Cardiff School of Engineering

Coursework Cover Sheet

Module Details

Module Name: PROJECT Module No: EN3100

Coursework Title: MACHINE LEARNING IN FINANCIAL ENGINEERING

Element: /

Lecturer: Dr: KENSUKE YOKOI

Personal Details

Name: CAMERON STUMPF Student No: 1673094

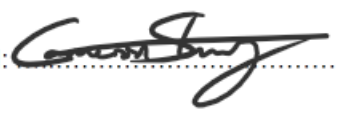
Personal Tutor: Dr: ZE JI Discipline: MMM / EEE

MMM

(please circle)

Declaration

I hereby declare that, except where I have made clear and full reference to the work of others, this submission, and all the material (*e.g.* text, pictures, diagrams) contained in it, is my own work, has not previously been submitted for assessment, and I have not knowingly allowed it to be copied by another student. In the case of group projects, the contribution of group members has been appropriately quantified.
I understand that deceiving, or attempting to deceive, examiners by passing off the work of another as my own is plagiarism. I also understand that plagiarising another's work, or knowingly allowing another student to plagiarise from my work, is against University Regulations and that doing so will result in loss of marks and disciplinary proceedings. I understand and agree that the University's plagiarism software 'Turnitin' may be used to check the originality of the submitted coursework.

Signed:  Date: 25/03/2020

Machine Learning in Financial Engineering

Cameron Stumpf - C1673094 - 2019-2020

Supervisor: Dr. Kensuke Yokoi

Abstract

Supervised machine learning methods are widely used in financial engineering applications to generate forecasts of future consumer trends and market behaviours, the accuracy of which may determine the future success of said business or venture; these methods have historically taken the form of decision tree ensemble methods, however, more cutting edge neural network methods are seeing increasing use. In this dissertation: gradient boosted decision trees, random forests, and ensemble neural network methods are used to generate forecasts for the behaviour of the NASDAQ100 stock market index; the project seeks to determine each model's relative suitability for financial forecasting applications by evaluating the sensitivity of their forecast accuracies subject to variations in model hyperparameter selection and data availability across short to medium ranges.

Table of Contents

1 - Introduction

Machine learning is a branch of artificial intelligence that sees a system's ability to iteratively improve its performance in a specific task without being explicitly reprogrammed; one of its most widely used applications is in time-series regression forecasting – to predict the future behaviour of an object or environment as a continuous variable based on its past behaviour.

If a business is able to accurately predict consumer trends, market behaviours, or competitor performance may influence business strategy and contribute towards higher profits.

The ability to accurately forecast environmental behaviour may find particular value in stock market forecasting where, historically, forecasts may have been generated by highly trained financial analysts guided loosely by basic linear regression tools.

Modern forecasting models utilise decision tree machine learning methods due to their low computing power requirements and accessibility through platforms such as Microsoft Azure. More cutting-edge neural network methods, however, are quickly finding more widespread use as the costs of high-powered computing lowers and accessibility of rentable machine learning servers grows.

This project seeks to evaluate and compare the relative performance of decision tree and neural network machine learning methods, to evaluate the effect on their forecast accuracies from variations in hyperparameter selection and variations in training data availability. The project will assess gradient boosted and randomized forest ensemble decision tree methods against averaged neural network models; 10 individual models will be

trained for each experiment, each forecasting in +0.5 day intervals up to a maximum of +5.0 days.

1.1 - Previous Work

The work contained in this project was inspired by the works of Jeremy Howard and Sylvian Gugger (Howard and Gugger, 2020) in development of the fast.ai neural network machine learning library and in the substantial support that they offer on the fast.ai forums.

This project additionally acts as an extension of the work done by Callum McIntyre's similarly titled Machine Learning in Financial Engineering (McIntyre, n.d.) within which experimentation was conducted to evaluate the relative efficacy of several machine learning methods in forecasting the behaviour of Apple's stock price using Microsoft's Azure Machine Learning platform; it was found that boosted decision tree models outperformed neural network models in comparable experiments.

2 - Theory

2.1 - Economic Theory

The stock market refers to a collection of markets and exchanges upon which a business may choose to publicly list itself. A business will release a limited number of shares in itself for sale publicly, the price of each share is dependant solely upon it's demand, which, in turn, is dependant on the number of shares released and the prospective buyer's confidence in the business's future performance.

Some believe that the market value of a single stock is an accurate reflection of all available information and, thus, it is impossible for a stock to be over or under valued; this is referred to as the Efficient Market Hypothesis (Sikorski, 1979). The implication of this hypothesis is that it is impossible to "beat the market", and that efforts to forecast the future behaviour of a stock's value are wasted as the future value of said stock will be dependent upon information that is not currently available; experimentation to prove this theory involves comparison between the performance of portfolios of randomly chosen stocks and stocks specifically chosen (Burton G. Malkiel, 2003). Figure 1 depicts various potential reactions of a stock's value following the release of "good news" about the subject company; the efficient market hypothesis is reflected by the "Efficient market" line.

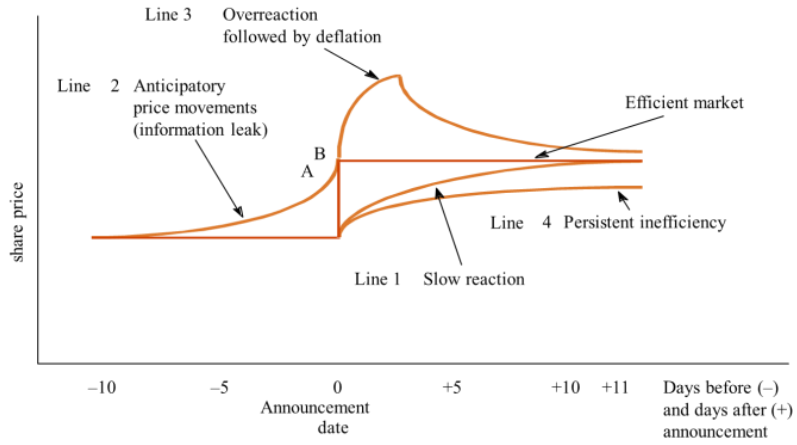


Figure 1: Visualization of a market's reaction to "good news" (Efficient Market Hypothesis, 2014)

Many reject this hypothesis, however, and view stock value as far more fallible. Those who do reject the Efficient Market Hypothesis view trading as having a competitive element, with the individual trader aiming to "beat" other traders by buying stocks that are perceived to be "under-valued" whilst selling when they appear to be "over-valued". Stock prices are dependent on supply and demand, and whilst the supply of shares may be controlled by the business itself, the demand is dependent on a wide range of environmental and economic

factors such as global GDP, housing prices, unemployment, and interest rates, however, stock values may also be dependent on consumer trends, business reputation, and global politics.

A stock index reflects the cumulative weighted value of a range of stocks listed on a single stock exchange. The value of an index is not directly dependant on the market itself, but rather on the weighted values of the stocks contained within the index, weightings determined by the index management. An index, then, can be viewed as carrying the accumulated information on the performance of a specific market sector, public trends, beliefs, and news.

The NASDAQ100 (NDX^) stock index reflects the top 100 valued businesses listed on the NASDAQ stock exchange; the NASDAQ is based in New York and consists mostly of businesses in the technology sector. The value of the NDX^ Index "…equals the aggregate value of the index share weights … of each of the Index Securities multiplied by each such security's Last Sale Price" (NASDAQ-100 INDEX METHODOLOGY, 2018). The relative stability of the NDX^ despite its complexity makes it the ideal candidate for forecasting experiments.

2.2 - Machine Learning Theory

2.2.1 - General Machine Learning Concepts

The fundamental objective of supervised machine learning is to produce a function capable of accurately fitting a set of input features to a single target feature by iteratively applying said function, assessing the difference between its output and the corresponding target value through a loss function, then incrementally updating the function itself to fit the training data. These incremental revisions to the function are conducted on each

input/target pair, or for each row of a dataset; one full pass through the training dataset is referred to as one epoch. Once the model is fully trained, it may then be applied to the testing dataset input values which tests the model's trained accuracy.

Different loss functions are better suited to certain machine learning problems than others, and thus consideration must be given towards the criterion chosen. Mean Squared Error Loss was chosen for this project as all features, once normalised, have relatively low numerical value (Jha, 2019).

$$\text{loss}(x, y) = (x - y)^2$$

Equation 1: Mean Squared Error Loss function

The two main risks of model training are to underfit and to overfit the model, the former resulting in an under-trained model that is unable to fit to the test data (Figure 2: Left) whilst the latter has over-trained to the training dataset and is thus overly sensitive to the data (Figure 2: Right); both result in high error rates when testing.

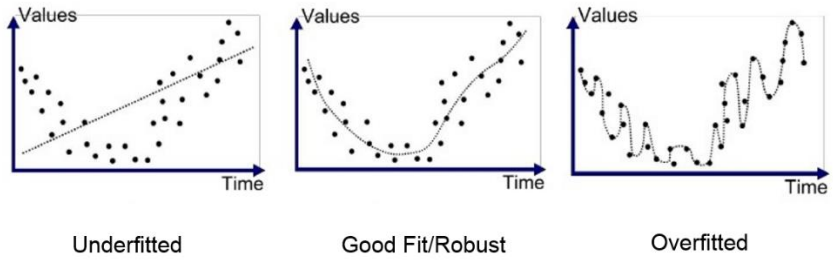


Figure 2: Graphs visualising the behaviour of underfit and overfit models (Bhande, 2018)

Ensemble learning methods see the combination of smaller, deliberately underfit models being combined to reduce noise, bias and variance from a forecast output. Two fundamental types of ensemble methods are averaging methods and boosting methods, the former seeing the outputs of multiple parallel models being averaged between one another,

the latter sees multiple models being combined with the intention of fitting the collected models towards a chosen loss function (1.11. Ensemble methods — scikit-learn 0.22.2 documentation, 2019).

2.2.2 - Set 1: Decision Tree Machine Learning

Decision tree machine learning is a method of predictive modelling that sees a model's arrival at a conclusion about a subject, in this case a continuous variable output, based on observations made on each input feature. The structure of a decision tree may be visualised as in Figure 3 where X represents a node, and F and T represent contrasting observations; a fully trained model may contain hundreds of these nodes and branches.

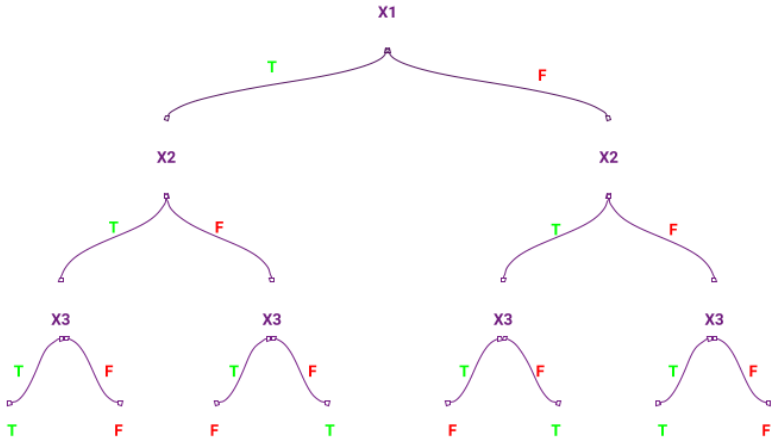


Figure 3: An example of a decision tree model (Decision Tree Tutorials & Notes | Machine Learning | HackerEarth, 2018)

2.2.2.1 - Set 1 Model 1: Gradient Boosting

Gradient boosted decision tree models are an example of a boosting ensemble method in which several decision tree models are trained independently then combined, fitting to a loss criterion. Hyperparameters of learning_rate and n_estimators are subject to optimisation in the gradient boosted decision tree, the former influencing the contribution

of each combined tree and the later influencing the number of trees trained in total and thus the time taken to fully train the model

2.2.2.2 - Set 1 Model 2: Random Forest

Random forest decision tree models are an example of averaging ensemble methods developed to produce unbiased estimates from large datasets (Breiman and Cutler, 2001); several models are trained independently, the outputs of which are then averaged to produce a single, more stable output. Similarly to the gradient boosted decision tree models, the n_estimators hyperparameter influences the number of individual models trained and combined.

2.2.3 - Set 2: Neural Network Machine Learning

Neural networks are a highly versatile method of machine learning developed with the intention of simulating the function of neurones in the human brain (F. Rosenblatt, 1957). A neural network consists of one input layer, a number of hidden layers, and an output layer, each layer consisting of n nodes, a simple feed forward neural network with 3 input features, one output value and 2 hidden layers can be seen in Figure 4.

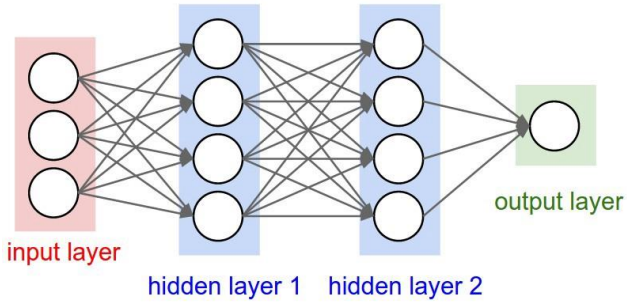


Figure 4: A simple feed forward neural network (Gupta, 2017)

Each input feature takes the form of a node on the input layer, the lines mapping each node to every other node of the following layer represents a matrix multiplication between the

node matrix and a randomly initialised weight matrix.

During a forward pass of the fast.ai neural network, each node will be multiplied against it's corresponding weight matrix, be passed through a rectified linear unit (ReLU) to induce non-linearity to the node values, then scaled; this process will be repeated for each hidden layer of the model.

At the point of creation of a new untrained neural network the weight matrices are randomly initialised, this initialisation state will yield great error after the first forward pass, however, it is through the iterative model training process of training that the values within the weight matrices are updated and the model will begin to fit to the training data.

Models with greater numbers of hidden layers and with more nodes per layer have a greater complexity and will thus behave differently to models with fewer, smaller hidden layers; this will later be explored as a model hyperparameter.

