Contents

Table of Figures

Introduction

Stock market prediction is a complicated, if not near impossible, task. Historical company performance combined with other factors such as market volatility and geopolitical and macroeconomics must be considered for an investor to guess best how an individual stock will perform in the future. Historically, this is a computationally intensive task; however, readily available computing power has resulted in the rise of machine learning algorithms that excel in modelling patterns. A vast body of research investigates potential machine learning applications in stock price prediction. One study examined the application of ANN and Random Forest. The study found ANN to be effective, evidenced by relatively low error metrics, and suggested future options for utilising deep learning [1]. Huang et al. compared several methods, including support vector machines [2]. The study found that SVM achieved a hit ratio of 73%, but this increased to 75% when SVM was used in combination with other classification models. Another study proposes a k-means clustering method to cluster different technical factors affecting stock prices. This approach leverages the clustering results to predict stock prices based on morphological similarity and hierarchical temporal memory [3].

Phase 1- Problem Description and Data Pre-processing

1.1 Methodology

This project will investigate the application of machine learning in predicting stock prices of five "big tech" companies. Three different machine learning approaches will be utilised, namely two supervised machine learning approaches in the form of regression and classification and clustering, which is an unsupervised machine learning approach.

This scenario will be framed from the perspective of a "retail investor, " a non-professional investor who generally invests irregularly in an investment such as Stocks and Shares ISA in the UK. This hypothetical investor would like to invest monthly into his stock portfolio. This investor would like to monitor their investments at the end of each month and will have three options on how to proceed, which are "buy more stocks if the price is increasing", "sell stocks if the price is decreasing", or "hold their position (don't buy or sell) if the stocks are relatively stable". The main aim of this project is to develop a machine learning model that can predict the future price or trend of a stock's price, allowing investors to make an informed decision on what actions they take.

As stated previously, five datasets will be considered "AMZN", "MSFT", "IBM", "GOOGL" and "FB". The methodology will be deployed to generate a model using one dataset and test whether this model can be generally applied to all the datasets. The advantage of this approach is a reduced model training time and computational costs; however, this approach assumes that all these stocks behave similarly.

1.2.1 – Dataset Description

A preliminary Exploratory data analysis is conducted on the "AMZN" dataset to understand its contents. The dataset contains 64 features and 2472 instances. The features are made up of various price indicators and stock market indicators. Each instance is representative of a single trading day. Fig 1 shows all the features of the dataset.

Feature Catergory	Dataset Features
Date and Time	Date, Date_col, Day, DayofWeek, DayofYear, Week, Is_month_end, Is_month_start, Is_quarter_end, Is_quarter_start, Is_year_end, Is_year_start, Is_leap_year, Year, Month
Price Indicators	Open, High, Low, Close(t), Lower_Band, S_Close(t-1), S_Close(t-2), S_Close(t-3), S_Close(t-5), S_Open(t-1), QQQ_Close, SnP_Close, DJIA_Close, Close_forcast
Technical Indicators	Volume, SD20, Upper_Band, MA5, MA10, MA20, MA50, MA200, EMA10, EMA20, EMA50, EMA100, EMA200, MACD, MACD_EMA, QQQ_MA10, QQQ_MA20, QQQ_MA50, ATR, ADX, CCI, ROC, RSI, William%R, SO%K, STD5, ForceIndex1, ForceIndex20
Comparative Indicators	QQQ(t-1), QQQ(t-2), QQQ(t-5), SnP(t-1)), SnP(t-5), DJIA(t-1)), DJIA(t-5)

Figure 1 - List of Features in the AMZN Dataset

The first challenge of this project is first to determine the target variable. This project aims to predict the stock price's value within the next 20 days. Figure 2 shows the stock price 20 days in advance.

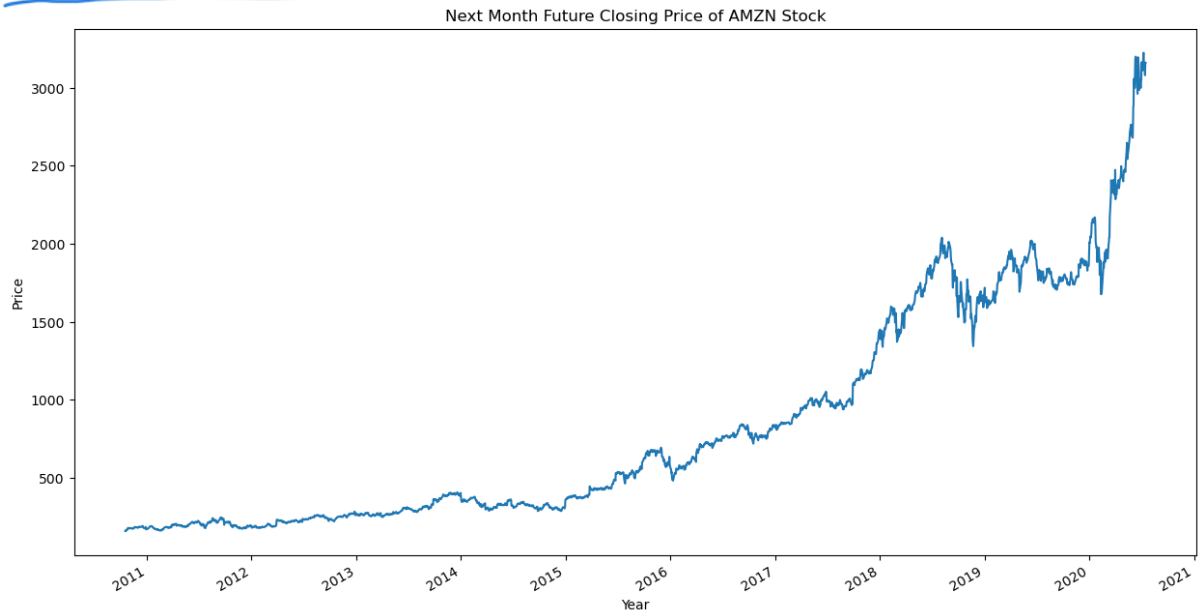

Figure 2 – Closing Price of AMZN dataset over time

As seen in Fig 2, the stock price between 2010-2021 shows four distinct trends. Between 2011 and 2015, there was a gradual increase in stock price until the period 2015-2019 offered an accelerated growth rate. Between 2019-2020, there is a volatile stage where the stock price fluctuates but still increases until finally, between 2020-2021, there is a dramatic increase in stock price where the price rises from approx.$2000 to $3100.

Fig 2 also highlights the daily volatility of the stock market prices, as shown by the daily rise and fall between instances. This introduces noise into the dataset, which may cause issues. One solution to reduce this noise would be to use a rolling average of the last 20 trading days [1]. This will maintain

the general trend of the dataset but filter out the noise, as seen in Fig 3. The blue line in Fig 3 is the "MA20" feature in the dataset.

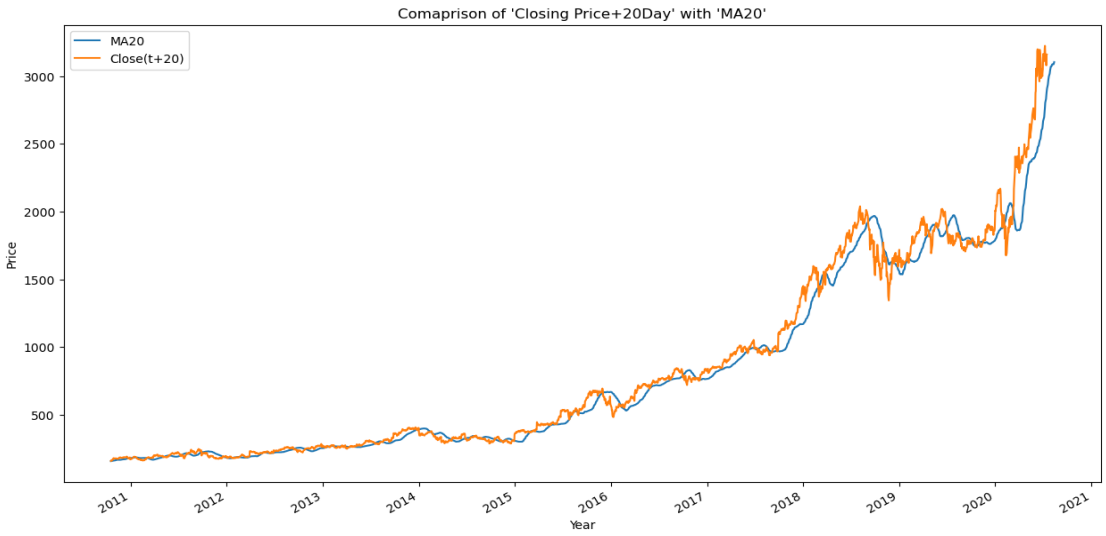

Figure 3 – Closing Price compared to MA20

Another issue must be addressed is splitting the target feature into a training and test dataset. The challenge with time series data is that the temporal nature of the dataset must be maintained to ensure the model isn't exposed to "look ahead bias" where future data is used during model training. This dataset produces the problem of the future data showing very different trends from the previous data. This is highlighted in Fig. 4, which shows the dataset's 80-20% split. The problem with using this test set is that a particular trend has never been seen in the dataset, especially the sharp spike in stock price towards 2021. Therefore, the machine learning model may not be able to predict accurately.

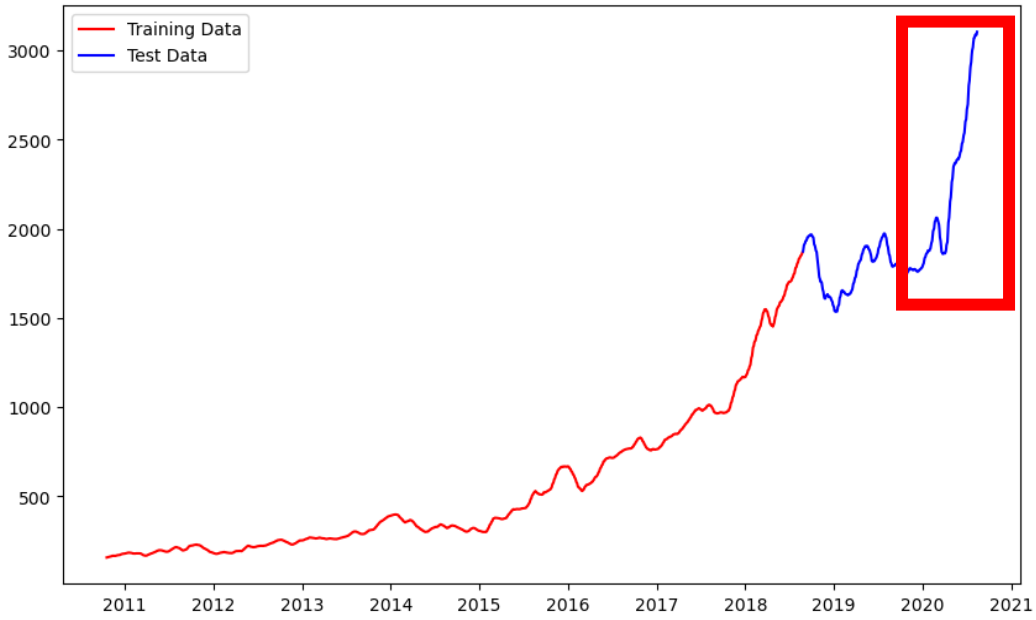

Figure 4 MA20 Spike in Price

Transforming this into percentage differences can transform and scale the data, as seen below in Fig 5.

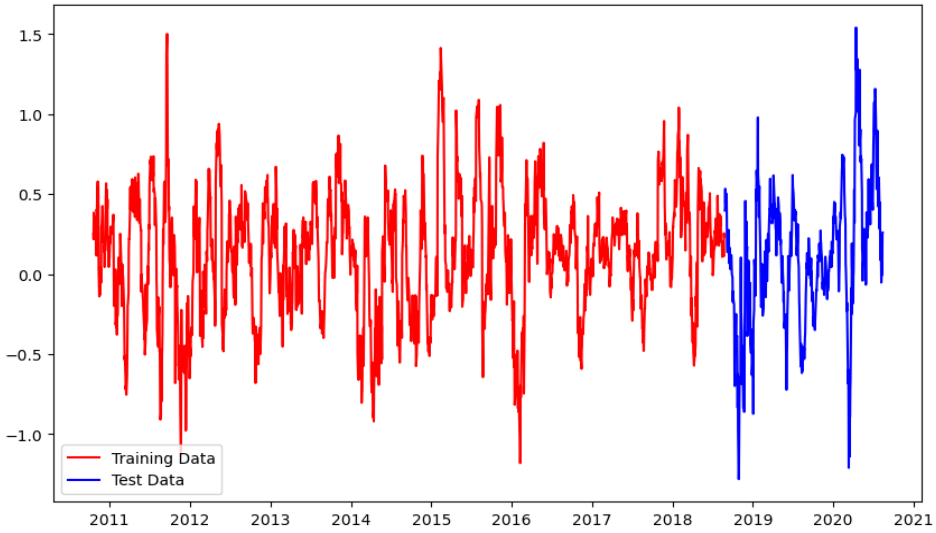

Figure 5 MA20 Percentage Change

Fig 6 displays a visual relationship between all the variables and the target variables. Fig demonstrates a range of different.

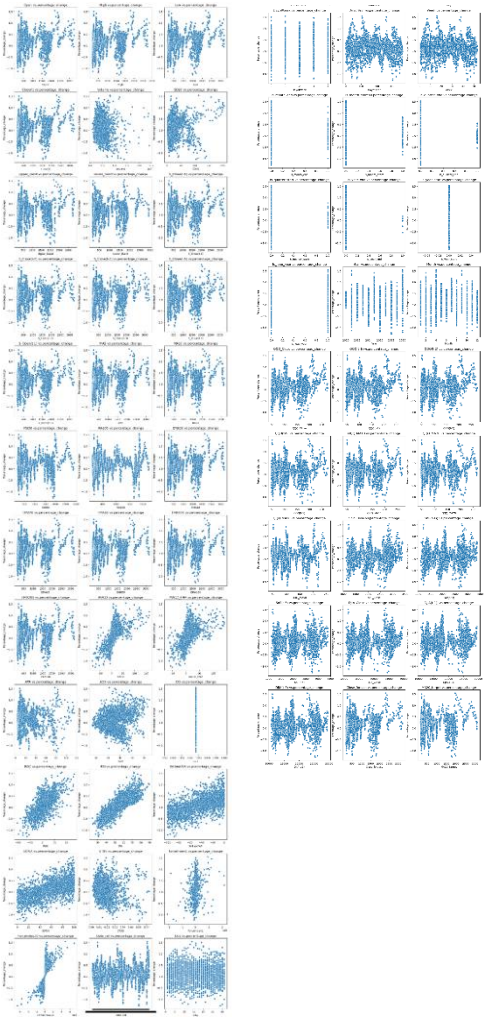

Figure 6 – Scatter plots showing the relationship between Predictor and Target Variables

As a preliminary feature selection, some features were removed as the EDA

The preditor features are scaled using the StandardScaler function in sklearn [4].

Phase 2 – Modelling

2.1 Regression Approach

2.1.1 Experiment Design

The regression approach in this project will use all the data to predict a future value for the daily percentage change. This approach will have five main experiments –

- **Experiment 1 –** A range of regression models will be trained using the training dataset on all the features. The
- **Experiment 2 –** The hyperparameters will be tuned using a search algorithm to find the optimal hyperparameters for each regression model. The optimised models will then be evaluated using all the features.
- **Experiment 3 -** Feature elimination will be done using the Recursive Feature Elimination algorithm. This function will train the model and remove the feature which provides the most negligible impact on the model results. This model utilises separate methods to determine feature elimination. The RFE will remove the feature with the smallest coefficient for linear regression models. The features will be ranked by feature importance for tree-based models and removed.
- **Experiment 4 –** Test final models on the last test set.
- **Experiment 5 –** Evaluate models' general Performance on the other four datasets.

This project will be using the Sk-learn Python library. The following regression models selected for these experiments can be seen in Fig 7 [5]–[10].

Regression Model Type	Python Package Used
Linear Regression	sklearn.linear_model.LinearRegression()
Ridge Regression	Sklearn.linear_model.Ridge()
Lasso Regression	Sklearn.linear_model.Lasso()
Support Vector Regression	Sklearn.svm.SVR()
Decision Tree Regression	Sklearn.tree.DescisionTreeRegressor()
Random Forest	Sklearn.ensemble.RandomForestRegressor

Figure 7 Regression Models used in this Experiment

For model evaluation, the Table below highlights the main metrics

- Mean Squared Errors (MSE)

- Root Mean Squared Errors (RMSE)
- Mean Absolute Error (MAE)
- R-Squared Score (R^2)

To evaluate the model's performance using a Time Series Split cross-validation [11]. This technique uses an "expanding-window approach to maintain the temporal nature of the time series. The training data will be split into five folds.

2.1.2 Results and Analysis

2.2.2.1 Results from Experiment 1 – Initial Baseline

Fig 8 below shows the average results from the five cross-validations, followed by Fig 9 to Fig 14 shows each model's performance over the five folds.

