure 4.31: Multi step model dense layer results

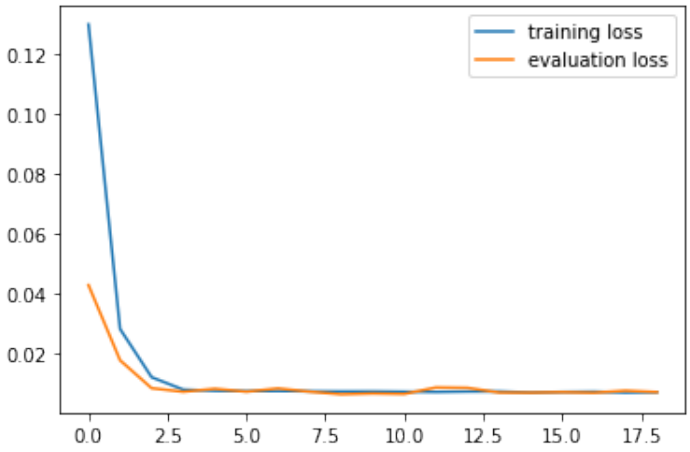


Figure 4.32: Multi step dense learning curve

The MAE values for the single step (multi input step) and the multi step dense layer model were converted from their normalised values shown in Table 4.1 to the MAE value corresponding to the original input data (using Equation 4.2). These values were 1.9 and 1.2 respectively. Despite the functionality of the multi input single step dense layer model, the pure multi step model still outperformed in terms of statistical results. This indicated that the model's algorithm was more reliant on a multi step output to utilise the extent of its predictive capabilities, rather than a multi step input configuration.

4.3.4 CNN results and analysis

4.3.4.1 Single step model

The single step CNN in Figure 4.33 was able to take in multiple time steps of data into the model. It is possible for a single step model to have the ability to take in multiple time steps of data, as the nomenclature of a single step model refers to its output ability not its data input intake.

The CNN model was superior in performance to the multi input step dense model. This was due to the CNN model's ability to run on an input of any length, as the layer was applied to a sliding window of inputs [40].

The learning curve in Figure 4.34 showed minor patterns suggestive of inadequate eval-

uation data [52]. This indicated that the model could improve given a larger data set that could apportion a greater amount of data to evaluation.

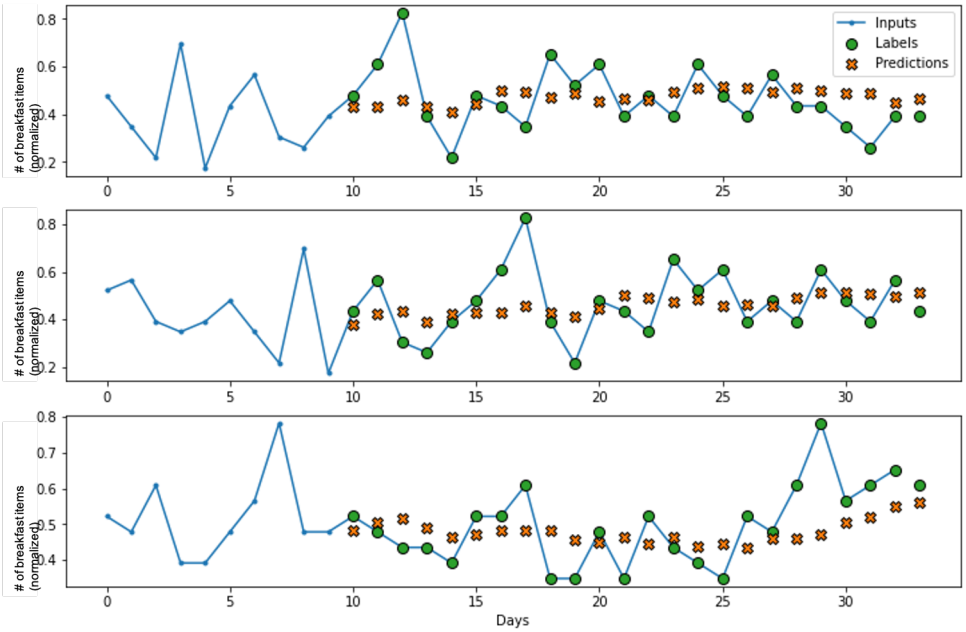


Figure 4.33: Single step CNN results

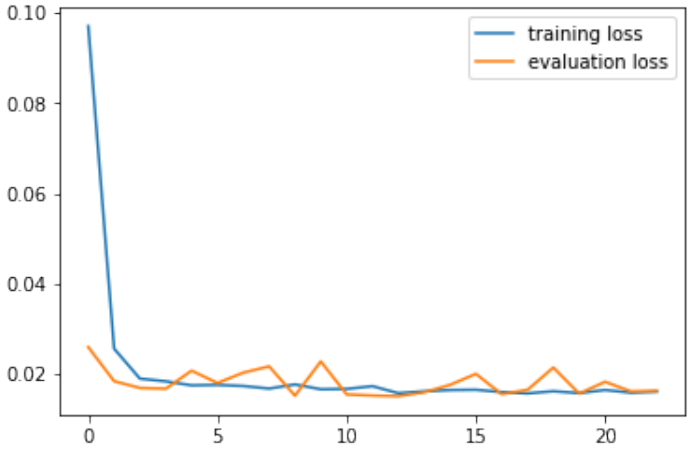


Figure 4.34: Single step CNN learning curve

4.3.4.2 Multi step model

The CNN model in Figure 4.35 was able to understand changes over time given its ability to input multiple time steps and use it predict multiple outputs at once. The model results indicated that it was superior to the dense multi step model. Figure 4.36 showed a "good fit" learning curve as described above [41].

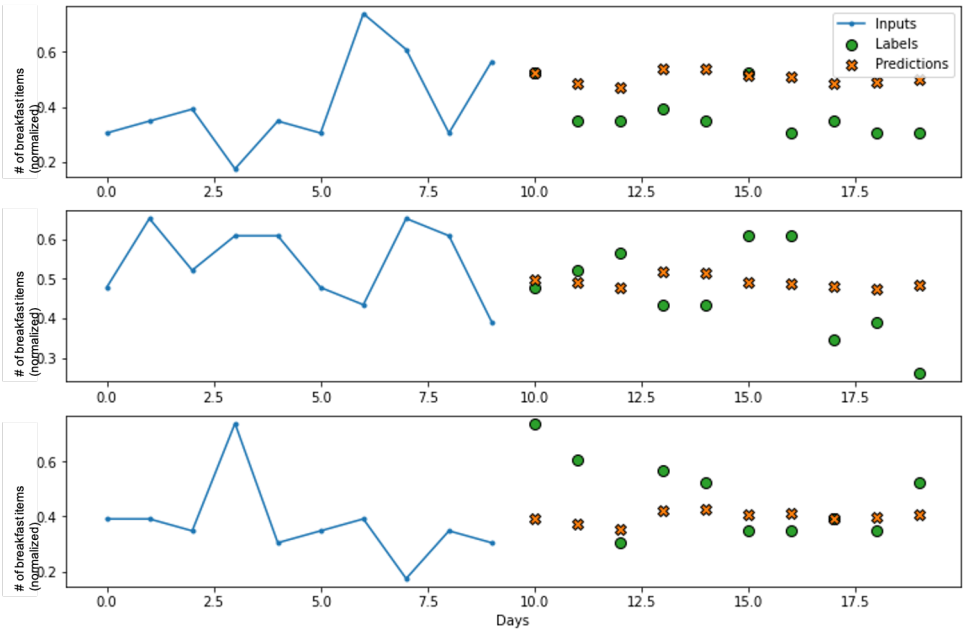


Figure 4.35: Multi step CNN results

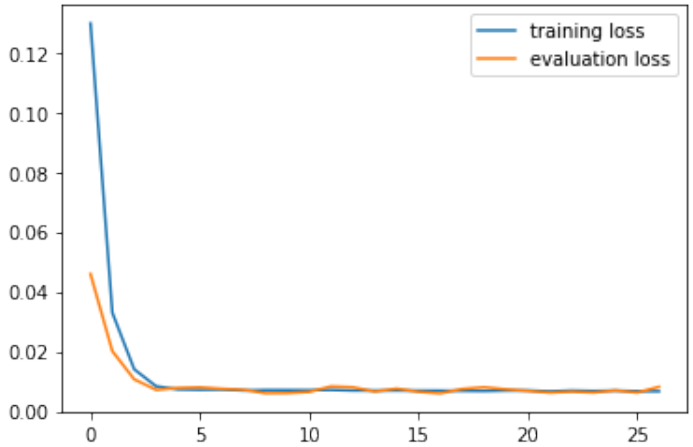


Figure 4.36: Multi step CNN learning curve

From Table 4.1 the MAE values revealed that apart from the linear model, the multi step CNN model had the best (lowest) error. This value was 0.050. When converted it represented just over one (1.2) of a Product 1 item as an average absolute error in predictions. The single step CNN shared an unsuccessful result similar to the ARIMA model.

4.3.5 RNN (LSTM) results and analysis

4.3.5.1 Single step model

Research indicated that RNNs, specifically LSTM have proven to be well suited for time series forecasting as the network was able to maintain an internal state from time step to time step [40]. The results of the single step LSTM model shown in Figure 4.37 along with the statistical results displayed in Table 4.1 indicated that the model did not perform as well as expected. This may be due to the simplistic single step model that does not allow the LSTM layer to properly display its abilities.

The learning curve for this model shown in Figure 4.38 however, followed the textbook definition of a good learning curve.

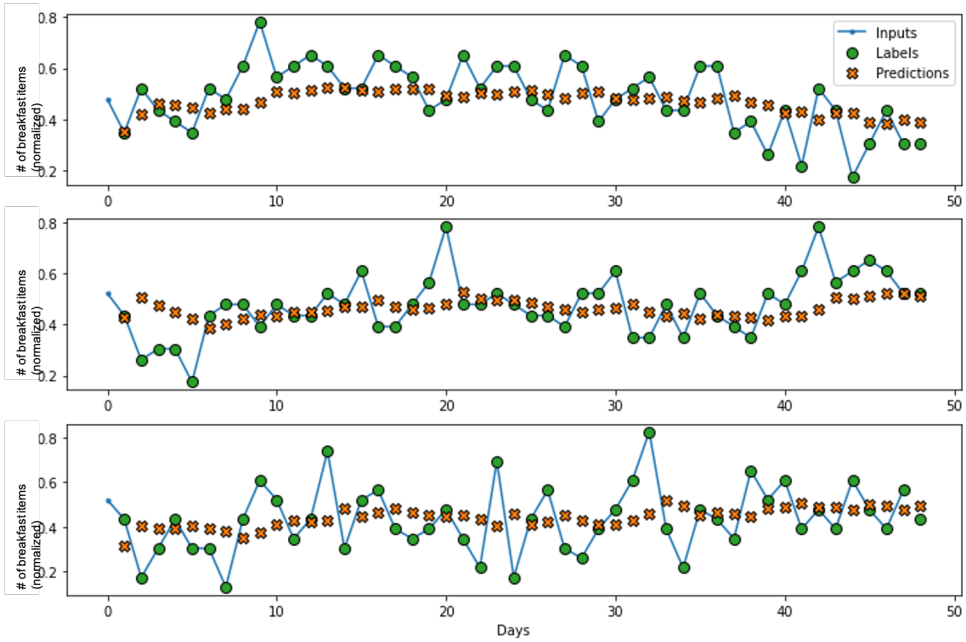


Figure 4.37: Single step LSTM results

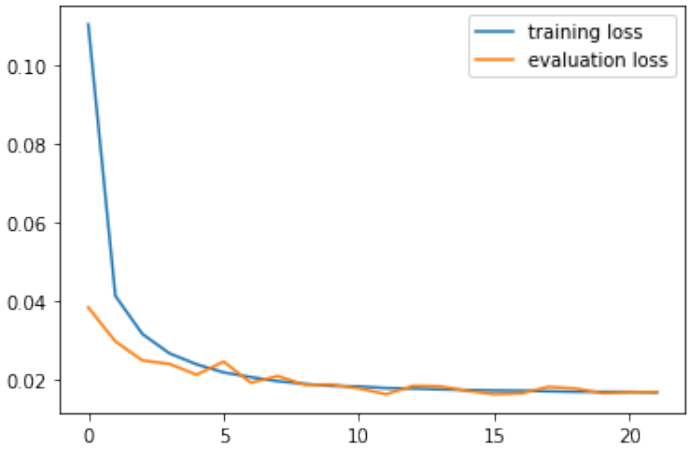


Figure 4.38: Single step LSTM learning curve

4.3.5.2 Multi step model

Single shot model

Figure 4.39 showed the results for a multi step model using the single shot methodology, i.e. all the predictions were made at once. The model results were an improvement from the single step model above, as observed through the learning curve in Figure 4.40.

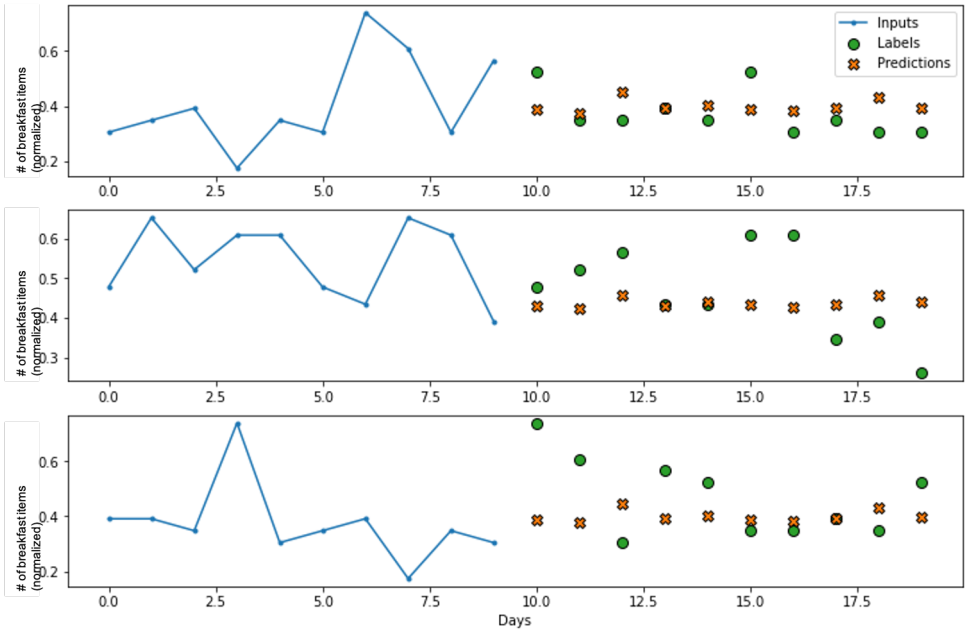


Figure 4.39: Multi step LSTM results

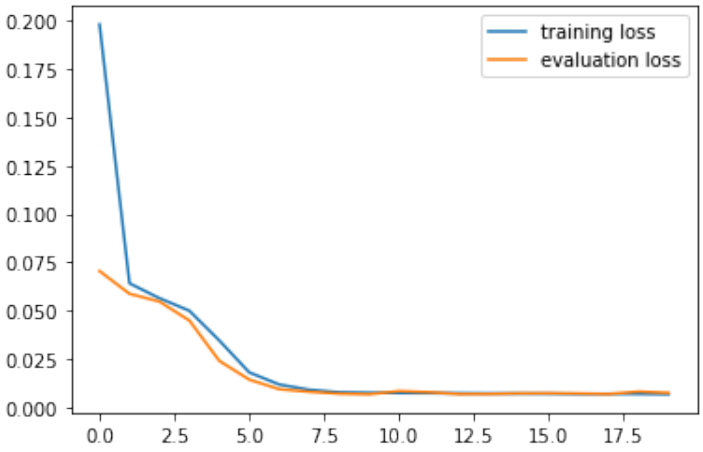


Figure 4.40: Multi step LSTM learning curve

Auto regressive model

Figure 4.41 and 4.42 below showed the results and the learning curve for the auto-regressive model respectively. The performance of this model compared to the single shot model above was very similar. Both have performed better than the single step version of the LSTM model.