2.2.3.1 - Neural Network Ensemble Methods

A consequence of the random initialisation of weight matrices of the neural network is that it suffered from high variability in the output layer. Two neural networks trained on identical datasets, both randomly initialised, will return very different predictions – these models have a low bias and a high variance. The low bias, high variance nature of neural network machine learning makes it an ideal application for the averaging ensemble methods, the idea being the combine individual models that are "good in different ways [meaning] they must make different prediction errors" - (Brownlee, 2018).

2.2.4 - Data Methods

A consequence of the high dimensionality of the chosen dataset is that different features will have different weighted significances. Naturally, those businesses listed on the NASDAQ stock exchange that have the highest market cap will hold the greatest influence over the NDX^ value compared with those with a lower market capacity. Dimensionality reduction will be carried out on the data to give selective priority to those data features that show the greatest influence in NDX^ value.

Due to the nature of the input data, the values of each feature will have drastically differing means and variances; in training, this would likely result in the model fitting poorly to the training data. This is avoided by scaling the data to ensure that each data feature has a mean and variance equal to 0 and 1 respectively.

3 - Methodology

3.1 - Resources

3.1.1 - Coding Platforms Used

Microsoft's Azure machine learning environment was used by McIntyre (McIntyre, n.d.) and was thus considered for this project, however, whilst the platform offered greatly simplified and intuitive access to advanced machine learning methods, it posed a lack of flexibility in data preparation and hyperparameter selection and was thus decided against in favour of the Python programming language.

Python was chosen as the primary language for this project as it is a widely used and easily accessible, with a wide selection of machine learning and data science libraries and

extensive and active support communities. SQL, R, and Matlab were additionally considered for this project, however, Python is the most likely to be utilised by small to medium sized financial engineering firm and was thus chosen.

Due to the high computational intensity of neural network machine learning, the neural network models must be trained on a computer with significant computational capacity; physical accessibility of such computers was limited, thus, a powerful dedicated machine learning server was rented to carry out much of the experimentation. The Salamander.ai platform was chosen as it offered several servers at different computational capacities and could be accessed via the Jupyter notebook web application. Figure 5 displays the user interface of both the Salamander.ai platform and an example Jupyter Notebook.

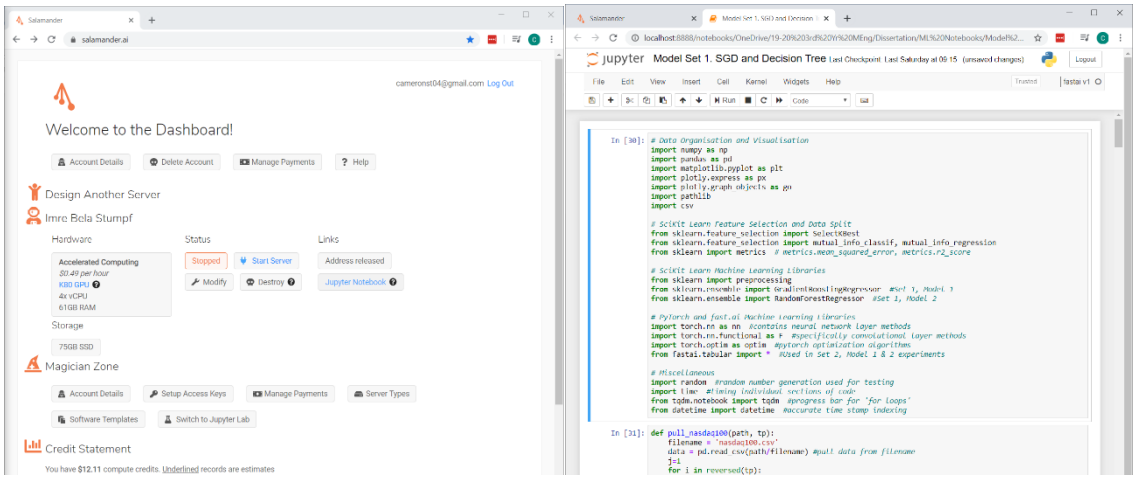


Figure 5: The Salamander.ai online platform (Left) allows access to the Jupyter Notebook web application (Right)

Jupyter Notebooks may also be accessed offline through the Anaconda programme; notebooks were developed offline then uploaded to the Salamander.ai server to execute.

3.1.2 - Libraries Used

All machine learning libraries used within this project can be found in Figure 6.

3.1.2.1 - Data Organisation and Visualisation

The Pandas library is used extensively throughout this project primarily for the DataFrame class; the DataFrame class is used ubiquitously in machine learning experimentation as the primary method of data tabulation. The NumPy library is used primarily for it's numpy-array class and the assignment of NaN (not a number) values used frequently in the experiment code.

The Pathlib and CSV libraries contain methods and functions for directory navigation and for reading data from Excel files respectively.

The Plotly and MatPlotLib libraries are versatile plotting libraries used widely for data analytics in Python. Plotly allows for the creation of simple interactive plots, valuable for testing and investigative data analysis, whilst MatPlotLib, a library used widely in machine learning experimentation, allows for much greater flexibility in plotting styles and structures, valuable for final results presentation.

3.1.2.2 - SciKit Learn library for feature selection and data splitting

The SciKit Learn library is one of the most widely used tabular machine learning and data preparation libraries for Python. The SciKit Learn data preparation functions used in this project are the dimensionality reduction and data normalization methods. Functionality for the mean absolute error and r2_score loss functions are additionally utilised for regression model scoring.

3.1.2.3 - SciKit Learn library for decision tree machine learning

the SciKit Learn library contains additional functionality for decision tree machine learning methods used in Set 1 experiments. The models utilised are the GradientBoostingRegressor and RandomForestRegressor models.

3.1.2.4 - Fast.ai and PyTorch

PyTorch is a machine learning library developed by Facebook primarily for neural network machine learning methods. Whilst the library is developed for the Python language, it's internal functions are run in C, allowing for significantly faster execution of matrix multiplication and gradient descent methods when compared to similar operations written exclusively in Python.

The Fast.ai library is an extension built onto the PyTorch library framework to make the neural network functionality more accessible and easier to optimise, working in more advanced neural network methods to improve model performance. The library contains functionality for more detailed data preparation methods in the form of the Fast.ai data block class.

```
# Data Organisation and Visualisation
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
import pathlib
import csv

# SciKit Learn Feature Selection and Data Split
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif, mutual_info_regression
from sklearn import metrics  # metrics.mean_squared_error, metrics.r2_score

# SciKit Learn Machine Learning Libraries
from sklearn import preprocessing
from sklearn.ensemble import GradientBoostingRegressor  #Set 1, Model 1
from sklearn.ensemble import RandomForestRegressor  #Set 1, Model 2

# PyTorch and fast.ai Machine Learning Libraries
import torch.nn as nn  #contains neural network layer methods
import torch.nn.functional as F  #specifically convolutional layer methods
import torch.optim as optim  #pytorch optimization algorithms
from fastai.tabular import *  #Used in Set 2, Model 1 & 2 experiments

# Miscellaneous
import random  #random number generation used for testing
import time  #timing individual sections of code
from tqdm.notebook import tqdm  #progress bar for 'for loops'
from datetime import datetime  #accurate time stamp indexing
```

Figure 6: Python code used to import all libraries used throughout this project

3.1.3 - Data

The dataset used by McIntyre (McIntyre, n.d.) was obtained from Yahoo Finance which contains functionality for downloading large amounts of free low fidelity historical data at intervals of one day over up to 10 years. More substantial, higher fidelity data may be found form previously conducted machine learning research experimentation, this data is typically pre-processed and requires little manipulation before it can be used in experimentation. The dataset chosen for these experiments was developed and used in the development of a recurrent neural network (Qin et al., 2017) to forecast the behaviour of the NASDAQ100 index. This dataset contains data for over 82 stocks listed on the NASDAQ100 index including the NDX^ itself in minute intervals between July 26th and December 22nd 2016. Figure 7 shows the NDX^ value across the full dataset; the red marker at Time index 33,000 represents the beginning of the designated testing portion of the dataset, the data before this point will be used for training and validation. It can be observed at Time index 38,000 the stock experiences a sudden jump in value; this represents a large market shift and will present a unique challenge for the models.

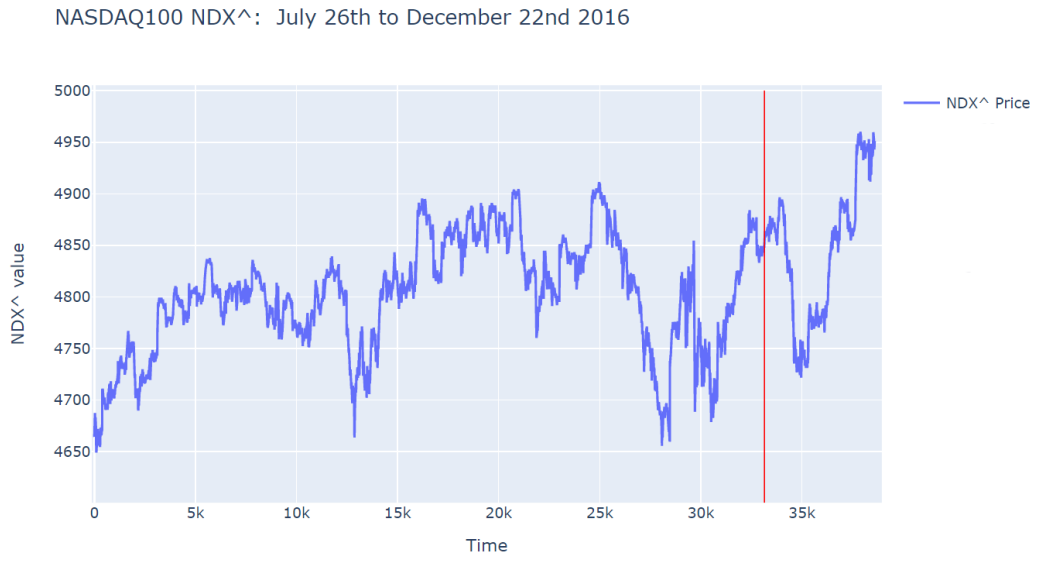


Figure 7: Graphs shows NDX^ values across full dataset, Test dataset begins at Time Index 33,000

The dataset is formatted as a Microsoft Excel spreadsheet with 82 individual columns and over 40,000 rows of data; Table 1 shows a portion of the full dataset in csv format (Excel).

Table 1: Initial dataset format as a .csv Excel file

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	AAL	AAPL	ADBE	ADI	ADP	ADSK	AKAM	ALXN	AMAT	AMGN	AMZN	ATVI	AVGO
2	35.98	96.8	97.8	61.15	95.4	58.18	58	127.04	26.68	165.81	742.12	42.3	161.41
3	35.931	96.89	97.8	61.61	95.4115	58.19	58.12	126.06	26.73	165.9101	741.92	42.44	161.83
4	35.9044	96.95	97.57	61.98	95.51	58.20333	57.95	126.51	26.712	166.18	739.6	42.46	162.5
5	35.89	96.97	97.55	62.09	95.52	58.21667	57.96	126.28	26.74	166.148	739.55	42.52	162.68
6	36.008	96.96	97.73	61.89	95.53	58.23	58.21	126.585	26.72	165.99	738.72	42.61	162.75
7	36.1	97.045	97.88	62.09	95.75	58.225	58.22	126.04	26.735	166.29	738.5	42.6	162.9
8	36.0922	97.095	97.93	62.07	95.75	58.22	58.35	126.4	26.7	165.93	738.49	42.46	163.08
9	36.05	97.145	97.94	62.04	95.73	58.25	58.34	126.41	26.715	166.315	739.68	42.52	163.08
10	36.05	97.23	98	61.99	95.69	58.25	58.37	126.96	26.7	166.169	739.2973	42.51	163.29
11	36.12	97.3	98.015	61.77	95.79	58.295	58.3677	126.74	26.64	166.25	739.125	42.47	162.845
12	36.14	97.24	97.985	61.81	95.84	58.34	58.24	126.645	26.63	166.25	739.38	42.43	162.7
13	36.21	97.3699	97.9792	61.84	95.83	58.385	58.3	126.67	26.63	166.555	739.8	42.38	162.76
14	36.08	97.35	97.98	61.85	95.81	58.38	58.29	126.35	26.65	166.49	739.5822	42.23	162.5407
15	36.1	97.27	97.96	61.93	95.81	58.36	58.36	126.29	26.67	166.8595	739.459	42.14	162.425
16	36.16	97.32	97.8807	62.06	95.89	58.455	58.34	126.225	26.67	166.87	739.1	42.09	162.7
17	36.16	97.46	98.02	62.06	95.85	58.53	58.415	126.53	26.71	166.83	738.97	42.1	162.82
18	36.0734	97.4	98.1075	62.07	95.9101	58.56	58.33	126.74	26.71	166.92	739.5	42.16	163.29

Each column of the dataset represents the close value of an individual stock listed on the NASDAQ stock exchange, and each row represents the value of said stock. According to the dataset documentation: “Each day contains 390 data points except for 210 data points on November 25 and 180 data points on December 22.” (Qin et al., 2017).

3.2 - Methodology

3.2.1 - Data Preparation

Before the NASDAQ100 dataset can be applied it must be imported, converted to the DataFrame format, and prepared through a series of functions written specifically for this project. The dataset is first downloaded from Cseweb.ucsd.edu/ (Qin et al., 2017) as an Excel document and saved as “nasdaq100.csv”; the smaller dataset variant is chosen for this project as, at the cost of dataset length, the data is already pre-processed, requiring no further cleaning for missing values.

3.2.1.1 - pull_nasdaq100

The contents of the nasdaq100.csv Excel file must first be extracted into the pandas DataFrame class for further preparation as the scikit-learn and fast.ai libraries are designed for input data in the pandas DataFrame class. The pull_nasdaq100 function takes two inputs, the directory path of the NASDAQ100 dataset (path), and a list of intervals into the future for which forecasts will later be made (tp).