Initial Test All Feautures default hyperparmeters								
Model	**Avg Training MSE**	**Avg Training RMSE**	**Avg Training MAE**	**Avg Training R2**	**Avg Validation MSE**	**Avg Validation RMSE**	**Avg Validation MAE**	**Avg Validation R2**
LinearRegression()	0.0117	0.1083	0.0821	0.9252	0.0696	0.2638	0.1660	0.3804
Ridge()	0.0163	0.1276	0.0986	0.8967	0.0652	0.2554	0.1675	0.4334
Lasso()	0.1585	0.3981	0.3201	0.0000	0.1394	0.3733	0.2882	-0.0486
DecisionTreeRegressor()	0.0000	0.0000	0.0000	1.0000	0.0491	0.2215	0.1370	0.6535
RandomForestRegressor()	0.0007	0.0261	0.0164	0.9958	0.0246	0.1569	0.1075	0.8093
SVR(kernel='linear')	0.0153	0.1238	0.0918	0.9026	0.0849	0.2913	0.1750	0.2419

Figure 8 Results after the initial test using all predictor features and default hyperparameters

The provided model evaluations reveal varied performance characteristics across the regression models. The Linear Regression model shows robust learning on the training data with an R^2 of 0.9252 but suffers from a substantial drop in R^2 to 0.3804 on the validation set, suggesting potential overfitting. Fig 9 confirms this as the graphs demonstrate the testing data diverging from the training data as the time-series cross-validation rolls forward.

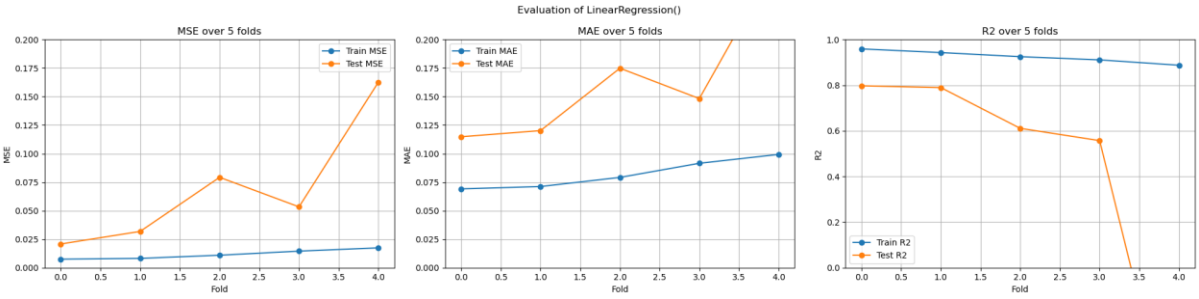

Figure 9 Linear Regression Metrics after Time Series Cross-validation

With slightly worse training metrics, Ridge Regression compensates by outperforming Linear Regression on the validation set on average. However, Fig 10 shows the test performance degrading

significantly across the five folds, indicating possible over-regularisation

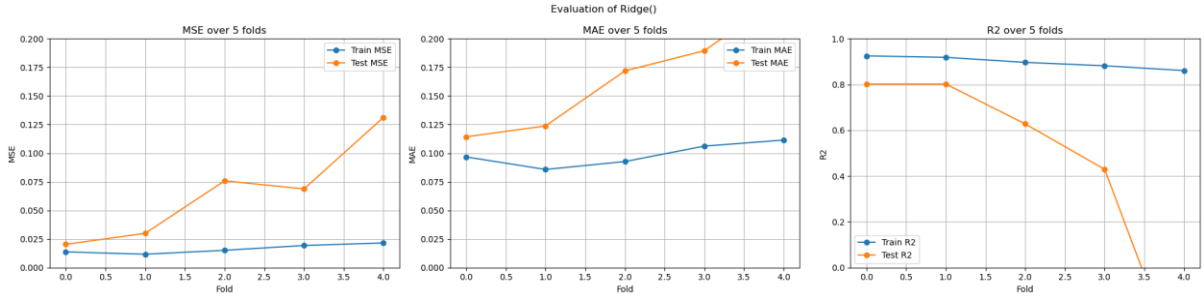

Figure 10 – Ridge Regression Metrics after Time Series Cross Validation

Lasso Regression underperforms significantly on training and validation data, with an R^2 of 0 on training and negative on validation, indicating it has likely been over-regularised, leading to serious underfitting.

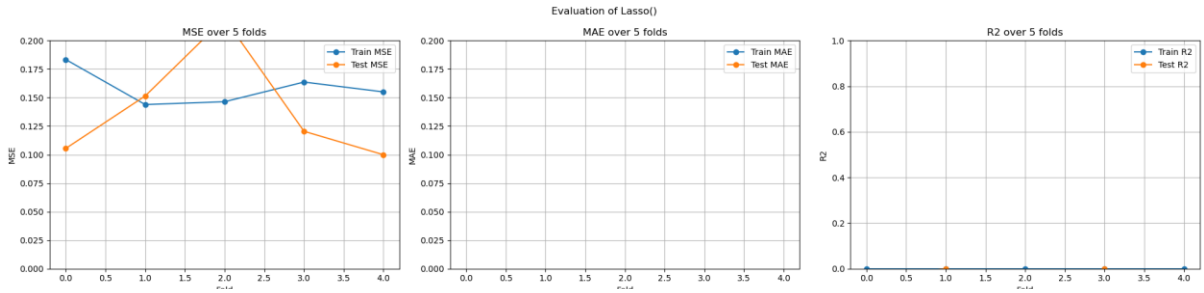

Figure 11- Lasso Regression After Time Series Cross Validation

The Decision Tree Regressor's perfect training performance hints at overfitting, but it maintains reasonable validation metrics, albeit with signs of overfitting.

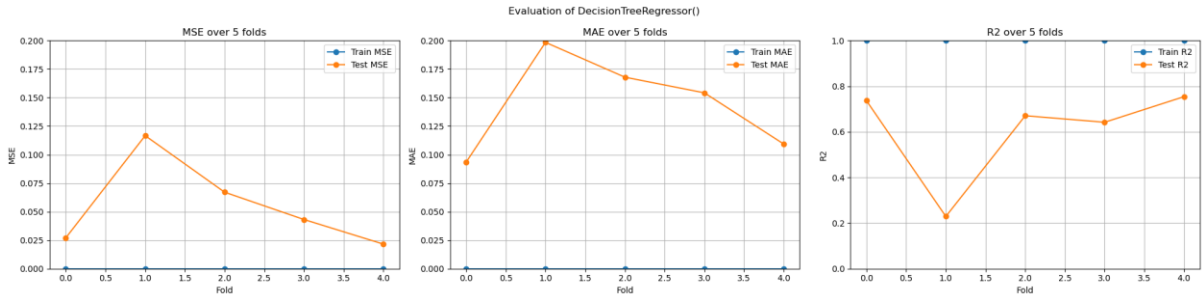

Figure 12 Decision Tree Metrics after Time Series Cross-validation

The RandomForestRegressor nearly mirrors the Decision Tree in training but demonstrates a robust ability to generalise, achieving the lowest validation MSE and RMSE and the highest R^2.

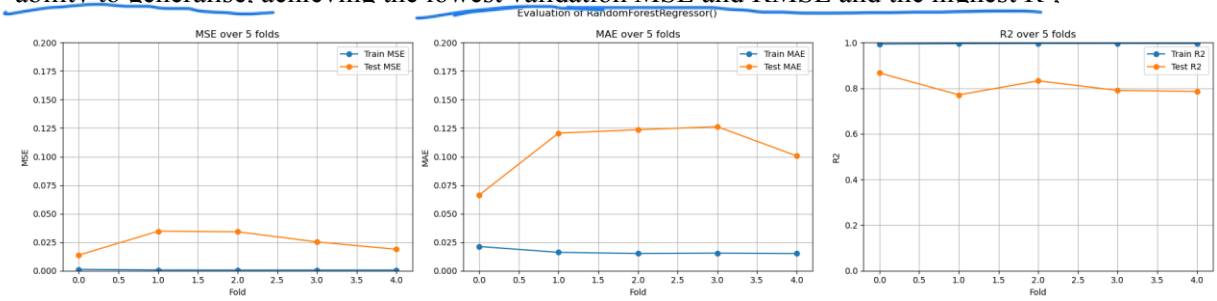

Figure 13 – Random Forest Regressor after Time Series Cross Validation

SVR with a linear kernel shows moderate training performance but fails to generalise effectively, as reflected by a lower validation R2 and a deterioration in validation performance across the folds.

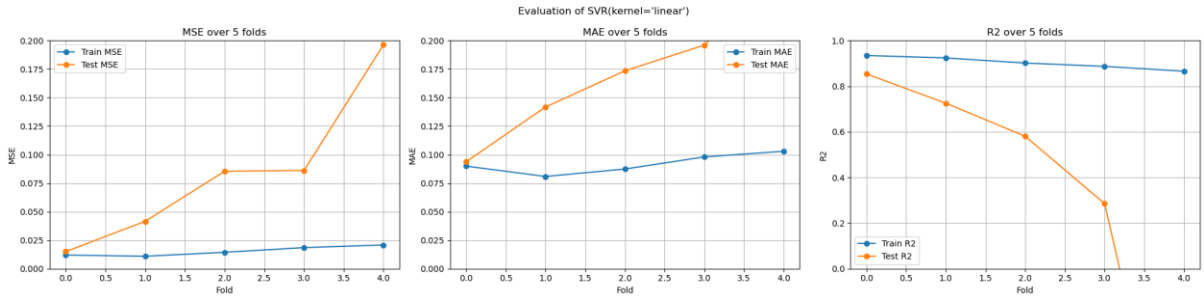

Figure 14 Support Vector Regressor after Time Series Cross Validation

Results from Experiments 2& 3– Hyperparameter Tuning and Feature Elimination

Fig 15 shows the results after Hyperparameter tuning, and RFECV has been carried out. Fig 16-21 shows the performance metrics of the model during cross-validation.

Evaluation AFTER Hyper-paramter tuning and Feature Elimination								
Model	**Avg Training MSE**	**Avg Training RMSE**	**Avg Training MAE**	**Avg Training R2**	**Avg Validation MSE**	**Avg Validation RMSE**	**Avg Validation MAE**	**Avg Validation R2**
LinearRegression()	0.0122	0.1103	0.0840	0.9225	0.0646	0.2542	0.1576	0.4207
Ridge(alpha=123)	0.0393	0.1982	0.1517	0.7537	0.0371	0.1927	0.1498	0.6821
Lasso(alpha=0.015)	0.0351	0.1873	0.1469	0.7768	0.0365	0.1910	0.1509	0.6942
SVR(C=0.0023, kernel='linear')	0.0400	0.1999	0.1536	0.7495	0.0362	0.1904	0.1499	0.6913
DecisionTreeRegressor(max_depth=50, min_samples_leaf=4)	0.0018	0.0421	0.0248	0.9890	0.0451	0.2124	0.1307	0.6781
RandomForestRegressor(max_depth=50, min_samples_leaf=4, min_samples_split=5)	0.0011	0.0330	0.0194	0.9933	0.0180	0.1342	0.0920	0.8638

Figure 15 Results after Hyper-Parameter Tuning and RFECV

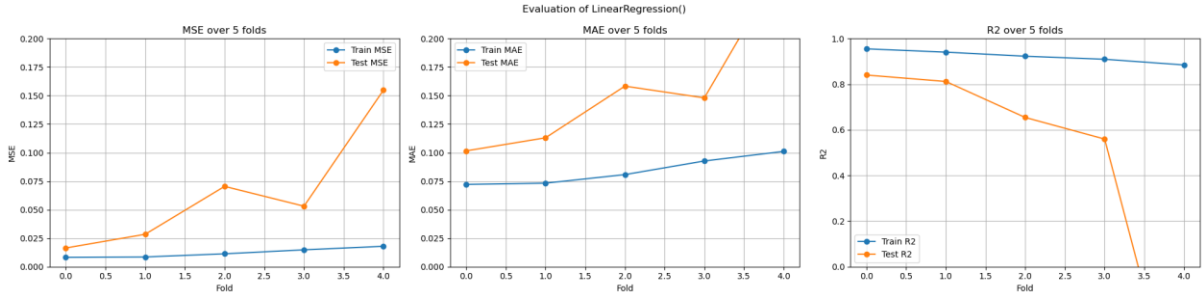

Figure 16 Linear Regression after Hyperparameter Tuning and RFECV

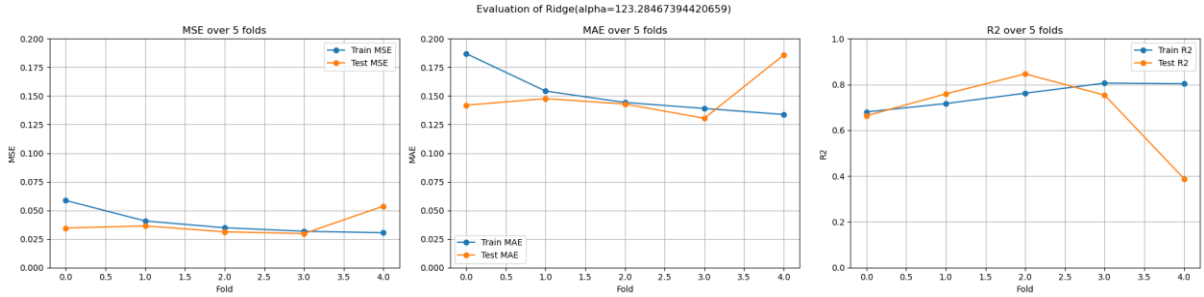

Figure 17 Ridge Regressor after Hyperparamter Tuning and RFECV

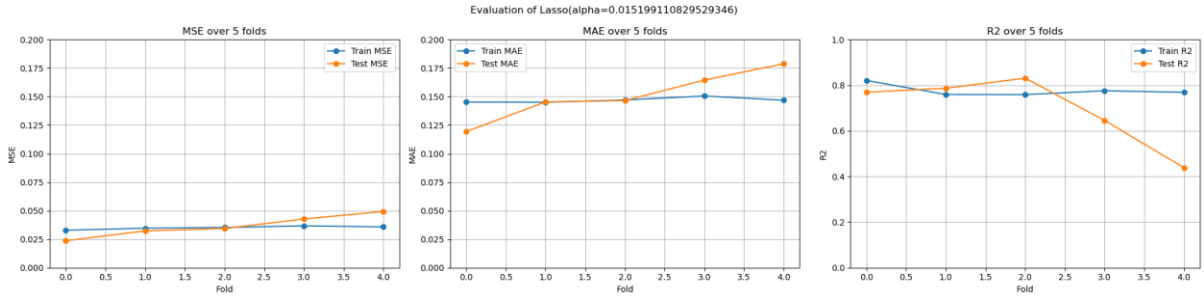

Figure 18 Lasso Regression after Hyperparamter Tuning and RFECV

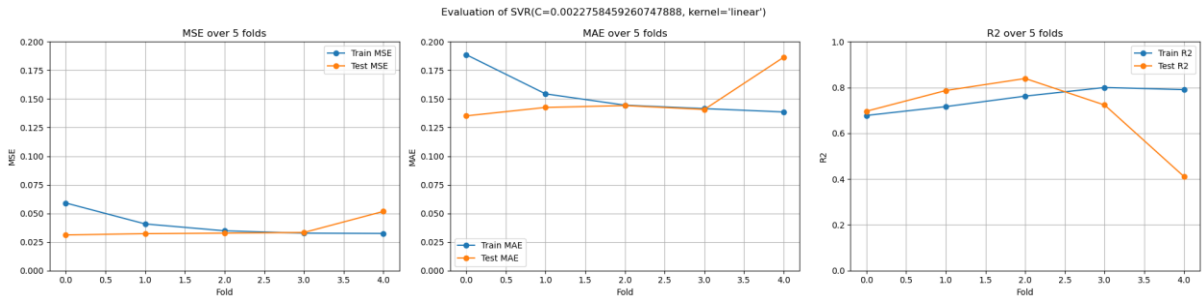

Figure 19 Support Vector Regressor after Hyperparameter tuning and RFECV

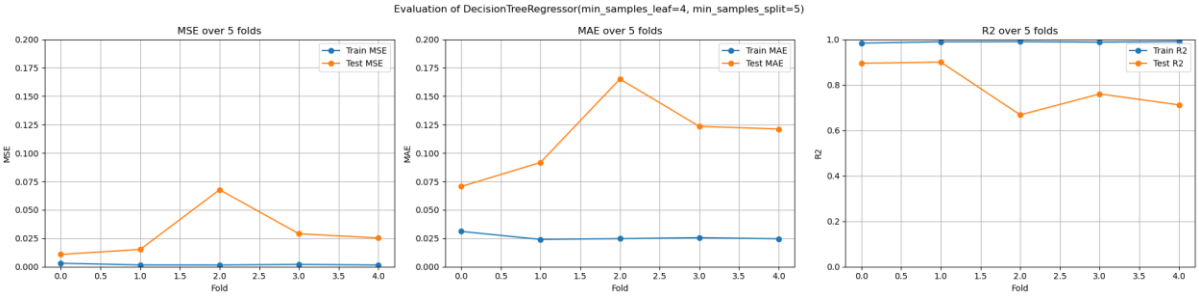

Figure 20 Decision Tree Regressor after Hyperparameter Tuning and RFECV

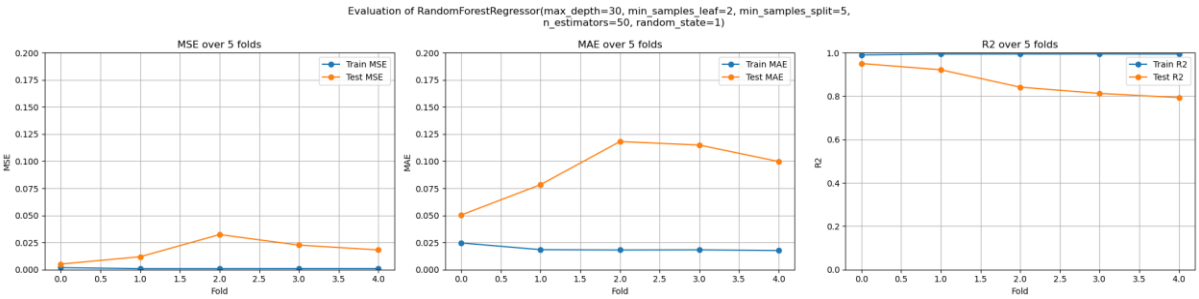

Figure 21 Random Forest Regressor after Hyperparameter tuning and RFECV

After hyper-parameter tuning and feature elimination, all models show improved validation performance, suggesting that these processes have effectively enhanced generalisation. The Linear Regression model's training R2 slightly decreased to 0.9225, but its validation R^2 increased to 0.4207, indicating a better balance between training and validation performance. Ridge Regression, with the specific tuning of alpha to 123, shows a much better validation R^2 of 0.6821 compared to its pre-tuning state, although training metrics indicate a considerable increase in error. Lasso Regression, tuned with an alpha of 0.015, also displays a substantial increase in validation R^2 to 0.6942, marking a significant improvement from its previous underfitting issue.