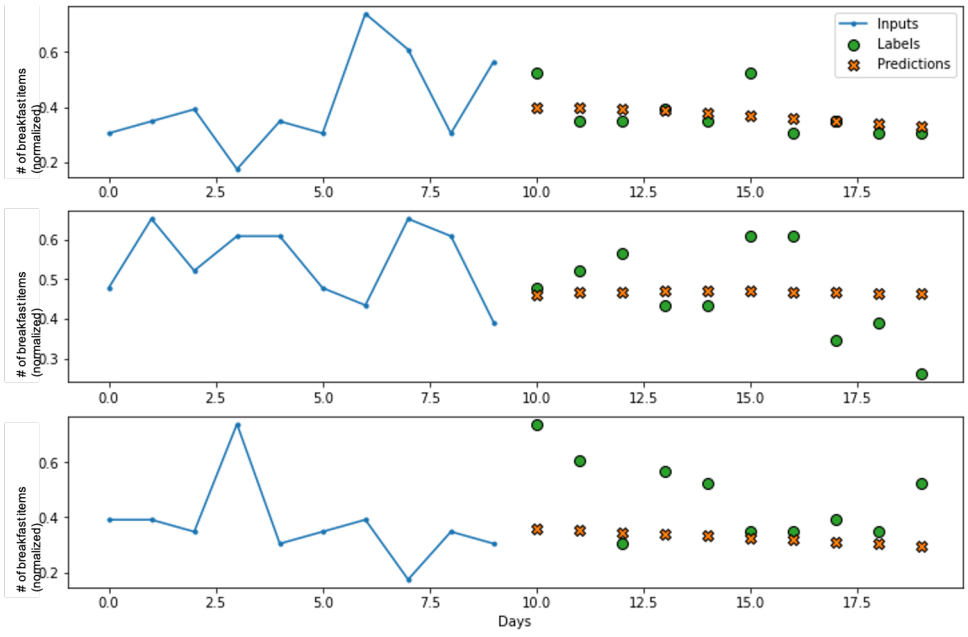


Figure 4.41: Multi step AR LSTM results

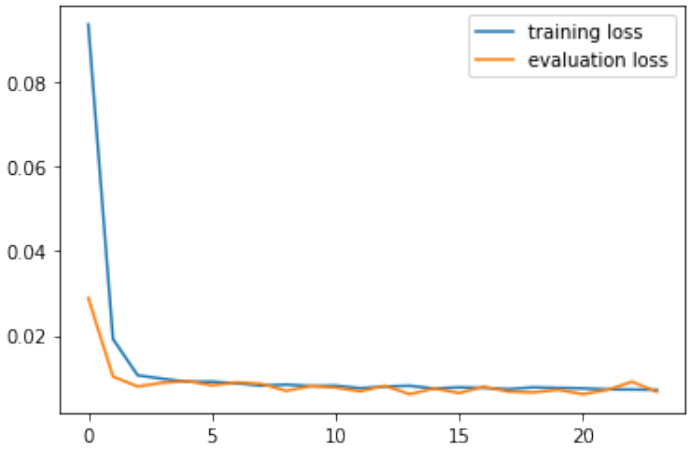


Figure 4.42: Multi step AR LSTM learning curve

The single step RNN (LSTM) model had an MAE similar to the single step linear model in Table 4.1. The MAE values for both the multi step models within RNN ranged from 0.054 to 0.060. Using the conversion from Equation 4.2 this resulted in an average absolute error in predictions of 1.2 to 1.4 units of Product 1. This indicated that the LSTM model had an average absolute error of ±1.3 units when predicting the demand of Product 1.

4.4 Further analysis on the top performing models

4.4.1 MAPE evaluation on the top performing models

The three best performing models were the multi step CNN, RNN (LSTM) and dense neural networks. To evaluate MAPE, these models were transformed back into their original form through an inverse scaling process built into the original normalisation function.

The MAPE values are calculated according to Equation 2.20. This equation indicated that each individual error is divided by the actual demand on a specific day, this summation is then averaged to result in an average absolute percentage error.

The multi step CNN model (Figure 4.43 below) resulted in a MAPE value of 29%, the RNN (LSTM) model (Figure 4.44) resulted in a MAPE value of 20% and the dense model

(Figure 4.45) resulted in a MAPE value of 28%. Through both graphical inspection and the statistical MAPE value, the RNN (LSTM) exhibited the best model behaviour.

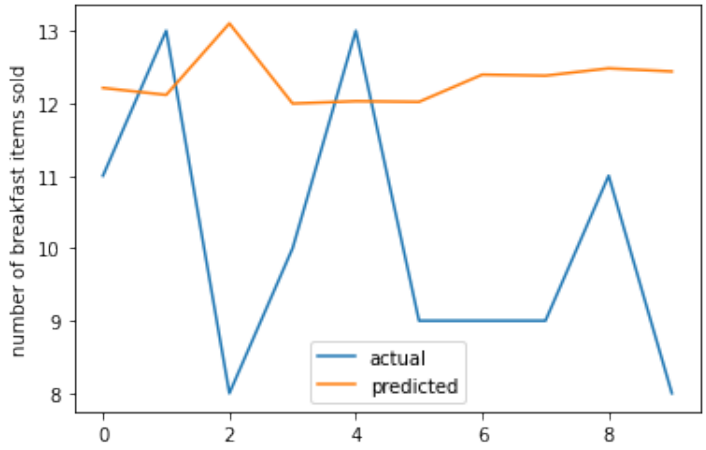


Figure 4.43: multi step CNN after inverse scaling - across 10 days of test data

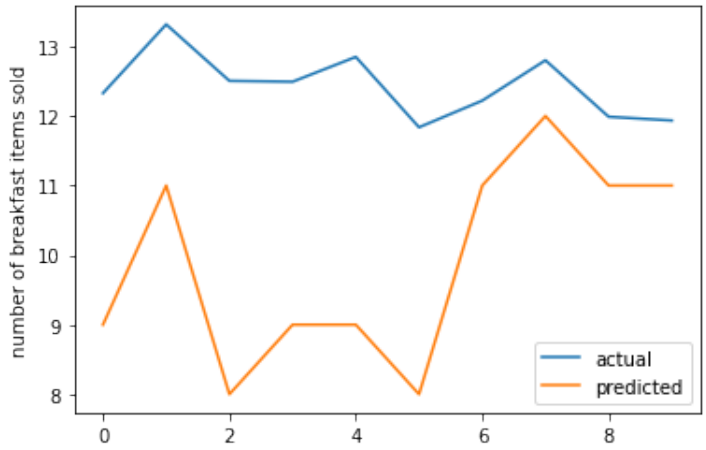


Figure 4.44: multi step RNN (LSTM) after inverse scaling - across 10 days of test data

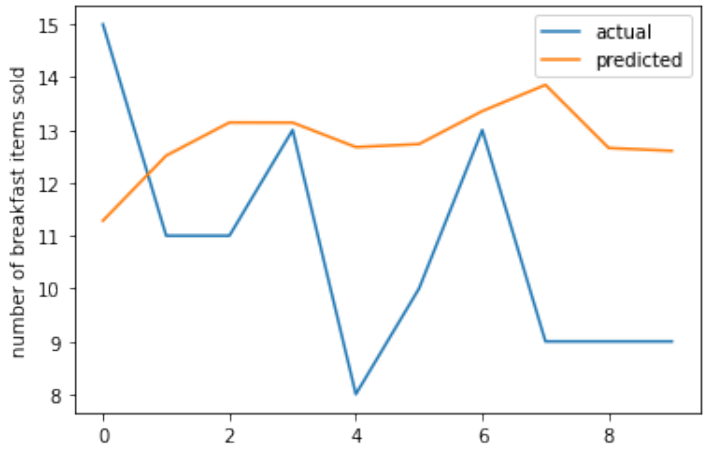


Figure 4.45: multi step dense layers after inverse scaling - across 10 days of test data

4.4.2 Product 2 results and analysis

Product 2 was tested on the top three models developed for Product 1. This was done to evaluate the reliability of the model on different product data.

Product 2 was tested with weather data to determine whether it improved results. Similar to the Product 1 data set, the addition of weather patterns did not assist in the models' ability to predict. Table 4.5 below shows the statistical measurements of the results from the models tested. These statistical measurements could not be compared to the Product 1 measurements as the input data was different.

Model:	MAE	MSE	RMSE
Multi step: Dense layers	0.048	0.0058	0.063
Multi step: CNN	0.041	0.0045	0.072
Multi step: RNN (LSTM)	0.049	0.0060	0.069

Table 4.5: Comparative statistical results for the tested methods using Product 2 data

An exercise similar to the one performed for Product 1 to convert the MAE to a value that could be interpreted was conducted. Table 4.6 below shows the summary of key metrics between the original and normalised data sets for Product 2.

Data type:	Count	Mean	Minimum value	Maximum value
Product 2	864	33	4	73
Product 2 normalised	864	0.42	0.0	1.0

Table 4.6: Key differences in Product 2 data after normalisation

Using the data in Table 4.6 this conversion resulted in following equation:

$$Product\ 2 : x = (x_{normalised} \times 69) + 4 \tag{4.3}$$

which indicated the method used to transform the mean, minimum value and maximum values back into its original form in Table 4.6.

The conversion stated in Equation 4.3 is substituted into the equation used to calculate MAE (Equation 2.18) similar to Product 1. The formula to convert the $MAE_{normalised}$ to the actual MAE value based on the original data is shown below:

$$Product\ 2 : MAE = 69 \times MAE_{normalised} \tag{4.4}$$

Using the equation above the converted MAE values for multi step dense layers, CNN and RNN (LSTM) are 7.3, 6.8 and 7.4 for Product 2 respectively.

In order to compare these values to Product 1, the MAE percentage was calculated. This percentage is different to MAPE as it is the MAE value divided by the mean/average demand [51]:

$$MAE\% = \frac{MAE}{Mean\,demand} \times 100 \tag{4.5}$$

Where the mean demand for Product 2 is captured in Table 4.6. This equation indicated that from an average of 33 items of Product 2 sold per day the predictions would roughly have an average absolute error of seven items. As an MAE percentage this equates to ±21%.

The MAE value for Product 1 converted to its original value was 1.2 for the three top models. Using Equation 4.5 to calculate the MAE percentage using the mean value for Product 1 captured in Table 4.2 resulted in an average percentage of ± 11%. Comparatively the MAE percentage for Product 1 proved to be better than that for Product 2 by a 10% difference.

Figure 4.46 to 4.48 show the graphical outputs for the models when predicting over a time window for Product 2.