As forecasts are made at intervals increasing by 195 rows (0.5 days) up to 1950 rows (+5.0 days) for each forecasting model, a duplicate NDX column is created, staggered upwards by 195 rows and concatenated with the main DataFrame to form a target column such that the new target value of row n would be equal to the NDX^ value of row n+195.

```
def pull_nasdaq100(path, tp):
    filename = 'nasdaq100.csv'
    data = pd.read_csv(path/filename) #pull data from filename
    j=1
    for i in reversed(tp):
        k = str(i/390)
        #CREATE STAGGERED TARGET COLUMN  (REGRESSION TARGET)
        target_list = data[target].tolist()  #creates list from target column
        target_list.extend([np.nan for x in range(i)])  #extends list with NaNs for specified time per
        target_list = target_list[i:]  #cuts the list back to size

        #CREATE UP/DOWN TARGET COLUMN  (CLASSIFICATION TARGET)
        classification_list = []
        for i in range(len(target_list)):
            if target_list[i] >= data['NDX'][i]:  item = 1
            else:  item = 0
            classification_list.append(item)

        data.insert(0, 'Classification Target: +'+k+' Days', classification_list) #insert
        data.insert(0+j, 'Regression Target: +'+k+' Days', target_list) #insert
        j+=1
    return data.dropna()  #remove rows with NaN
```

Figure 8: Python code used in pull_nasdaq100 function

3.2.1.2 - feature_selection

The scikit-learn machine learning library contains functionality for feature selection as a method of dimensionality reduction. The feature_selection function takes arguments for an

input DataFrame (data), the number of desired features (num_features), and which application the output dataframe is intended for, either regression or classification (mode). The SelectKBest function selects the <num_features> most relevant data features from the dataset as selected against the mutual_info_regression metric, discarding the remaining columns.

```
def feature_selection(data, num_features, mode):
    #mode either classif or regress
    #split full df into x and y, feature select, to df then rejoin  -  we lose data in column names, bu
    columns,index = data.columns.to_list(), data.index.values  #extract columns and index of x
    if mode == "classification":
        data = data.drop(data.columns[range(10,20)], axis=1)  #drop the regression targets
        X,y = data.drop(data.columns[range(0,10)], axis=1), data[data.columns[range(0,10)]]
        X = SelectKBest(mutual_info_classif, k=num_features).fit_transform(X,y.iloc[:,0])  #this reduce

    if mode == "regression":
        data = data.drop(data.columns[range(0,10)], axis=1)  #drop the classification targets
        X,y = data.drop(data.columns[range(0,10)], axis=1), data[data.columns[range(0,10)]]
        X = SelectKBest(mutual_info_regression, k=num_features).fit_transform(X,y.iloc[:,0])  #this rec

    X = pd.DataFrame(X, index=index)
    data = pd.concat([y, X], axis=1, sort=False)
    return data
```

Figure 9: Python code used in feature_selection function

3.2.1.3 - sequence_input

For the sequenced input neural network experiments of Set 2 Model 2, within which a neural network is fed with data from multiple time periods as a single input, a single row (n) of the input DataFrame must contain the independent variables of time n in addition to the stock market values of time , n-x, n-2x, n-3x, etc. where x represents a constant time interval spacing. The sequence_input function receives arguments for the dataset to be modified (data), how many time points are to be contained in a single row of data (sequence_length), and the row spacing between time points (sequence_spacing). Similarly to the pull_nasdaq100 function, the independent variable columns are duplicated, staggered downwards by <sequence_spacing> rows, and re-joined with the full dataset; this is repeated <sequence_length> times.

```
def sequence_input(data, sequence_length, sequence_spacing):
    df_main, df_x = data.iloc[:,:10], data.iloc[:,10:]
    nan_unit = pd.DataFrame(np.nan, columns=df_x.columns, index=range(sequence_spacing)) #create nan row df
    y_cols = df_main.columns.tolist()

    for i in range(sequence_length):  #for i we create and concat a new temp_x
        nan_block = nan_unit.iloc[:0, :]  #create
        for j in range(i):   #create nan block to add to start of temp_x
            nan_block = pd.concat([nan_block, nan_unit], ignore_index=True)
        temp_x = pd.concat([nan_block, df_x], ignore_index=True)  #make overlength x
        df_main = pd.concat([df_main, temp_x], axis=1, ignore_index=True)
        df_cols = y_cols + list(range(len(df_main.columns)-10))
        df_main.columns = df_cols
    return df_main.dropna()
```

Figure 10: Python code used in sequence_input function

3.2.1.4 - split

The full dataset must be split into training and testing datasets; the former being used to train and validate the learning model whilst the latter is used to generate the experimental forecasts and scoring. 5000 rows of data are used for the test set, leaving over 35000 rows of data for training and validation.

```
def split(data):   #split data into train, valid and test sets
    data_train, data_test = data[:-5000], data[-5000:]
    return data_train, data_test
```

Figure 11: Python code used in split function

3.2.1.5 - scale

Normalization and feature scaling methods are applied to the dataset through the SciKit-Learn scale functionality. The testing dataset must be scaled using a scaler fit to the train dataset as otherwise the testing dataset would influence the scaled training data, causing artificially high accuracies.

```python
def scale(data_train, data_test):
    #set up normalization on train, apply to valid and test
    train_x = data_train.iloc[:, 10:]  #pull x values from train data
    scaler = preprocessing.StandardScaler().fit(train_x) #fit scale model to train set

    data, scaled_data = [data_train, data_test], []
    for df in data:
        df_y, df_x = df.iloc[:, :10], df.iloc[:, 10:]  #splits data into y and x columns
        columns,index = df_x.columns.to_list(), df_x.index.values  #extract columns and index of x
        x_scaled = scaler.transform(df_x)  #returns np.array
        df_x = pd.DataFrame(x_scaled, columns=columns, index=index)
        df = pd.concat([df_y, df_x], axis=1, sort=False)
        scaled_data.append(df)

    return scaled_data[0], scaled_data[1]  #train, valid, test
```

Figure 12: Python code used in scale function

3.2.1.6 - truncate

Data availability experiments will consider the effect of limited training data availability on each machine learning model, thus, a function is written to reduce the size of the train dataset. It receives arguments for the DataFrame to be reduced (data), and the targeted length of the DataFrame in rows (n_rows).

```python
def truncate(data, n_rows):
    #REDUCE df_nasdaq100 INTO THE SMALLER dataframe
    data = data.iloc[-n_rows:]
    return data
```

Figure 13: Python code used in truncate function

3.2.1.7 - split_Xy

The SciKit-Learn decision tree experiments require the training and test datasets to each be split into separate DataFrames for independent and dependent variables, this split is conducted via the split_Xy function which takes in arguments for both the train and test DataFrames (data_train and data_test respectively) and returns 4 DataFrames, each containing independent and dependant variables for the train and test datasets.

```
def split_Xy(data_train, data_test):
    data_list = [data_train, data_test]
    X_list, y_list = [], []
    for data in data_list:  X_list.append(data.iloc[:, 10:]), y_list.append(data.iloc[:, :10])
    return [X_list[0],y_list[0]],  [X_list[1],y_list[1]]
```

Figure 14: Python code used in split_Xy function

3.2.2 - Model Scoring and Results Plotting

The intended output format of a single experiment is a single DataFrame with 11 columns and 5,000 rows, this output table will contain the predicted NDX^ behaviour made 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, and 5.0 days in advance, the values in this DataFrame are to be scored against a second DataFrame of identical dimensions containing true NDX^ values; these two DataFrames are referred to as df_pred and df_targ respectively. A portion of the df_pred DataFrame can be seen in Table 2, the df_targ DataFrame is of similar format.

Table 2: Example of df_pred prediction DataFrame, dimensions: 5000 rows x 11 columns

	NDX	Regression Prediction: +0.5 Days	Regression Prediction: +1.0 Days	Regression Prediction: +1.5 Days	Regression Prediction: +2.0 Days	Regression Prediction: +2.5 Days	Regression Prediction: +3.0 Days	Regression Prediction: +3.5 Days	Regression Prediction: +4.0 Days	Regression Prediction: +4.5 Days	Regression Prediction: +5.0 Days
33610	4870.649	4867.151526	4888.924928	4846.850135	4833.598930	4774.877774	4735.870290	4730.091577	4735.872060	4775.425663	4775.389800
33611	4870.848	4867.084527	4888.775509	4848.342041	4833.678805	4774.770194	4735.596285	4731.317690	4735.777061	4774.432101	4775.339446
33612	4870.985	4867.006437	4888.767917	4849.639998	4833.967542	4777.574381	4735.692022	4733.197900	4735.880009	4773.819898	4775.399648
33613	4871.131	4867.332321	4888.902598	4849.292662	4834.206414	4775.570678	4736.172013	4732.922359	4735.722466	4773.839550	4775.666929
33614	4871.014	4867.047193	4888.857446	4849.489089	4834.062986	4776.918706	4736.112444	4733.101907	4735.645050	4763.524486	4775.544962
...	...	...	...	...	...	...	...	...	...	...	...
38605	4948.508	4863.977200	4856.912242	4878.157984	4875.111633	4812.167821	4796.150096	4684.051999	4702.077479	4785.035719	4791.456022
38606	4948.896	4863.978118	4856.911909	4878.156589	4875.113715	4812.167259	4795.950598	4684.051999	4702.088756	4784.973266	4795.451663
38607	4948.945	4863.977415	4856.911970	4878.160307	4875.112926	4812.169606	4795.935544	4684.051999	4702.020705	4784.980395	4791.293833
38608	4948.065	4863.961642	4856.912304	4878.155691	4875.103790	4812.177998	4796.153790	4684.051999	4702.050664	4785.032549	4786.249532
38609	4949.618	4863.952670	4856.911909	4878.148639	4875.113882	4812.177840	4798.394700	4684.051999	4702.088756	4785.013880	4791.381039

5000 rows × 11 columns

3.2.2.1 - score_regress

The accuracy of the trained models is calculated and returned via the score_regress function, which will receive arguments for a list of target values (target) and a list of

corresponding predictions made by the trained model (prediction). The model returns a list containing the coefficient of determination and the mean squared error between the values in the two lists.

```
def score_regress(target, prediction):  #target is as a series, prediciton is as an np.array
    for i in range(1, len(target)):
        score_list  = [metrics.r2_score(target, prediction),
                       np.sqrt(metrics.mean_squared_error(target, prediction))]  # [r2, mse]
    return score_list
```

Figure 15: Python code used in score_regress function

3.2.2.2 – plot_col and plot_row

Two methods of forecast visualisation are used, one to plot a single column of predicted values from one of the 10 trained models in df_pred against its corresponding target values in df_targ, and the other to plot values from each of the 10 trained models from a single row of df_pred and df_targ DataFrames to compare the predicted and actual behaviour of said stock over the 5 days following on from the chosen time point. Both plot_col and plot_row functions receive arguments for df_targ (targ) and df_pred (pred) as well as (col) for column index and (row) for row index respectively.

```
def plot_col(targ, pred, col):  #pring regression results
    #plot specified columns in pred vs. targ
    x_axis = list(range(len(df_test)))
    # Data for plotting

    for i in range(len(col)):
        fig, a = plt.subplots()  #1,len(col))
        a.plot(x_axis, targ.iloc[:, col[i]], label='NDX^ Target Value')
        a.plot(x_axis, pred.iloc[:, col[i]], label='NDX^ Predicted Value')
        a.set(xlabel='Test Dataset Index', ylabel='NDX^ Value',
              title='Forecast: +'+str((tp[col[i]-1])/390)+' Days')
        a.legend(loc='upper right')
        #plt.tight_layout()
        plt.show()


def plot_row(targ, pred, row):
    fig = go.Figure()
    x_axis = ['NDX Base Truth']
    for i in pred.columns.tolist()[1:]: x_axis.append(i.split(" ", 2)[2])

    for j in range(len(row)):
        fig, a = plt.subplots() #1,len(row))
        a.plot(x_axis, targ.iloc[row[j]], label='NDX^ Target Value')
        a.plot(x_axis, pred.iloc[row[j]], label='NDX^ Predicted Value')
        a.set(xlabel='Time (Days)', ylabel='NDX^ Value',
              title='0.5-5 Day Forecast: Row Index: '+str(row[j]))
        a.legend(loc='upper right')
        a.set_xticklabels(x_axis, rotation=40, ha='right')
        plt.show()
```

Figure 16: Python code used in plot_col and plot_row functions

3.2.3 - Set 1: Decision Trees and SciKit Learn

Experiment parameters are first defined in addition to lists that will be appended to with predictions and scores. The dataset is import and prepared using the functions described in section 3.2.2; additionally, the input and target columns of the DataFrame are separated for later use in the creation of the target and prediction DataFrames

```
#####
## Set 1, Models 1 and 2
#####

tp = [195, 390, 585, 780, 975, 1170, 1365, 1560, 1755, 1950]  #10 models up to 5 days
input_features = 20
learning_rate = 0.1
data_rows = 30000
n_estimators = 100

s1_m1 = []
s1_m2 = []

df_full = pull_nasdaq100(path, tp)  #import data with y_c and y_r cols
df = feature_selection(df_full, input_features, 'regression')  #reduce columns
df_train, df_test = split(df)  #split train/valid/test
df_train, df_test = scale(df_train, df_test)  #normalize and scale data
df_train = truncate(df_train, data_rows)
train_R, test_R = split_Xy(df_train, df_test)
[X_train,y_train], [X_test,y_test] = train_R, test_R

input_list = df_train.columns.tolist()[10:]  #isolate list of inputs
target_list = df_train.columns.tolist()[:10]  #list of target column names
ndx_df = pd.DataFrame(df_full['NDX'])  #create df with just ndx
ndx_df.rename(columns={"NDX": "NDX Base Truth"})  #rename ndx column

##################
```