With reduced features, the tuned SVR shows very competitive validation metrics with an R^2 of 0.6913, mirroring the performance improvements in Ridge and Lasso Regression. The Decision Tree Regressor maintains an excellent training R^2 of 0.9890 and boosts its validation R2 to 0.6781, reducing overfitting significantly compared to its initial configuration.

The tuned RandomForestRegressor achieves a validation R2 of 0.8638 alongside the lowest validation MSE and RMSE, reflecting its highly effective learning and generalisation capabilities. These results highlight the considerable impact of hyper-parameter tuning and feature selection on the model's ability to predict unseen data accurately.

Experiment 4 – Final Model Evaluation

Fig 22 shows the results of the final models using the test dataset.

Final Model Testing with Test Set								
Model Name	**Training MSE**	**Training RMSE**	**Training MAE**	**Training R2-Score**	**Testing MSE**	**Testing RMSE**	**Testing MAE**	**Testing R2-Score**
LinearRegression()	0.0234	0.1528	0.1158	0.8395	0.1802	0.4245	0.2774	0.0490
Ridge(alpha=123)	0.0323	0.1796	0.1376	0.7784	0.1119	0.3345	0.2707	0.4096
Lasso(alpha=0.015)	0.0374	0.1934	0.1493	0.7429	0.0334	0.1829	0.1353	0.8235
SVR(C=0.0023, kernel='linear')	0.0346	0.1861	0.1426	0.7621	0.1089	0.3300	0.2667	0.4253
DecisionTreeRegressor(max_depth=50, min_samples_leaf=4)	0.0013	0.0364	0.0234	0.9909	0.0441	0.2101	0.1359	0.7671
RandomForestRegressor(max_depth=50, min_samples_leaf=4, min_samples_split=5)	0.0010	0.3162	0.0188	0.9931	0.0305	0.1747	0.1070	0.8389

Figure 22 Results for tuned models on the Test Dataset

The Random Forest Regressor demonstrates its superiority as the best model through its high R^2 score and lowest test RMSE and MAE score, indicating excellent variance capture and prediction accuracy. It achieves the lowest mean squared error and root mean squared error on the test set, signifying its predictions are consistently close to the actual values. Its performance shows minimal overfitting, suggesting it has effectively learned the general patterns in the data.

Experiment 5

Figure 23 shows the results after the final Random Forest model used on the remaining four datasets.

Dataset	**Training MSE**	**Training RMSE**	**Training MAE**	**Training R2-Score**	**Testing MSE**	**Testing RMSE**	**Testing MAE**	**Testing R2-Score**
AMZN	0.0010	0.3162	0.0188	0.9931	0.0305	0.1747	0.1070	0.8389
FB	0.00291	0.053941	0.035937	0.974727	0.118706	0.344537	0.258255	0.613632
MSFT	0.002501	0.050012	0.034671	0.975464	0.039911	0.199778	0.147267	0.539702
IBM	0.002402	0.049015	0.033178	0.980417	0.040745	0.201855	0.119424	0.709266
GOOGL	0.003069	0.055401	0.037952	0.980373	0.050707	0.225181	0.149433	0.629261

Figure 23 Final Models tested on different Stock Datasets

The model's ability to predict stock price movements for FB and MSFT is notably weaker, with accuracy and F1 scores dropping substantially in the test dataset. However, the model achieves a better level of generalisation with IBM and GOOGL data, although it still does not match its performance on the AMZN. This variability indicates that stock-specific features could impact the model's effectiveness, and adapting it to each stock might require retraining or additional tuning.

2.2 Classification Approach

2.2.1 Experiment Design

The classification approach in this project is that the percentage difference will be transformed into three classes: “up”, “down”, and “stable”. A stable threshold is set at 0.25% as this gives the best balance between the classes. The distribution of the classes is seen in Fig 24.

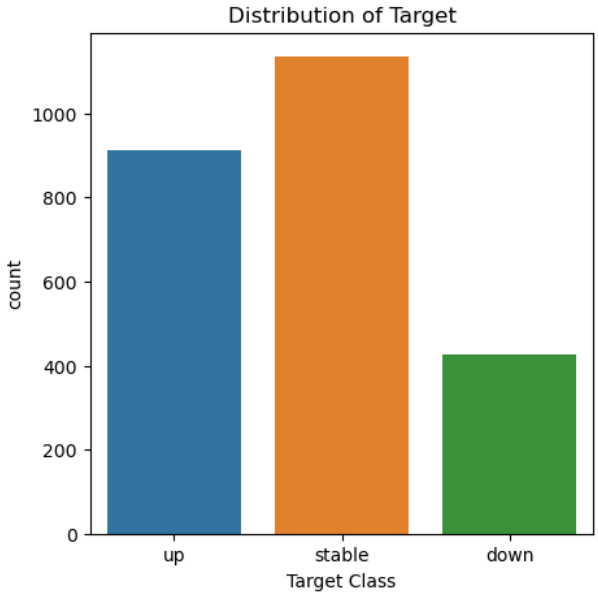

Figure 24 – Distrubtuion of target variable classes

The target classes are imbalanced, with the “stable” class having a frequency of 1135 instances and the “up” and “down” classes having frequencies of 991 and 426, respectively. This approach will have five main experiments:

- **Experiment 1 –** A range of classification models will be trained using the training dataset on all the features.
- **Experiment 2 –** The hyperparameters will be tuned using a search algorithm to find the optimal hyperparameters for each classification model. The optimised models will then be evaluated using all the features.
- **Experiment 3 -** Feature elimination will be done using the Recursive Feature Elimination algorithm.
- **Experiment 4 –** Test final models on the last test set.
- **Experiment 5 –** Evaluate the model's general Performance on the other four datasets.

The models selected for these experiments can be seen in Fig 25 [12]–[16].

Regression Model Name	Python Package Used
Logistic Regression	sklearn.linear_model.LogisticRegression()
Ridge Classification	Sklearn.linear_model.RidgeClassifier()
Support Vector Classification	Sklearn.svm.SVC()
Decision Tree Classifier	Sklearn.tree.DescisionTreeClassifier()
Random Forest Classifier	Sklearn.ensemble.RandomForestClassifier

Figure 25 Classification models used in this experiment

For model evaluation, the Table below highlights the primary metrics used:

- Balanced Accuracy
- (RMSE)
- Mean Absolute Error (MAE)
- R-Squared Score (R^2)
- ROC Curve (One Vs Many)
- AUC

These metrics will be weighted to ensure that the imbalance does not introduce errors in the metrics.

2.2.2 Results and Analysis

Experiment 1 – Initial Evaluation

Figure 26 displays the initial results for the classification models. Fig 27-31 shows the performance of the model during cross-validation.

Initial Classifier Test								
Model	**Mean Accuracy Train**	**Mean F1 Train**	**Mean Precision Train**	**Mean Recall Train**	**Mean Accuracy Test**	**Mean F1 Test**	**Mean Precision Test**	**Mean Recall Test**
LogisticRegression (max_iter=1000, multi_class='multinomial')	0.8389	0.8463	0.848	0.8466	0.8396	0.8467	0.8507	0.8492
RidgeClassifier()	0.7863	0.796	0.8003	0.7978	0.7866	0.7971	0.8137	0.7994
SVC(kernel='linear', probability=True)	0.8622	0.8702	0.8725	0.8705	0.8588	0.8689	0.8725	0.8705
DecisionTreeClassifier()	1	1	1	1	0.5817	0.5688	0.5852	0.6316
RandomForestClassifier(random_state=1)	1	1	1	1	0.5643	0.5874	0.6249	0.6365

Figure 26 Initial test on the models using default Hyperparamters and all predictor features

Logistic Regression performs well and consistently across training and testing datasets, indicating good fit and generalisation.

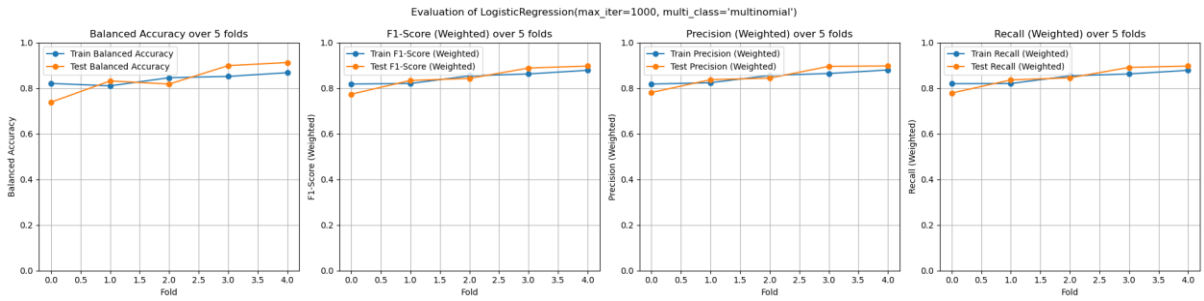

Figure 27 Logistic Regression Initial Evaluation after Time Series Cross Validation

The Ridge Classifier shows stable but slightly inferior performance to Logistic Regression, suggesting less capability in handling data complexity.

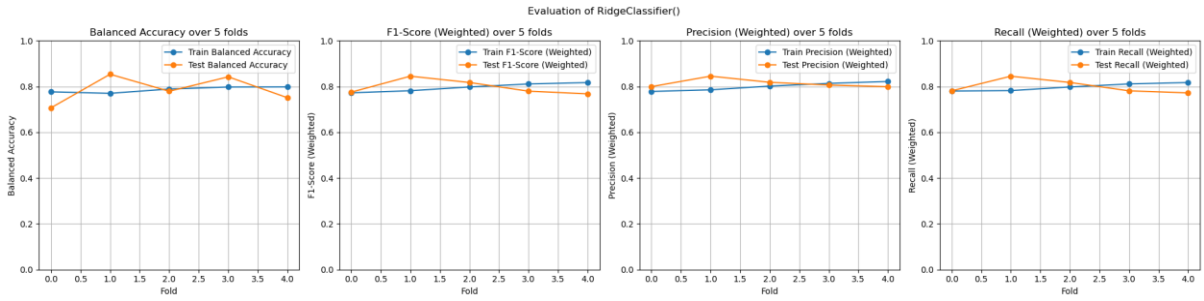

Figure 28 Ridge Classifier Initial Evaluation after Time Series Cross Validation

SVC demonstrates the best performance on the training data and maintains high performance on the testing data, suggesting it is the most robust and generalisable model among those tested.

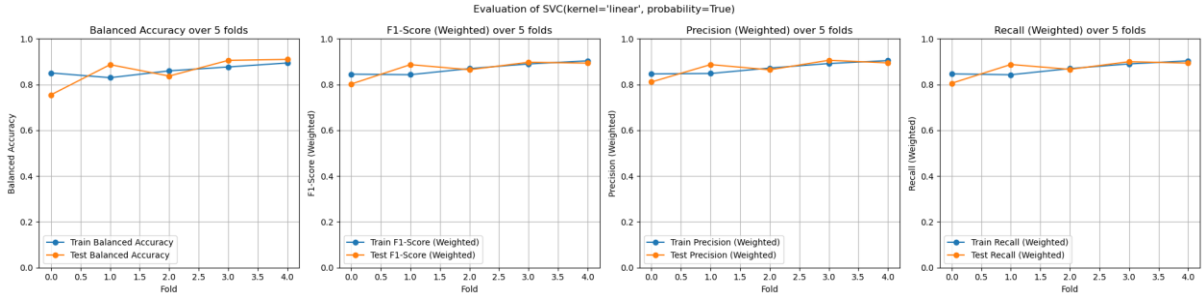

Figure 29 Support Vector Classifier Initial Evaluation after Time Series Cross Validation

Decision Tree Classifier exhibits perfect training scores but significant performance degradation on the testing set, indicating overfitting to the training data.

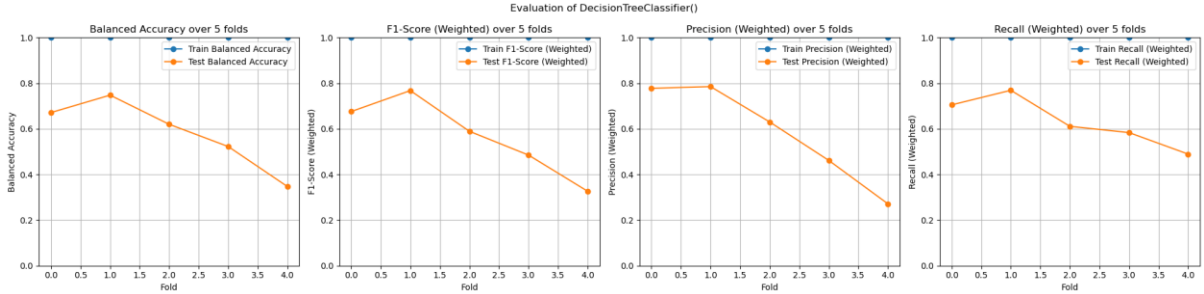

Figure 30 Decision Tree Classifier Initial Evaluation after Time Series Cross Validation

While also perfect on training data, the Random Forest Classifier suffers a notable performance drop on the testing set, even more so than the Decision Tree, pointing towards substantial overfitting.

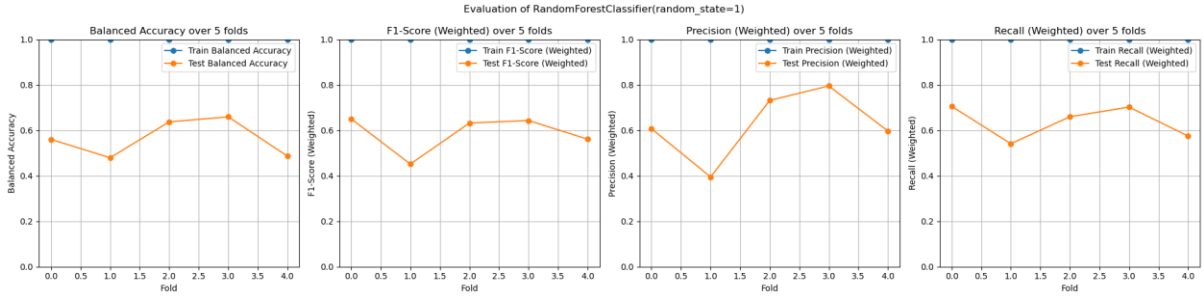

Figure 31 Random Forest Classifier Initial Evaluation after Time Series Cross Validation

Experiment 2 & 3 – Hyperparameter tuning and Feature Elimination

Figure 32 displays the results after Hyperparameter tuning and feature elimination. Fig 33-37 shows the model's performance across the cross-validation folds.

Results after Hyperparameter Tuning and Feature Elimination								
Model	**Mean Accuracy Train**	**Mean F1 Train**	**Mean Precision Train**	**Mean Recall Train**	**Mean Accuracy Test**	**Mean F1 Test**	**Mean Precision Test**	**Mean Recall Test**
LogisticRegression(max_iter=1000, multi_class='multinomial')	0.8148	0.8192	0.8209	0.8201	0.8427	0.8459	0.8476	0.8474
RidgeClassifier(alpha=4.328761281083062)	0.7668	0.7752	0.7801	0.7777	0.7819	0.8017	0.8102	0.8036
SVC(C=19.306977288832496, kernel='linear', probability=True)	0.9597	0.9585	0.959	0.9585	0.8927	0.9055	0.9135	0.907
DecisionTreeClassifier(max_depth=10, min_samples_leaf=4, min_samples_split=5, splitter='random')	0.9045	0.8992	0.9003	0.8993	0.7977	0.7685	0.7806	0.7708
RandomForestClassifier(bootstrap=False, max_depth=10, min_samples_leaf=4, n_estimators=200)	0.9856	0.9841	0.9842	0.9841	0.8003	0.8273	0.847	0.834

Figure 32 Classifier Models Evaluation after Hyperparameter Tuning and RFECV

After feature elimination, Logistic Regressions' training performance slightly decreased, but testing performance improved, suggesting that the model is now less overfit to the training data and generalises better to unseen data.