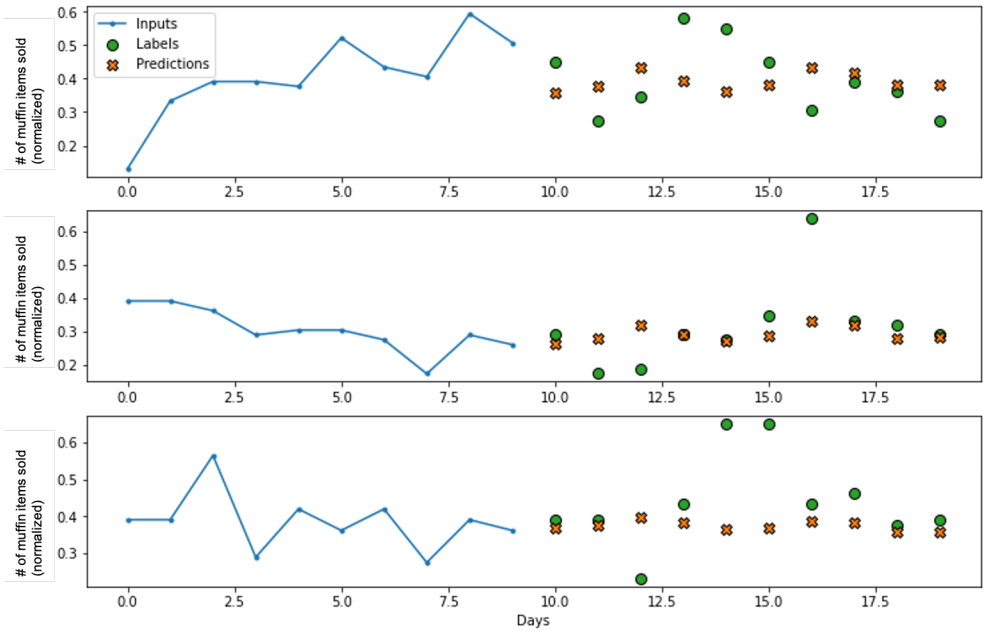


Figure 4.46: multi step CNN with Product 2

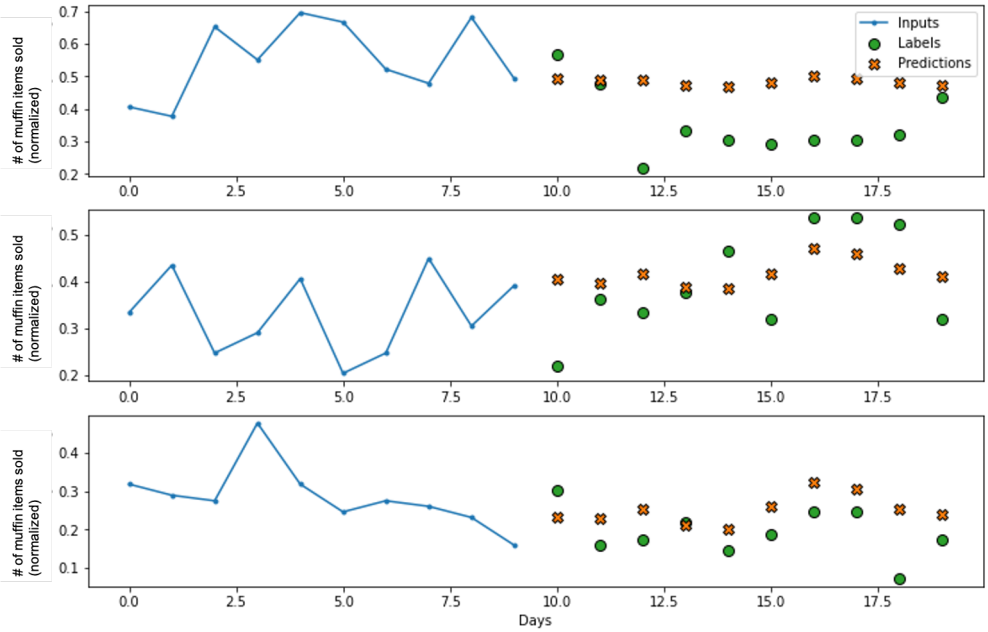


Figure 4.47: multi step RNN with Product 2

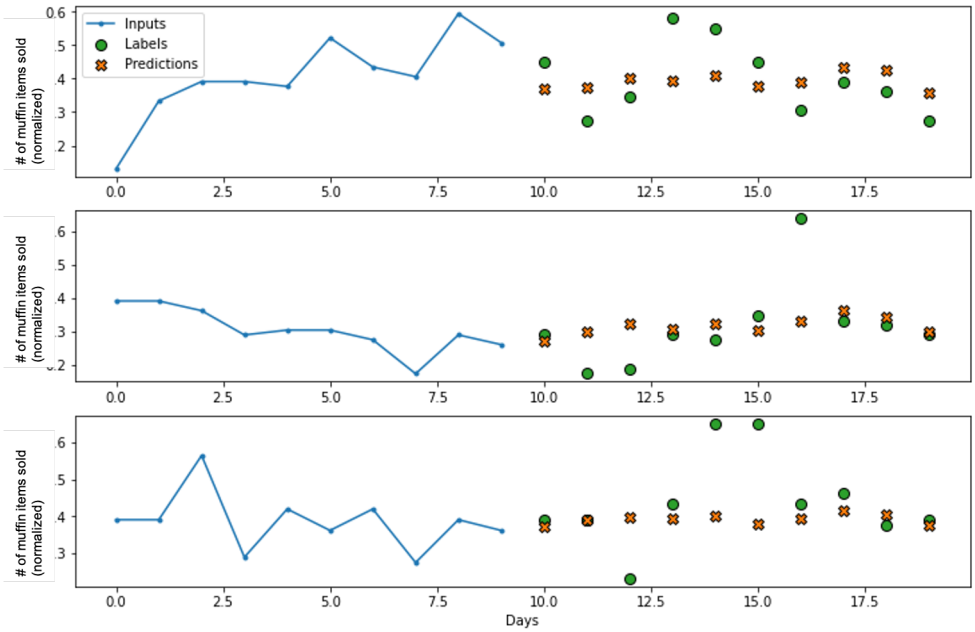


Figure 4.48: multi step dense layers with Product 2

Chapter 5

Discussion

5.1 Discussion on the models developed for Product 1

5.1.1 Overview

Model:	Converted MAE
ARIMA	2.1
SVR	1.6
Single step: linear	2.2
Single step: Dense layers	1.9
Single step: CNN	2.1
Single step: RNN (LSTM)	2.2
Multi step: Linear	0.92
Multi step: Dense layers	1.2
Multi step: CNN	1.2
Multi step: RNN (LSTM)	1.2
Multi step: AR RNN (LSTM)	1.4

Table 5.1: Converted MAE for Product 1 models analysed in Chapter 4

Table 5.1 summarises the converted MAE values calculated in Chapter 4. These values, the RMSE and MSE values in Table 4.1 and the theoretical background behind the models evaluated, was discussed in this section to further understand the results obtained.

The results obtained from the models tested with Product 2 data were also discussed in relation to the indicative reliability of the models. Further to this the potential business processes that could be altered in order to improve inventory management were discussed.

5.1.2 Discussion based on the model architecture

From the models developed for Product 1 there was a clear performance gap between the machine learning methods and the traditional statistical technique of ARIMA. Therefore, it could be deduced that for this type of data ARIMA has been superseded by the methodologies found in machine learning.

The (p,d,q) ARIMA model discussed in Section 2.4 accounted for a demand pattern that fell within the boundaries of a level of auto-regression, integration (differencing) and moving average residual errors. These were represented by the respective p, d and q values that formed inputs into the model. It was clear that extreme "noise" or volatility in the data could not be captured by this model and the three components on which it was based on.

In addition, the input data used for the ARIMA model was solely past time series data. This was unlike the neural networks where input feature data that assisted the model in understanding the demand behaviours was incorporated. For this reason the ARIMA methodology may not be rigorous enough to predict the future demand of volatile sales data.

The SVR model performed better than the single step neural networks and closely followed the performance of the multi step neural networks. Although it is widely known that neural networks have proven to be superior in forecasting compared to other machine

learning algorithms [53], SVR offered a model configuration that could be mathematically tracked throughout the process. A difficulty with the building of neural networks was the inability to track the calculation process from neuron to neuron within layers. This is because the neural network became too complex in its computation.

The single output step configuration of the neural network proved to be too simplistic for the data set utilised. For the neural network to compute complex data, a large enough input size needed to be computed for a set output. This was proven by the results in Table 5.1 that clearly indicated the superior performance of the multi output step neural network models.

It is also important to reiterate that even though the converted MAE performance of the multi step linear model seen in Table 5.1 indicated it had the lowest MAE value, the graphical outputs shown in Chapter 4 highlighted that the model was predicting values close to the average value. Therefore this outputted a lucrative MAE score that did not reflect the same success in its graphical plot. This was expected given the linear model was created as a baseline model due to the simplicity of the architecture.

The success of both the CNN and RNN (LSTM) models was due to the complexity of the architecture that allowed the models to handle volatile data well. Both methods have a different way of handling their input data; CNN through the use of smoothing layers and RNN (LSTM) through the use of feedback loops and Long Short Term Memory. Both these methods proved to be successful in their ability to predict demand for Product 1. The Auto-regressive RNN (LSTM) model was not as successful as the single shot methodology within the multi output step configuration. This indicated that the data patterns were more accurately predicted at once rather than through the use of auto-regression.

The MSE and RMSE values for the multi step RNN (LSTM) model were lower than that of the multi step CNN model. Opposite to this was the results of MAE, where the CNN model had the lowest value between the two models. This was an indication that the prediction patterns differed slightly between the models. This was discussed further in

the next section.

5.1.3 Discussion on statistical measures

The conclusions drawn for the top performing models based on Product 1 were different based on the evaluation of MAE, MSE and RMSE versus the evaluation of MAPE. It was not surprising that the best performing model determined by the MAE, MSE and RMSE metrics did not output the best MAPE. This was because of the specific nuances in each of the statistical calculations. The MAPE measure calculated the average of the individual percentage errors across the predicted range [51]. This meant that a high error when demand was low would significantly impact the MAPE value [51]. Compared to MAE where the absolute difference between actual and predicted was first calculated before averaging the values.

The RMSE and MSE were calculated for all the models tested. The square root of the MSE seen in Equation 2.19 is heavily impacted by a large error. In this way the RMSE was useful in understanding large outlying errors in predictive models. These errors may have gone undetected in other statistical calculations as the errors are averaged and weighted equally (especially in the calculation of MAE). Values from Table 4.1 therefore indicated that the multi step linear model had some of the larger errors across models. The opposite was true for the multi step RNN (LSTM) model which had the lowest RMSE value. This indicated that the model did not have any major outlying errors in its predictions.

The lowest MAPE value was calculated for the RNN (LSTM) model, this equated to a value of 20%. According to a paper written on electricity demand predictions [14] this value was deemed to be within an acceptable range for accurate forecasting. The paper stated a MAPE between 10% and 20% could produce accurate forecasting. And a MAPE of 10% or less would result in highly accurate forecasts.

5.2 Discussion on the models developed for Product 2

Product 2 was tested on the top performing models developed, in order to determine the reliability of the models using another perishable product data set. The tuning from Product 1 was kept constant, therefore it was expected that Product 2 would not perform as well as the tuning was tested on Product 1 results only.

Even though the results of Product 2 testing were not as good as Product 1, both the repetition in Product 1 testing and Product 2 testing indicated that the models were reliable in their performance results [54]. By definition if the results can be replicated several times the models can be accepted as reliable. In addition, both products exhibited similar MAE percentages that had a 10% difference between their respective values.

Both products did not show an improvement to prediction results when weather data was incorporated. One can assume given these results that weather dependent food items such as hot chocolate will do even better using all the periodicity curves and temperature. But given that the wastage of these items was rare, predicting the demand for hot chocolate for the coffee franchise would be unnecessary.

5.3 Discussion on the business operations

A research objective stated in Section 1.3 was to evaluate the coffee franchise holistically. This formed part of the objectives, because building a machine learning tool stemmed from the need to manage inventory effectively. Apart from comparing forecasting models that are able to estimate the demand numbers of a given perishable item depending on the time of year, external patterns etc. a number of other business initiatives can be implemented to better understand demand in the various branches. The following two paragraphs outlined two broad areas for business improvement:

From discussions with the coffee franchise owner, it was evident that the process for reporting low stock levels was manual. It depended on the employees willingness to

report feedback to the central operations team. If this process is streamlined into a central reporting requirement and set as a KPI for staff members, an accurate view of how quickly or slowly perishable food items leave the shelves can be understood. Therefore at a branch-by-branch level, the movement of products can be better understood and prepared for.

The location of the coffee shops in the central business areas around Johannesburg makes it possible to ring fence a loyal customer base. By doing this an investigation into what the typical coffee franchise customer expects or frequents the coffee franchise for, using ethnographic research can be carried out. A better understanding of customer needs will ultimately allow the company to improve their product offering.

These initiatives along with a machine learning tool that harnesses AI can create optimised and efficient business processes. The machine learning model has the ability to be scaled to forecast the perishable stock of the coffee franchise using the methodology outlined in this paper. Scaling the predictive models will result in an inventory management tool for the coffee franchise across branches.

Chapter 6

Conclusion and Recommendations

6.1 Conclusion

This research investigation set out to build an inventory management tool using machine learning for an SME. The SME/coffee franchise provided their POS data from which several predictive tools were built. These tools used traditional statistical techniques and varied forms of machine learning, to test whether a prediction tool could accurately predict the volatile demand patterns that occur in the coffee franchise's data set. It was found that the best performing model at its optimised state for the current data set had a 20% MAPE. This implied that at an average number of 11 Product 1 units sold per day the model could potentially have an average error of 20% on predictions. Further analysis would be required to understand the opportunity cost of over stocking versus under stocking, a feat described by the Newsvendor Problem in Chapter 2. The franchise would need to evaluate the marginal profit from additional units versus the wastage or loss that would occur when perishable items are not purchased within the required time period for the product.

The neural networks in machine learning proved to outperform both traditional and other types of machine learning methods (ARIMA and SVR). This indicated that the methodology in neural networking has potential to be very successful within the field of demand

forecasting in the future.

The results in Chapter 4 show that the ARIMA model, which did not utilise input features, did not perform as well as the machine learning methodologies. It could therefore be concluded from this investigation that machine learning performs better on data sets with high volatility than traditional techniques in demand forecasting.

The hypothesis that there would be a strong correlation between weather patterns and the demand proved to be incorrect for the two products tested. This can be expected given the products are not typically associated with certain types of weather as closely as an ice tea or hot chocolate are. However, as the investigation aimed to improve inventory management to reduce waste and meet demand, building neural networks that predicted long life items would not assist the business in terms of profitability, as these items and their ingredients have a longer shelf life.

While testing the hypothesis that weather and product demand demonstrated a correlation, was not successful. Periodicity through the use of sine and cosine curves improved the predictions of all the machine learning models. The periodicity enabled the models to build patterns around seasonal or cyclical trends, which proved to have more of a correlation to demand than weather for the perishable products tested.

Over and above the development of a machine learning method for demand predictions, the coffee shop was also evaluated holistically to determine whether the implementation of small initiatives could have a significant impact on the understanding and management of demand. Currently, based on the size of the franchise, initiatives around staff training and customer surveys could change the way the business manages their inventory. The way in which inventory is currently distributed could be revised to potentially offer a more frequent restock to avoid the need for future planning and inevitable wastage.

6.2 Recommendations and future work

The machine learning tool built as part of this investigation will need to be developed for all perishable products at all branches for the coffee franchise. The tool would need to have an easily accessible front end user interface, that at a weekly interval could be run given the POS data of the previous weeks. This output would be indicative to assist the central kitchen in their meal preparations.

Metrics such as cost of sale and profit margins should be factored into the model such that the lesser of the two evils can be determined and modelled with accordingly.

The input data for the model was approximately 860 data points as the data set spanned two and half years (on weekends the coffee franchise was not open). With more data the model's accuracy and ability to learn will improve. This was observed in the TensorFlow time series example which utilised 7 years of weather data [40]. Most neural networks take in tens of thousands of line items of input data. Therefore it can be extrapolated that the 20% predictive error on the current model will improve as more data is fed into the model. This hypothesis can only be confirmed when more data is accumulated from the franchise.

Another element to consider given recent events is the inclusion of a national disaster as a feature column. In this way the model is able to correlate a national disaster to a drop in demand and quickly change demand predictions when there is no national disaster present. This would be a long term consideration if the model is trained with 2016 to 2021 data.

Similar to the concept of a feature column for a national disaster, a column could also be created for other public events. Events such as public holidays, school holidays, election periods and ceremonial occasions. These events could have an effect on the typical customer that frequents the franchise which may display differences in demand over those specific periods.