Figure 17: SciKit-Learn decision tree (Set 1) experiment code: initialisation

Specific target (df_targ), prediction (df_pred), and score (df_score) DataFrames are created for each model, these will be appended to later with predicted values and model scores. The gradient boosting model is optimised towards the Friedman mean squared error (friedman_mse) criterion, a variant of the standard MSE that is optimised for gradient boosting applications. Experimental hyperparameters of n_estimators and learning_rate are defined at this point.

```
##############################

#Set 1, Model 1
df_targ = pd.concat([ndx_df, y_test], axis=1).dropna()  #creates target dataframe, dropna to remove excess ndx
df_pred = df_targ.iloc[:, :1]  #creates empty prediction dataframe
df_score = pd.DataFrame(index=["R^2 Score", "Mean Square Error"])  #create score df


for column in tqdm(target_list, leave=False):  #train for each target column
    col_num = column.split(" ", 2)[2]

    model = GradientBoostingRegressor(loss='ls', learning_rate=learning_rate, n_estimators=n_estimators, subsample=1.0,
                                      criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1,
                                      min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0,
                                      min_impurity_split=None, init=None, random_state=None, max_features=None,
                                      alpha=0.9, verbose=0, max_leaf_nodes=None, warm_start=False, presort='deprecated',
                                      validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0)

    model.fit(X_train, y_train[column])   #fit on y column of this for loop
    y_pred = model.predict(X_test)  #predict and release as an array
    df_pred.insert(len(df_pred.columns), "Regression Prediction: "+col_num, y_pred) #join df_pred with predictions
    score = score_regress(y_test[column], y_pred)  #score model  (series, np.array)
    df_score.insert(len(df_score.columns), col_num, score )  #insert score into score df

s1_m1.append([df_targ, df_pred, df_score, tag])


##############################
```

Figure 18: SciKit-Learn Gradient Boosted Decision Tree (Set 1 Model 1) experiment code: training and prediction

The RandomForestRegressior model's training and prediction process is identical to that of the GradientBoostingRegressor aside from the different initialisation function. Once again, the mean squared error is used as the loss criterion against which the tree is fit to. The experimental hyperparameter of n_estimators is defined at this point.

```
##############################

#Set 1, Model 2:  Random Forest
df_targ = pd.concat([ndx_df, y_test], axis=1).dropna()  #creates target dataframe, dropna to remove exc
df_pred = df_targ.iloc[:, :1]  #just ndx column, will be added to later on
df_score = pd.DataFrame(index=["R^2 Score", "Mean Square Error"])  #create score df

for column in tqdm(target_list, leave=False):  #initialise model 2 learners
    col_num = column.split(" ", 2)[2]  #exctract iteration name

    model = RandomForestRegressor(n_estimators=100, criterion='mse', max_depth=None,
                                  min_samples_split=2, min_samples_leaf=1,
                                  min_weight_fraction_leaf=0.0, max_features='auto',
                                  max_leaf_nodes=None, min_impurity_decrease=0.0,
                                  min_impurity_split=None, bootstrap=True, oob_score=False,
                                  n_jobs=None, random_state=None, verbose=0, warm_start=False,
                                  ccp_alpha=0.0, max_samples=None)

    model.fit(X_train, y_train[column])   #fit on y column of this for loop
    y_pred = model.predict(X_test)  #predict and release as an array
    df_pred.insert(len(df_pred.columns), "Regression Prediction: "+col_num, y_pred) #join df_pred with
    score = score_regress(y_test[column], y_pred)  #score model  (series, np.array)
    df_score.insert(len(df_score.columns), col_num, score )  #insert score into score df

s1_m2.append([df_targ, df_pred, df_score])  #returns df_pred, df_targ, and df_score
```

Figure 19: SciKit-Learn Randomized Forest (Set 1 Model 2) experiment code: training and prediction

For each of the model types in Set 1 10 individual models are trained, forecasting over intervals of up to 5 days. The trained values from each model are appended to the df_pred DataFrames, the scores of each individually trained model is calculated and appended to the df_score DataFrames.

3.2.4 - Set 2: Neural Networks and Fast.ai

Similarly to the models of Set 1, the neural network experiments begin by first defining the experimental hyperparameters that will be used in each experiment, then by importing and preparing the datasets.

```
filename = 'nasdaq100.csv'      # data file to be used
path = pathlib.Path.cwd()  #path for salamander.ai
target = 'NDX'  #target column name to be predicted
tp  = [195, 390, 585, 780, 975, 1170, 1365, 1560, 1755, 1950]  #10 models up to 5 days

input_features = 20
data_rows = 30000
layers = [1000,1000,1000]  #neural net hidden layer arrangement
s2_m1 = []  #list that will be appended with experiment results

#CREATE AND PREPARE DATAFRAME
df_full = pull_nasdaq100(path, tp)  #import data with y_c and y_r cols
df_selected = feature_selection(df_full, input_features, 'regression')  #reduce columns and cut classif
df_train, df_test = split(df_selected)  #split train/test
df_train, df_test = scale(df_train, df_test)  #normalize and scale data
df_train = truncate(df_train, data_rows)
#train_R, test_R = split_Xy(df_train, df_test)  #split data betwen X & y

################################
```

Figure 20: Fast.ai Single Input Neural Network (Set 2 Model 1) experiment code: initialisation

The training dataset is further split into training and validation at a ratio of 80:20 the validation set will be used to assess the convergence of the model during training. A batch size of 64 is chosen as the default value, this will not be varied during these experiments. The training metrics of coefficient of determination (r2_score) and mean squared error are chosen as the validation metrics, these will be returned during the training process to gauge the convergence of the train and validation.

Similarly to the decision tree models of experiment Set 1, 10 ensemble models will be trained, forecasting in intervals increasing up to 5 days in advance; each ensemble model will be the averaged results of 5 neural network models trained with the same hyperparameters and on the same training dataset.

The Fast.ai TabularDataBunch function is supplied with the subject DataFrame and arguments for independent and dependant variable columns to create an object of the Fast.ai DataBunch class which will then be used to create the Fast.ai Learner model. Learning models are created for each of the 10 forecasting models which are appended to

the learner_list.

Each individual learner may now be trained using the fit_one_cycle function. This function receives arguments for number of training epochs and the desired learning rate as a value between 0 and 1 in standard form; the selected learning rate may be modulated to return higher training accuracy, however, as learning rate optimisation for neural network training lies outside of the scope of this project, a learning rate of 3e-3 was chosen as a constant value across all experimentation. Figure 21 reflects the training cycle of 6 training epochs of a neural network; it can be seen that each epoch results in lower train and valid loss and reduced error rates.

epoch	train_loss	valid_loss	r2_score	root_mean_squared_error	time
0	22731882.000000	22891108.000000	-3144126.500000	4784.258789	00:03
1	13613126.000000	12437185.000000	-1717497.125000	3526.280029	00:03
2	2568140.750000	1948321.625000	-273394.531250	1395.081787	00:03
3	52477.675781	42977.156250	-6435.310547	204.167496	00:03
4	128.286560	7668.255859	-1459.211792	74.174698	00:03
5	61.396290	5021.986816	-844.039062	62.358093	00:03

Figure 21: 6 training epochs of a neural network

Following the successful training of each individual model, the trained model is applied to the test dataset through the predict function. The output values from this function are then appended to the df_pred DataFrame.

```
################################

s2_m1_list=[]  #iterations for averaging appended to this list
for itt in range(5):
    #params for databunch
    df = df_train  #df is train and validation combined
    valid_idx = range(int(0.8*data_rows),data_rows)  #validation range within df
    test_df = df_test
    bs=64
    train_metrics = [r2_score, root_mean_squared_error]

    #CREATE LIST OF LEARNERS, ONE PER ITEM IN TP
    input_list = df_train.columns.tolist()[10:]   #isolate inputs
    target_list = df_train.columns.tolist()[:10]   #isolate targets
    learner_list = []   #to be appended to with learners
    i = 0
    for column in target_list:   #number of models we're going for
        df = pd.concat([df_train[column], df_train[input_list]], axis=1, sort=False)
        data = TabularDataBunch.from_df(path, df=df, dep_var=column, valid_idx=valid_idx, bs=bs)
        learner_list.append(tabular_learner(data, layers=layers, metrics=train_metrics))  #create learner


    #Train Phase 1
    for learn in learner_list:
        learn.fit_one_cycle(6,3e-3)  #fast train initially
    print("all models trained for iteration ", itt+1 , "!")

    #Create prediction and target dataframes
    ndx_df = pd.DataFrame(df_full['NDX'])  #create df with just ndx
    ndx_df.rename(columns={"NDX": "NDX Base Truth"})  #rename ndx column

    df_targ = pd.concat([ndx_df, df_test.iloc[:, :10]], axis=1).dropna()  #creates target dataframe, dropna
    df_pred = df_targ.iloc[:, :1]  #just ndx column, will be added to later on
    df_score = pd.DataFrame(index=["R^2 Score", "Mean Square Error"])  #create score df

    X = df_test.iloc[:, 10:]  #isolates inputs from test dataframe
    i = 0  #ticker indicates the half days
    for learn in learner_list:
        y_pred = []  #every predicted value is appended to this list
        for j in range(0, len(df_test)):  #run through length of test_df
            y = learn.predict(X.iloc[j])[1]  # calls the current colmn
            y_pred.append(y.item())  #extract value from torch.tensor and append to test
        df_pred.insert(len(df_pred.columns), 'Regression Prediction: +'+str(tp[i]/390)+' Days', y_pred)

        score = score_regress(df_targ.iloc[:, i+1], y_pred)  #score model  (series, np.array)
        df_score.insert(len(df_score.columns), str(tp[i]/390)+' Days', score )  #insert score into score df
        i +=1

    s2_m1_list.append([df_targ, df_pred, df_score])
    print(itt+1 , " Complete!")

####################
```

Figure 22: Fast.ai Single Input Neural Network (Set 2 Model 1) experiment code: training and prediction

Following the completion of 5 iterations of individually trained forecasting models, the predicted values in each of the df_pred DataFrames are averaged across all 5 iterations. The resulting values are appended to a new dataframe which is then scored against the df_targ DataFrame. The resulting DataFrames are then appended to the s2_m1 experiment list.

```
####################

#AVERAGE RESULTS OF ALL 5 ITERATIONS
averaged_pred = pd.DataFrame(0, index=range(len(s2_m1_list[0][1])), columns=s2_m1_list[0][1].columns)
for i in range(len(s2_m1_list[0][1].columns)):  #go through each column
    for j in range(len(s2_m1_list[0][1].iloc[:,0])):  #go through each element in the column
        val = 0  #this will be summed up
        for k in range(len(s2_m1_list)):
            val += s2_m1_list[k][1].iloc[j,i]
        val = val/len(s2_m1_list)  #averagted out
        averaged_pred.iloc[j,i] = val   #swap av_pred value with val

df_score = pd.DataFrame(index=["R^2 Score", "Mean Square Error"])  #create score df
for i in range(len(averaged_pred.columns[1:])):
    score = score_regress(df_targ.iloc[:, i+1], averaged_pred.iloc[:,i+1])  #score model  (series, np.array,
    df_score.insert(len(df_score.columns), str(tp[i]/390)+' Days', score )  #insert score into score df

s2_m1.append([df_targ, averaged_pred, df_score, tag])
```

Figure 23: Fast.ai Single Input Neural Network (Set 2 Model 1) experiment code: prediction averaging

The sequenced input neural network experiments (Set 2 Model 2) are conducted similarly to those of the single input neural network (Set 2 Model 1), however, with definition of hyperparameters for sequence length and sequence separation, and in the preparation of the dataset the sequence_input function is used.

```
filename = 'nasdaq100.csv'     # data file to be used
path = pathlib.Path.cwd()  #path for salamander.ai
target = 'NDX'  #target column name to be predicted
tp  = [195, 390, 585, 780, 975, 1170, 1365, 1560, 1755, 1950]  #10 models up to 5 days

input_features = 20
data_rows = 30000
seq_len = 2
seq_sep = 200
layers = [1000,1000,1000]  #neural net hidden layer arrangement
s2_m2 = []  #list that will be appended with experiment results

#CREATE REGRESSION DATA SETS
df_full = pull_nasdaq100(path, tp)  #import data with y_c and y_r cols
df_mod1 = feature_selection(df_full, input_features, 'regression')  #reduce columns and cut classif
df_mod2 = sequence_input(df_mod1, seq_len, seq_sep)
df_train, df_test = split(df_mod2)  #split train/valid/test
df_train, df_test = scale(df_train, df_test)  #normalize and scale data
df_train = truncate(df_train, data_rows)
#train_R, test_R = split_Xy(df_train, df_test)  #split data betwen X & y

#################################
```

Figure 24: Fast.ai Sequenced Input Neural Network (Set 2 Model 2) experiment code: initialisation

4 - Experimentation and Results

Hyperparameter experiments are conducted with 20 unique input features and 30,000 rows of data with forecasts made at a range of +5.0 days. Results are presented as column plots which indicate the closeness of predicted to target NDX^ behaviour.

Data Availability experiments are conducted across forecast ranges up to +5.0 days. Results are presented as two column plots for +0.5 day forecasts and +5.0 day forecasts, and as two row plots for rows indexes 1000 and 4500 of the test datasets.

Coefficient of determination (r2_score) and mean squared error (MSE) score tables can be found in Appendix A.

4.1 - Hyperparameter selection

The hyperparameters of each learning model represent elements of their setup that will influence the accuracy of their forecasts. The intention of these experiments is to evaluate the sensitivity of each model's accuracy to its setup conditions.

4.1.1 - Gradient Boosted Decision Tree Experiments (Set 1 Model 1):

n_estimators and learning rate

The primary hyperparameters of the gradient boosted decision trees are the model's learning rate and training epochs. Nine experiments are carried out with learning rates of 0.01, 0.1, and 1 at n_estimator values of 100, 200, and 300. Results are displayed in Figure 25.