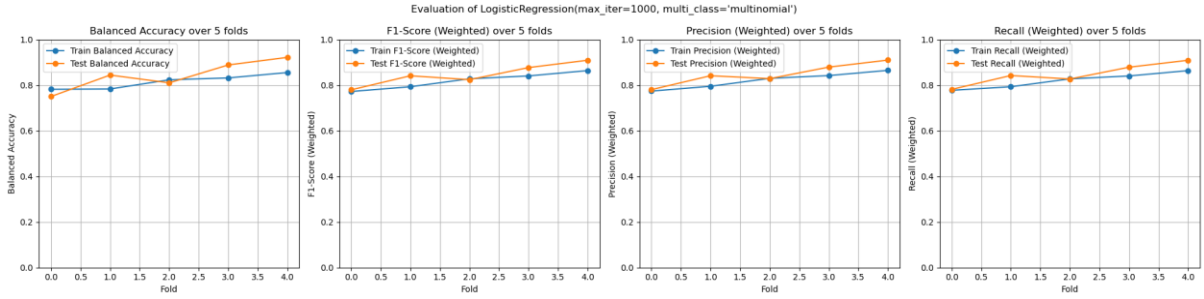

Figure 33 Logistic Regression metrics after feature elimination

For the Ridge Classifier, both training and testing performance decreased slightly. Despite this, increased F1, precision, and recall testing suggest better generalisation after feature selection and tuning.

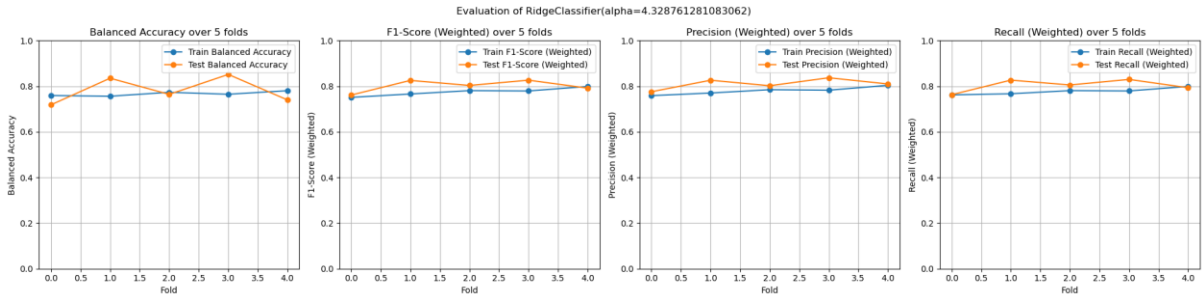

Figure 34 Ridge Classifier metrics after Hyperparameter Tuning and RFECV

For SVC, there is a slight decrease in training performance but a significant increase in testing performance, especially in F1 score and precision, indicating a substantial improvement in the model's generalisation ability.

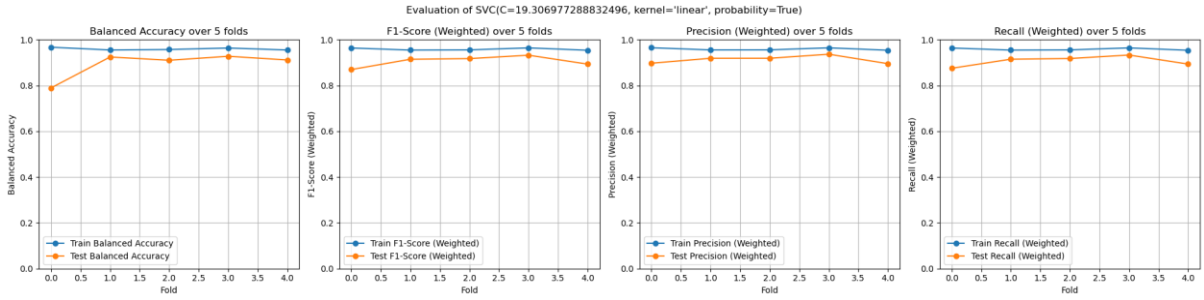

Figure 35 Support Vector Classifier Metrics after Hyperparameter Tuning and RFECV

Decision Tree Classifier significantly improves testing performance with much higher accuracy and F1 scores, indicating that hyperparameter tuning and feature elimination have effectively mitigated overfitting.

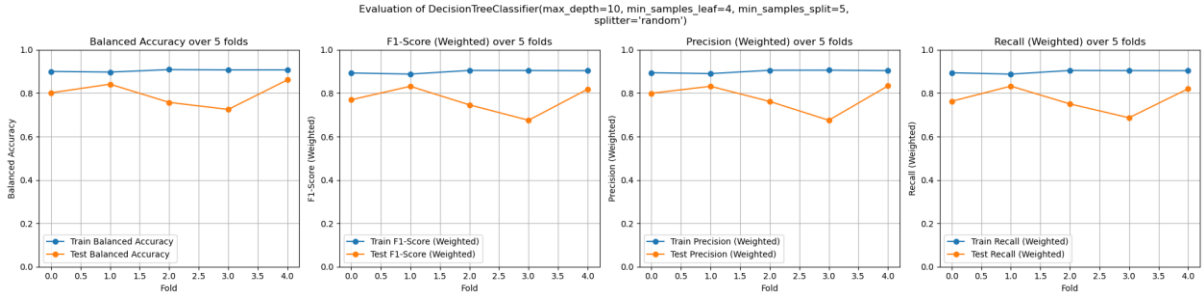

Figure 36 Decision Tree Classifier metrics after Hyperparameter Tuning and RFECV

While the Random Forest Classifier is still nearly perfect in training performance, the testing accuracy and other metrics have improved, showing better generalisation than the initial model. However, there may still be some overfitting.

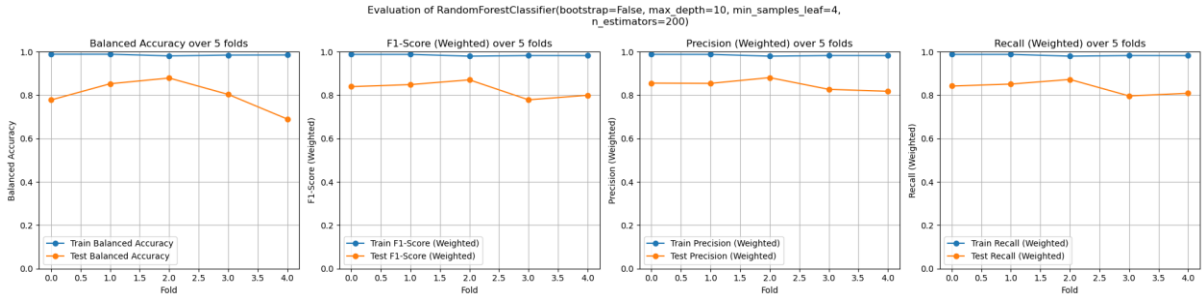

Figure 37 Random Forest Classifier after Hyperparameter tuning and RFECV

Experiment 4 – Final Model Evaluation

Fig 38 below shows the results for the final models on the test set.

Final Model Testing with Test Set								
Model Name	**Mean Accuracy Train**	**Mean F1 Train**	**Mean Precision Train**	**Mean Recall Train**	**Mean Accuracy Test**	**Mean F1 Test**	**Mean Precision Test**	**Mean Recall Test**
LogisticRegression(max_iter=1000, multi_class='multinomial')	0.8599	0.8695	0.8703	0.8695	0.9125	0.9093	0.9135	0.9091
RidgeClassifier(alpha=4.328761281083062)	0.7725	0.7991	0.8040	0.7997	0.8226	0.7989	0.8013	0.8000
SVC(C=19.306977288832496, kernel='linear', probability=True)	0.9424	0.9393	0.9394	0.9393	0.9085	0.9250	0.9273	0.9253
DecisionTreeClassifier(max_depth=10, min_samples_leaf=4, min_samples_split=5, splitter='random')	0.8842	0.8854	0.8860	0.8852	0.6988	0.7074	0.7147	0.7051
RandomForestClassifier(bootstrap=False, max_depth=10, min_samples_leaf=4, n_estimators=200)	0.9827	0.9808	0.9808	0.9808	0.9090	0.8891	0.8947	0.8889

Figure 38 Final Classifier Models Evaluation on Test Dataset

Among the evaluated models, the Logistic Regression and SVC perform best on the test set, with Logistic Regression achieving a balanced accuracy of 0.9124 and SVC slightly lower at 0.9085. Yet, SVC surpasses the highest weighted F1-score of 0.9253. The Ridge Classifier lags with the most inferior test performance, a balanced accuracy of 0.8226, and a weighted F1-score of 0.7989, indicating it might be less effective for complex predictions. The Decision Tree Classifier significantly underperforms on the test set, suggesting overfitting issues. The Random Forest Classifier, while showing high training performance, does not match the Logistic Regression or SVC on the test set, with a balanced accuracy of 0.9090 and a weighted F1-score of 0.8891. Fig shows the ROC curves for each model (except Ridge). Each class is plotted using the One vs Rest approach [17]. Fig 39 shows that the models can distinguish between the classes well, with AUC scores ranging between 0.93-0.99

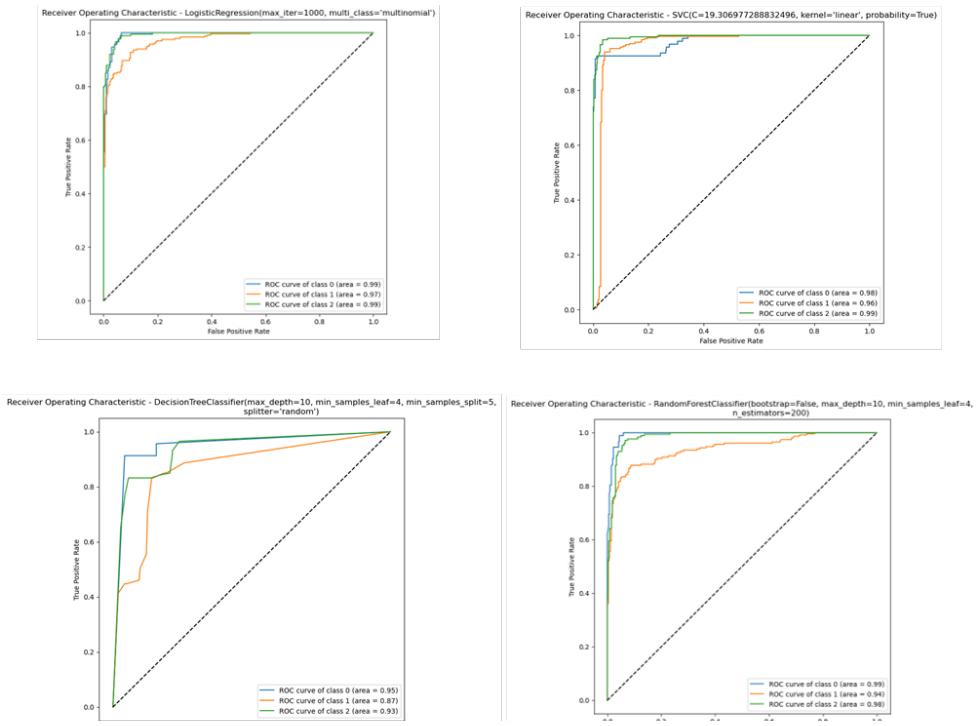

Figure 39 – ROC & AUC Scores for the final models (except Ridge Classifier)

The final model chosen is the Support Vector Classifier (SVC) due to its robust test set performance. It exhibits a high balanced accuracy of 0.9085 and the highest weighted F1-score of 0.9250, indicating

its strong predictive power and generalisation capability. Despite the Logistic Regression model's slightly higher balanced accuracy, the SVC's superior F1 score reflects a better balance between precision and recall, making it the more reliable choice for accurately classifying the direction of stock price movements. The SVC's performance metrics suggest it is particularly good at handling the nuances of the dataset, making it the preferred model for this application.

Experiment 5 – Testing on remaining datasets

Dataset	Mean Accuracy Train	Mean F1 Train	Mean Precision Train	Mean Recall Train	Mean Accuracy Test	Mean F1 Test	Mean Precision Test	Mean Recall Test
AMZN	0.9424	0.9393	0.9394	0.9393	0.9085	0.9250	0.9273	0.9253
FB	0.8741	0.8715	0.8716	0.8714	0.7667	0.6976	0.7336	0.6955
MSFT	0.8158	0.8524	0.8545	0.8543	0.7630	0.7528	0.8031	0.7684
IBM	0.8296	0.8662	0.8666	0.8674	0.8031	0.8550	0.8621	0.8577
GOOGL	0.8325	0.8501	0.8517	0.8506	0.8048	0.7817	0.7925	0.7802

Figure 40 Classifier Models tested on different Stock datasets.

As seen in Fig 40, the SVC models trained on AMZN data show varying effectiveness when applied to other stock datasets. It tends to generalise well to IBM's data, as demonstrated by the but shows limitations with FB, MSFT, and Google, highlighting potential issues with overfitting and the need for model adjustments to account for the unique characteristics of each company's stock price behaviour.

2.3 Clustering Approach

2.3.1 Experiment design

For this approach, the K-means algorithm will be used to cluster the dataset. The mean percentage change of each cluster will then be calculated, and a test set will be used to determine how close the predicted mean averages for each cluster are to the percentage change. Before clustering, the five datasets will be merged into a single dataset to capture all the patterns. The temporal nature of the time series data is not required, so the combined dataset is split using a random train-test split with 20 retained for testing.

To determine the optimal k value for K-Means clustering, an elbow plot is used with the cluster inertia used as the metric. As seen in Fig 41, the optimal k is four clusters.

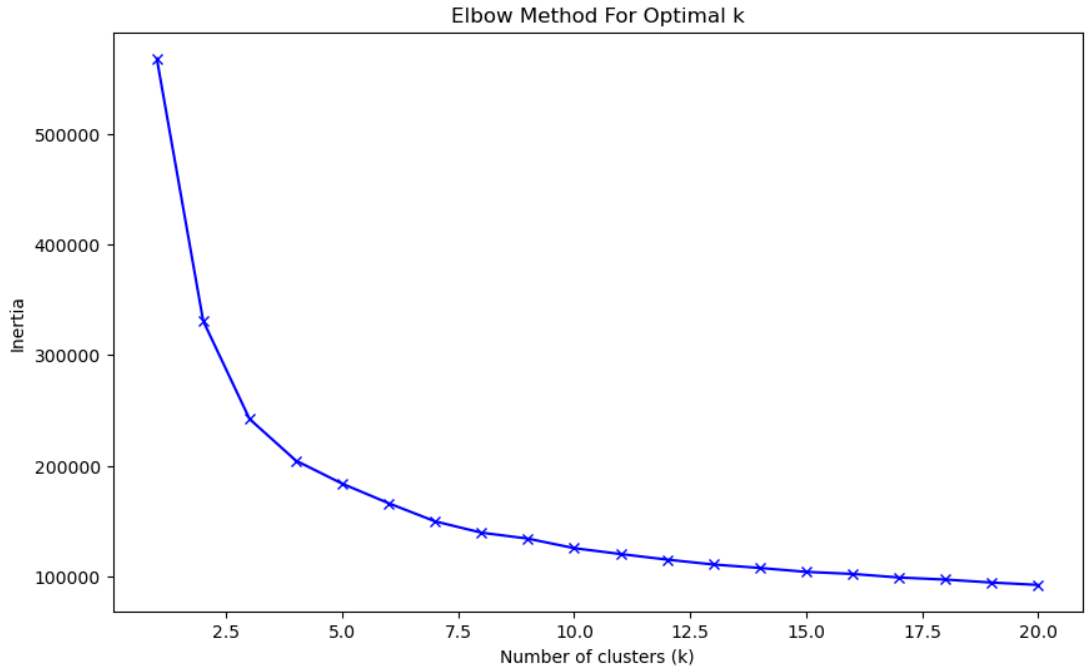

Figure 41 Elbow Plot to determine optimal k-value

Principal Component Analysis is carried out to reduce the dimensionality of the data. Fig 42 displays the Scree plot for the PCA, in which three principal components account for 80% of the variance. Both training and test sets are transformed.

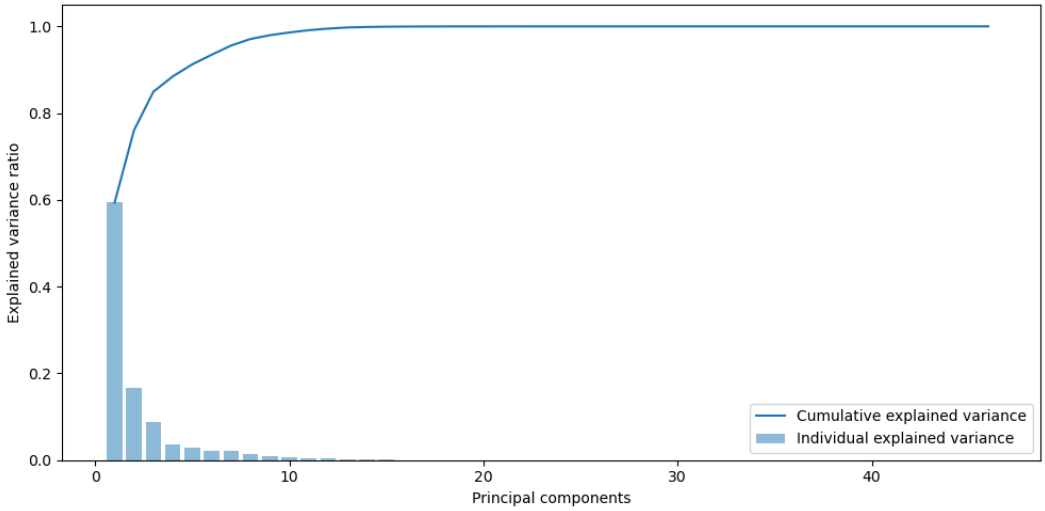

Figure 42 Explains the Variance Plot to determine the optimal number of principal components.