In conclusion, this paper has demonstrated that machine learning, specifically neural networks, has the ability to predict volatile demand patterns better than any statistical technique that has been available in the past. The accuracy of demand predictions will continue to improve as the understanding and building of neural networks continues to grow. Implementing a similar small scale solution can assist small businesses in operational efficiency and profitability without the high investment cost of software tools that are geared towards larger corporations.

Bibliography

[1] Sebastian Saniuk, Sandra Grabowska, and Bozena Gajdzik. "Social expectations and market changes in the context of developing the industry 4.0 concept". In: *Sustainability (Switzerland)* 12.4 (2020).

[2] Shirley Coleman, Rainer Göb, Giuseppe Manco, Antonio Pievatolo, Xavier Tort-Martorell, and Marco Seabra Reis. "How Can SMEs Benefit from Big Data? Challenges and a Path Forward". In: *Quality and Reliability Engineering International* 32.6 (2016), pp. 2151–2164.

[3] DemandCaster. *Demand Management: A critical success factor in the food and beverages industry*. URL: https://www.demandcaster.com/demand-management-a-critical-success-factor-in-the-food-and-beverages-industry/ (visited on 04/14/2020).

[4] Saleem Khatri. *How POS systems are Shaping the Restaurant Industry*. 2020. URL: https://www.fsrmagazine.com/expert-takes/how-pos-systems-are-shaping-restaurant-industry (visited on 06/20/2020).

[5] Ann I. Ogbo, Onekanma Ifeyinwa Victoria, and Wilfred I. Ukpere. "The impact of effective inventory control management on organisational performance: A study of 7up bottling company Nile Mile Enugu, Nigeria". In: *Mediterranean Journal of Social Sciences* (2014), pp. 109–118.

[6] Tahir Suleman. "Improving the understanding of a well-known Johannesburg-based coffee business with data science". PhD thesis. Johannesburg: University of Witwatersrand, 2019.

[7] Hasaan Sohail Haji Idris. “Improving the understanding of a well-known Johannesburg-based coffee business with data science”. PhD thesis. Johannesburg: University of Witswatersrand, 2019.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[9] Julian Luengo, Salvador Garcia, and Francisco Herrera. “A study on the use of statistical tests for experimentation with neural networks: Analysis of parametric test conditions and non-parametric tests”. In: *Expert systems with applications* 36 (2008), pp. 7798–7808.

[10] Karin Kandananond. “A comparison of various forecasting methods for autocorrelated time series”. In: *International Journal of Engineering Business Management* 4.1 (2012), pp. 1–6.

[11] Rob Hyndman and George Athanasopoulos. *Forecasting: Principles and Practice*. 2nd. Melbourne: OTexts, 2018. URL: `https://otexts.com/fpp2/#` (visited on 06/04/2020).

[12] Takashi Tanizaki, Tomohiro Hoshino, Takeshi Shimmura, and Takeshi Takenaka. “Demand forecasting in restaurants using machine learning and statistical analysis”. In: *Procedia CIRP* 79.ii (2019), pp. 679–683.

[13] Elcio Tarallo, Getúlio K. Akabane, Camilo I. Shimabukuro, Jose Mello, and Douglas Amancio. “Machine learning in predicting demand for fast-moving consumer goods: An exploratory research”. In: *IFAC-PapersOnLine* 52.13 (2019), pp. 737–742.

[14] Hyojoo Son and Changwan Kim. “A deep learning approach to forecasting monthly demand for residential-sector electricity”. In: *Sustainability (Switzerland)* 12.8 (2020), p. 3103.

[15] F. L. Chen and T. Y. Ou. “Sales forecasting system based on Gray extreme learning machine with Taguchi method in retail industry”. In: *Expert Systems with Applications* 38.3 (2011), pp. 1336–1345.

[16] Afshin Oroojlooyjadid, Lawrence V. Snyder, and Martin Takáč. “Applying deep learning to the newsvendor problem”. In: *IISE Transactions* 52.4 (2020), pp. 444–463.

[17] Ajinkya Athlye and Angad Bashani. “Multivariate Demand Forecasting of Sales Data”. In: *International Journal for Research in Applied Science and Engineering Technology (IJRASET)* 6 (2018), pp. 198–211.

[18] Hossein Abbasimehr, Mostafa Shabani, and Mohsen Yousefi. “An optimized model using LSTM network for demand forecasting”. In: *Computers and Industrial Engineering* 143.March (2020), p. 106435.

[19] Jason Brownlee. *A Gentle Introduction to Autocorrelation and Partial Autocorrelation*. 2020. URL: https://machinelearningmastery.com/gentle-introduction-autocorrelation-partial-autocorrelation/ (visited on 06/10/2020).

[20] Jason Brownlee. *How to Create an ARIMA Model for Time Series Forecasting in Python*. 2017. URL: https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/ (visited on 10/01/2020).

[21] Jason Brownlee. *How to Use StandardScaler and MinMaxScaler Transforms in Python*. 2020. URL: https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/ (visited on 11/16/2020).

[22] Harris Drucker, Chris J.C. Surges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. “Support vector regression machines”. In: *Advances in Neural Information Processing Systems* 1 (1997), pp. 155–161.

[23] Tatiana V Tatarinova, Yuri Nikolsky Editors, Sebastian Raschka, Claus Führer Jan Erik Solem Olivier Verdier, John Hearty, Jared Huffman, and Ashwin Pajankar. *Python Machine Learning*. Birmingham: PackIt Publishing Ltd., 2015, p. 278.

[24] David R. Tobergte and Shirley Curtis. *Introduction to Machine Learning with Python*. Vol. 53. 9. 2016, pp. 1689–1699.

[25] Haiqin Yang, Kaizhu Huang, Irwin King, and Michael R. Lyu. “Localized support vector regression for time series prediction”. In: *Neurocomputing* 72.10-12 (2009), pp. 2659–2669.

[26] Oscar Claveria, Enric Monte, and Salvador Torra. “Regional Forecasting with Support Vector Regressions: The Case of Spain”. In: *SSRN Electronic Journal* (2018).

[27] Skikit-learn. *sklearn.svm.SVR*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html (visited on 10/01/2020).

[28] Clare Liu. *SVM Hyperparameter tuning using GridSearchCV*. URL: https://www.vebuso.com/2020/03/svm-hyperparameter-tuning-using-gridsearchcv/ (visited on 02/10/2021).

[29] Jannifer Zaino. *How Artificial Neural Networks Types Can Change Business*. 2018. URL: https://biztechmagazine.com/article/2018/09/how-artificial-neural-network-types-can-change-business-perfcon (visited on 04/14/2020).

[30] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalker. *Foundations of Machine Learning*. Ed. by Francis Bach. second. London: The MIT Press, 2018, p. 300.

[31] Harrison Kinsley. *Introduction to Neural Networks*. URL: https://pythonprogramming.net/introduction-deep-learning-neural-network-pytorch/ (visited on 05/06/2020).

[32] Larry Snyder. *Here's what you need to know about the Newsvendor Problem*. 2016. URL: https://medium.com/opex-analytics/extra-extra-heres-what-you-need-to-know-about-the-newsvendor-problem-be1d462822c (visited on 06/04/2020).

[33] Mohammed Rampurawala. *Classification with TensorFlow and Dense Neural Networks*. 2019. URL: https://heartbeat.fritz.ai/classification-with-tensorflow-and-dense-neural-networks-8299327a818a (visited on 08/15/2020).

[34] TensorFlow. *TensorFlow Playground*. 2020. URL: https://playground.tensorflow.org/ (visited on 05/08/2020).

[35] Assaad Moawad. *Dense Layers explained in a simple way*. 2019. URL: https://medium.com/datathings/dense-layers-explained-in-a-simple-way-62fe1db0ed75 (visited on 08/10/2020).

[36] Alex. *1-d Convolutional Neural Networks for Time series:Basic intuition*. 2020. URL: https://boostedml.com/2020/04/1-d-convolutional-neural-networks-for-time-series-basic-intuition.html (visited on 08/10/2020).

[37] Allison Koenecke. "Applying Deep Neural Networks to Financial Time Series Forecasting". PhD thesis. California: Institute for Computational and Mathematical Engineering, Standford, pp. 1–22.

[38] DeepAI. *What is the vanishing gradient problem?* 2020. URL: https://deepai.org/machine-learning-glossary-and-terms/vanishing-gradient-problem (visited on 12/02/2020).

[39] Víctor Manuel Landassuri-moreno and Carmen L Bustillo-hernández. "Single-Step-Ahead and Multi-Step-Ahead Prediction with Evolutionary Artificial Neural Networks". In: *LNCS 8258* (2013), pp. 65–72.

[40] TensorFlow. *Time Series Forecasting*. 2020. URL: https://www.tensorflow.org/tutorials/structured_data/time_series (visited on 05/08/2020).

[41] Jason Brownlee. *How to use Data Scaling Improve Deep Learning Model Stability and Performance*. 2019. URL: https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/ (visited on 08/15/2020).

[42] Nagesh Singh Chauhan. *Optimisation Algorithms in Neural Networks*. 2020. URL: https://www.kdnuggets.com/2020/12/optimization-algorithms-neural-networks.html (visited on 11/01/2020).

[43] Renu Khandelwal. *Overview of different Optimizers for neural networks*. 2019. URL: https://medium.datadriveninvestor.com/overview-of-different-optimizers-for-neural-networks-e0ed119440c3 (visited on 11/30/2020).

[44] MissingLink. *7 Types of Neural Network Activation Functions: How to Choose?* URL: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/ (visited on 05/05/2020).

[45] Dishashree Gupta. *Fundamentals of Deep Learning - Activation Functions and When to use them?* 2020. URL: https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/ (visited on 11/30/2020).

[46] Joydeep Bhattacharjee. *Common metrics for Time Series Analysis*. 2019. URL: https://medium.com/@joydeepubuntu/common-metrics-for-time-series-analysis-f3ca4b29fe42 (visited on 09/03/2020).

[47] Tutiempo. *South Africa - Weather Data*. 2020. URL: https://en.tutiempo.net/records/fapr (visited on 05/07/2020).

[48] Medium. *An introduction to Grid Search*. 2019. URL: https://medium.com/datadriveninvestor/an-introduction-to-grid-search-ff57adcc0998 (visited on 08/15/2020).

[49] Anita Yadav, C. K. Jha, and Aditi Sharan. "Optimizing LSTM for time series prediction in Indian stock market". In: *Procedia Computer Science* 167.2019 (2020), pp. 2091–2100.

[50] Michael Clements and David Hendry. "Evaluating forecast accuracy". In: *Forecasting Economic Time Series* (2009), pp. 52–78.

[51] Nicolas Vandeput. *Data Science for Supply Chain Forecast*. 2019. URL: http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=137392898&site=eds-live (visited on 05/06/2020).

[52] Jason Brownlee. *How to use Learning Curves to Diagnose Machine Learning Performance*. URL: https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/ (visited on 12/02/2020).

[53] Ishmael S. Msiza, Fulufhelo V. Nelwamondo, and Tshilidzi Marwala. “Artificial neural networks and support vector machines for water demand time series forecasting”. In: *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics* (2007), pp. 638–643.

[54] OER Services. *Chapter 7 Scale Reliability and Validity*. 2020. URL: https://courses.lumenlearning.com/suny-hccc-research-methods/chapter/chapter-7-scale-reliability-and-validity/ (visited on 01/05/2021).

[55] Nachiketa Hebbar. *Time series forecasting with ARIMA model in Python for Temperature Prediction*. 2020. URL: https://medium.com/swlh/temperature-forecasting-with-arima-model-in-python-427b2d3bcb53 (visited on 09/10/2020).

[56] Medium. *A practical guide to getting started with Machine Learning*. 2018. URL: https://medium.datadriveninvestor.com/a-practical-guide-to-getting-started-with-machine-learning-3a6fcc0f95aa (visited on 10/02/2020).

Appendix A

Coffee franchise SQL output summary

Table A.1 summarises the SQL database data structure from the coffee franchise. Data was split amongst numerous tables, in order to link tables together the colour coded variables were used as linkages between tables.

Table A.1: Summary of SQL output from coffee franchise data (colours indicate link variables across tables)

Table name:	Variables:														
Accesscreditsordered															
buyingitems															
comps	cid	compid	compname	sumid											
config															
dayparts	dpid	dpcode	dpname	dpstarthour	dpstartmin	dpendhour	dpendmin	sumid							
empcompsbreakdown	ecbrid	empno	compname	compamount	sumid										
groups	grpid	grpname	grpstatus												
hourlysales	hsid	hsstarttime	hsgrosssales	hsguests	hsguestsave	hschecks	hscheckave	hslaborhours	hsgrossperhour	sumid					
hourlysales_copy															
iscdetails	iscid	iscdate	isccosttype	iscopen	iscclose	iscpurchases	iscusage	iscdiscount	iscsales	strid					
itemrecipes	irid	grpid	inid	siid	irquantity	irdescription									
itemsbreakdown	ibrid	ibrcategoryid	ibrcategoryname	ibritenplu	ibritemname	ibrnumsold	ibrpricesold	ibramount	ibrsalespercent	ibrtimesold	ibrcheckno	ibrcost	sumid		
itemsbreakdownsummary	ibsid	ibsplu	ibsitemname	ibsqty	sumid										
itemsdistinct	itdname														
itemsdistinctaus	itdname														
log	logid	logowner	usrid	logdescription	logdate	logtime									
nonsalescategories	nscid	nsccatid	nscname	nscnumsold	nscpersales	sumid									
payments															
paymentsbreakdown	pbrid	pbrpaymenttype	pbramount	pbrchargetips	pbrautogratuity	pbrsales	pbrcash	sumid							
paymentslineitems	pliid	pmttype	pmttypeid	checknum	qty	amount	grat	tip	total	emp	sumid				
paymentssummary	pmsid	pmttype	pmttypeid	qty	amount	grat	tip	total	percentagetotal	sumid					
prepaid	prepaidid	custid	custname	custnum	prepaidloaded	prepaidused	prepaidbalance	contribution	sumid						
promos															
quickcount															
salesbycategoryexvat	sbcid	sbccategoryname	sbccategoryid	sbcamount	sbccomps	sbcpromos	sbcsurord	sbctaxes	sumid						
salescategories	scid	salescatid	salescatname	sales	sumid										
sellingitems	siid	grpid	siname	siplu	sicostprice	sistatus									
speedofservice	sosid	sostime	sosseconds	empid	sumid										
stockvalues	itemid	categoryname	itemname	svopencount	svopencost	svpurchases	svsales	svadjustments	svclosecount	svorderqty					
storebudgets	stbid	stbmonth	stbyear	stbamount	strid										
stores	strid	strname	strcode	straddress	strsuburb	strstate	strzip	strstatus	strlastimpdate	strlastsmsdate	strversion	strsms	...		
storesgroups	stgid	grpid	strid	stgstatus											
storeshld															
storesisc	strid	strname	strcode	...											
summary	sumid	sumdate	sumgrosssales	sumnettsales	sumbankingsales	sumcomps	sumrefunds	sumvat	sumvoids	sumheadcount	sumaveperheade..	sumaveperheadi...	sumcheckcount	sumavepercheck..	strid
summarydaily	sumhourid	sumdate	sumtime	sumgrosssales	sumnetsales	sumbankingsales	strid								
summaryextras	seid	addcharges	ordercharges	promos	refunds	sumid									
taxes	tid	taxid	taxname	sumid											
turnovervalues	turnovervaluesid	tvdate	foodcost	tvopenstockvalue	tvpurchases	tvcloseststockvale	tvsales	tvdiscount	strid						
useraccess															
users															
vouchers															