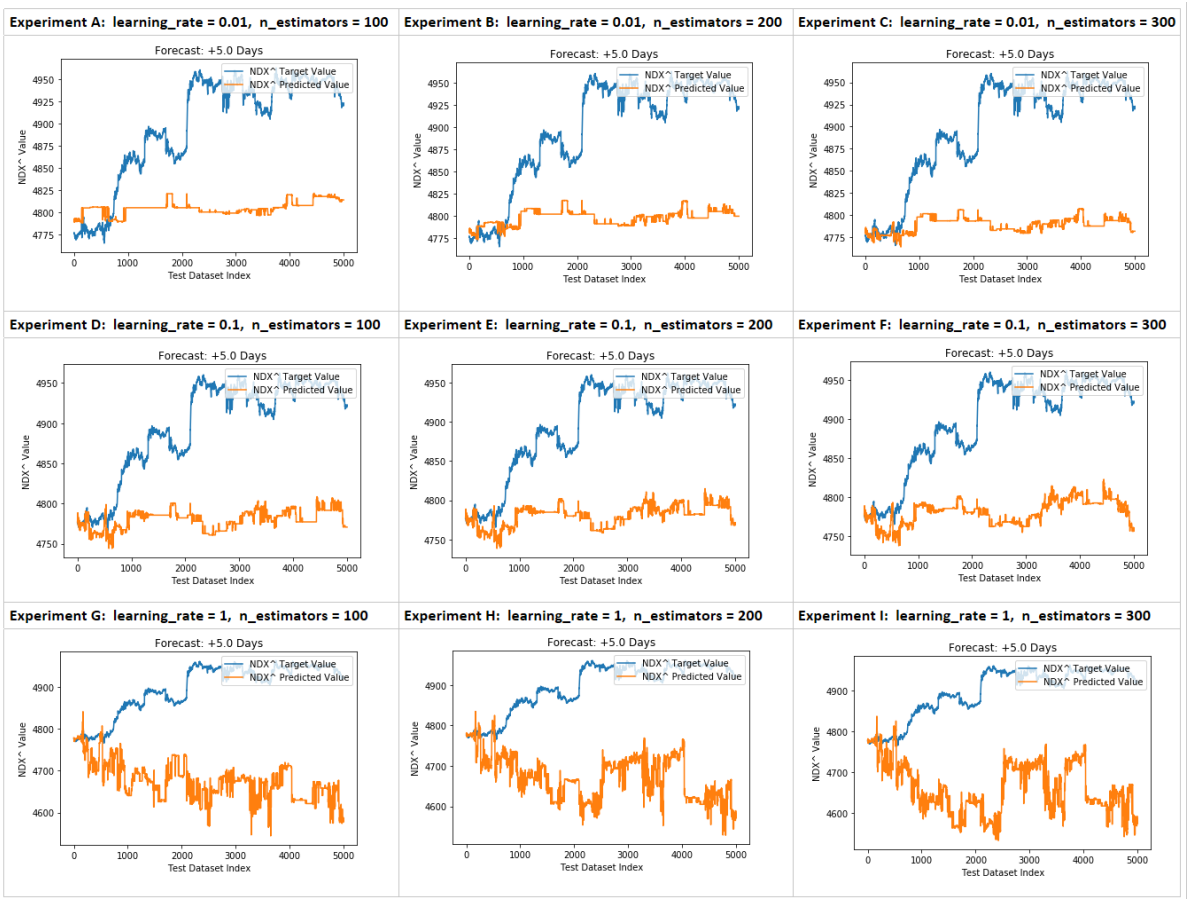


Figure 25: Gradient Boosted Decision Tree (Set 1 Model 1) estimator quantity and learning rate experiment graphs

4.1.2 - Random Forest Experiments (Set 1 Model 2): n_estimators

The random forest decision tree models are trained with n_estimators of 100, 200, and 300.

The results are presented in Figure 26.

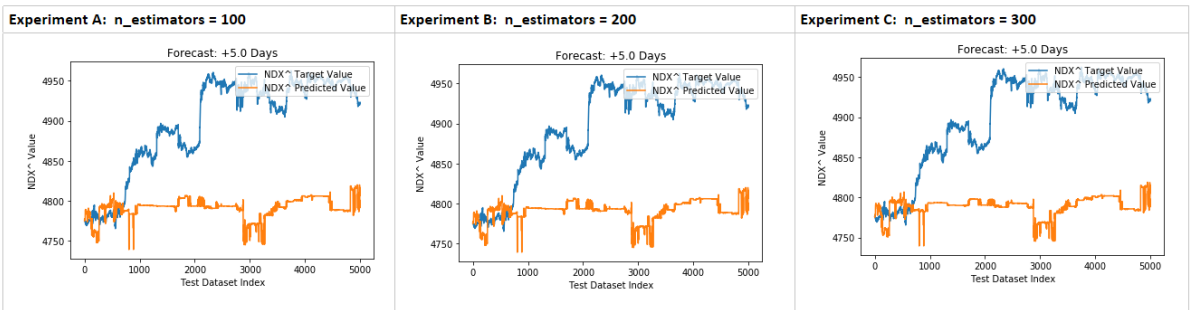


Figure 26: Randomized Forest (Set 1 Model 2) estimator quantity and learning rate experiment graphs

4.1.3 - Sequenced Input Neural Network Experiments (Set 2 Model 2): seq_len and seq_sep

Sequenced input neural network hyperparameter selection experiments are conducted to evaluate the effect of sequence length and spacing of the multiple time series inputs. Nine experiments are conducted across sequence lengths (seq_len) of 2, 4, and 6 and sequence separations (seq_sep) of 100, 200, and 300 rows, the results are shown in Figure 27.

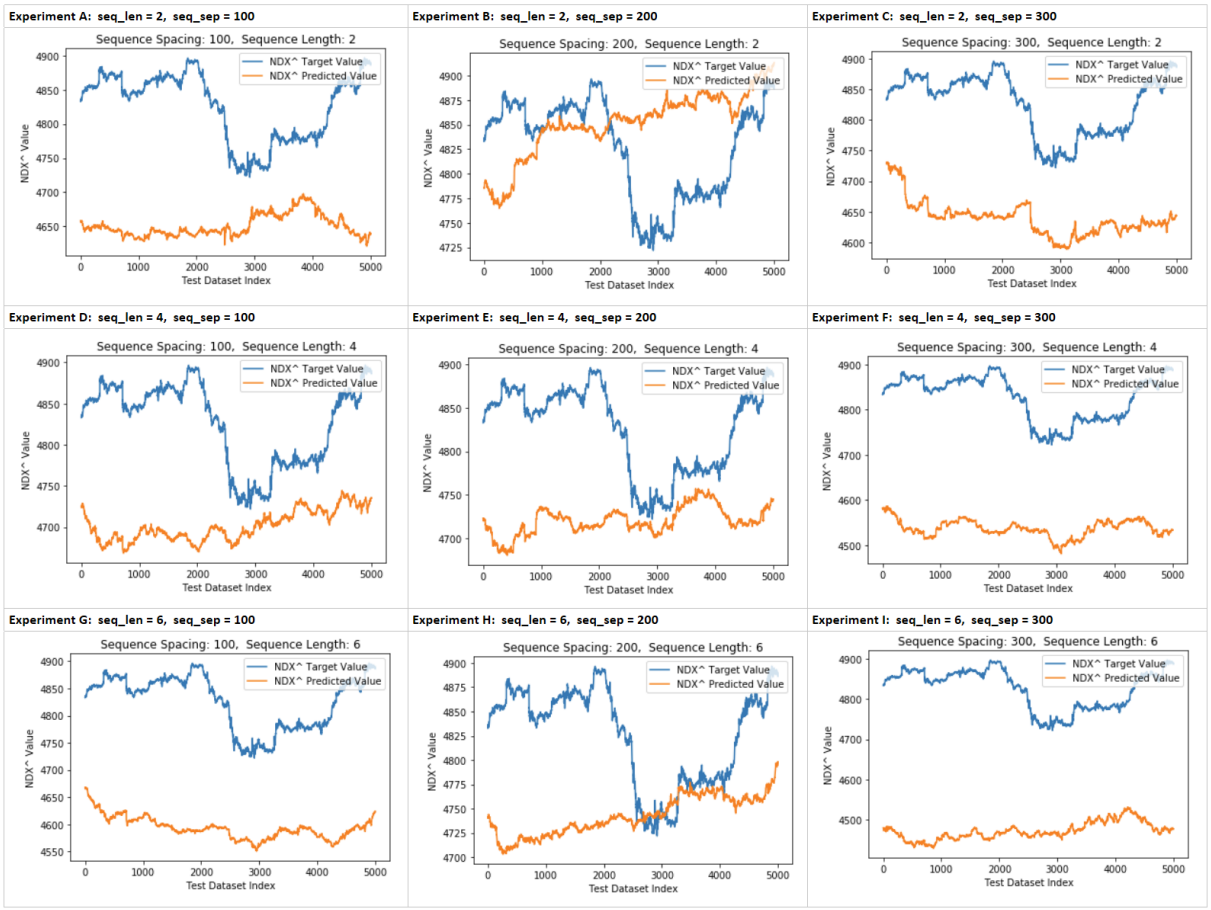


Figure 27: Sequenced Input Neural Network (Set 2 Model 2) Sequence Length and Separation experiment graphs

4.1.4 - Single Input Neural Network Experiments (Set 2 Model 1): Hidden Layer Dimensions

The hidden layer dimensions of each of the model variants in set 2 are defined at the point of learner initialisation through a list argument that controls the number of hidden layers and the number of nodes in each. 12 experiments are conducted with 1, 2, 3, and 4 layers over layer sizes of 500, 1000, and 1500 nodes per layer; the results of these experiments are shown in Figure 28

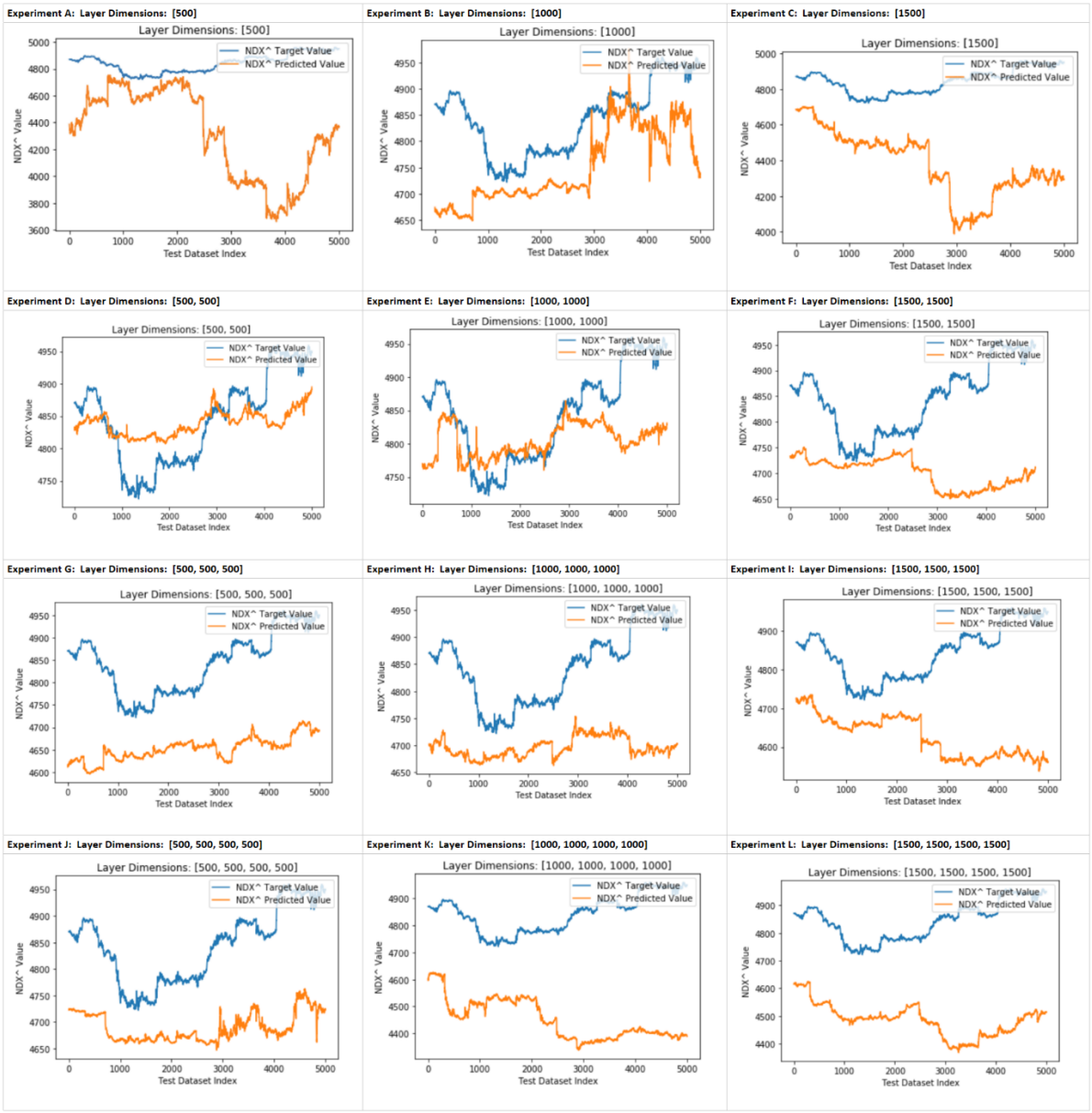


Figure 28: Single Input Neural Network (Set 2 Model 1) hidden layer dimension experiment graphs

4.1.5 - Sequenced Input Neural Network Experiments (Set 2 Model 2): Hidden Layer Dimensions

Identical hidden layer dimension experiments are carried out on the sequenced input neural

network model. 12 experiments with 1, 2, 3, and 4 layers over layer sizes of 500, 1000, and 1500 nodes per layer. The results of these experiments are shown in Figure 29.