The dataset will be segmented into 4 clusters, of which the mean percentage change will be calculated for each. The test set is clustered using the same model.

2.3.2 Results and Analysis

The resultant clusters for the training are shown in Fig 43. The left plot shows a 2D representation of the cluster compared to each principal component, and the right plot shows the clusters in 3D space.

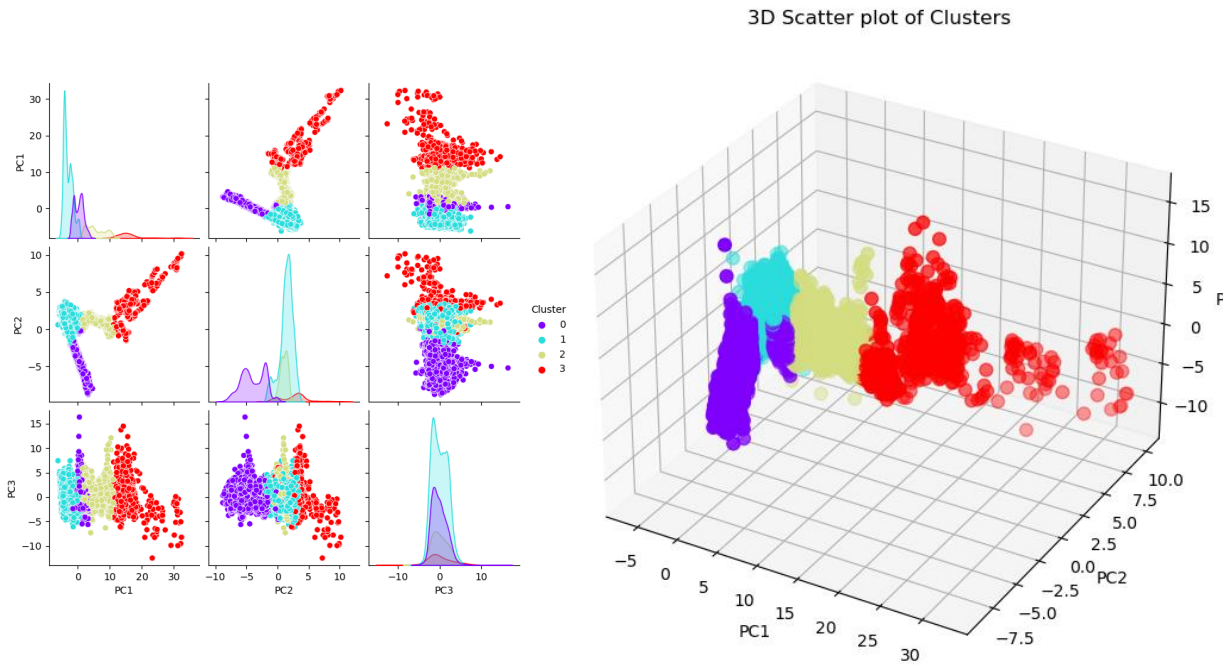

Figure 43- Resultant Cluster Visualisations

The mean average percentage difference is seen in fig 44. These mean values are then compared to the percentage difference of each instance in the test set, and the resultant metrics can be seen in fig 45. These results show a relatively high error for the clustering model, which highlights that the mean of this clustering may not be suitable for predicting the stock price.

Cluster	Mean Value
0	0.0646
1	0.0297
2	0.0940
3	0.1789

Figure 44 – Means values for each cluster

k	Mean Absolute Error (MAE)	Mean Squared Error (MSE)	Root Mean Squared Error (RMSE)
4	0.2940	1.7120	1.3084

Figure 45 - Metrics after cluster comparison with the test set

Phase 3 – Discussion & Conclusion

3.1 Comparison of Machine Learning Methods and Recommended Approach

In Fig 46 are the final models and their respective results when evaluated using a test set

Regression Approach Final Model								
Model Name	**Training MSE**	**Training RMSE**	**Training MAE**	**Training R2-Score**	**Testing MSE**	**Testing RMSE**	**Testing MAE**	**Testing R2-Score**
RandomForestRegressor(max_depth=50, min_samples_leaf=4, min_samples_split=5)	0.0010	0.3162	0.0188	0.9931	0.0305	0.1747	0.1070	0.8389

Classification Final Model								
Model Name	**Mean Accuracy Train**	**Mean F1 Train**	**Mean Precision Train**	**Mean Recall Train**	**Mean Accuracy Test**	**Mean F1 Test**	**Mean Precision Test**	**Mean Recall Test**
SVC(C=19.306977288832496, kernel='linear', probability=True)	0.9424	0.9393	0.9394	0.9393	0.9085	0.9250	0.9273	0.9253

Clustering Final Model				
Model Name	**k**	**Mean Absolute Error (MAE)**	**Mean Squared Error (MSE)**	**Root Mean Squared Error (RMSE)**
Kmeans(k=4)	4	0.2940	1.7120	1.3084

Figure 46 Best Models for results for each Machine Learning Approach

The SVC model excels in classification tasks, showcasing high accuracy and F1 scores, making it ideal for predicting the direction of price movement but not the exact values. K-means clustering is unsuited for direct predictions as it performed poorly. The hypothetical investor mentioned in the introduction will likely want to see estimated price values when making investment decisions. In this case, the final Random Forest Regressor appears to be the most suitable model for predicting the 20-day rolling average price thanks to its ability to output continuous values directly related to the task. It demonstrates superior performance, with low error metrics and high R2-Score on both training and testing sets, indicating its capability to generalise effectively to new data. This model is excellent at handling complex nonlinear relationships within the dataset and can be fine-tuned to mitigate overfitting. However, it may require more computational power and offers less interpretability than simpler models.

However, this model will only work when predicting AMZN stock price; therefore, its usefulness is limited to the investor. Further work would be required to investigate a more general predictive model or a custom solution could be explored, which can include data more relevant to the individual investors' stock portfolio. A feedback loop mechanism could also be investigated so real-time data stock data could update the model.

3.3 Conclusion

This project tested three approaches to predicting stock price using AMZN stock. Multiple models were tested for each approach, and three final models were selected, including Random Forest Regressor, Support Vector Classifier and k-means. The Random Forest Regressor was chosen as the best approach for this problem. However, its inability to be used accurately on other datasets limits its usefulness.

Final Word Count – 3278 words

References

[1] M. Vijh, D. Chandola, V. A. Tikkiwal, and A. Kumar, 'Stock Closing Price Prediction using Machine Learning Techniques', in *Procedia Computer Science*, Elsevier B.V., 2020, pp. 599–606. doi: 10.1016/j.procs.2020.03.326.

[2] W. Huang, Y. Nakamori, and S. Y. Wang, 'Forecasting stock market movement direction with support vector machine', *Comput Oper Res*, vol. 32, no. 10, pp. 2513–2522, Oct. 2005, doi: 10.1016/j.cor.2004.03.016.

[3] X. Wang, K. Yang, and T. Liu, 'Stock Price Prediction Based on Morphological Similarity Clustering and Hierarchical Temporal Memory', *IEEE Access*, vol. 9, pp. 67241–67248, 2021, doi: 10.1109/ACCESS.2021.3077004.

[4] Scikit-learn Developers, 'sklearn. preprocessing.StandardScaler — scikit-learn 1.3.2 documentation'. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[5] Scikit-learn Developers, 'sklearn.linear_model.LinearRegression — scikit-learn 1.3.2 documentation'. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

[6] Scikit-learn Developers, 'sklearn.linear_model.Ridge — scikit-learn 1.3.2 documentation'. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html

[7] Scikit-learn Developers, 'sklearn.linear_model.Lasso — scikit-learn 1.3.2 documentation'. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

[8] Scikit-learn Developers, 'sklearn.svm.SVR — scikit-learn 1.3.2 documentation'. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

[9] Scikit-learn Developers, 'sklearn.tree.DecisionTreeRegressor — scikit-learn 1.3.2 documentation'. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

[10] Scikit-learn Developers, 'sklearn.ensemble.RandomForestRegressor — scikit-learn 1.3.2 documentation'. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

[11] Scikit-learn Developers, 'sklearn.model_selection.TimeSeriesSplit — scikit-learn 1.3.2 documentation'. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html

[12] Scikit-learn Developers, 'sklearn.linear_model.LogisticRegression — scikit-learn 1.3.2 documentation'. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[13] Scikit-learn Developers, 'sklearn.linear_model.RidgeClassifier — scikit-learn 1.3.2 documentation'. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RidgeClassifier.html

[14] Scikit-learn Developers, ‘sklearn.svm.SVC — scikit-learn 1.3.2 documentation’. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

[15] Scikit-learn Developers, ‘sklearn.tree.DecisionTreeClassifier — sci-kit-learn 1.3.2 documentation’. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

[16] Scikit-learn Developers, ‘sklearn.ensemble.RandomForestClassifier — sci-kit-learn 1.3.2 documentation’. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[17] Scikit-learn Developers, ‘Multiclass Receiver Operating Characteristic (ROC) — sci-kit-learn 1.3.2 documentation’. Accessed: Nov. 08, 2023. [Online]. Available: https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

Appendix A – Project Python Code

```
#!/usr/bin/env python
# coding: utf-8

# ECS8051 - Machine Learning Jupyter Notebook
#

# Mark Daly - 40418892

# In[1]:



#Load Python Packages
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt



# In[2]:



#Load AMZN dataset
df_amzn = pd.read_csv("AMZN.csv")
pd.set_option('display.max_columns', None)
df_amzn.head()



# In[3]:
```

```
#Set Date to index which allows for easier visualisation of the data over time
df_amzn['Date'] = pd.to_datetime(df_amzn['Date'])
df_amzn.set_index('Date', inplace=True)


# In[4]:


#Goal is to predict what the stocks closing price is in the the next month
#Therfore, the target feature should be Close(t+20) for Closing Price + 20 traiding days
#Create Target Variable by shifting the close(t) variable forward by twenty days to find next months price
df_amzn["Close(t+20)"] = df_amzn['Close(t)'].shift(-20)
df_amzn.head()


# In[5]:


df_amzn.describe()


# DETERMINE THE TARGET VARIABLE

# In[6]:


#Plot the Closing price +20Day
plt = df_amzn["Close(t+20)"].plot(figsize=(15,8))
plt.set_xlabel("Year")
```

```
plt.set_ylabel("Price")
plt.set_title("Next Month Future Closing Price of AMZN Stock")



# The graph shows that AMZN's stock price at a steady rate until approx 2015 then experiences the rate of changes increases exponentialy until 2021
# The data set appears to have periods of peaks and troughs which is standard due to the volatile nature of stocks
# Interestingly, the dataset also contains a "MA20" which corresponds to the Moving Average over the last 20 Days


# In[7]:



#Compare the Closing price +20Day with MA20
targets = ['MA20', "Close(t+20)" ]


for i in targets:
    plt = df_amzn[i].plot(figsize=(15,8))
    plt.set_xlabel("Year")
    plt.set_ylabel("Price")
    plt.set_title("Comaprison of 'Closing Price+20Day' with 'MA20'")
    plt.legend()



# The MA20 feature captures the same general trend of the Individual Stock prices however it reduces the noise caused by the daily volatility of the stock price. Stock prices are influenced daily by macroeconomic, microeconomic and geopolitical events therefore it is theoretically very difficult predict the exact stock price at a certian point in future. However, for the purposes of the hypothetical investor, the knowing the estimated price at certian time should be sufficent to make a descison wether to buy, sell or hold stocks


# For the purposes for this the MA20 will be set as the target and the "Close(t+20)" will be dropped
```

In[8]:

```
#Set MA20 as target by creating a new column
df_amzn["MA20_target"] = df_amzn["MA20"]

# Drop the "Close(t+20)" and redundant "MA20" columns
df_amzn = df_amzn.drop(columns=["Close(t+20)", "MA20"])
```

In[9]:

```
plt = df_amzn["MA20_target"].plot(figsize=(15,8))
plt.set_xlabel("Year")
plt.set_ylabel("20-Day Moving Average Price")
plt.set_title("MA20-Target")
plt.legend()
```

The next problem with this dataset is it's non-stationary nature (i.e its statiscal properoties mean, variance etc are changing over time). The problem is that to maintain the temporl nature of this time series requires the test set to be a future time to the training set to ensure bias is not introduced into the model. The problem in this case is that the test set will include the period between 2020-2021 where the price sees a dramatic rise in price not seen previously. Therefore the model's training process has not been trainined to see this trend and will likely provide in accurate predictions.

#

One method to stabilise the data would be to convert the price into a percentage change using the pandas pcd_change() function which calculates the percentage between the current and previous row in a dataframe

In[10]:

```
df_amzn["percentage_change"]= df_amzn["MA20_target"].pct_change() * 100
df_amzn.head()



# In[33]:



df_amzn.info()



# In[11]:



#Plot Percentage Change
plt = df_amzn["percentage_change"].plot(figsize=(15,8))
plt.set_xlabel("Year")
plt.set_ylabel("Percentage Change")
plt.set_title("20-Day Moving Average Percentage Change")
plt.legend()



# As seen above, the data has been transformed and captures the rise and fall of the rolling 20 day average price over the timeframe
#
# To highlight the effect, a simple linear regression model will be train as baseline and to high light the effect this transformation will have on the performance of the models



# EDA on MA20 with other variables



# In[12]:
```

```
#Exploratory Data Analysis to determine relationships between each feature and percentage change
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings(action='ignore', category=FutureWarning)
# Calculate the number of features
num_features = len(df_amzn.columns)
num_rows = -(-num_features // 3)

# Create a grid of subplots with 3 columns and required number of rows
fig, axes = plt.subplots(nrows=num_rows, ncols=3, figsize=(15, 6*num_rows))

# Flatten the axes for easier indexing
axes = axes.ravel()

for idx, feat in enumerate(df_amzn.columns):
    sns.scatterplot(data=df_amzn, x=feat, y="percentage_change", ax=axes[idx])
    axes[idx].set_title(f"{feat} vs.percentage_change")
    axes[idx].set_xlabel(f"{feat}")
    axes[idx].set_ylabel("Percentage_change")



# In[13]:



# First some basic data pre-processing is required to run the Random Forest regression model
# Drop the columns below
cols_to_drop = [
    'Date_col', #Duplicate of the index
    'Close_forcast', #Predicted forecast alread in dataset - removed so biased is not introduced into the
model
```

```
    'CCI', # This is constant for all instances therefore will have no impact on model prediction performance
    'Day', 'DayofWeek', 'DayofYear', 'Week',
        'Is_month_end', 'Is_month_start', 'Is_quarter_end', 'Is_quarter_start',
        'Is_year_end', 'Is_year_start', 'Is_leap_year', 'Year', 'Month' # These don't appear to have any relevance to the predicting the stock price



]
df_amzn = df_amzn.drop(columns=cols_to_drop)



# In[14]:



# Check for missing values
df_amzn.isna().sum()



# In[15]:



#Drop missing values
df_amzn = df_amzn.dropna(axis=0)



# In[17]:



df_amzn.describe()
```

In[18]:

```
#Plot a correlation matrix to see realtionships between target
corr = df_amzn.corr()
print("Correlation Matrix")
print(corr)
```

In[422]:

```
fig, ax = plt.subplots(figsize=(35,35))
sns.heatmap(corr, annot=True, ax=ax)
```

In[19]:

```
#Separate into Target and Feature - this will be done twice to highlight the effect of the data transformation

y_MA20 = df_amzn["MA20_target"]
y_pc = df_amzn["percentage_change"]

X = df_amzn.drop(columns=["MA20_target", "percentage_change"])
```

In[20]:

```
#Normalise the Dataset for model generation - not required for linear reagression but for more complex modeeling

from sklearn.preprocessing import StandardScaler


scaler = StandardScaler()


X_scaled = scaler.fit_transform(X)


#Convert back to DataFrame to return column names back to X_scaled

X_scaled = pd.DataFrame(X_scaled, columns=X.columns)




# In[21]:



# Separate the dataset into a training which will be used for training and tuning the model. The test set will be retainined for the final model evaluation

# 80% of the data will be used for training, validation and tuning while the reamining will be retained for final model evaluation

split_data = int(len(X_scaled) * 0.80)


X_train = X_scaled[:split_data]

X_test = X_scaled[split_data:]



y_train_MA20 = y_MA20[:split_data]

y_test_MA20 = y_MA20[split_data:]


y_train_pc = y_pc[:split_data]

y_test_pc = y_pc[split_data:]
```

In[23]:

```
#Plot Y_train and Y_test to verify data haas split correctly
plt.figure(figsize=(10,6))
plt.plot(y_train_MA20, color="red", label="Training Data")
plt.plot(y_test_MA20, color="blue", label="Test Data")
plt.legend()

plt.figure(figsize=(10,6))
plt.plot(y_train_pc, color="red", label="Training Data")
plt.plot(y_test_pc, color="blue", label="Test Data")
plt.legend()
```