Appendix B

Python code: ARIMA

The following code was developed using the methodologies outlined in [20] and [55].

```
# -*- coding: utf-8 -*-
"""print ARIMA.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1WB8xTvIV8C7zK-eWPhGx2uSZm5Rbidj6

# data preprocessing
"""

import tensorflow as tf
import pandas as pd
import datetime
import os
import IPython
import IPython.display
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
```

```
import seaborn as sns

#graph config below
mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['axes.grid'] = False

!pip install pmdarima

from pmdarima import auto_arima
from statsmodels.tsa.stattools import adfuller
from sklearn.metrics import mean_squared_error
from math import sqrt
from pandas.plotting import autocorrelation_plot

from statsmodels.tsa.arima.model import ARIMA

from pandas import DataFrame

from google.colab import files
uploaded = files.upload()

def parse(x):
  return datetime.datetime.strptime(x, '%Y/%m/%d')
df = pd.read_csv('strid_237_Jan_Dec_18_breakfast2503.csv',sep=',',
    parse_dates=["sumdate"], date_parser=parse)
df.head()

df.rename(columns={'sumdate':'Date'},inplace=True)
df.rename(columns={'Date ':'Dateinmonth'},inplace=True)
df.rename(columns={'Month ':'Month'},inplace=True)
df.rename(columns={'Year ':'Year'},inplace=True)
for col in df.columns:
  print(col)

df = df.groupby(by='Date').agg({
```

```
                                'ibrnumsold': 'sum',}).reset_index()
df.head()

"""Trying to Normalise"""

df.set_index('Date', inplace=True)
dfscaled = df
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
dfscaled= scaler.fit_transform(dfscaled)

dfscaled = pd.DataFrame({
                          'ibrnumsold': dfscaled[:,0],
                         })

dfscaled1 = dfscaled.melt(var_name='Column', value_name='Normalized
    ')
plt.figure(figsize=(12, 6))
ax = sns.violinplot(x='Column', y='Normalized', data=dfscaled1)
_ = ax.set_xticklabels(df.keys(), rotation=90)

df = dfscaled

"""using method from: https://medium.com/swlh/temperature-
    forecasting-with-arima-model-in-python-427b2d3bcb53"""

df['ibrnumsold'].plot(figsize=(12,5))

def adf_test(dataset):
  dftest = adfuller(dataset, autolag = 'AIC')
  print("1. ADF : ",dftest[0])
  print("2. P-Value : ", dftest[1])
  print("3. Num Of Lags : ", dftest[2])
  print("4. Num Of Observations Used For ADF Regression and
    Critical Values Calculation :", dftest[3])
```

```
  print("5. Critical Values :")
  for key, val in dftest[4].items():
      print("\t",key, ": ", val)

adf_test(df['ibrnumsold'])

import warnings
warnings.filterwarnings("ignore")

stepwise_fit = auto_arima(df['ibrnumsold'],
                          suppress_warnings=True,
                          approximation=False,
                          stepwise=False
                          )

stepwise_fit.summary()

"""https://machinelearningmastery.com/arima-for-time-series-
   forecasting-with-python/

"""

from statsmodels.graphics import tsaplots

df.head()

df = df.squeeze()

autocorrelation_plot(df)
plt.show()

tsaplots.plot_pacf(df)
plt.show()

tsaplots.plot_acf(df)
```

```
121 plt.show()
122
123 model = ARIMA(df, order=(4,1,1)) #114
124 #model_fit = model.fit(disp=0)
125 model_fit = model.fit()
126 print(model_fit.summary())
127 # plot residual errors
128 residuals = DataFrame(model_fit.resid)
129 residuals.plot()
130 plt.show()
131 residuals.plot(kind='kde')
132 plt.show()
133 print(residuals.describe())
134
135 X = df.values
136 size = int(len(X) * 0.66)
137 train, test = X[0:size], X[size:len(X)]
138 history = [x for x in train]
139 predictions = list()
140 for t in range(len(test)):
141   model = ARIMA(history, order=(4,1,1))#114
142   model_fit = model.fit()
143   output = model_fit.forecast()
144   yhat = output[0]
145   predictions.append(yhat)
146   obs = test[t]
147   history.append(obs)
148   print('predicted=%f, expected=%f' % (yhat, obs))
149 error = mean_squared_error(test, predictions)
150 print('Test MSE: %.3f' % error)
151 # plot
152 plt.plot(test, label = 'Actual')
153 plt.plot(predictions, color='red', label = 'Predicted')
154 plt.show()
155
```

```
156 from sklearn.metrics import mean_absolute_error
157 from sklearn.metrics import mean_squared_error
158
159 error = mean_absolute_error(test, predictions)
160 print('Test MAE: %.3f' % error)
161
162 error = mean_squared_error(test, predictions, squared=False)
163 print('Test RMSE: %.3f' % error)
```

Appendix C

Python code: SVR

The following code was developed using the methodologies outlined in [56] and [48].

```
# -*- coding: utf-8 -*-
"""print SVR.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1aY3JGLnXtc7f_jqK-bKrRxP25gyaEC6O

# data preprocessing
"""

import tensorflow as tf
import pandas as pd
import datetime
import os
import IPython
import IPython.display
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
```

```
import seaborn as sns


#graph config below
mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['axes.grid'] = False

from google.colab import files
uploaded = files.upload()

def parse(x):
  return datetime.datetime.strptime(x, '%Y/%m/%d')
df = pd.read_csv('strid_237_Jan_Dec_18_breakfast2503.csv',sep=',',
    parse_dates=["sumdate"], date_parser=parse)

df.rename(columns={'sumdate':'Date'},inplace=True)
df.rename(columns={'Date ':'Dateinmonth'},inplace=True)
df.rename(columns={'Month ':'Month'},inplace=True)
df.rename(columns={'Year ':'Year'},inplace=True)
for col in df.columns:
  print(col)

df = df.groupby(by='Date').agg({
                                'Dateinmonth': 'mean',
                                'Month': 'mean',
                                'Day of the week': 'mean',
                                'mean wind speed 10m above ground (m/
    s)':'mean',
                                'Avg day temp (deg C)':'mean',
                                'Rain':'mean',
                                'Cloud cover':'mean',
                                'ibrnumsold': 'sum',}).reset_index()

df.drop([
         'mean wind speed 10m above ground (m/s)',
```

```
        'Cloud cover'
        ], inplace=True, axis=1)

def getDayOfYear(dateTimeVect):
  dayOfYearVect =[]
  for datetime in dateTimeVect:
    dayOfYear = 0
    for i in range(1,datetime.month):
      dayOfYear += daysInMonth[i]
    dayOfYear += datetime.day
    dayOfYearVect.append(int(dayOfYear))
  return np.array(dayOfYearVect)

timestamp_s = df.Date
daysInMonth = [31, 29, 31, 30,31, 30, 31, 31, 30, 31, 30, 31]
print(getDayOfYear(df.Date))

month = 12
daysinyear = 366 #52.14*5 #52.14 weeks x 5 days a week
daysinmonth = 30.5


df['Year sin'] = np.sin(getDayOfYear(df.Date).astype(int) * (2 * np
    .pi / daysinyear))
df['Year cos'] = np.cos(getDayOfYear(df.Date).astype(int) * (2 * np
    .pi / daysinyear))

df['Month sin'] = np.sin(df.Dateinmonth.astype(int) * (2 * np.pi /
    daysinmonth))
df['Month cos'] = np.cos(df.Dateinmonth.astype(int) * (2 * np.pi /
    daysinmonth))

df['MonthYear sin'] = np.sin(df.Month.astype(int) * (2 * np.pi /
    month))
```

```
df['MonthYear cos'] = np.cos(df.Month.astype(int) * (2 * np.pi /
    month))

df.drop([
          'Dateinmonth',
          'Month',
          ], inplace=True, axis=1)

"""Trying to normalise"""

df.set_index('Date', inplace=True)

dfscaled = df
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
dfscaled= scaler.fit_transform(dfscaled)

df

dfscaled = pd.DataFrame({'Day of the week': dfscaled[:,0],
                         'Avg day temp (deg C)': dfscaled[:,1],
                         'Rain': dfscaled[:,2],
                         'ibrnumsold': dfscaled[:,3],
                         'Year sin': dfscaled[:,4],
                         'Year cos': dfscaled[:,5],
                         'Month sin': dfscaled[:,6],
                         'Month cos': dfscaled[:,7],
                         'MonthYear sin': dfscaled[:,8],
                         'MonthYear cos': dfscaled[:,9]
                        })

dfscaled

df = dfscaled

```

```
117 df.head()
118
119 df.reset_index(drop=True, inplace=True)
120 df.head()
121
122 """#grid search"""
123
124 df.shape
125
126 df= df.head(200)
127
128 from sklearn.model_selection import train_test_split
129
130 df['index1'] = df.index
131
132 df.head()
133
134 df_days = df.loc[:,['index1',
135                     'Rain',
136                     'Day of the week',
137                     'Avg day temp (deg C)',
138                     'Year sin','Year cos','Month sin','Month cos','
    MonthYear sin','MonthYear cos']]
139 df_ibrnumsold = df.loc[:,'ibrnumsold']
140
141 df_days
142
143 X= df_days
144 y= df_ibrnumsold.values.reshape(-1, 1)
145
146 X.shape
147
148 y.shape
149
150 y=y.ravel()
```

```
151
152 X_train, X_test, y_train, y_test = train_test_split(
153     X, y, shuffle=False)
154
155 import math
156 import pandas
157 from sklearn.preprocessing import MinMaxScaler
158 from sklearn.svm import SVR
159 from sklearn.model_selection import GridSearchCV, cross_validate
160 from sklearn.utils import shuffle
161
162 """for rbf"""
163
164 gsc = GridSearchCV(
165         estimator=SVR(kernel='rbf'),
166         param_grid={
167             'C': [1, 10,0.1,0.01],
168             'gamma': [1,10,100,150]
169         },
170         cv=5, scoring='neg_mean_squared_error', verbose=0, n_jobs
    =-1)
171
172 grid_result = gsc.fit(X, y)
173 best_params = grid_result.best_params_
174 best_svr = SVR(kernel='rbf', C=best_params["C"],
175                gamma=best_params["gamma"],
176                    coef0=0.1, shrinking=True,
177                    tol=0.001, cache_size=200, verbose=False,
    max_iter=-1)
178 scoring = {
179                'abs_error': 'neg_mean_absolute_error',
180                'squared_error': 'neg_mean_squared_error'}
181
182 scores = cross_validate(best_svr, X, y, cv=10, scoring=scoring,
183                         return_train_score=True)
```

```
184 print("MAE :", abs(scores['test_abs_error'].mean()), "| RMSE :",
185       math.sqrt(abs(scores['test_squared_error'].mean())))
186
187 best_params
188
189 """for poly"""
190
191 gsc = GridSearchCV(
192         estimator=SVR(kernel='poly'),
193         param_grid={
194             'C': [150,200,250,300],
195             'degree':[4,5]
196         },
197         cv=5, scoring='neg_mean_squared_error', verbose=True,
    n_jobs=-1)
198
199 grid_result = gsc.fit(X, y)
200 best_params = grid_result.best_params_
201 best_svr = SVR(kernel='poly', C=best_params["C"],
202                degree=best_params["degree"],
203                    coef0=0.1, shrinking=True,
204                    tol=0.001, cache_size=200, verbose=True,
    max_iter=-1)
205 scoring = {
206                'abs_error': 'neg_mean_absolute_error',
207                'squared_error': 'neg_mean_squared_error'}
208
209 scores = cross_validate(best_svr, X, y, cv=10, scoring=scoring,
    return_train_score=True)
210 print("MAE :", abs(scores['test_abs_error'].mean()), "| RMSE :",
    math.sqrt(abs(scores['test_squared_error'].mean())))
211
212 best_params
213
214 """for linear"""
```

```
215
216 gsc = GridSearchCV(
217         estimator=SVR(kernel='linear'),
218         param_grid={
219             'C': [1,10],
220             'gamma': [10,1]
221         },
222         cv=5, scoring='neg_mean_squared_error', verbose=2, n_jobs
    =-1)
223
224 grid_result = gsc.fit(X, y)
225 best_params = grid_result.best_params_
226 best_svr = SVR(kernel='linear',
227                C=best_params["C"],
228                gamma=best_params["gamma"],
229                    coef0=0.1, shrinking=True,
230                    tol=0.001, cache_size=200, verbose=False,
    max_iter=-1)
231 scoring = {
232                'abs_error': 'neg_mean_absolute_error',
233                'squared_error': 'neg_mean_squared_error'}
234
235 scores = cross_validate(best_svr, X, y, cv=10, scoring=scoring,
    return_train_score=True)
236 print("MAE :", abs(scores['test_abs_error'].mean()), "| RMSE :",
    math.sqrt(abs(scores['test_squared_error'].mean())))
237
238 best_params
239
240 """#method 3
241
242 https://www.youtube.com/watch?v=C64BIMx7Slw
243 """
244
245 import numpy as np
```

```
246 from sklearn.svm import SVR
247 import matplotlib.pyplot as plt
248 plt.style.use('fivethirtyeight')
249 import pandas as pd
250
251 df.shape
252
253 df['index1'] = df.index
254
255 df
256
257 df = df.head(60)
258
259 #get and print the last bit of data
260 actual_ibrnumsold=df.tail(10)
261 actual_ibrnumsold
262
263 #prep data to train
264 df = df.head(len(df)-10)
265
266 #create list to store indep and dep data
267 days_list = list()
268 ibrnumsold_list = list()
269
270 #get the dates and ibrnumsold
271 df_days = df.loc[:,['index1',
272                     'Rain',
273                     'Day of the week',
274                     'Avg day temp (deg C)',
275                     'Year sin','Year cos','Month sin','Month cos','
    MonthYear sin','MonthYear cos']]
276 df_ibrnumsold = df.loc[:,'ibrnumsold']
277
278 df_index = df.loc[:,'index1']
279
```

```
days_list = df_days.values

#create dep dataset
for ibrnumsold in df_ibrnumsold:
  ibrnumsold_list.append(float(ibrnumsold))

#print the lists
print(days_list)
print(ibrnumsold_list)

#create SVM

#create and train a svr using linear kernel
lin_svr = SVR(kernel='linear', C=10, gamma=10)
lin_svr.fit(days_list, ibrnumsold_list)

#create and train a svr using poly kernel
poly_svr = SVR(kernel='poly', C=250 , degree=3)
poly_svr.fit(days_list, ibrnumsold_list)

#create and train a svr using rbf kernel
rbf_svr = SVR(kernel='rbf', C=0.2, gamma=200)
rbf_svr.fit(days_list, ibrnumsold_list)

#plot
plt.figure(figsize=(16,8))
plt.scatter(df_index, ibrnumsold_list, color='red', label='data')
plt.plot(df_index, rbf_svr.predict(days_list), color='green', label
   ='RBF model')
plt.plot(df_index, poly_svr.predict(days_list), color='orange',
   label='polynomial model')
plt.plot(df_index, lin_svr.predict(days_list), color='blue', label=
   'linear model')
plt.legend()
plt.show
```

```
312
313 actual_ibrnumsold
314
315 days_test = actual_ibrnumsold.loc[:,['index1',
316                     'Rain',
317                     'Day of the week',
318                     'Avg day temp (deg C)',
319                     'Year sin','Year cos','Month sin','Month cos','
    MonthYear sin','MonthYear cos']]
320
321 days_test = days_test.values
322
323 from sklearn.metrics import mean_squared_error
324
325 from sklearn.metrics import mean_absolute_error
326
327 ibrnumsold_test = actual_ibrnumsold.loc[:,'ibrnumsold']
328
329 ibrnumsold_test = ibrnumsold_test.ravel()
330 ibrnumsold_test = ibrnumsold_test.reshape(-1,1)
331
332 error = mean_absolute_error(ibrnumsold_test,rbf_svr.predict(
    days_test))
333 print('Test MAE rbf: %.3f' % error)
334
335 error = mean_absolute_error(ibrnumsold_test,lin_svr.predict(
    days_test))
336 print('Test MAE lin: %.3f' % error)
337
338 error = mean_absolute_error(ibrnumsold_test,poly_svr.predict(
    days_test))
339 print('Test MAE poly: %.3f' % error)
340
341 error = mean_squared_error(ibrnumsold_test,rbf_svr.predict(
    days_test))
```

```
342 print('Test MSE rbf: %.3f' % error)
343
344 error = mean_squared_error(ibrnumsold_test,lin_svr.predict(
    days_test))
345 print('Test MSE lin: %.3f' % error)
346
347 error = mean_squared_error(ibrnumsold_test,poly_svr.predict(
    days_test))
348 print('Test MSE poly: %.3f' % error)
349
350 error = mean_squared_error(ibrnumsold_test,rbf_svr.predict(
    days_test),squared=False)
351 print('Test RMSE rbf: %.3f' % error)
352
353 error = mean_squared_error(ibrnumsold_test,lin_svr.predict(
    days_test),squared=False)
354 print('Test RMSE lin: %.3f' % error)
355
356 error = mean_squared_error(ibrnumsold_test,poly_svr.predict(
    days_test),squared=False)
357 print('Test RMSE poly: %.3f' % error)
```