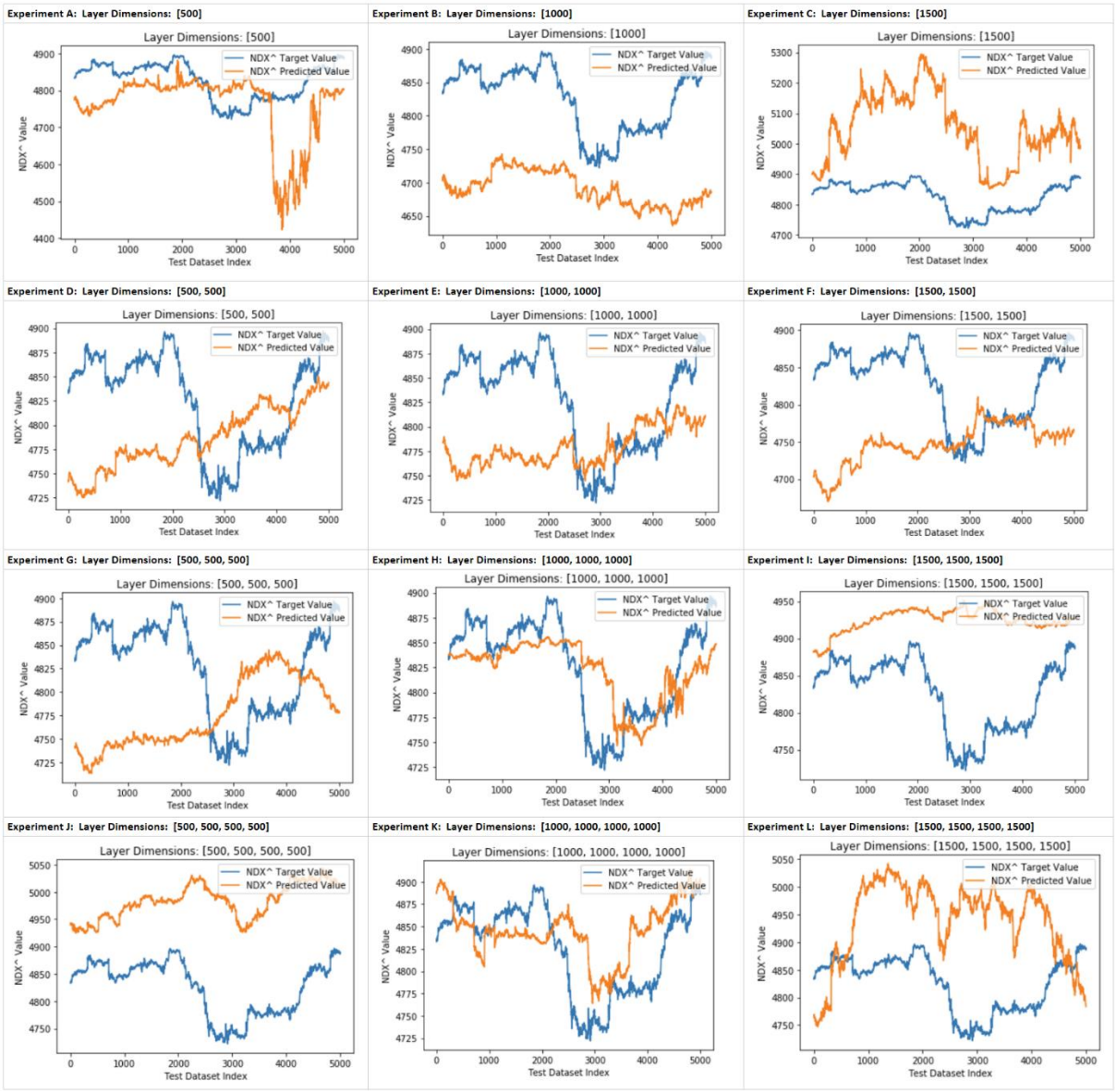


Figure 29: Sequenced Neural Network (Set 2 Model 2) hidden layer dimension experiment graphs

4.2 - Data Availability Experiments

For each machine learning model variant, 4 experiments are conducted to assess the relative effects of reduced data availability. Experiments are conducted on datasets of

15000 and 30000 rows of data across 20 and 80 unique data features. The results for the Set 1 gradient boosted and random forest models are shown in Figures 30 and 31 respectively; The results for Set 2 single input and sequence input neural network models are shown in Figures 32 and 33 respectively. Coefficient of determination (r2_score) and mean squared error (MSE) score tables for each experiment may be found in Appendix A.

Figure 30: Gradient Boosted Decision Tree (Set 1 Model 1) data availability experiment graphs

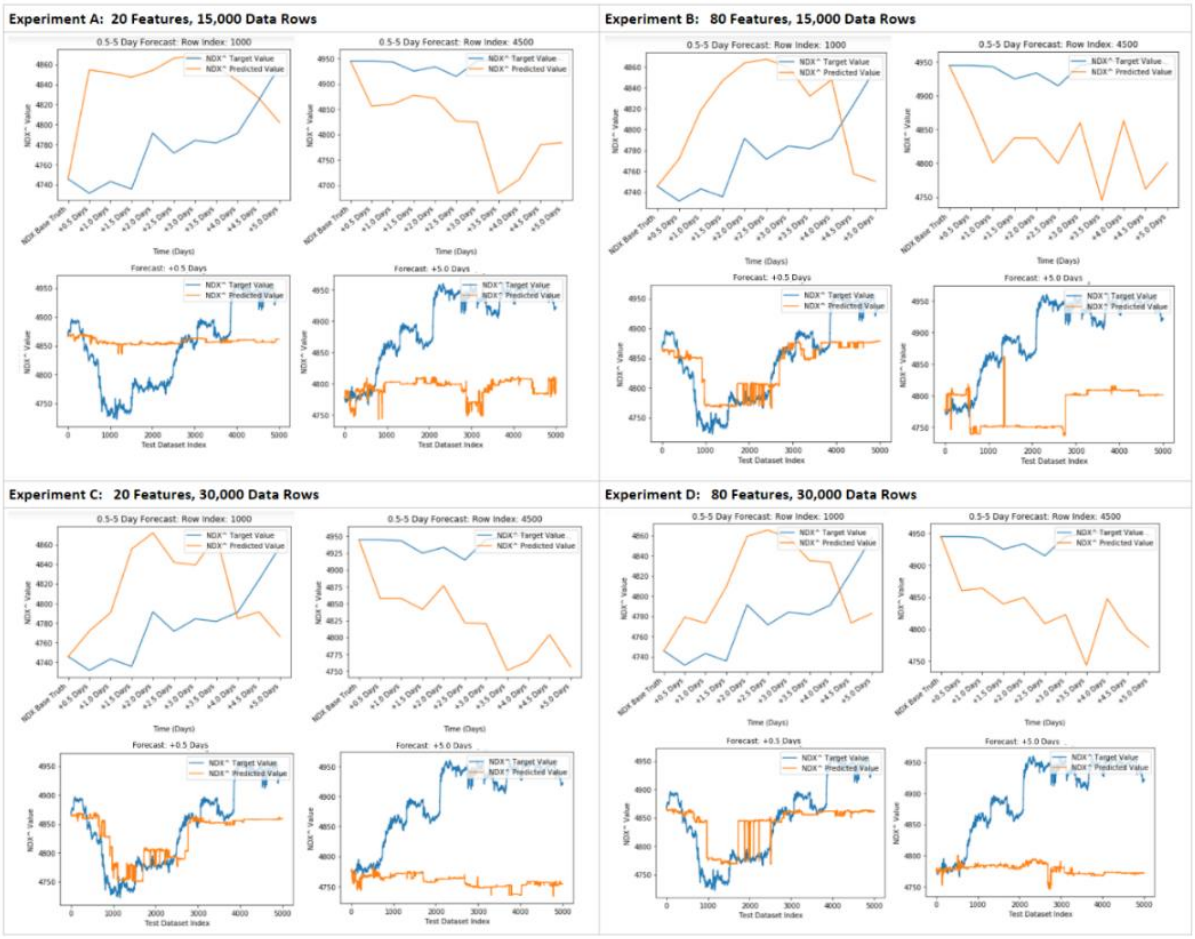


Figure 31: Randomized Forest (Set 1 Model 2) data availability experiment graphs

Figure 32: Single Input Neural Network (Set 2 Model 1) data availability experiment graphs

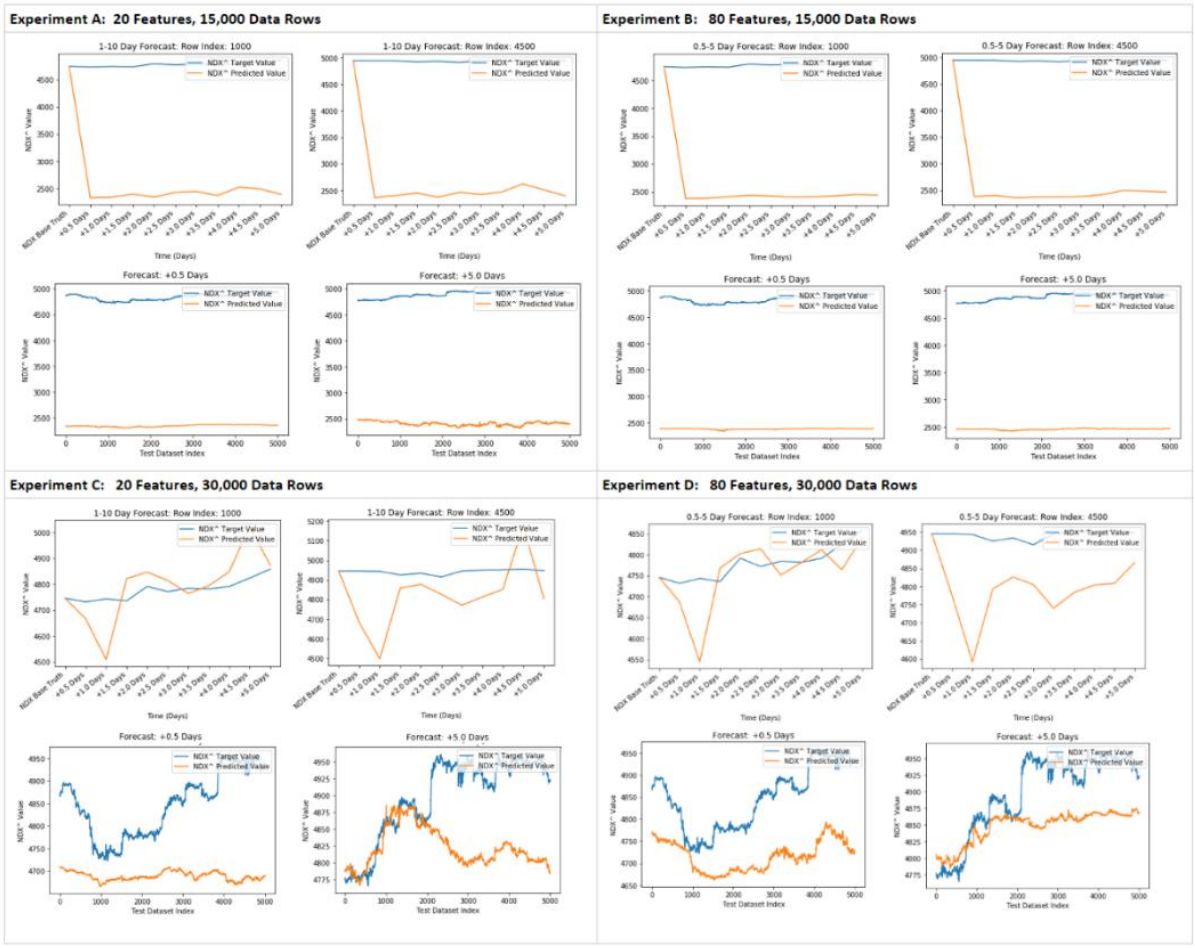


Figure 33: Sequenced Input Neural Network (Set 2 Model 2) data availability experiment graphs

5 - Discussion

5.1 - Hyperparameter Selection Experiments

5.1.1 - Set 1 Models 1 and 2 - Number of Estimators

Experimentation conducted to assess the influence of the n_estimators hyperparameter shows that there is little effect of varying this parameter on either the gradient boosting models (Set 1 Model 1, Figure 25) or the random forest models (Set 1 Model 2, Figure 26). From Appendix A, Table 3 results show that for experiments conducted with similar n_estimators values consistently return mean squared error values within a close range; Experiments conducted at a learning rate of 0.1 with n_estimators of 100 (experiment D), 200 (experiment E), and 300 (experiment F) return MSE values of 129.7, 128.8, and 128.3 respectively.

Randomized forest results (Figure 26) show similar results with experiments across 100 (experiment A), 100 (experiment B), and 300 (experiment C) estimators returning MSE values of 118.3, 121.0, and 123.4 respectively (Appendix A - Table 4).

5.1.2 - Set 1 Model 1 - Learning Rate

Variations in learning rate are found to have great effect on the accuracy and fit of the gradient boosting models (Set 1 Model 2, Figure 25). At learning rates of 0.01 (experiments A, B, C), and 0.1 (experiments D, E, F) MSE scores lie consistently within the range 108.0 to 130.0 (Appendix A Table 3). When the learning rate is increased to 1, however, (experiments G, H, I) the model drastically overfits, as evident from the noise in the column plots of these experiments (Figure 25).

5.1.3 - Set 2 Model 2 - Sequence Separation and Length

Experimentation revealed that varying the number of time periods considered as inputs and their separation has only a marginal impact on the effectiveness of the sequenced neural network model (Set 2 Model 2 – Figure 27). It can be seen in Figure 33 that in considering more time points the complexity of the learning model is drastically increased, thus causing the models to underfit to data, reflected in higher recorded MSE values of 358 and 236 from experiments G and I (Appendix A - Table 5). Optimum results are observed when two time points spaced 200 rows of data apart (separation of roughly half a day) are taken as inputs to the model, this showing the lowest MSE loss of 72.4 (Appendix A – Table 5 - Experiment B).

5.1.4 - Set 2 Models 1 and 2 - Layer Dimensions

The single input (Figure 28) and sequenced input (Figure 29) neural network experiments reveal that layer dimensions have great influence over the effectiveness of the trained model. The single input neural network experiments reveal optimal results when 2 layers of 500 – 1000 nodes per layer are used, returning MSE losses of 60.4 and 67.8 respectively (Appendix A – Table 6 – Experiments D and E); this is in contrast with when fewer layers are considered, such as a single layer of 500 nodes in experiment A returning a MSE value of 590.5, at these layer dimensions the model tends to overfit significantly to the data with significantly higher variance in forecasted values. Higher hidden layers of higher dimensions similarly show underfitting to the data: Experiments K and L represent experiments with 4 layers of 1000 and 1500 nodes per layer respectively, the former returning an MSE error of 378.3, the latter returning 343.2 (Appendix A – Table 6).

Sequenced input neural network experimentation (Set 2 Model 2, Figure 29) shows similar trends in forecast accuracy; with smaller dimensionality hidden layers overfitting to the training data returning MSE values of 115.2 (Appendix A – Table 7 – Experiment A); whilst higher dimensionality layers underfit to the data with MSE values of 148.6 when 4 layers of 1500 nodes per layer are trained (Appendix A – Table 7 – Experiment L). Optimum results are found in experiments H and K at 1000 nodes per layer over 3 and 4 hidden layers respectively, both experiments producing significantly lower MSE values of 37.7 and 46.5 respectively.