In[24]:

```
# Test on Random Forest regression model with all features as a benchmark to test which is best target to use
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from math import sqrt

lr = RandomForestRegressor(bootstrap=True, random_state=1)

# First test the MA20 Target
lr.fit(X_train, y_train_MA20)

y_pred = lr.predict(X_test)
```

```
# Metrics for MA20 as Target
mse = mean_squared_error(y_test_MA20, y_pred)
rmse = sqrt(mse)
mae = mean_absolute_error(y_test_MA20, y_pred)
r2 = r2_score(y_test_MA20, y_pred)

print("WITH MA20 as Target")
print(f"MSE = {mse}")
print(f"RMSE = {rmse}")
print(f"MAE = {mae}")
print(f"R2-Score = {r2}")



# Plot for MA20 as Target
plt.figure(figsize=(12, 6))
plt.scatter(y_test_MA20, y_pred, color='blue')
plt.plot([min(y_test_MA20), max(y_test_MA20)], [min(y_test_MA20), max(y_test_MA20)],
color='red')  # diagonal line
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Actual vs Predicted for MA20 Target')
plt.show()

lr.fit(X_train, y_train_pc)

y_pred = lr.predict(X_test)

# Metrics for Percentage Change as Target
mse_pc = mean_squared_error(y_test_pc, y_pred)
rmse_pc = sqrt(mse_pc)
```

```
mae_pc = mean_absolute_error(y_test_pc, y_pred)
r2_pc = r2_score(y_test_pc, y_pred)

print("\nWITH Percentage Change as Target")
print(f"MSE = {mse_pc}")
print(f"RMSE = {rmse_pc}")
print(f"MAE = {mae_pc}")
print(f"R2-Score = {r2_pc}")



# Plot for Percentage Change as Target
plt.figure(figsize=(12, 6))
plt.scatter(y_test_pc, y_pred, color='green')
plt.plot([min(y_test_pc), max(y_test_pc)], [min(y_test_pc), max(y_test_pc)], color='red')  # diagonal line
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Actual vs Predicted for Percentage Change Target')
plt.show()




# Percentage change generalises better to model as hown by higher r2-score


# Regression Model


# In[25]:
```

```
# Define Function
from sklearn.model_selection import TimeSeriesSplit, cross_validate
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from math import sqrt

#To keep notebook tidy a "initial model test" function is created. the model will be defined in each notebook cell and passed into the function


def int_model_testing(model, X_train, y_train):
    #To carry out cross-validation on the training data while ensuring that the data isn't randomly shuffled TimeSeries Split from sklearn.model_selection will be used
    #This will carry out a walk forward validation

    #Define number of splits
    n_splits=5
    tscv = TimeSeriesSplit(n_splits=n_splits)

    #Define the metrics that the cross validation will use to test the model (neg prefix inicates that lower value is better)
    reg_metrics = {
        "mean_squared_error": "neg_mean_squared_error",
        "mean_absolute_error": "neg_mean_absolute_error",
        "r2_score": "r2"
    }

    #Use the cross_validate function to calculate training and testing scores
    results = cross_validate(model, X_train, y_train, cv=tscv, return_train_score=True, return_estimator= True, scoring=reg_metrics)

    #Extract training set scores into lists
```

```
mse_train_scores = results["train_mean_squared_error"]
abs_err_train_scores = results["train_mean_absolute_error"]
r2_score_train_scores = results["train_r2_score"]

#Extract Test scores into lists
mse_test_scores = results["test_mean_squared_error"]
abs_err_test_scores = results["test_mean_absolute_error"]
r2_score_test_scores = results["test_r2_score"]

# Merge lists into pandas Dataframe to create table

train_test_results = {

    "Training MSE Score" : (-mse_train_scores),
    "Training MAE Score": (-abs_err_train_scores),
    "Training R2-Score": (r2_score_train_scores),
    "Testing MSE Score": (-mse_test_scores),
    "Testing MAE Score": (-abs_err_test_scores),
    "Testing R2-Score": (r2_score_test_scores)
}
fold_results_table = pd.DataFrame(train_test_results)

mean_mse_train_score = -1* (mse_train_scores.mean())
mean_rmse_train_score = sqrt(mean_mse_train_score)
mean_mae_train_score = -1* (abs_err_train_scores.mean())
mean_r2_train_score = r2_score_train_scores.mean()

mean_mse_test_score = -1* (mse_test_scores.mean())
mean_rmse_test_score = sqrt(mean_mse_test_score)
mean_mae_test_score = -1 * (abs_err_test_scores.mean())
mean_r2_test_score = r2_score_test_scores.mean()
```

```
mean_scores = {
    "Average Training MSE Score" : (mean_mse_train_score),
    "Average Training RMSE Score": (mean_rmse_train_score),
    "Average Training MAE Score": (mean_mae_train_score),
    "Average Training R2-Score": (mean_r2_train_score),
    "Average Validation MSE Score" : (mean_mse_test_score),
    "Average Validation RMSE Score": (mean_rmse_test_score),
    "Average Validation MAE Score": (mean_mae_test_score),
    "Average Validation R2-Score": (mean_r2_test_score)
}

mean_results_table = pd.DataFrame([mean_scores], index=[str(model)])

# Plotting the scores
metrics = [("MSE", "Training MSE Score", "Testing MSE Score"),
           ("MAE", "Training MAE Score", "Testing MAE Score"),
           ("R2", "Training R2-Score", "Testing R2-Score")]

plt.figure(figsize=(20,5))
plt.suptitle(F"Evaluation of {model}")




y_limits=[(0, 0.2), (0, 0.2), (0, 1)]
for i, (metric_name, train_label, test_label) in enumerate(metrics, 1):  # Start index from 1
    plt.subplot(1, 3,i)  # One row, three columns, current index
    plt.plot(fold_results_table[train_label], '-o', label='Train ' + metric_name)
    plt.plot(fold_results_table[test_label], '-o', label='Test ' + metric_name)
    plt.title(f"{metric_name} over {str(n_splits)} folds")
    plt.ylim(y_limits[i-1])
```

```
        plt.xlabel('Fold')
        plt.ylabel(metric_name)
        plt.legend()
        plt.grid(True)
        plt.tight_layout()

    # Plotting residuals
    plt.figure(figsize=(10, 5))
    for i, (train_idx, test_idx) in enumerate(tscv.split(X_train)):
        estimator = results['estimator'][i]
        y_pred = estimator.predict(X_train.iloc[test_idx])
        residuals = y_train_MA20.iloc[test_idx] - y_pred
        plt.scatter(y_pred, residuals, label=f'Fold {i+1}')
    plt.axhline(0, color='red', linestyle='--')
    plt.xlabel('Predicted Values')
    plt.ylabel('Residuals')
    plt.title('Residual Plot using TimeSeries CV')
    plt.legend()
    plt.show()

    return fold_results_table, mean_results_table



# In[26]:



#Test a linear Regression Model
from sklearn.linear_model import LinearRegression
```

```
lr = LinearRegression()

lr_fold_results_table, lr_mean_results_table = int_model_testing(lr, X_train, y_train_pc)

print(lr_fold_results_table)
lr_mean_results_table


# In[27]:


#Test a Ridge Regression Model
from sklearn.linear_model import Ridge


ridge = Ridge()

ridge_fold_results_table, ridge_mean_results_table = int_model_testing(ridge,  X_train, y_train_pc)

print(ridge_fold_results_table)
ridge_mean_results_table


# In[28]:


#Test a Lasso Regression Model
from sklearn.linear_model import Lasso


lasso =Lasso()
```

```
lasso_fold_results_table, lasso_mean_results_table = int_model_testing(lasso,  X_train, y_train_pc)

print(lasso_fold_results_table)
lasso_mean_results_table


# In[29]:


#Test a Decision Tree Model
from sklearn.tree import DecisionTreeRegressor


tree = DecisionTreeRegressor()

tree_fold_results_table, tree_mean_results_table = int_model_testing(tree,  X_train, y_train_pc)

print(tree_fold_results_table)
tree_mean_results_table


# In[30]:


#Test a Random Forest Model
from sklearn.ensemble import RandomForestRegressor


rf = RandomForestRegressor()
```

```
rf_fold_results_table, rf_mean_results_table = int_model_testing(rf,  X_train, y_train_pc)

print(rf_fold_results_table)
rf_mean_results_table


# In[31]:


#Test a SVR Model
from sklearn.svm import SVR


svr = SVR(kernel="linear")

svr_fold_results_table, svr_mean_results_table = int_model_testing(svr,  X_train, y_train_pc)

print(svr_fold_results_table)
svr_mean_results_table


# Hyperparamter tuning - Carry Out inital Round of hyper parameter tuning so model can explore interations with entire dataset

# In[32]:


param = "Linear Regression "  # No hyperparameters to tune for Linear Regression


# In[437]:
```

```
from sklearn.model_selection import GridSearchCV



#Ridge regression - change regularisation strength

model = Ridge()

param = {

    'alpha': np.logspace(-3,3,100)

    }

tscv = TimeSeriesSplit(n_splits=5)

grid_search = GridSearchCV(model, param, cv=tscv, scoring='neg_mean_squared_error',
return_train_score=True, verbose=10)

grid_search.fit(X_train, y_train_pc)


    # Save the best model

optimised_ridge = grid_search.best_estimator_


    # Store results for plotting

results_ridge = grid_search.cv_results_


print(f"Best hyperparameters for Ridge: {grid_search.best_params_}")



# In[438]:



#Test a Ridge Regression Model

from sklearn.linear_model import Ridge


ridge_fold_results_table, ridge_mean_results_table = int_model_testing(optimised_ridge,  X_train,
y_train_pc)
```

```
print(ridge_fold_results_table)
ridge_mean_results_table


# In[439]:


#Lasso regression - change regularisation strength
model = Lasso()
param = {
    'alpha': np.logspace(-4,4,100)
    }
tscv = TimeSeriesSplit(n_splits=5)
grid_search = GridSearchCV(model, param, cv=tscv, scoring='neg_mean_squared_error',
return_train_score=True, verbose=10)
grid_search.fit(X_train, y_train_pc)

    # Save the best model
optimised_lasso = grid_search.best_estimator_

    # Store results for plotting
results_lasso = grid_search.cv_results_

print(f"Best hyperparameters for Lasso: {grid_search.best_params_}")


# In[440]:


#Test a Lasso Regression Model
```

```
lasso_fold_results_table, lasso_mean_results_table = int_model_testing(optimised_lasso,  X_train, y_train_pc)


print(lasso_fold_results_table)

lasso_mean_results_table



# In[441]:



#SVR - change regularisation

model = SVR(kernel="linear")

param = {

   'C': np.logspace(-3,2,15),

}

tscv = TimeSeriesSplit(n_splits=5)

grid_search = GridSearchCV(model, param, cv=tscv, scoring='neg_mean_squared_error', return_train_score=True, verbose=10)

grid_search.fit(X_train, y_train_pc)


   # Save the best model

optimised_svr = grid_search.best_estimator_


   # Store results for plotting

results_svr = grid_search.cv_results_


print(f"Best hyperparameters for SVR: {grid_search.best_params_}")



# In[442]:
```

```
#Test a SVR Regression Model


svr_fold_results_table, svr_mean_results_table = int_model_testing(optimised_svr,  X_train,
y_train_pc)


print(svr_fold_results_table)

svr_mean_results_table



# In[443]:



#Decision Tree - change max_depth, min_samples_split, min_samples_leaf

model = DecisionTreeRegressor()

param = {

   'max_depth': [None, 10, 20, 30, 40, 50],

   'min_samples_split': [2, 5, 10],

   'min_samples_leaf': [1, 2, 4],

   'splitter': ['best', 'random'],


}

tscv = TimeSeriesSplit(n_splits=5)

grid_search = GridSearchCV(model, param, cv=tscv, scoring='neg_mean_squared_error',
return_train_score=True, verbose=10)

grid_search.fit(X_train, y_train_pc)


   # Save the best model

optimised_tree = grid_search.best_estimator_
```

```
    # Store results for plotting
results_tree = grid_search.cv_results_

print(f"Best hyperparameters for Decision Tree: {grid_search.best_params_}")



# In[444]:



#Test a Deccision Tree Regression Model



tree_fold_results_table, tree_mean_results_table = int_model_testing(optimised_tree,  X_train,
y_train_pc)

print(tree_fold_results_table)

tree_mean_results_table



# In[445]:



from sklearn.model_selection import RandomizedSearchCV

#Random Forest - change n_stimators, max_depth, min_samples_split, min_samples_leaf

model = RandomForestRegressor(bootstrap=True, random_state=1)

param= {

    'n_estimators': [1,5, 10, 25, 50, 100, 200],

    'max_depth': [1,5, 10, 20, 30, 40, 50],

    'min_samples_split': [2, 5, 10],

    'min_samples_leaf': [1, 2, 4],
```

```
}
tscv = TimeSeriesSplit(n_splits=5)
grid_search = RandomizedSearchCV(model, param, cv=tscv, scoring='neg_mean_squared_error', return_train_score=True,  n_iter=50, verbose=10, random_state=1)
grid_search.fit(X_train, y_train_pc)

    # Save the best model
optimised_rf = grid_search.best_estimator_

    # Store results for plotting
results_rf = grid_search.cv_results_

print(f"Best hyperparameters for Random Forest: {grid_search.best_params_}")



# In[446]:



#Test a Random Forest Regression Model



rf_fold_results_table, rf_mean_results_table = int_model_testing(optimised_rf,  X_train, y_train_pc)

print(rf_fold_results_table)
rf_mean_results_table



# FEATURE ELIMINATION USING Backwards Eliminaton using Recurssive Feature Elimination - Importance Measure is set to auto (coef_ for linear regression models & feature importance for tree based models)

# In[447]:
```

```
from sklearn.feature_selection import RFECV
# Define function for Feature elimation
def backwards_elim(model, X_train, y_train, X_test):
    tscv = TimeSeriesSplit(n_splits=5)
    selector = RFECV(model, step=1, cv=tscv, scoring="neg_mean_squared_error", verbose=10)
    selector = selector.fit(X_train, y_train)
    selected_feats = pd.DataFrame(X_train).columns[selector.support_].to_list()

#Transform both training and testing sets
    X_train_red = selector.transform(X_train)
    X_train_red = pd.DataFrame(X_train_red, columns=selected_feats)

    X_test_red = selector.transform(X_test)
    X_test_red = pd.DataFrame(X_test_red, columns=selected_feats)

    return X_train_red, selected_feats, X_test_red



# In[448]:



# Test Linear Regression Model after RFECV

lr_X_train_red, lr_selected_feats, lr_X_test_red = backwards_elim(lr, X_train, y_train_pc, X_test)
print(lr_selected_feats)

lr_fold_results_table_feats_select, lr_mean_results_table_feats_select = int_model_testing(lr,
lr_X_train_red, y_train_pc)
```

```
print(lr_fold_results_table_feats_select)

lr_mean_results_table_feats_select



# In[449]:



# Test Ridge Regression Model after RFECV



ridge_X_train_red, ridge_selected_feats, ridge_X_test_red = backwards_elim(optimised_ridge,
X_train, y_train_pc, X_test)

print(ridge_selected_feats)


ridge_fold_results_table_feats, ridge_mean_results_table_feats = int_model_testing(optimised_ridge,
ridge_X_train_red, y_train_pc)


print(ridge_fold_results_table_feats)

ridge_mean_results_table_feats



# In[ ]:









# In[450]:



# Test Lasso Regression Model after RFECV
```

```
lasso_X_train_red, lasso_selected_feats, lasso_X_test_red = backwards_elim(optimised_lasso, X_train, y_train_pc, X_test)

print(lasso_selected_feats)


lasso_fold_results_table_feats, lasso_mean_results_table_feats = int_model_testing(optimised_lasso, lasso_X_train_red, y_train_pc)


print(lasso_fold_results_table_feats)

lasso_mean_results_table_feats



# In[451]:



from sklearn.svm import SVR

# Test SVR Model after RFECV



svr_X_train_red, svr_selected_feats, svr_X_test_red = backwards_elim(optimised_svr, X_train, y_train_pc, X_test)

print(svr_selected_feats)


svr_fold_results_table_feats, svr_mean_results_table_feats = int_model_testing(optimised_svr, svr_X_train_red, y_train_pc)


print(svr_fold_results_table_feats)

svr_mean_results_table_feats



# In[452]:
```

```
# Test Decision Tree Model after RFECV


tree_X_train_red, tree_selected_feats, tree_X_test_red = backwards_elim(optimised_tree, X_train, y_train_pc, X_test)

print(tree_selected_feats)


tree_fold_results_table_feats, tree_mean_results_table_feats = int_model_testing(optimised_tree, tree_X_train_red, y_train_pc)


print(tree_fold_results_table_feats)

tree_mean_results_table_feats



# In[453]:



# Test Random Forest Model after RFECV


rf_X_train_red, rf_selected_feats, rf_X_test_red = backwards_elim(optimised_rf, X_train, y_train_pc, X_test)

print(rf_selected_feats)


forest_fold_results_table, forest_mean_results_table = int_model_testing(optimised_rf, rf_X_train_red, y_train_pc)


print(forest_fold_results_table)

forest_mean_results_table



# In[454]:
```