Appendix D

Python code: Neural networks

The following code was developed using the methodologies outlined in [40].

```
# -*- coding: utf-8 -*-
"""to print data preprocessing.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1
    czJbh074s8qlvORC7mhggDxPUDENQoVd

# Data processing
"""

import tensorflow as tf
import pandas as pd
import datetime
import os
import IPython
import IPython.display
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
```

```
import seaborn as sns

#graph config below
mpl.rcParams['figure.figsize'] = (12, 12)
mpl.rcParams['axes.grid'] = False

from google.colab import files
uploaded = files.upload()

def parse(x):
  return datetime.datetime.strptime(x, '%Y/%m/%d')
df = pd.read_csv('strid_237_Jan_Dec_18_breakfast2503.csv',sep=',',
    parse_dates=["sumdate"],
                 date_parser=parse,
                 )
df.head()

df.rename(columns={'sumdate':'Date'},inplace=True)
df.rename(columns={'Date ':'Dateinmonth'},inplace=True)
df.rename(columns={'Month ':'Month'},inplace=True)
df.rename(columns={'Year ':'Year'},inplace=True)
for col in df.columns:
  print(col)

df = df.groupby(by='Date').agg({
                               'Dateinmonth': 'mean',
                               'Month': 'mean',
                               'Day of the week': 'mean',
                               'mean wind speed 10m above ground (m/
    s)':'mean',
                               'Avg day temp (deg C)':'mean',
                               'Rain':'mean',
                               'Cloud cover':'mean',
                               'ibrnumsold': 'sum',}).reset_index()
df.head()
```

```
54
55 df.drop([
56           'mean wind speed 10m above ground (m/s)',
57           'Cloud cover'
58           ], inplace=True, axis=1)
59
60 df.head()
61
62 for col in df.columns:
63   print(col)
64
65 plot_cols = [
66               'Dateinmonth',
67               'Month',
68               'Day of the week',
69               'Avg day temp (deg C)',
70               'ibrnumsold',
71               'Rain',
72               ]
73
74 plot_features = df[plot_cols]
75 plot_features.index = df.Date #note if you index date then you will
       have to remove this
76 _ = plot_features.plot(subplots=True)
77
78 plot_features = df[plot_cols][:100]
79 plot_features.index = df.Date[:100]
80 _ = plot_features.plot(subplots=True)
81
82 df.describe().transpose()
83
84 def getDayOfYear(dateTimeVect):
85   dayOfYearVect =[]
86   for datetime in dateTimeVect:
87     dayOfYear = 0
```

```
    for i in range(1,datetime.month):
      dayOfYear += daysInMonth[i]
    dayOfYear += datetime.day
    dayOfYearVect.append(int(dayOfYear))
  return np.array(dayOfYearVect)

timestamp_s = df.Date
daysInMonth = [31, 29, 31, 30,31, 30, 31, 31, 30, 31, 30, 31]
print(getDayOfYear(df.Date))

month = 12
daysinyear = 366 #52.14*5 #52.14 weeks x 5 days a week
daysinmonth = 30.5


df['Year sin'] = np.sin(getDayOfYear(df.Date).astype(int) * (2 * np
    .pi / daysinyear))
df['Year cos'] = np.cos(getDayOfYear(df.Date).astype(int) * (2 * np
    .pi / daysinyear))

df['Month sin'] = np.sin(df.Dateinmonth.astype(int) * (2 * np.pi /
    daysinmonth))
df['Month cos'] = np.cos(df.Dateinmonth.astype(int) * (2 * np.pi /
    daysinmonth))

df['MonthYear sin'] = np.sin(df.Month.astype(int) * (2 * np.pi /
    month))
df['MonthYear cos'] = np.cos(df.Month.astype(int) * (2 * np.pi /
    month))

plt.plot(np.array(df['Month sin'])[:31])
plt.legend('sin')
plt.plot(np.array(df['Month cos'])[:31])
plt.legend('cos')
plt.xlabel('days')
```

```
plt.title('Time of year signal')
plt.show()

plt.plot(np.array(df['Year sin'])[:366])
plt.legend('sin')
plt.plot(np.array(df['Year cos'])[:366])
plt.legend('cos')
plt.xlabel('days')
plt.title('Time of year signal')
plt.legend
plt.show()

plt.plot(np.array(df['MonthYear sin'])[:366])
plt.legend('sin')
plt.plot(np.array(df['MonthYear cos'])[:366])
plt.legend('cos')
plt.xlabel('days')
plt.title('Time of year signal')
plt.legend
plt.show()

df.drop([
          'Dateinmonth',
          'Month',
          'Avg day temp (deg C)',
          'Rain',
          'Day of the week',
          'MonthYear sin',
          'MonthYear cos',
          'Month sin',
          'Month cos',
          ], inplace=True, axis=1)

df.head()

```

```
"""Split the data"""

df.set_index('Date', inplace=True)

dfscaled = df
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
dfscaled= scaler.fit_transform(dfscaled)

column_indices = {name: i for i, name in enumerate(df.columns)}

n = len(dfscaled)
train_df = dfscaled[0:int(n*0.60)]
val_df = dfscaled[int(n*0.60):int(n*0.90)]
test_df = dfscaled[int(n*0.90):]

num_features = dfscaled.shape[1]

"""Normalize the data"""

dfscaled = pd.DataFrame({
                         'ibrnumsold': dfscaled[:,0],
                         'Year sin': dfscaled[:,1],
                         'Year cos': dfscaled[:,2],
                        })

dfscaled.describe().transpose()

train_df = pd.DataFrame({
                         'ibrnumsold': train_df[:,0],
                         'Year sin': train_df[:,1],
                         'Year cos': train_df[:,2],
                        })
val_df = pd.DataFrame({
                         'ibrnumsold': val_df[:,0],
```

```
                            'Year sin': val_df[:,1],
                            'Year cos': val_df[:,2],
                           })
test_df = pd.DataFrame({
                            'ibrnumsold': test_df[:,0],
                            'Year sin': test_df[:,1],
                            'Year cos': test_df[:,2],
                           })

dfscaled1 = dfscaled.melt(var_name='Column', value_name='Normalized')
plt.figure(figsize=(12, 6))
ax = sns.violinplot(x='Column', y='Normalized', data=dfscaled1)
_ = ax.set_xticklabels(df.keys(), rotation=90)

drive.mount('/drive', force_remount=True)

test_df.to_csv('/drive/My Drive/test_df.csv',index=True)
train_df.to_csv('/drive/My Drive/train_df.csv', index=True)
val_df.to_csv('/drive/My Drive/val_df.csv', index=True)
df.to_csv('/drive/My Drive/df.csv', index=True)
```

```
# -*- coding: utf-8 -*-
"""print modelling.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1QHMuUq5MQsuj5CXhP2-HwkVGb4dh2du8

#  **0.Get data**
"""

import tensorflow as tf
import pandas as pd
```

```
import datetime
import os
import IPython
import IPython.display
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

#graph config below
mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['axes.grid'] = False

from google.colab import drive
drive.mount('/drive')

train_df = pd.read_csv('/drive/My Drive/train_df.csv')
test_df = pd.read_csv('/drive/My Drive/test_df.csv')
val_df = pd.read_csv('/drive/My Drive/val_df.csv')
df = pd.read_csv('/drive/My Drive/df.csv')

X_test = test_df.loc[:,[
                    'Year sin',
                    'Year cos'
                    ]]
Y_test = test_df.loc[:,'ibrnumsold']

from google.colab import drive
drive.mount('/content/drive')

train_df.set_index('Unnamed: 0', inplace=True)
test_df.set_index('Unnamed: 0', inplace=True)
val_df.set_index('Unnamed: 0', inplace=True)
df.set_index('Date', inplace=True)