5.1.5 - Summary of Hyperparameter Selection

The results from these experiments reveals the significantly greater sensitivity of Neural Network machine learning methods to their initialisation hyperparameters. Variations in hidden layer dimensions may account for significant improvements in MSE loss and may be modulated to both induce and reduce model overfitting. The more complex sequenced input neural network experiments (Set 2 Model 2) reveals preference for higher dimensionality hidden layers of 3 to 4 layers when compared to the single input neural network experiments (Set 2 Model 1) showing optimum results with 2 hidden layers.

Traditional decision tree methods, whilst returning higher average error values, are more consistent in their output accuracy, requiring very little experimentation to find optimum hyperparameter values.

5.2 - Data Availability Experiments

5.2.1 - Variations in dataset size

Both gradient boosted (Figure 30) and random forest (Figure 31) decision tree methods show slight improvement in forecast accuracy across each forecast range when trained on a dataset with 30,000 rows of data over 15,000 rows. Gradient boosted trees, in experiments with 20 features, show an improvement in MSE from 66.1 to 53.0 in +0.5 day forecasts and from 99.0 to 81.6 in +3.0 day forecasts (Appendix A – Table 8 – Experiments A and C). Randomized forest models show similar improvement, returning changes in MSE of 69.6 to 50.5 and 95.0 to 81.3 for +0.5 and +3.0 day forecasts respectively. The improvements in the results are minimal, however, and indicate the strong capabilities of decision tree methods to produce consistent accuracy when less data is available.

The single input (Figure 32) and sequenced input (Figure 33) neural network experiments show a significant change in accuracy between experiments conducted on 15,000 and 30,000 rows of training data; the former causing both models to strongly underfit, returning MSE loss values consistently over 3500 across all forecast ranges for both 20 and 80 input feature experiments (Appendix A – Tables 10 and 11 – Experiments A and B).

Whilst it can be concluded that decision tree methods provide significantly more consistent forecast accuracy when provided with limited training data, the neural network methods produce significantly more accurate results in higher forecast ranges of 4 to 5 days when supplied with sufficient data, the single input Neural Network producing MSE errors of 67.8 for a +4.0 day forecast when trained on 30,000 rows of data with 80 features in comparison with 130.0 of the Gradient boosted decision tree model (Appendix A – Table 8 and 10 – Experiment D).

5.2.2 - Variations in Number of Features

The gradient boosted (Figure 30) and random forest (Figure 31) decision tree models show negligible difference in accuracy when trained on 20 and 80 unique input features, showing similar if not worse fitting and MSE losses as a result (Appendix A – Tables 8 and 9).

Both single input (Figure 32) and sequenced input (Figure 33) neural network methods, however, show improved forecast fitting and MSE losses (Appendix A – Tables 10 and 11) when supplied with more input features, indicating the neural network's strength when considering more complex applications where higher dimensional data is available. The neural network itself can assign relative weightings to each input features, and thus, provided the models are trained on a sufficiently sized dataset, benefit greatly from more input features. The increase in forecast accuracy is most evident on the singular input neural network experiment (Figure 32), showing an increase in accuracy from 212.6 to 136.8 and 209.9 to 79.2 for forecasts of +1.0 and +5.0 days respectively (Appendix A – Table 10). The sequenced neural network experiments showed a less noticeable improvement in MSE from increased dimensionality: from 317.0 to 285.2 and 98.4 to 63.6 for forecasts of +1.0 and +5.0 days respectively (Appendix A – Table 11).

5.2.3 - Variations in of Forecast Range

Both single input and sequence input neural network methods show consistently lower error rates than comparable decision tree experiments across all 10 forecasting intervals when trained on a sufficiently large dataset, with the greatest improvement shown in forecasts from 3 to 5 days. These results point towards the neural network's suitability for

longer term forecasting than comparable gradient boosting and random forest decision tree methods.

Whist both neural network methods produced relatively competent 5 day forecasts in the row index 1000 plots (Figures 32 and 33), neither they nor the decision tree methods were able to respond to the visible market shift that occurs at around row index 4000 of the test dataset, visible at index 38,000 in Figure 7. This was a real market shift (Oyedele, 2016) that occurred, and represents an occurrence that would likely have been unforeseen on the stock market; this points towards a larger weakness of the models trained for this project in forecasts are formed based purely upon the behaviour of those stock listed on the NASDAQ stock exchange without considering external factors, and without the ability to adapt to market shifts.

The results of this project have shown that whilst decision tree ensemble methods are able to produce more consistent results in a fraction of the time and with significantly less available data, neural network methods are better able to take advantage of higher dimensionality problems in producing forecasts with significantly higher accuracies provided a sufficiently sized training dataset is provided. Ultimately, however, models have a limited usable lifetime once trained, and to ensure strong forecasts are continued to be made, models must be regularly retrained on modern data that reflects current market behaviours.

6 - Conclusion

The results of this project have shown that decision tree ensemble methods are able to produce forecasts of a consistent accuracy from significantly less training data and in a fraction of the time of comparable neural networks, whilst neural networks require significantly more training data and computing power to produce useable forecasts. Neural network methods, however, are better able to take advantage of higher dimensionality input data and consider more complex trends between each input feature, resulting in significantly higher accuracies across short to medium range forecasts.

The results of these experiments additionally subvert the conclusions of McIntyre's work (McIntyre, n.d.), showing that once correctly initialised and supplied with sufficient training data, neural network methods are able to marginally outperform decision tree methods.

Neural network machine learning is strongly suited towards financial engineering applications in spite of its weaknesses, as a financial firm will have access to vast reservoirs of relevant market data already collected; an individual stock market trader, however, may be required to obtain such data at cost from subscription based online data vendors (TRADING ECONOMICS).

The limitations of both decision tree and neural network methods were met when a significant market shift occurred later in the test data, for which none of the models were able to account for in their forecasts; the machine learning models trained in this project will no longer produce relevant forecasts and thus will need to be retrained on modern data. Once a significant market shift has occurred a firm may be required to operate solely on decision tree models until sufficient data has been collected to train a full neural

network, thus, neural network methods are not yet able to fully replace traditional decision tree methods.

6.1 - Further Work

Having proven the relative effectiveness of neural network machine learning methods against traditional decision tree methods, further work may be devoted to implementing and evaluating more complex applications of neural network machine learning; specifically, in being able to fully train a model on smaller datasets.

For this project only a single iteration of hyperparameter selection was undertaken, however, the relative impact of each hyperparameter is dependent on the complexity of the problem the model is applied to; higher dimensionality problems may benefit from deeper neural networks, thus several iterations of hyperparameter selection may be conducted.

Recurrent Neural Network (RNN) models are typically used in Natural Language Processing applications where an ordered sequence of word is more relevant than just one when predicting the next word in a sequence. RNNs are beginning to see applications in financial forecasting, the work done on Attention-Based RNNs conducted by the University of California (Qin et al., 2017); implementation of an RNN was attempted for this project but was not completed as it was determined to lay outside the scope of this project. Future work may evaluate the potential benefits it holds over a standard neural network model.

7 - References

[1] McIntyre, C., n.d. Machine Learning in Financial Engineering.

[2] Sikorski, D. (1979). The efficient market hypothesis. Republic of Singapore: Institute of Economics and Business Studies, College of Graduate Studies, Nanyang University.

[3] Malkiel, B. (2003). The Efficient Market Hypothesis and Its Critics. Journal of Economic Perspectives, 17(1), pp.59-82.

[4] Seadnafinnegan.blogspot.com. 2014. Efficient Market Hypothesis. [online] Available at: <http://seadnafinnegan.blogspot.com/2014/10/efficient-market-hypothesis.html>

[5] 2018. NASDAQ-100 INDEX METHODOLOGY. [3] Howard, J. and Gugger, S., 2020. Fastai: A Layered API for Deep Learning. Information, 11(2), p.108.

[6] Jha, P., 2019. A Brief Overview Of Loss Functions In Pytorch. [online] Medium. Available at: <https://medium.com/udacity-pytorch-challengers/a-brief-overview-of-loss-functions-in-pytorch-c0ddb78068f7>

[7] Bhande, A., 2018. What Is Underfitting And Overfitting In Machine Learning And How To Deal With It.. [online] Medium. Available at: <https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76>

[8] Scikit-learn.org. 2019. 1.11. Ensemble Methods — Scikit-Learn 0.22.2 Documentation. [online] Available at: <https://scikit-learn.org/stable/modules/ensemble.html#ensemble>

[9] HackerEarth. 2018. Decision Tree Tutorials & Notes | Machine Learning | Hackerearth. [online] Available at: https://www.hackerearth.com/practice/machine-learning/machine-learning-algorithms/ml-decision-tree/tutorial/

[10] Breiman, L. and Cutler, A., 2001. Random Forests.

[11] Brownlee, J., 2018. Ensemble Learning Methods For Deep Learning Neural Networks. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/ensemble-methods-for-deep-learning-neural-networks/

[12] F. Rosenblatt. The perceptron, a perceiving and recognizing automaton Project Para. Cornell Aeronautical Laboratory, 1957.

[13] Gupta, R., 2017. Getting Started With Neural Network For Regression And Tensorflow. [online] Medium. Available at: <https://medium.com/@rajatgupta310198/getting-started-with-neural-network-for-regression-and-tensorflow-58ad3bd75223>

[14] Tradingeconomics.com. 2020. TRADING ECONOMICS. [online] Available at: <https://tradingeconomics.com/>

[15] Qin, Y., Song, D., Cheng, H., Jiang, G. and Cottrell, G., 2017. A Dual-Stage Attention-Based Recurrent Neural Network for Time Series Prediction. International Joint Conference on Artificial Intelligence (IJCAI)

[16] Oyedele, A., 2016. Stocks Are Jumping Towards New Highs. [online] Business Insider. Available at: <https://www.businessinsider.com/stock-market-update-december-15-2016-2016-12?r=US&IR=T> [Accessed 27 March 2020].

Appendix A: Experiment Scores

Table 3: Gradient Boosted Decision Tree (Set 1 Model 1) estimator quantity and learning rate experiment scores

Experiment A: learning_rate = 0.01, n_estimators = 100	**Experiment B: learning_rate = 0.01, n_estimators = 200**	**Experiment C: learning_rate = 0.01, n_estimators = 300**
+5.0 Days R^2 Score -2.303506 Mean Square Error 108.608935	+5.0 Days R^2 Score -2.659164 Mean Square Error 114.305971	+5.0 Days R^2 Score -3.183062 Mean Square Error 122.215176
Experiment D: learning_rate = 0.1, n_estimators = 100	**Experiment E: learning_rate = 0.1, n_estimators = 200**	**Experiment F: learning_rate = 0.1, n_estimators = 300**
+5.0 Days R^2 Score -3.714592 Mean Square Error 129.747821	+5.0 Days R^2 Score -3.647607 Mean Square Error 128.822788	+5.0 Days R^2 Score -3.609745 Mean Square Error 128.296984
Experiment G: learning_rate = 1, n_estimators = 100	**Experiment H: learning_rate = 1, n_estimators = 200**	**Experiment I: learning_rate = 1, n_estimators = 300**
+5.0 Days R^2 Score -15.340770 Mean Square Error 241.554016	+5.0 Days R^2 Score -15.647870 Mean Square Error 243.813274	+5.0 Days R^2 Score -17.209490 Mean Square Error 254.992203

Table 4: Randomized Forest (Set 1 Model 2) estimator quantity experiment scores

Experiment A: n_estimators = 100	**Experiment B: n_estimators = 200**	**Experiment C: n_estimators = 300**
+5.0 Days R^2 Score -2.916544 Mean Square Error 118.257726	+5.0 Days R^2 Score -3.103617 Mean Square Error 121.049056	+5.0 Days R^2 Score -3.265722 Mean Square Error 123.416801

Table 5: Sequence Input Neural Network (Set 2 Model 2) sequence length and separation selection experiment scores

Experiment A: seq_len = 2, seq_sep = 100	**Experiment B: seq_len = 2, seq_sep = 200**	**Experiment C: seq_len = 2, seq_sep = 300**
5.0 Days R^2 Score -12.961715 Mean Square Error 185.142760	5.0 Days R^2 Score -1.132652 Mean Square Error 72.359793	5.0 Days R^2 Score -13.885276 Mean Square Error 191.168261
Experiment D: seq_len = 4, seq_sep = 100	**Experiment E: seq_len = 4, seq_sep = 200**	**Experiment F: seq_len = 4, seq_sep = 300**
5.0 Days R^2 Score -6.544020 Mean Square Error 136.093933	5.0 Days R^2 Score -4.801457 Mean Square Error 119.345452	5.0 Days R^2 Score -33.722830 Mean Square Error 291.974486
Experiment G: seq_len = 6, seq_sep = 100	**Experiment H: seq_len = 6, seq_sep = 200**	**Experiment I: seq_len = 6, seq_sep = 300**
5.0 Days R^2 Score -21.713616 Mean Square Error 236.145942	5.0 Days R^2 Score -3.282383 Mean Square Error 102.536884	5.0 Days R^2 Score -51.306086 Mean Square Error 358.354991

Table 6: Single Input Neural Network (Set 2 Model 1) hidden layer dimension experiment scores

Experiment A: Layer Dimensions: [500]	**Experiment B: Layer Dimensions: [1000]**	**Experiment C: Layer Dimensions: [1500]**
+5 Days R^2 Score -141.043249 Mean Square Error 590.537720	+5 Days R^2 Score -5.592301 Mean Square Error 127.220136	+5 Days R^2 Score -89.681032 Mean Square Error 471.840888
Experiment D: Layer Dimensions: [500, 500]	**Experiment E: Layer Dimensions: [1000, 1000]**	**Experiment F: Layer Dimensions: [1500, 1500]**
+5 Days R^2 Score -0.489016 Mean Square Error 60.462627	+5 Days R^2 Score -0.869747 Mean Square Error 67.753045	+5 Days R^2 Score -5.943038 Mean Square Error 130.560598
Experiment G: Layer Dimensions: [500, 500, 500]	**Experiment H: Layer Dimensions: [1000, 1000, 1000]**	**Experiment I: Layer Dimensions: [1500, 1500, 1500]**
+5 Days R^2 Score -12.648469 Mean Square Error 183.054041	+5 Days R^2 Score -7.357561 Mean Square Error 143.244222	+5 Days R^2 Score -16.083777 Mean Square Error 204.799691
Experiment J: Layer Dimensions: [500, 500, 500, 500]	**Experiment K: Layer Dimensions: [1000, 1000, 1000, 1000]**	**Experiment L: Layer Dimensions: [1500, 1500, 1500, 1500]**
+5 Days R^2 Score -7.588122 Mean Square Error 145.206628	+5 Days R^2 Score -57.282311 Mean Square Error 378.273333	+5 Days R^2 Score -46.961810 Mean Square Error 343.150887