#Define a final model evaluation function which will test optimised model on the reduced Train and Test model

```
def final_model_eval(model,X_train,y_train, X_test, y_test):

    model.fit(X_train, y_train)
    train_pred = model.predict(X_train)

    final_train_mse = mean_squared_error(y_train, train_pred)
    final_train_rmse = sqrt(mean_squared_error(y_train, train_pred))
    final_train_mae = mean_absolute_error(y_train, train_pred)
    final_train_r2 = r2_score(y_train, train_pred)



    print(f"Model Name = {model}")

    print(f"Training MSE = {final_train_mse}")
    print(f"Training RMSE = {final_train_rmse}")
    print(f"Training MAE = {final_train_mae}")
    print(f"Training R2-Score = {final_train_r2}")



    pred = model.predict(X_test)

    final_mse = mean_squared_error(y_test, pred)
    final_rmse = sqrt(mean_squared_error(y_test, pred))
    final_mae = mean_absolute_error(y_test, pred)
    final_r2 = r2_score(y_test, pred)

    print(f"MSE = {final_mse}")
    print(f"RMSE = {final_rmse}")
```

```
    print(f"MAE = {final_mae}")
    print(f"R2-Score = {final_r2}")

    eval_list = [model, final_rmse, final_rmse, final_mae, final_r2]

    # Plotting residuals
    residuals = y_test - pred
    plt.figure(figsize=(10, 5))
    plt.scatter(y_test, residuals)
    plt.axhline(0, color='red', linestyle='--')
    plt.xlabel('Time Index')
    plt.ylabel('Residuals')
    plt.title('Residual Plot')
    plt.show()

    return eval_list, residuals, pred

# In[455]:

# Linear Regression Final Model Evaluation

final_model_eval(lr, lr_X_train_red, y_train_pc, lr_X_test_red, y_test_pc)

# In[456]:

#Final Model evaluation on optimised ridge
```

```
final_model_eval(optimised_ridge, ridge_X_train_red, y_train_pc, ridge_X_test_red, y_test_pc)
```

```
# In[457]:
```

```
#Final Model Evaluation on Lasso
final_model_eval(optimised_lasso, lasso_X_train_red, y_train_pc, lasso_X_test_red, y_test_pc)
```

```
# In[458]:
```

```
#Final Model Evaluation on SVR
final_model_eval(optimised_svr, svr_X_train_red, y_train_pc, svr_X_test_red, y_test_pc)
```

```
# In[459]:
```

```
#Final Model Evaluation on optimised Tree
final_model_eval(optimised_tree, tree_X_train_red, y_train_pc, tree_X_test_red, y_test_pc)
```

```
# In[461]:
```

```
#Final Model Evaluation on Optimised Random Forest
inal_model_eval(optimised_rf, rf_X_train_red, y_train_pc, rf_X_test_red, y_test_pc)
```

Final Hyperparameter Tune on Random Forest

In[473]:

```
# Define a function that will transform a new dataset then fit Random Forest model to the new datasets
def new_dataset(dataset, model):
    #read dataset
    df_new = pd.read_csv(dataset)
    df_new['Date'] = pd.to_datetime(df_new['Date'])
    df_new.set_index('Date', inplace=True)

    #Set MA20 as target by creating a new column
    df_new["MA20_target"] = df_new["MA20"]

    # Drop the "Close(t+20)" and redundant "MA20" columns
    df_new = df_new.drop(columns=[ "MA20"])
    # Convert to percentage change
    df_new["percentage_change"]= df_new["MA20_target"].pct_change() * 100


    cols_to_drop = [
    "MA20_target",
    'Date_col', #Duplicate of the index
    'Close_forcast', #Predicted forecast alread in dataset - removed so biased is not introduced into the model
    'CCI', # This is constant for all instances therefore will have no impact on model prediction performance
   ]
    df_new = df_new.drop(columns=cols_to_drop)
```

```
#Drop missing values
df_new= df_new.dropna(axis=0)

backwards_elim_feats = rf_selected_feats

df_new = df_new.drop(columns=backwards_elim_feats)

#Split into target and feats

y= df_new["percentage_change"]
X = df_new.drop(columns=["percentage_change"])

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)

#Convert back to DataFrame to return column names back to X_scaled
X_scaled = pd.DataFrame(X_scaled, columns=X.columns)

#Split dataset into training and testing
split_data = int(len(X_scaled) * 0.80)

X_train = X_scaled[:split_data]
X_test = X_scaled[split_data:]



y_train = y[:split_data]
y_test= y[split_data:]

#Fit model on training  and evaluate
model.fit(X_train,y_train)
```

```
train_pred = model.predict(X_train)

final_train_mse = mean_squared_error(y_train, train_pred)
final_train_rmse = sqrt(mean_squared_error(y_train, train_pred))
final_train_mae = mean_absolute_error(y_train, train_pred)
final_train_r2 = r2_score(y_train, train_pred)



print(f"Model Name = {model}")

print(f"Training MSE = {final_train_mse}")
print(f"Training RMSE = {final_train_rmse}")
print(f"Training MAE = {final_train_mae}")
print(f"Training R2-Score = {final_train_r2}")

pred = model.predict(X_test)

final_mse = mean_squared_error(y_test, pred)
final_rmse = sqrt(mean_squared_error(y_test, pred))
final_mae = mean_absolute_error(y_test, pred)
final_r2 = r2_score(y_test, pred)



print(f"Testing MSE = {final_mse}")
print(f"Testing RMSE = {final_rmse}")
print(f"Testing MAE = {final_mae}")
print(f"Testing R2-Score = {final_r2}")
```

```
    # Plotting residuals
    residuals = y_test - pred
    plt.figure(figsize=(10, 5))
    plt.scatter(y_test, residuals)
    plt.axhline(0, color='red', linestyle='--')
    plt.xlabel('Time Index')
    plt.ylabel('Residuals')
    plt.title('Residual Plot')
    plt.show()
```

```
# In[474]:


#Apply model to FB dataset
new_dataset("FB.csv", optimised_rf)


# In[475]:


#Apply Model to MSFT data set
new_dataset("MSFT.csv", optimised_rf)


# In[476]:
```

#Apply model to IBM Dataset

new_dataset("IBM.csv", optimised_rf)

In[477]:

#Apply model to GOOGL

new_dataset("GOOGL.csv", optimised_rf)

Classification

In[ ]:

Recall same y and X for Classification problem

df_amzn_class = df_amzn.copy()

In[ ]:

df_amzn_class.head()

In[ ]:

```
# Set threshold to 1% - if greater than 0.5% Price is "UP", if less than 1% price "DOWN" else its "STABLE "
df_amzn_class['Trend_Direction'] = 'stable'
df_amzn_class.loc[df_amzn_class["percentage_change"] > 0.25, 'Trend_Direction'] = 'up'
df_amzn_class.loc[df_amzn_class["percentage_change"] < -0.25, 'Trend_Direction'] = 'down'
df_amzn_class = df_amzn_class.drop(columns=["percentage_change"])



# In[ ]:



#Set Target variable
y_class = df_amzn_class["Trend_Direction"]
X_class = df_amzn_class.drop(columns=["MA20_target", "Trend_Direction"])



# In[ ]:



df_amzn_class.head()



# In[ ]:



#Check Distribution of target varaible

plt.figure(figsize=(5,5))
sns.countplot(df_amzn_class, x="Trend_Direction")
plt.title("Distribution of Target")
plt.xlabel("Target Class")
```

```
print(df_amzn_class["Trend_Direction"].value_counts())


# In[ ]:


#Normalise the Dataset for model generation
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_class_scaled = scaler.fit_transform(X_class)


# In[ ]:


#Split Data into train and test set
split_data = int(len(X_scaled) * 0.80)

X_train_class = X_class_scaled[:split_data]
X_test_class = X_class_scaled[split_data:]

y_train_class = y_class[:split_data]
y_test_class = y_class[split_data:]


# In[ ]:
```

```
from sklearn.model_selection import TimeSeriesSplit, cross_validate
from sklearn.metrics import balanced_accuracy_score, f1_score, precision_score, recall_score
import matplotlib.pyplot as plt
import pandas as pd
# Define a function that will evaluate a classification model on the trianing dataset
def classifier_model_testing(model, X_train, y_train):

    n_splits = 5
    tscv = TimeSeriesSplit(n_splits=n_splits)

    # Define classification metrics for imbalanced datasets
    cls_metrics = {
        "balanced_accuracy": "balanced_accuracy",
        "f1_score_weighted": "f1_weighted",
        "precision_weighted": "precision_weighted",
        "recall_weighted": "recall_weighted"
    }

    results = cross_validate(model, X_train, y_train, cv=tscv, return_train_score=True,
return_estimator=True, scoring=cls_metrics)

    # Extract scores
    accuracy_train_scores = results["train_balanced_accuracy"]
    f1_train_scores = results["train_f1_score_weighted"]
    precision_train_scores = results["train_precision_weighted"]
    recall_train_scores = results["train_recall_weighted"]

    accuracy_test_scores = results["test_balanced_accuracy"]
    f1_test_scores = results["test_f1_score_weighted"]
    precision_test_scores = results["test_precision_weighted"]
    recall_test_scores = results["test_recall_weighted"]
```

```
train_test_results = {
    "Training Balanced Accuracy": accuracy_train_scores,
    "Training Weighted F1-Score": f1_train_scores,
    "Training Weighted Precision": precision_train_scores,
    "Training Weighted Recall": recall_train_scores,
    "Testing Balanced Accuracy": accuracy_test_scores,
    "Testing Weighted F1-Score": f1_test_scores,
    "Testing Weighted Precision": precision_test_scores,
    "Testing Weighted Recall": recall_test_scores,
}

fold_results_table = pd.DataFrame(train_test_results)

# Plotting the scores
metrics = [
    ("Balanced Accuracy", "Training Balanced Accuracy", "Testing Balanced Accuracy"),
    ("F1-Score (Weighted)", "Training Weighted F1-Score", "Testing Weighted F1-Score"),
    ("Precision (Weighted)", "Training Weighted Precision", "Testing Weighted Precision"),
    ("Recall (Weighted)", "Training Weighted Recall", "Testing Weighted Recall")
]

plt.figure(figsize=(20,5))
plt.suptitle(f'Evaluation of {model}")
for i, (metric_name, train_label, test_label) in enumerate(metrics, 1):
    plt.subplot(1, 4, i)
    plt.plot(fold_results_table[train_label], '-o', label='Train ' + metric_name)
    plt.plot(fold_results_table[test_label], '-o', label='Test ' + metric_name)
    plt.title(f"{metric_name} over {str(n_splits)} folds")
    plt.ylim(0, 1)
    plt.xlabel('Fold')
```

```
        plt.ylabel(metric_name)
        plt.legend()
        plt.grid(True)
        plt.tight_layout()

    plt.show()

    print(f"Model: {model}")
    print(f"Mean Accuracy Train Score: {accuracy_train_scores.mean()}")
    print(f"Mean F1 Train Score: {f1_train_scores.mean()}")
    print(f"Mean Precision Train Score: {precision_train_scores.mean()}")
    print(f"Mean Recall Train Score: {recall_train_scores.mean()}")
    print("================================================")

    print(f"Mean Accuracy Test Score: {accuracy_test_scores.mean()}")
    print(f"Mean F1 Test Score: {f1_test_scores.mean()}")
    print(f"Mean Precision Test Score: {precision_test_scores.mean()}")
    print(f"Mean Recall Test Score: {recall_test_scores.mean()}")



    return fold_results_table


# In[ ]:


from sklearn.linear_model import LogisticRegression

#Evaluate Logistice Regression
log_reg = LogisticRegression(multi_class="multinomial", max_iter=1000)
```

```
tscv = TimeSeriesSplit(n_splits=5)
classifier_model_testing(log_reg, X_train_class, y_train_class)


# In[ ]:


from sklearn.linear_model import RidgeClassifier
#Evaluate Ridge_Classifier
ridge_c = RidgeClassifier()
classifier_model_testing(ridge_c, X_train_class, y_train_class)


# In[ ]:


from sklearn.svm import SVC
#Evaluate SVC
svc_c = SVC(kernel="linear", probability=True)
classifier_model_testing(svc_c, X_train_class, y_train_class)


# In[ ]:


from sklearn.tree import DecisionTreeClassifier
#Evaluate Decsion Tree Classifier
tree_c = DecisionTreeClassifier()
classifier_model_testing(tree_c, X_train_class, y_train_class)
```

In[478]:

```
from sklearn.ensemble import RandomForestClassifier

#Evaluate Random Forest Classifier
rfc = RandomForestClassifier(bootstrap=True, random_state=1)

classifier_model_testing(rfc, X_train_class, y_train_class)
```

Hyperparameter tuning

In[ ]:

```
from sklearn.model_selection import GridSearchCV

#Ridge Classifier - change regularisation strength
param = {
    'alpha': np.logspace(-3,3,100)
    }
tscv = TimeSeriesSplit(n_splits=5)
grid_search = GridSearchCV(ridge_c, param, cv=tscv, scoring='balanced_accuracy',
return_train_score=True, verbose=10)
grid_search.fit(X_train_class, y_train_class)

    # Save the best model
optimised_ridge_c = grid_search.best_estimator_
```

```
    # Store results for plotting
results_ridge = grid_search.cv_results_

print(f"Best hyperparameters for Ridge: {grid_search.best_params_}")



# In[479]:



# Evaluate Tuned Ridge Classifier
classifier_model_testing(optimised_ridge_c, X_train_class, y_train_class)



# In[ ]:



#SVC - change regularisation strength

param = {
    'C': np.logspace(-3,2,15),
}
tscv = TimeSeriesSplit(n_splits=5)
grid_search = GridSearchCV(svc_c, param, cv=tscv, scoring='balanced_accuracy',
return_train_score=True, verbose=10)
grid_search.fit(X_train_class, y_train_class)

    #Save the best model
optimised_svc = grid_search.best_estimator_

    #Store results for plotting
results_svc = grid_search.cv_results_
```

```
print(f"Best hyperparameters for SC: {grid_search.best_params_}")


# In[ ]:


#Test a SVC Model

classifier_model_testing(optimised_svc,  X_train_class, y_train_class)


# In[ ]:


#Decision Tree - change max_depth, min_samples_split, min_samples_leaf
param = {
    'max_depth': [None, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'splitter': ['best', 'random'],

}
tscv = TimeSeriesSplit(n_splits=5)
grid_search = GridSearchCV(tree_c, param, cv=tscv, scoring="balanced_accuracy",
return_train_score=True, verbose=10)
grid_search.fit(X_train_class, y_train_class)

    # Save the best model
optimised_tree_c = grid_search.best_estimator_
```

```
    # Store results for plotting
results_tree = grid_search.cv_results_

print(f"Best hyperparameters for Decision Tree: {grid_search.best_params_}")



# In[480]:



#Test a Deccision Tree Classifier Model



classifier_model_testing(optimised_tree_c,  X_train_class, y_train_class)



# In[356]:



from sklearn.model_selection import RandomizedSearchCV

#Random Forest - change n_estimators, max_depth, min_samples_split, min_samples_leaf

param= {
    'n_estimators': [1,5, 10, 25, 50, 100, 200],
    'max_depth': [1,5, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
}
tscv = TimeSeriesSplit(n_splits=5)
grid_search = RandomizedSearchCV(rfc, param, cv=tscv, scoring='balanced_accuracy',
return_train_score=True,  n_iter=50, verbose=10, random_state=1)
```

```
grid_search.fit(X_train_class, y_train_class)

    # Save the best model
optimised_rfc = grid_search.best_estimator_

    # Store results for plotting
results_rfc = grid_search.cv_results_

print(f"Best hyperparameters for Random Forest: {grid_search.best_params_}")



# In[481]:



#Test a Random Forest Classifier Model

classifier_model_testing(optimised_rfc,  X_train_class, y_train_class)



# Feature Selection

# In[358]:



from sklearn.feature_selection import RFECV
# Define feature elinination function for classification models
def backwards_elim_class(model, X_train, y_train, X_test):
    tscv = TimeSeriesSplit(n_splits=5)
    #Use balanced_accuracy to account for unbalanced dataset
    selector = RFECV(model, step=1, cv=tscv, scoring="balanced_accuracy", verbose=10)
    selector = selector.fit(X_train, y_train)
```

```
    selected_feats = pd.DataFrame(X_train).columns[selector.support_].to_list()


    X_train_class_red = selector.transform(X_train)
    X_train_class_red = pd.DataFrame(X_train_class_red, columns=selected_feats)

    X_test_class_red = selector.transform(X_test)
    X_test_class_red = pd.DataFrame(X_test_class_red, columns=selected_feats)

    return X_train_class_red, selected_feats, X_test_class_red


# In[359]:


log_reg_X_train_class_red, log_reg_selected_feats, log_reg_X_test_class_red =
backwards_elim_class(log_reg, X_train_class, y_train_class, X_test_class)


# In[360]:


#Test an Logistic Regression Model
classifier_model_testing(log_reg,  log_reg_X_train_class_red, y_train_class)


# In[488]:


#Feature Elinination on Tuned Ridge Classifier
ridge_c_X_train_class_red, ridge_selected_feats, ridge_c_X_test_class_red =
backwards_elim_class(optimised_ridge_c, X_train_class, y_train_class, X_test_class)
```

In[489]:

#Test the reduced ridge Model

classifier_model_testing(optimised_ridge_c, ridge_c_X_train_class_red, y_train_class)

In[363]:

from sklearn.preprocessing import LabelEncoder

#SVC doesn't work with non-numeric targets using label encoder to convert

label_encoder = LabelEncoder()

y_train_class_encoded = label_encoder.fit_transform(y_train_class)

svc_X_train_class_red, svc_selected_feats, svc_X_test_class_red = backwards_elim_class(optimised_svc, X_train_class, y_train_class_encoded, X_test_class)