```

```
column_indices = {name: i for i, name in enumerate(df.columns)}
num_features = df.shape[1]

"""# **1.Single Step models**

## Data windowing
"""

class WindowGenerator():
  def __init__(self, input_width, label_width, shift,
               train_df=train_df, val_df=val_df, test_df=test_df,
               label_columns=None):
    # Store the raw data.
    self.train_df = train_df
    self.val_df = val_df
    self.test_df = test_df

    # Work out the label column indices.
    self.label_columns = label_columns
    if label_columns is not None:
      self.label_columns_indices = {name: i for i, name in
                                    enumerate(label_columns)}
    self.column_indices = {name: i for i, name in
                           enumerate(train_df.columns)}

    # Work out the window parameters.
    self.input_width = input_width
    self.label_width = label_width
    self.shift = shift

    self.total_window_size = input_width + shift

    self.input_slice = slice(0, input_width)
    self.input_indices = np.arange(self.total_window_size)[self.
   input_slice]
```

```
83
84      self.label_start = self.total_window_size - self.label_width
85      self.labels_slice = slice(self.label_start, None)
86      self.label_indices = np.arange(self.total_window_size)[self.
        labels_slice]
87
88    def __repr__(self):
89      return '\n'.join([
90          f'Total window size: {self.total_window_size}',
91          f'Input indices: {self.input_indices}',
92          f'Label indices: {self.label_indices}',
93          f'Label column name(s): {self.label_columns}'])
94
95  w1 = WindowGenerator(input_width=22, label_width=1, shift=1,
96                       label_columns=['ibrnumsold'])
97  w1
98
99  def split_window(self, features):
100   inputs = features[:, self.input_slice, :]
101   labels = features[:, self.labels_slice, :]
102   if self.label_columns is not None:
103     labels = tf.stack(
104         [labels[:, :, self.column_indices[name]] for name in self.
        label_columns],
105         axis=-1)
106
107   # Slicing doesn't preserve static shape information, so set the
        shapes
108   # manually. This way the `tf.data.Datasets` are easier to inspect
        .
109   inputs.set_shape([None, self.input_width, None])
110   labels.set_shape([None, self.label_width, None])
111
112   return inputs, labels
113
```

```
WindowGenerator.split_window = split_window

# Stack three slices, the length of the total window:
example_window = tf.stack([np.array(train_df[:w1.total_window_size]),
                           np.array(train_df[100:100+w1.total_window_size]),
                           np.array(train_df[200:200+w1.total_window_size])])


example_inputs, example_labels = w1.split_window(example_window)

print('All shapes are: (batch, time, features)') #batch is number of training examples utilised, time and features, width and height
print(f'Window shape: {example_window.shape}')
print(f'Inputs shape: {example_inputs.shape}')
print(f'labels shape: {example_labels.shape}')

"""## Plot
visualisation of split window
"""

w1.example = example_inputs, example_labels

def plot(self, model=None, plot_col='ibrnumsold', max_subplots=3):
  inputs, labels = self.example
  plt.figure(figsize=(12, 8))
  plot_col_index = self.column_indices[plot_col]
  max_n = min(max_subplots, len(inputs))
  for n in range(max_n):
    plt.subplot(3, 1, n+1)
    plt.ylabel(f'{plot_col} [normed]')
    plt.plot(self.input_indices, inputs[n, :, plot_col_index],
```

```
            label='Inputs', marker='.', zorder=-10)

    if self.label_columns:
      label_col_index = self.label_columns_indices.get(plot_col,
    None)
    else:
      label_col_index = plot_col_index

    if label_col_index is None:
      continue

    plt.scatter(self.label_indices, labels[n, :, label_col_index],
                edgecolors='k', label='Labels', c='#2ca02c', s=64)
    if model is not None:
      predictions = model(inputs)
      plt.scatter(self.label_indices, predictions[n, :,
    label_col_index],
                  marker='X', edgecolors='k', label='Predictions',
                  c='#ff7f0e', s=64)

    if n == 0:
      plt.legend()

  plt.xlabel('Days')

WindowGenerator.plot = plot

w1.plot()

"""## Create tf.data.datasets"""

def make_dataset(self, data):
  data = np.array(data, dtype=np.float32)
  ds = tf.keras.preprocessing.timeseries_dataset_from_array(
      data=data,
```

```
      targets=None,
      sequence_length=self.total_window_size,
      sequence_stride=1,
      shuffle=True,
      batch_size=32,)

  ds = ds.map(self.split_window)

  return ds

WindowGenerator.make_dataset = make_dataset

@property
def train(self):
  return self.make_dataset(self.train_df)

@property
def val(self):
  return self.make_dataset(self.val_df)

@property
def test(self):
  return self.make_dataset(self.test_df)

@property
def example(self):
  """Get and cache an example batch of `inputs, labels` for
   plotting."""
  result = getattr(self, '_example', None)
  if result is None:
    # No example batch was found, so get one from the `.train`
   dataset
    result = next(iter(self.train))
    # And cache it for next time
    self._example = result
```

```
  return result

WindowGenerator.train = train
WindowGenerator.val = val
WindowGenerator.test = test
WindowGenerator.example = example

# Each element is an (inputs, label) pair
w1.train.element_spec

for example_inputs, example_labels in w1.train.take(1):
  print(f'Inputs shape (batch, time, features): {example_inputs.
    shape}')
  print(f'Labels shape (batch, time, features): {example_labels.
    shape}')

"""## Single step model
predict a single features value one timestep into the future
"""

single_step_window = WindowGenerator(
    input_width=1, label_width=1, shift=1,
    label_columns=['ibrnumsold'])
single_step_window

for example_inputs, example_labels in single_step_window.train.take
    (1):
  print(f'Inputs shape (batch, time, features): {example_inputs.
    shape}')
  print(f'Labels shape (batch, time, features): {example_labels.
    shape}')

"""## Baseline

model that returns current value as prediction
```

```
"""

class Baseline(tf.keras.Model):
  def __init__(self, label_index=None):
    super().__init__()
    self.label_index = label_index

  def call(self, inputs):
    if self.label_index is None:
      return inputs
    result = inputs[:, :, self.label_index]
    return result[:, :, tf.newaxis]

import keras as keras

keras.backend.set_epsilon(1)
baseline = Baseline(label_index=column_indices['ibrnumsold'])

baseline.compile(loss=tf.losses.MeanSquaredError(),
                 metrics=[tf.metrics.MeanAbsoluteError()])

val_performance = {}
performance = {}
val_performance['Baseline'] = baseline.evaluate(single_step_window.
    val)
performance['Baseline'] = baseline.evaluate(single_step_window.test
    , verbose=0)

"""create a wide window to display output below"""

wide_window = WindowGenerator(
    input_width=48, label_width=48, shift=1,
    label_columns=['ibrnumsold'])

wide_window
```

```
273
274 print('Input shape:', wide_window.example[0].shape)
275 print('Output shape:', baseline(wide_window.example[0]).shape)
276
277 wide_window.plot(baseline)
278
279 """## Linear model
280
281 simplest trainable model - linear transformation from input to
    output
282 """
283
284 linear = tf.keras.Sequential([
285     tf.keras.layers.Dense(units=1)
286 ])
287
288 print('Input shape:', single_step_window.example[0].shape)
289 print('Output shape:', linear(single_step_window.example[0]).shape)
290
291 MAX_EPOCHS = 75
292
293 def compile_and_fit(model, window, patience=10):
294   early_stopping = tf.keras.callbacks.EarlyStopping(monitor='
    val_loss',
295                                                     patience=
    patience,
296                                                     mode='min')
297
298   model.compile(loss=tf.losses.MeanSquaredError(),
299                 optimizer=tf.optimizers.RMSprop(),
300                 metrics=[tf.metrics.MeanAbsoluteError()])
301
302   history = model.fit(window.train, epochs=MAX_EPOCHS,
303                       validation_data=window.val,
304                       callbacks=[early_stopping])
```

```
  return history

history = compile_and_fit(linear, single_step_window)

val_performance['Linear'] = linear.evaluate(single_step_window.val)
performance['Linear'] = linear.evaluate(single_step_window.test,
    verbose=0)

# Plot loss per iteration

plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='evaluation loss')
plt.legend()

print('Input shape:', wide_window.example[0].shape)
print('Output shape:', baseline(wide_window.example[0]).shape)

"""used wide window in a similar manner to plot outcomes - each
    prediction is independant of the other"""

wide_window.plot(linear)

plt.bar(x = range(len(train_df.columns)),
        height=linear.layers[0].kernel[:,0].numpy())
axis = plt.gca()
axis.set_xticks(range(len(train_df.columns)))
_ = axis.set_xticklabels(train_df.columns, rotation=90)

"""## Dense
more powerful but still a single input model
"""

dense = tf.keras.Sequential([
    tf.keras.layers.Dense(units=64, activation='relu'),
    tf.keras.layers.Dense(units=64, activation='relu'),
```

```
    tf.keras.layers.Dense(units=1)
])

history = compile_and_fit(dense, single_step_window)

val_performance['Dense'] = dense.evaluate(single_step_window.val)
performance['Dense'] = dense.evaluate(single_step_window.test,
    verbose=0)

# Plot loss per iteration

plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='evaluation loss')
plt.legend()

"""## Multi step dense
access to multiple time steps when making a prediction
"""

CONV_WIDTH = 10
conv_window = WindowGenerator(
    input_width=CONV_WIDTH,
    label_width=1,
    shift=1,
    label_columns=['ibrnumsold'])

conv_window

conv_window.plot()
plt.title("Given 10 days as input, predict 1 day into the future.")

multi_step_dense = tf.keras.Sequential([
    # Shape: (time, features) => (time*features)
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=32, activation='relu'),
```

```
    tf.keras.layers.Dense(units=32, activation='relu'),
    tf.keras.layers.Dense(units=1),
    # Add back the time dimension.
    # Shape: (outputs) => (1, outputs)
    tf.keras.layers.Reshape([1, -1]),
])

print('Input shape:', conv_window.example[0].shape)
print('Output shape:', multi_step_dense(conv_window.example[0]).
    shape)

history = compile_and_fit(multi_step_dense, conv_window)

#IPython.display.clear_output()
val_performance['Multi step dense'] = multi_step_dense.evaluate(
    conv_window.val)
performance['Multi step dense'] = multi_step_dense.evaluate(
    conv_window.test, verbose=0)

# Plot loss per iteration

plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='evaluation loss')
plt.legend()

conv_window.plot(multi_step_dense)

print('Input shape:', wide_window.example[0].shape)
try:
  print('Output shape:', multi_step_dense(wide_window.example[0]).
    shape)
except Exception as e:
  print(f'\n{type(e).__name__}:{e}')

"""downside - input windows have to be exactly this shape. Conv.
```

```
    solves this

## Convolution Neural Network
with multiple time steps
"""

conv_model = tf.keras.Sequential([
    tf.keras.layers.Conv1D(filters=32,
                           kernel_size=(CONV_WIDTH,),
                           activation='relu'),
    tf.keras.layers.Dense(units=32, activation='relu'),
    tf.keras.layers.Dense(units=1),
])

"""run on sample batch to check outputs are the expected shape

"""

print("Conv model on `conv_window`")
print('Input shape:', conv_window.example[0].shape)
print('Output shape:', conv_model(conv_window.example[0]).shape)

history = compile_and_fit(conv_model, conv_window)

#IPython.display.clear_output()
val_performance['Conv'] = conv_model.evaluate(conv_window.val)
performance['Conv'] = conv_model.evaluate(conv_window.test, verbose
    =0)

# Plot loss per iteration

plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='evaluation loss')
plt.legend()
```

```
436 print("Wide window")
437 print('Input shape:', wide_window.example[0].shape)
438 print('Labels shape:', wide_window.example[1].shape)
439 print('Output shape:', conv_model(wide_window.example[0]).shape)
440
441 LABEL_WIDTH = 24
442 INPUT_WIDTH = LABEL_WIDTH + (CONV_WIDTH - 1)
443 wide_conv_window = WindowGenerator(
444     input_width=INPUT_WIDTH,
445     label_width=LABEL_WIDTH,
446     shift=1,
447     label_columns=['ibrnumsold'])
448
449 wide_conv_window
450
451 print("Wide conv window")
452 print('Input shape:', wide_conv_window.example[0].shape)
453 print('Labels shape:', wide_conv_window.example[1].shape)
454 print('Output shape:', conv_model(wide_conv_window.example[0]).
    shape)
455
456 wide_conv_window.plot(conv_model)
457
458 """## Recurrent Neural Network
459
460 using LSTM
461 """
462
463 lstm_model = tf.keras.models.Sequential([
464     # Shape [batch, time, features] => [batch, time, lstm_units]
465     tf.keras.layers.LSTM(32, return_sequences=True),
466     # Shape => [batch, time, features]
467     tf.keras.layers.Dense(units=1)
468 ])
469
```

```
"""return sequence above - if true - returns output for each input,
    if false the layer only returns the output of the final
    timestep. Which gives the model a chance to warm up."""

print('Input shape:', wide_window.example[0].shape)
print('Output shape:', lstm_model(wide_window.example[0]).shape)

history = compile_and_fit(lstm_model, wide_window)

#IPython.display.clear_output()
val_performance['LSTM'] = lstm_model.evaluate(wide_window.val)
performance['LSTM'] = lstm_model.evaluate(wide_window.test, verbose
    =0)

wide_window.plot(lstm_model)

# Plot loss per iteration

plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='evaluation loss')
plt.legend()

"""## Performance"""

x = np.arange(len(performance))
width = 0.3
metric_name = 'mean_absolute_error'
metric_index = lstm_model.metrics_names.index('mean_absolute_error'
    )
val_mae = [v[metric_index] for v in val_performance.values()]
test_mae = [v[metric_index] for v in performance.values()]

plt.ylabel('MAE')
plt.bar(x - 0.17, val_mae, width, label='Validation')
plt.bar(x + 0.17, test_mae, width, label='Test')
```

```
plt.xticks(ticks=x, labels=performance.keys(),
           rotation=45)
_ = plt.legend()

for name, value in performance.items():
  print(f'{name:12s}: {value[1]:0.4f}')

"""#**2. Multi Step Models**"""

OUT_STEPS = 10
multi_window = WindowGenerator(input_width=10,
                               label_width=OUT_STEPS,
                               shift=OUT_STEPS)

multi_window.plot()
multi_window

"""## Baselines

just repeating the last input time step
"""

class MultiStepLastBaseline(tf.keras.Model):
  def call(self, inputs):
    return tf.tile(inputs[:, -1:, :], [1, OUT_STEPS, 1])

last_baseline = MultiStepLastBaseline()
last_baseline.compile(loss=tf.losses.MeanSquaredError(),
                      metrics=[tf.metrics.MeanAbsoluteError()])

multi_val_performance = {}
multi_performance = {}

multi_val_performance['Last'] = last_baseline.evaluate(multi_window
   .val)
```

```
multi_performance['Last'] = last_baseline.evaluate(multi_window.test, verbose=0)
multi_window.plot(last_baseline)

"""below just repeats the last "window" like a stamp"""

class RepeatBaseline(tf.keras.Model):
  def call(self, inputs):
    return inputs

repeat_baseline = RepeatBaseline()
repeat_baseline.compile(loss=tf.losses.MeanSquaredError(),
                        metrics=[tf.metrics.MeanAbsoluteError()])

multi_val_performance['Repeat'] = repeat_baseline.evaluate(multi_window.val)
multi_performance['Repeat'] = repeat_baseline.evaluate(multi_window.test, verbose=0)
multi_window.plot(repeat_baseline)

"""## Single shot model

### Linear
"""

multi_linear_model = tf.keras.Sequential([
    # Take the last time-step.
    # Shape [batch, time, features] => [batch, 1, features]
    tf.keras.layers.Lambda(lambda x: x[:, -1:, :]),
    # Shape => [batch, 1, out_steps*features]
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros),
    # Shape => [batch, out_steps, features]
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
```

```
])

history = compile_and_fit(multi_linear_model, multi_window)

#IPython.display.clear_output()
multi_val_performance['Linear'] = multi_linear_model.evaluate(
    multi_window.val)
multi_performance['Linear'] = multi_linear_model.evaluate(
    multi_window.test, verbose=0)
multi_window.plot(multi_linear_model)

# Plot loss per iteration

plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='evaluation loss')
plt.legend()

"""### Dense"""

multi_dense_model = tf.keras.Sequential([
    # Take the last time step.
    # Shape [batch, time, features] => [batch, 1, features]
    tf.keras.layers.Lambda(lambda x: x[:, -1:, :]),
    # Shape => [batch, 1, dense_units]
    tf.keras.layers.Dense(512, activation='relu'),
    # Shape => [batch, out_steps*features]
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros)
    ,
    # Shape => [batch, out_steps, features]
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])

history = compile_and_fit(multi_dense_model, multi_window)

```

```
IPython.display.clear_output()
multi_val_performance['Dense'] = multi_dense_model.evaluate(multi_window.val)
multi_performance['Dense'] = multi_dense_model.evaluate(multi_window.test, verbose=0)
multi_window.plot(multi_dense_model)

# Plot loss per iteration

plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='evaluation loss')
plt.legend()

"""### CNN"""

CONV_WIDTH = 3
multi_conv_model = tf.keras.Sequential([
    # Shape [batch, time, features] => [batch, CONV_WIDTH, features]
    tf.keras.layers.Lambda(lambda x: x[:, -CONV_WIDTH:, :]),
    # Shape => [batch, 1, conv_units]
    tf.keras.layers.Conv1D(256, activation='relu', kernel_size=(CONV_WIDTH)),
    # Shape => [batch, 1,  out_steps*features]
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros),
    # Shape => [batch, out_steps, features]
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])

history = compile_and_fit(multi_conv_model, multi_window)

IPython.display.clear_output()