Table 7: Sequence Input Neural Network (Set 2 Model 2) hidden layer dimension experiment scores

Experiment A: Layer Dimensions: [500]	**Experiment B: Layer Dimensions: [1000]**	**Experiment C: Layer Dimensions: [1500]**
5.0 Days R^2 Score -4.401882 Mean Square Error 115.162187	5.0 Days R^2 Score -7.174280 Mean Square Error 141.664848	5.0 Days R^2 Score -23.919583 Mean Square Error 247.347615
Experiment D: Layer Dimensions: [500, 500]	**Experiment E: Layer Dimensions: [1000, 1000]**	**Experiment F: Layer Dimensions: [1500, 1500]**
5.0 Days R^2 Score -1.456220 Mean Square Error 77.655283	5.0 Days R^2 Score -1.120977 Mean Square Error 72.161455	5.0 Days R^2 Score -3.384840 Mean Square Error 103.756244
Experiment G: Layer Dimensions: [500, 500, 500]	**Experiment H: Layer Dimensions: [1000, 1000, 1000]**	**Experiment I: Layer Dimensions: [1500, 1500, 1500]**
5.0 Days R^2 Score -2.405559 Mean Square Error 91.439014	5.0 Days R^2 Score 0.421234 Mean Square Error 37.695427	5.0 Days R^2 Score -4.284308 Mean Square Error 113.902018
Experiment J: Layer Dimensions: [500, 500, 500, 500]	**Experiment K: Layer Dimensions: [1000, 1000, 1000, 1000]**	**Experiment L: Layer Dimensions: [1500, 1500, 1500, 1500]**
5.0 Days R^2 Score -10.377200 Mean Square Error 167.130241	5.0 Days R^2 Score 0.120363 Mean Square Error 46.471770	5.0 Days R^2 Score -7.990450 Mean Square Error 148.568953

Table 8: Gradient Boosted Decision Tree (Set 1 Model 1) data availability experiment scores

Experiment A: 20 Features, 15,000 Data Rows

	+0.5 Days	+1.0 Days	+1.5 Days	+2.0 Days	+2.5 Days	+3.0 Days	+3.5 Days	+4.0 Days	+4.5 Days	+5.0 Days
R^2 Score	0.134394	0.021822	-0.134475	-0.125833	-0.524321	-0.731225	-2.461234	-1.133191	-5.084990	-1.822972
Mean Square Error	66.062415	71.656308	78.056668	79.944517	94.759852	99.000596	134.004663	99.548464	156.148493	100.399432

Experiment B: 80 Features, 15,000 Data Rows

	+0.5 Days	+1.0 Days	+1.5 Days	+2.0 Days	+2.5 Days	+3.0 Days	+3.5 Days	+4.0 Days	+4.5 Days	+5.0 Days
R^2 Score	0.537954	-0.002640	-0.125335	-0.288218	-1.045188	0.135085	-2.483329	0.108112	-5.548836	-3.158737
Mean Square Error	48.265529	72.546766	77.741629	85.515796	109.762206	69.975786	134.431706	64.368728	161.990650	121.859310

Experiment C: 20 Features, 30,000 Data Rows

	+0.5 Days	+1.0 Days	+1.5 Days	+2.0 Days	+2.5 Days	+3.0 Days	+3.5 Days	+4.0 Days	+4.5 Days	+5.0 Days
R^2 Score	0.442729	0.379448	0.283574	-0.082671	0.579966	-0.174932	-3.409189	-1.415713	-3.254463	-6.525370
Mean Square Error	53.006328	57.073533	62.029505	78.397074	49.742573	81.558182	151.245965	105.935705	130.566052	163.923772

Experiment D: 80 Features, 30,000 Data Rows

	+0.5 Days	+1.0 Days	+1.5 Days	+2.0 Days	+2.5 Days	+3.0 Days	+3.5 Days	+4.0 Days	+4.5 Days	+5.0 Days
R^2 Score	0.415047	0.192385	-0.101663	-0.120191	-0.023270	-0.389384	-2.644584	-2.640453	-2.307175	-5.994205
Mean Square Error	54.306901	65.109988	76.919593	79.743934	77.639261	88.689493	137.508145	130.046128	115.116200	158.032783

Table 9: Randomized Forest (Set 1 Model 2) data availability experiment scores

Experiment A: 20 Features, 15,000 Data Rows

	+0.5 Days	+1.0 Days	+1.5 Days	+2.0 Days	+2.5 Days	+3.0 Days	+3.5 Days	+4.0 Days	+4.5 Days	+5.0 Days
R^2 Score	0.037632	0.015072	-0.143019	-0.020101	0.013023	-0.597005	-3.864389	-3.134511	-1.530169	-2.97275
Mean Square Error	69.657040	71.903132	78.350068	76.098014	76.249987	95.085486	158.861476	138.589964	100.689147	119.10325

Experiment B: 80 Features, 15,000 Data Rows

	+0.5 Days	+1.0 Days	+1.5 Days	+2.0 Days	+2.5 Days	+3.0 Days	+3.5 Days	+4.0 Days	+4.5 Days	+5.0 Days
R^2 Score	0.617018	-0.186231	-0.077807	-0.214433	-0.716703	-0.021305	-2.455046	-0.091723	-4.298165	-3.832302
Mean Square Error	43.942391	78.909664	76.082224	83.030665	100.561976	76.039430	133.884821	71.215743	145.703705	131.357555

Experiment C: 20 Features, 30,000 Data Rows

	+0.5 Days	+1.0 Days	+1.5 Days	+2.0 Days	+2.5 Days	+3.0 Days	+3.5 Days	+4.0 Days	+4.5 Days	+5.0 Days
R^2 Score	0.493604	0.297415	-0.092115	-0.097811	-0.312519	-0.167894	-2.465958	-2.389603	-2.624343	-5.572309
Mean Square Error	50.528873	60.728846	76.585540	78.943323	87.930408	81.313524	134.096088	125.485665	120.509844	153.192320

Experiment D: 80 Features, 30,000 Data Rows

	+0.5 Days	+1.0 Days	+1.5 Days	+2.0 Days	+2.5 Days	+3.0 Days	+3.5 Days	+4.0 Days	+4.5 Days	+5.0 Days
R^2 Score	0.415919	0.319367	0.085149	-0.037703	-0.558794	-0.176618	-2.564737	-0.393868	-2.788545	-4.079780
Mean Square Error	54.266375	59.772577	70.095138	76.751738	95.825383	81.616658	135.993522	80.469375	123.209472	134.679176

Table 10: Single Input Neural Network (Set 2 Model 1) data availability experiment scores

Experiment A: 20 Features, 15,000 Data Rows

	0.5 Days	1.0 Days	1.5 Days	2.0 Days	2.5 Days	3.0 Days	3.5 Days	4.0 Days	4.5 Days	5.0 Days
R^2 Score	-2511.50963	-2557.700250	-2335.991122	-2327.897166	-2123.122561	-2262.843898	-2517.966506	-2588.222328	-2994.342174	-3580.713417
Mean Square Error	3559.16630	3664.841641	3542.755344	3636.024664	3537.333406	3580.010768	3615.061083	3468.202113	3464.424450	3576.213359

Experiment B: 80 Features, 15,000 Data Rows

	0.5 Days	1.0 Days	1.5 Days	2.0 Days	2.5 Days	3.0 Days	3.5 Days	4.0 Days	4.5 Days	5.0 Days
R^2 Score	-2567.603284	-2486.135501	-2315.129170	-2231.550949	-2181.490093	-2276.440708	-2535.681723	-2745.816068	-3096.073091	-3665.746031
Mean Square Error	3598.677510	3554.641252	3526.907086	3560.019309	3585.604958	3590.745575	3627.750690	3572.189721	3522.764423	3618.415371

Experiment C: 20 Features, 30,000 Data Rows

	0.5 Days	1.0 Days	1.5 Days	2.0 Days	2.5 Days	3.0 Days	3.5 Days	4.0 Days	4.5 Days	5.0 Days
R^2 Score	-13.730775	-7.609730	-4.889662	-2.489426	-7.699789	-8.164249	-1.946602	-1.209808	-1.695252	-11.340675
Mean Square Error	272.525545	212.588745	177.851611	140.743611	226.381310	227.776826	123.641762	101.320426	103.922013	209.916925

Experiment D: 80 Features, 30,000 Data Rows

	0.5 Days	1.0 Days	1.5 Days	2.0 Days	2.5 Days	3.0 Days	3.5 Days	4.0 Days	4.5 Days	5.0 Days
R^2 Score	-8.839714	-2.567441	-0.420819	-0.643769	-0.609321	-0.021208	-6.218253	0.010023	-2.302883	-0.757773
Mean Square Error	222.733555	136.843416	87.353849	96.598886	97.366055	76.035823	193.517606	67.816001	115.041486	79.224474

Table 11: Sequenced Input Neural Network (Set 2 Model 2) data availability experiment scores

Experiment A: 20 Features, 15,000 Data Rows

	0.5 Days	1.0 Days	1.5 Days	2.0 Days	2.5 Days	3.0 Days	3.5 Days	4.0 Days	4.5 Days	5.0 Days
R^2 Score	-1237.053162	-1166.445359	-1094.434913	-1092.536754	-981.372864	-1029.631070	-1171.779244	-1154.603989	-1428.939526	-1735.894146
Mean Square Error	2498.412015	2475.504893	2425.525918	2491.542825	2405.605392	2415.531525	2466.678674	2316.990446	2393.683409	2490.374611

Experiment B: 80 Features, 15,000 Data Rows

	0.5 Days	1.0 Days	1.5 Days	2.0 Days	2.5 Days	3.0 Days	3.5 Days	4.0 Days	4.5 Days	5.0 Days
R^2 Score	-1206.060589	-1149.566680	-1126.259833	-1057.782890	-1023.548366	-1080.428110	-1175.592110	-1268.300587	-1470.654757	-1667.073413
Mean Square Error	2466.942052	2457.544577	2460.507237	2451.631097	2456.701790	2474.343109	2470.685175	2428.297917	2428.347547	2440.538058

Experiment C: 20 Features, 30,000 Data Rows

	0.5 Days	1.0 Days	1.5 Days	2.0 Days	2.5 Days	3.0 Days	3.5 Days	4.0 Days	4.5 Days	5.0 Days
R^2 Score	-4.935471	-18.151272	-0.183783	0.258195	0.142000	-1.658099	-0.825358	-0.302971	-10.405154	-1.710549
Mean Square Error	172.990396	317.062088	79.734939	64.892821	71.093491	122.672341	97.314692	77.801354	213.775754	98.379956

Experiment D: 80 Features, 30,000 Data Rows

	0.5 Days	1.0 Days	1.5 Days	2.0 Days	2.5 Days	3.0 Days	3.5 Days	4.0 Days	4.5 Days	5.0 Days
R^2 Score	-2.931043	-14.496791	-3.497938	-0.404982	-0.686935	-2.711161	-1.664156	-1.115316	-2.288979	-0.134507
Mean Square Error	140.782378	285.211081	155.424452	89.307328	99.686275	144.949265	117.566683	99.130517	114.799078	63.647538

Appendix B: Recording of Project Meetings

Record of Project Meetings
CARDIFF SCHOOL OF ENGINEERING

NAME: Cameron Stumpf **STUDENT NUMBER: C1673094**

SUPERVISOR: Kensuke YOKOI

TEACHING DISCIPLINE: MMM

Date of meeting	Supervisor's assessment of progress	Actions by next meeting	Supervisor signature
03/10/2019	1st meeting	Determine your project	よこい けんすけ
10/10/2019	Discussed on your project	Complete project brief	よこい けんすけ
17/10/2019	He drafted project brief and continued his study of machine learning.	Submit project brief. Start project according to the plan in the project brief.	よこい けんすけ
24/10/2019	He submitted brief.	Updata your project brief (avoid exam period, etc.). Recreate the previous experiment Use the following link: https://gallery.cortanaintelligence.com/Experiment/Data-Analysis-Using-Machine-Learning-in-Financial-Engineering	よこい けんすけ
31/10/2019	Review what Callum did and reproduced some of his results.	Literature reviews of past works. Try to improve prediction based on Azure platform. Coursera machine learning week1-2.	よこい けんすけ
07/11/2019	Coursera week 1 was completed. A literature review of Boris work. He	Complete cousera ML, week 2. Try to replicate Boris work.	よこい けんすけ

	tried Azure but he decides not to use Azure.		
14/11/2019	He has very good holiday last week. He partially did week 2.	Try to replicate Boris work and show a result. Complete coursera course week3	よこい けんすけ
21/11/2019 Week8	Completed coursera week3. Some parts of FastAI course	Complete FastAI part 1. Decide targets clearly.	よこい けんすけ
28/11/2019	Completed FastAI course part 1.	Prepare presentation, create results as much as possible. Complete coursera by week5 by the presentation.	よこい けんすけ
05/12/2019	Up to Coursera week5 was completed. He showed draft of the presentation.	Improve presentation.	よこい けんすけ
04/02/2020	Implemented new data set and got some results	Create pytorch result. Please understand clearly how you predict stock price.	よこい けんすけ
11/02/2020	Show some results.	Follow gannet chard. Try RNN, maybe check TCN.	よこい けんすけ
18/02/2020	He is implementing RNN.	Complete RNN implementation and follow gannet chart	よこい けんすけ
25/02/2020	Implemented RNN, trained Started writing	Show results of the comparisons NN based methods and decision tree and regression.	よこい けんすけ
03/03/2020	RNN did not work. Conducted decision tree experiments.	Finalize conclusions. Create results as much as possible.	よこい けんすけ
10/03/2020	He got results and finalized conclusions	Write report.	よこい けんすけ