```

```
multi_val_performance['Conv'] = multi_conv_model.evaluate(
    multi_window.val)
multi_performance['Conv'] = multi_conv_model.evaluate(multi_window.
    test, verbose=0)
multi_window.plot(multi_conv_model)

# Plot loss per iteration

plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='evaluation loss')
plt.legend()

"""### RNN"""

multi_lstm_model = tf.keras.Sequential([
    # Shape [batch, time, features] => [batch, lstm_units]
    # Adding more `lstm_units` just overfits more quickly.
    tf.keras.layers.LSTM(32, return_sequences=False),
    # Shape => [batch, out_steps*features]
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros)
    ,
    # Shape => [batch, out_steps, features]
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])

history = compile_and_fit(multi_lstm_model, multi_window)

IPython.display.clear_output()

multi_val_performance['LSTM'] = multi_lstm_model.evaluate(
    multi_window.val)
multi_performance['LSTM'] = multi_lstm_model.evaluate(multi_window.
    test, verbose=0)
multi_window.plot(multi_lstm_model)
```

```
658
659 # Plot loss per iteration
660
661 plt.plot(history.history['loss'], label='training loss')
662 plt.plot(history.history['val_loss'], label='evaluation loss')
663 plt.legend()
664
665 """## Advanced: Autoregressive model"""
666
667 class FeedBack(tf.keras.Model):
668   def __init__(self, units, out_steps):
669     super().__init__()
670     self.out_steps = out_steps
671     self.units = units
672     self.lstm_cell = tf.keras.layers.LSTMCell(units)
673     # Also wrap the LSTMCell in an RNN to simplify the `warmup`
    method.
674     self.lstm_rnn = tf.keras.layers.RNN(self.lstm_cell,
    return_state=True)
675     self.dense = tf.keras.layers.Dense(num_features)
676
677 feedback_model = FeedBack(units=32, out_steps=OUT_STEPS)
678
679 def warmup(self, inputs):
680   # inputs.shape => (batch, time, features)
681   # x.shape => (batch, lstm_units)
682   x, *state = self.lstm_rnn(inputs)
683
684   # predictions.shape => (batch, features)
685   prediction = self.dense(x)
686   return prediction, state
687
688 FeedBack.warmup = warmup
689
690 prediction, state = feedback_model.warmup(multi_window.example[0])
```

```
691 prediction.shape
692
693 def call(self, inputs, training=None):
694   # Use a TensorArray to capture dynamically unrolled outputs.
695   predictions = []
696   # Initialize the lstm state
697   prediction, state = self.warmup(inputs)
698
699   # Insert the first prediction
700   predictions.append(prediction)
701
702   # Run the rest of the prediction steps
703   for n in range(1, self.out_steps):
704     # Use the last prediction as input.
705     x = prediction
706     # Execute one lstm step.
707     x, state = self.lstm_cell(x, states=state,
708                               training=training)
709     # Convert the lstm output to a prediction.
710     prediction = self.dense(x)
711     # Add the prediction to the output
712     predictions.append(prediction)
713
714   # predictions.shape => (time, batch, features)
715   predictions = tf.stack(predictions)
716   # predictions.shape => (batch, time, features)
717   predictions = tf.transpose(predictions, [1, 0, 2])
718   return predictions
719
720 FeedBack.call = call
721
722 print('Output shape (batch, time, features): ', feedback_model(
      multi_window.example[0]).shape)
723
724 history = compile_and_fit(feedback_model, multi_window)
```

```
725
726 IPython.display.clear_output()
727
728 multi_val_performance['AR LSTM'] = feedback_model.evaluate(
    multi_window.val)
729 multi_performance['AR LSTM'] = feedback_model.evaluate(multi_window
    .test, verbose=0)
730 multi_window.plot(feedback_model)
731
732 # Plot loss per iteration
733
734 plt.plot(history.history['loss'], label='training loss')
735 plt.plot(history.history['val_loss'], label='evaluation loss')
736 plt.legend()
737
738 """## Performance"""
739
740 x = np.arange(len(multi_performance))
741 width = 0.3
742
743
744 metric_name = 'mean_absolute_error'
745 metric_index = lstm_model.metrics_names.index('mean_absolute_error'
    )
746 val_mae = [v[metric_index] for v in multi_val_performance.values()]
747 test_mae = [v[metric_index] for v in multi_performance.values()]
748
749 plt.bar(x - 0.17, val_mae, width, label='Validation')
750 plt.bar(x + 0.17, test_mae, width, label='Test')
751 plt.xticks(ticks=x, labels=multi_performance.keys(),
752            rotation=45)
753 plt.ylabel(f'MAE')
754 _ = plt.legend()
755
756 for name, value in multi_performance.items():
```

```
757   print(f'{name:8s}: {value[1]:0.4f}')
```

Appendix E

Parameter tuning: Numerical results

Tables E.1 to E.3 summarises the results captured during the parameter tuning process for the neural network models. This data was used for detailed result analysis.

Table E.1: MAE results for parameter tuning numerical analysis

	Single step						Multistep						
	baseline	linear	dense	multi step de	conv	LSTM	last	repeat	linear	dense	conv	LSTM	AR LSTM
all features	0.1101	0.1268	0.1126	0.11	0.107	0.0968	0.1476	0.123	0.1584	0.0902	0.0873	0.1089	0.1064
just sin and cos features (batch 32 and epochs 50)	0.1101	0.3214	0.109	0.0969	0.1163	0.0994	0.0728	0.1026	0.1625	0.0403	0.0423	0.0487	0.0411
batch 64 (epochs 50)	0.1101	0.3102	0.1091	0.103	0.0999	0.0949	0.0728	0.1026	0.2133	0.0454	0.0418	0.0491	0.0503
epochs 100	0.1101	0.1565	0.1044	0.0969	0.0959	0.0974	0.0728	0.1026	0.0889	0.0442	0.0429	0.0446	0.0449
epochs 20	0.1101	0.3084	0.1088	0.1104	0.1008	0.0986	0.0728	0.1026	0.2265	0.0443	0.0422	0.0512	0.0487
epochs 150	0.1101	0.314	0.1103	0.1122	0.0983	0.0992	0.0728	0.1026	0.052	0.0452	0.0444	0.0479	0.0423
patience 5 (from 10) epoch 50	0.1101	0.1846	0.1083	0.094	0.0999	0.0966	0.0728	0.1026	0.1627	0.044	0.0418	0.0486	0.0447
patience 5 (from 10) epoch 100	0.1101	0.1099	0.1239	0.1031	0.104	0.0947	0.0728	0.1026	0.0886	0.045	0.0431	0.0477	0.0487
32b 75epoch pat 10 - optimiser adam	0.1101	0.1548	0.0963	0.1099	0.0983	0.0949	0.0728	0.1026	0.1209	0.0452	0.0443	0.0478	0.0468
32b 75epoch pat 10 - optimiser rmsprop	0.1101	0.1086	0.0925	0.0956	0.0948	0.1154	0.0728	0.1026	0.042	0.0561	0.0493	0.0524	0.0884
32b 75epoch pat 10 - optimiser SGD	0.1101	0.1277	0.1106	0.096	0.107	0.1133	0.0728	0.1026	0.2416	0.2154	0.2038	0.2523	0.0715
windows increased	0.1101	0.1023	0.1009	0.0915	0.0948	0.1059	0.0868	0.1321	0.0437	0.049	0.0498	0.0638	0.0541
windows decreased	0.1101	0.3221	0.1086	0.1098	0.1007	0.0998	0.058	0.0709	0.0419	0.0488	0.0514	0.0515	0.0515
activation = sigmoid	0.1101	0.111	0.139	0.149	0.1057	0.1104	0.0728	0.1026	0.042	0.1253	0.0867	0.0526	0.0532
activation = tanh	0.1101	0.1528	0.1052	0.1028	0.096	0.1051	0.0728	0.1026	0.0425	0.0663	0.071	0.0434	0.0488
results with data split	0.1072	0.0979	0.0849	0.0944	0.0934	0.0958	0.0674	0.0937	0.0395	0.0528	0.0493	0.0541	0.0597

Table E.2: MSE results for parameter tuning numerical analysis

	Single step						Multistep						
	baseline	linear	dense	multi step de	conv	LSTM	last	repeat	linear	dense	conv	LSTM	AR LSTM
all features	0.0212	0.0689	0.0207	0.0195	0.0163	0.0187	0.063	0.0421	0.0423	0.0234	0.023	0.0316	0.0318
just sin and cos features (batch 32 and epochs 50)	0.0212	0.0537	0.0173	0.015	0.015	0.0153	0.0115	0.0175	0.0348	0.0067	0.0063	0.0076	0.0062
batch 64 (epochs 50)	0.0212	0.1076	0.0165	0.0186	0.017	0.0165	0.0115	0.0175	0.0608	0.0069	0.0062	0.0077	0.0064
epochs 100	0.0212	0.2071	0.0282	0.0176	0.0159	0.0152	0.0115	0.0175	0.0117	0.0073	0.0062	0.0063	0.0061
epochs 20	0.0212	0.1733	0.0213	0.0173	0.0151	0.0169	0.0115	0.0175	0.0702	0.0075	0.0059	0.0075	0.0062
epochs 150	0.0212	0.0197	0.0175	0.0172	0.0184	0.016	0.0115	0.0175	0.0072	0.0069	0.006	0.0076	0.0056
patience 5 (from 10) epoch 50	0.0212	0.1394	0.0181	0.0182	0.0156	0.0173	0.0115	0.0175	0.0349	0.0069	0.0058	0.0078	0.0068
patience 5 (from 10) epoch 100	0.0212	0.0547	0.0195	0.0165	0.0167	0.0165	0.0115	0.0175	0.0117	0.0066	0.0059	0.0077	0.0072
32b 75epoch pat 10 - optimiser adam	0.0212	0.0285	0.0179	0.0185	0.0156	0.0156	0.0115	0.0175	0.0194	0.0071	0.006	0.0065	0.0074
32b 75epoch pat 10 - optimiser rmsprop	0.0212	0.0173	0.0244	0.015	0.0193	0.0183	0.0115	0.0175	0.0069	0.0079	0.0063	0.0086	0.0078
32b 75epoch pat 10 - optimiser SGD	0.0212	0.0178	0.0235	0.0177	0.0227	0.0245	0.0115	0.0175	0.0866	0.0624	0.0558	0.0975	0.0211
windows increased	0.0212	0.0171	0.0144	0.0223	0.0209	0.0164	0.014	0.0248	0.0069	0.0065	0.0079	0.0078	0.0102
windows decreased	0.0212	0.0185	0.0158	0.0393	0.0154	0.0168	0.0097	0.0115	0.0071	0.0072	0.0072	0.0072	0.0065
activation = sigmoid	0.0212	0.0168	0.022	0.0184	0.0248	0.0225	0.0115	0.0175	0.0069	0.0288	0.0101	0.0068	0.0077
activation = tanh	0.0212	0.0186	0.0204	0.0155	0.0237	0.015	0.0115	0.0175	0.0068	0.0082	0.0077	0.0066	0.0083
results with data split	0.0179	0.0152	0.0124	0.0154	0.0172	0.0133	0.0092	0.0134	0.0059	0.0086	0.0075	0.0065	0.0071

Table E.3: RMSE results for parameter tuning numerical analysis

	Single step						Multistep						
	baseline	linear	dense	multi step de	conv	LSTM	last	repeat	linear	dense	conv	LSTM	AR LSTM
all features	0.1457	0.2561	0.1467	0.1484	0.1329	0.1304	0.2509	0.2053	0.2058	0.1518	0.1506	0.177	0.181
just sin and cos features (batch 32 and epochs 50)	0.1457	0.5282	0.131	0.1332	0.1212	0.122	0.1071	0.1323	0.1868	0.0808	0.0863	0.0889	0.078
batch 64 (epochs 50)	0.1475	0.1524	0.1317	0.135	0.125	0.1313	0.1071	0.1323	0.2465	0.0822	0.0778	0.0846	0.0813
epochs 100	0.1457	0.2314	0.1337	0.1237	0.1177	0.1324	0.1071	0.1323	0.1081	0.0789	0.0814	0.0863	0.0874
epochs 20	0.1475	0.155	0.1379	0.1238	0.1299	0.1347	0.1071	0.1323	0.2646	0.0821	0.0802	0.0874	0.0861
epochs 150	0.1475	0.4124	0.1643	0.127	0.1303	0.1257	0.1071	0.1323	0.0852	0.0843	0.0798	0.082	0.083
patience 5 (from 10) epoch 50	0.1457	0.2593	0.147	0.1469	0.1237	0.1294	0.1071	0.1323	0.1868	0.0826	0.0784	0.0855	0.0833
patience 5 (from 10) epoch 100	0.1457	0.2277	0.1393	0.138	0.1327	0.1237	0.1071	0.1323	0.1084	0.0822	0.0819	0.0828	0.0817
32b 75epoch pat 10 - optimiser adam	0.1457	0.4299	0.1444	0.1272	0.1256	0.1263	0.1071	0.1323	0.1398	0.0817	0.0826	0.0853	0.0844
32b 75epoch pat 10 - optimiser rmsprop	0.1475	0.1494	0.1296	0.1345	0.1409	0.1329	0.1071	0.1323	0.0822	0.0856	0.0847	0.0884	0.0777
32b 75epoch pat 10 - optimiser SGD	0.1457	0.1347	0.1629	0.136	0.1361	0.1578	0.1071	0.1323	0.2943	0.2479	0.2323	0.2785	0.1015
windows increased	0.1457	0.1825	0.126	0.1564	0.1447	0.1234	0.1184	0.1574	0.0832	0.08	0.0887	0.0866	0.1093
windows decreased	0.1457	0.2067	0.1815	0.1389	0.1316	0.1242	0.0984	0.1072	0.085	0.0976	0.0812	0.0883	0.0894
activation = sigmoid	0.1457	0.1343	0.1385	0.1314	0.1383	0.1364	0.1071	0.1323	0.083	0.1322	0.1087	0.0905	0.0938
activation = tanh	0.1457	0.1798	0.1497	0.1356	0.1335	0.1285	0.1071	0.1323	0.0831	0.1003	0.0843	0.0801	0.0818
results with data split	0.1338	0.1519	0.1184	0.1109	0.1689	0.118	0.0961	0.1158	0.0779	0.0794	0.0749	0.0806	0.0907