In[482]:

#Evaluate SCV after Feature Elinination

classifier_model_testing(optimised_svc, svc_X_train_class_red, y_train_class)

In[365]:

#Feature Elimination Tree_Classifier

tree_c_X_train_class_red, tree_c_selected_feats, tree_c_X_test_class_red = backwards_elim_class(optimised_tree_c, X_train_class, y_train_class, X_test_class)

In[366]:

#Evaluate an Optimised Tree Classifier

classifier_model_testing(optimised_tree_c, tree_c_X_train_class_red, y_train_class)

In[483]:

rfc_X_train_class_red, rfc_selected_feats, rfc_X_test_class_red = backwards_elim_class(optimised_rfc, X_train_class, y_train_class, X_test_class)

In[484]:

#Evaluate Random Forest after Feature Elimination

classifier_model_testing(optimised_rfc, rfc_X_train_class_red, y_train_class)

Final Model Evaluation

In[485]:

```
#Test Random Forest Classifier Model
classifier_model_testing(optimised_svc, svc_X_train_class_red, y_train_class)


# In[678]:


from sklearn.metrics import balanced_accuracy_score, f1_score, precision_score, recall_score, confusion_matrix, roc_curve, auc
from sklearn.preprocessing import label_binarize
#Define a funtion that will carry out an final evaluation of the classifer model on the test set

def final_classifier_eval(model, X_train, y_train, X_test, y_test):

    #Fit the model and predict
    model.fit(X_train, y_train)
    train_pred = model.predict(X_train)
    pred = model.predict(X_test)

    accuracy_train_scores = balanced_accuracy_score(y_train, train_pred)
    f1_train_scores = f1_score(y_train, train_pred, average='weighted')
    precision_train_scores = precision_score(y_train, train_pred, average='weighted')
    recall_train_scores = recall_score(y_train, train_pred, average='weighted')

    print(f"Model Name = {model}")
    print(f"Training Balanced Accuracy = {accuracy_train_scores}")
    print(f"Training Weighted F1-Score = {f1_train_scores}")
    print(f"Training Weighted Precision = {precision_train_scores}")
    print(f"Training Weighted Recall = {recall_train_scores}")
```

```
#Calculate metrics
accuracy_test_scores = balanced_accuracy_score(y_test, pred)
f1_test_scores = f1_score(y_test, pred, average='weighted')
precision_test_scores = precision_score(y_test, pred, average='weighted')
recall_test_scores = recall_score(y_test, pred, average='weighted')

#Print metrics

print(f"Testing Balanced Accuracy = {accuracy_test_scores}")
print(f"Testing Weighted F1-Score = {f1_test_scores}")
print(f"Testing Weighted Precision = {precision_test_scores}")
print(f"Testing Weighted Recall = {recall_test_scores}")

#Plot confusion matrix
cm = confusion_matrix(y_test, pred)
sns.heatmap(cm, annot=True)

#Plot ROC curve for multi-class
fpr = {}
tpr = {}
roc_auc = {}
classes = 3

#Encode target variable for One vs Rest for ROC curve
y_test_encoded = label_binarize(y_test, classes=["down", "stable", "up"])
y_prob = model.predict_proba(X_test)

for i in range(classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_encoded[:, i], y_prob[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
```

```
    # Plotting the ROC curves for each class
    plt.figure(figsize=(8,8))
    for i in range(classes):
        plt.plot(fpr[i], tpr[i], label=f'ROC curve of class {i} (area = {roc_auc[i]:.2f})')

    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f"Receiver Operating Characteristic - {model}")
    plt.legend(loc="lower right")
    plt.show()

    eval_list = [model, accuracy_test_scores, f1_test_scores, precision_test_scores, recall_test_scores]
    return eval_list


# In[679]:


#Final Evalution of Logistic Regression
final_classifier_eval(log_reg,log_reg_X_train_class_red,y_train_class, log_reg_X_test_class_red, y_test_class)


# In[514]:


#Final Evaluation of Ridge Classifier - ROC Curv won't work because Ridge doesn't output probabilities
final_classifier_eval(optimised_ridge_c, ridge_c_X_train_class_red, y_train_class, ridge_c_X_test_class_red, y_test_class)
```

In[680]:

#Final Evaluation of SVC

final_classifier_eval(optimised_svc, svc_X_train_class_red, y_train_class, svc_X_test_class_red, y_test_class)

In[681]:

#Final Evaluation of Tree Classifier

final_classifier_eval(optimised_tree_c, tree_c_X_train_class_red, y_train_class, tree_c_X_test_class_red, y_test_class)

In[682]:

#Evaluation of Random Forest Classifier

final_classifier_eval(optimised_rfc, rfc_X_train_class_red, y_train_class, rfc_X_test_class_red, y_test_class)

In[505]:

Define function that can take a new dataset, transform it and evaluate the classification model

```
def new_dataset_class(dataset, model):
    #read dataset
    df = pd.read_csv(dataset)
```

```
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)

#Set MA20 as target by creating a new column
df["MA20_target"] = df["MA20"]

# Drop the "Close(t+20)" and redundant "MA20" columns
df = df.drop(columns=[ "MA20"])
# Convert to percentage change
df["percentage_change"]= df["MA20_target"].pct_change() * 100

# Set threshold to 1% - if greater than 0.5% Price is "UP", if less than 1% price "DOWN" else its "STABLE "
df['Trend_Direction'] = 'stable'
df.loc[df["percentage_change"] > 0.25, 'Trend_Direction'] = 'up'
df.loc[df["percentage_change"] < -0.25, 'Trend_Direction'] = 'down'
df = df.drop(columns=["percentage_change"])


cols_to_drop = [
"MA20_target",
'Date_col', #Duplicate of the index
'Close_forcast', #Predicted forecast alread in dataset - removed so biased is not introduced into the model
'CCI', # This is constant for all instances therefore will have no impact on model prediction performance
]
df = df.drop(columns=cols_to_drop)

#Drop missing values
df= df.dropna(axis=0)
```

```
backwards_elim_feats = rf_selected_feats

df = df.drop(columns=backwards_elim_feats)

#Split into target and feats

y= df["Trend_Direction"]
X = df.drop(columns=["Trend_Direction"])

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)

#Convert back to DataFrame to return column names back to X_scaled
X_scaled = pd.DataFrame(X_scaled, columns=X.columns)

#Split dataset into training and testing
split_data = int(len(X_scaled) * 0.80)

X_train = X_scaled[:split_data]
X_test = X_scaled[split_data:]


y_train = y[:split_data]
y_test= y[split_data:]

model.fit(X_train,y_train)
train_pred = model.predict(X_train)
pred = model.predict(X_test)
```

```
accuracy_train_scores = balanced_accuracy_score(y_train, train_pred)
f1_train_scores = f1_score(y_train, train_pred, average='weighted')
precision_train_scores = precision_score(y_train, train_pred, average='weighted')
recall_train_scores = recall_score(y_train, train_pred, average='weighted')

print(f"Training Balanced Accuracy = {accuracy_train_scores}")
print(f"Training Weighted F1-Score = {f1_train_scores}")
print(f"Training Weighted Precision = {precision_train_scores}")
print(f"Training Weighted Recall = {recall_train_scores}")

accuracy_test_scores = balanced_accuracy_score(y_test, pred)
f1_test_scores = f1_score(y_test, pred, average='weighted')
precision_test_scores = precision_score(y_test, pred, average='weighted')
recall_test_scores = recall_score(y_test, pred, average='weighted')

# Print metrics
print(f"Model Name = {model}")
print(f"Testing Balanced Accuracy = {accuracy_test_scores}")
print(f"Testing Weighted F1-Score = {f1_test_scores}")
print(f"Testing Weighted Precision = {precision_test_scores}")
print(f"Testing Weighted Recall = {recall_test_scores}")

# Plot confusion matrix
cm = confusion_matrix(y_test, pred)
sns.heatmap(cm, annot=True)

# Plot ROC curve for multi-class
fpr = {}
tpr = {}
roc_auc = {}
classes = 3
```

```
    y_test_encoded = label_binarize(y_test, classes=["down", "stable", "up"])
    y_prob = model.predict_proba(X_test)

    for i in range(classes):
        fpr[i], tpr[i], _ = roc_curve(y_test_encoded[:, i], y_prob[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # Plotting the ROC curves for each class
    plt.figure(figsize=(8,8))
    for i in range(classes):
        plt.plot(fpr[i], tpr[i], label=f'ROC curve of class {i} (area = {roc_auc[i]:.2f})')

    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic')
    plt.legend(loc="lower right")
    plt.show()

    # Return metrics
    eval_list = [model, accuracy_test_scores, f1_test_scores, precision_test_scores, recall_test_scores]
    return eval_list


# In[507]:


#Evaluate SVC on MSFT dataset
new_dataset_class("MSFT.csv", optimised_svc)
```

In[509]:

#Evaluate SVC on GOOGL dataset

new_dataset_class("GOOGL.csv", optimised_svc)

In[510]:

#Evaluate SVC on FB dataset

new_dataset_class("FB.csv", optimised_svc)

In[511]:

#Evaluate SVC on IBM dataset

new_dataset_class("IBM.csv", optimised_svc)

Clustering

In[647]:

#Load all Datasets

df_amzn_clust = pd.read_csv("AMZN.csv")

df_msft_clust = pd.read_csv("MSFT.csv")

df_fb_clust = pd.read_csv("FB.csv")

```
df_ibm_clust = pd.read_csv("IBM.csv")
df_googl_clust = pd.read_csv("GOOGL.csv")

datasets = [df_amzn_clust, df_msft_clust, df_fb_clust, df_ibm_clust, df_googl_clust]

#Add company name to each Dataset
df_amzn_clust["Name"] = "AMZN"
df_msft_clust["Name"] = "MSFT"
df_fb_clust["Name"] = "FB"
df_ibm_clust["Name"] = "IBM"
df_googl_clust["Name"] = "GOOGL"

#Merge all Datasets

df_clust = pd.concat(datasets, ignore_index=True)



# In[648]:


#Set Date to index which allows for easier visualisation of the data over time
df_clust['Date'] = pd.to_datetime(df_clust['Date'])
df_clust.set_index('Date', inplace=True)

#Set MA20 as target by creating a new column
df_clust["MA20_target"] = df_clust["MA20"]


 # Convert to percentage change
df_clust["percentage_change"]= df_clust["MA20_target"].pct_change() * 100
```

```
# Drop the "Close(t+20)" and redundant "MA20" columns
df_clust = df_clust.drop(columns=[ "MA20", "MA20_target"])



cols_to_drop = [
    'Date_col', #Duplicate of the index
    'Close_forcast', #Predicted forecast alread in dataset - removed so biased is not introduced into the model
    'CCI', # This is constant for all instances therefore will have no impact on model prediction performance
    'Day', 'DayofWeek', 'DayofYear', 'Week',
       'Is_month_end', 'Is_month_start', 'Is_quarter_end', 'Is_quarter_start',
       'Is_year_end', 'Is_year_start', 'Is_leap_year', 'Year', 'Month' # These don't appear to have any relevance to the predicting the stock price


]
df_clust = df_clust.drop(columns=cols_to_drop)

#Drop missing values
df_clust = df_clust.dropna(axis=0)

df_clust.to_csv("Cluster.csv")



# In[649]:



df_clust.describe()
```

In[651]:

#Define Features anf Target

X_clust = df_clust.drop(columns=["percentage_change", "Name"])

y_clust = df_clust["percentage_change"]

In[652]:

#Normalise the Dataset for model generation - not required for linear reagression but for more complex modeeling

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_cluster_scaled = scaler.fit_transform(X_clust)

X_cluster_scaled = pd.DataFrame(X_cluster_scaled, columns=X_clust.columns)

In[653]:

from sklearn.model_selection import train_test_split

#Time Series doesn't matter for clustering so Random train test split should work fine

X_train_clust, X_test_clust, y_train_clust, y_test_clust = train_test_split(X_cluster_scaled, y_clust, test_size=0.2, random_state=1)

In[654]:

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# Assuming X_train is your training data
inertia = []
K = range(1, 21)  # check for up to 10 clusters

#Calculate Interia for all k values
# Inertia: Sum of squared distances of samples to their closest cluster center
for k in K:
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(X_train_clust)
    inertia.append(kmeans.inertia_)

plt.figure(figsize=(10, 6))
plt.plot(K, inertia, 'bx-')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Inertia')
plt.title('Elbow Method For Optimal k')
plt.show()
```

In[655]:

```
#Elbow plot shows k =4 to be optimium
n_clusters = 4
```

```
# Cluster data using Kmeans
kmeans = KMeans(n_clusters=n_clusters, random_state=1, n_init=50 )
train_labels = kmeans.fit_predict(X_train_clust)


# In[656]:


#Reduce Dataset using PCA
from sklearn.decomposition import PCA

#Reduce dataset dimensionality using PCA
pca = PCA()
X_train_pca = pca.fit_transform(X_train_clust)


# In[657]:


#Use cumlative Variance to determine the optimial number of Principal Components required - Threshold = 80% explained variancee
explained_variance = pca.explained_variance_ratio_
cumulative_explained_variance = np.cumsum(explained_variance)

#Plot explained and cumulative on same plot
plt.figure(figsize=(10, 5))
plt.bar(range(1, len(explained_variance) + 1), explained_variance, alpha=0.5, label='Individual explained variance')
plt.plot(range(1, len(explained_variance) + 1), cumulative_explained_variance, label='Cumulative explained variance')
plt.ylabel('Explained variance ratio')
```

```
plt.xlabel('Principal components')
plt.legend(loc='best')
plt.tight_layout()
plt.show()


# In[658]:


# 3 PC's should explain 80% of the variance
pca_n_components = 3
pca = PCA(n_components=pca_n_components)

#Transform X_train and X_test (Don't fit X_test only transform)
X_train_cluster_pca_reduced = pca.fit_transform(X_train_clust)
X_test_cluster_pca_reduced = pca.transform(X_test_clust)


# In[659]:


n_clusters = 4

kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=50 )
train_labels = kmeans.fit_predict(X_train_cluster_pca_reduced)


# In[660]:


#Visualise Clusters
```

```
df = pd.DataFrame(X_train_cluster_pca_reduced, columns=[f"PC{i+1}" for i in
range(pca_n_components)])


#Add the cluster labels as a new column to the dataframe

df['Cluster'] = train_labels


#Create the pair plot colored by the cluster labels

sns.pairplot(df, hue='Cluster', palette='rainbow', diag_kind='kde')


plt.show()



# In[661]:



fig = plt.figure(figsize=(10, 7))



ax = fig.add_subplot(111, projection='3d')


# 3D Scatter plot of Cluster

ax.scatter(X_train_cluster_pca_reduced[:, 0], X_train_cluster_pca_reduced[:, 1],
X_train_cluster_pca_reduced[:, 2], c=train_labels, cmap='rainbow', s=60)

ax.set_xlabel('PC1')

ax.set_ylabel('PC2')

ax.set_zlabel('PC3')

ax.set_title('3D Scatter plot of Clusters')

plt.show()



# In[662]:
```

```
# Add cluster labels back orginal training data by creating a new dataframe
train_cluster = pd.DataFrame({'Cluster': train_labels, 'Percentage_Change_target': y_train_clust})

#Group
mean_pc_per_train_cluster = train_cluster.groupby('Cluster')['Percentage_Change_target'].mean()



# In[663]:



#Display Cluster Means
mean_pc_per_train_cluster



# In[664]:



# Predict Test Labels for X_test
test_labels = kmeans.predict(X_test_cluster_pca_reduced)



# In[665]:



# Predict the mean MA20 for the test set based on the clusters
predicted_pc = mean_pc_per_train_cluster[test_labels].values

predicted_pc
```

In[666]:

#Plot predicted v actial in a table

```
results_cluster = pd.DataFrame({'Actual_Percentage_Change': y_test_clust.values,
'Predicted_Percentage_Change': predicted_pc})
```

In[667]:

```
results_cluster.head()
```

In[668]:

```
# Calculate error metrics

mae = mean_absolute_error(results_cluster['Actual_Percentage_Change'],
results_cluster['Predicted_Percentage_Change'])

mse = mean_squared_error(results_cluster['Actual_Percentage_Change'],
results_cluster['Predicted_Percentage_Change'])

rmse = np.sqrt(mse)

print(f"Mean Absolute Error (MAE): {mae}")

print(f"Mean Squared Error (MSE): {mse}")

print(f"Root Mean Squared Error (RMSE): {rmse}")
```

In[676]:

```
# Test how  different k values affect the metrics
# Create a function that can be looped that will

def k_value_test(X_train_cluster_pca_reduced, y_train_clust, X_test_cluster_pca_reduced,
y_test_clust, k_value):

    kmeans = KMeans(n_clusters=k_value, random_state=42, n_init=50 )
    train_labels = kmeans.fit_predict(X_train_cluster_pca_reduced)

    # Add cluster labels back orginal training data by creating a new dataframe
    train_cluster = pd.DataFrame({'Cluster': train_labels, 'percentage_change': y_train_clust})

    mean_ma20_per_train_cluster = train_cluster.groupby('Cluster')['percentage_change'].mean()

    test_labels = kmeans.fit_predict(X_test_cluster_pca_reduced)

    # Predict the mean MA20 for the test set based on the clusters
    predicted_ma20 = mean_ma20_per_train_cluster[test_labels].values

    results_cluster = pd.DataFrame({'Actual_Percentage_Change': y_test_clust.values,
'Predicted_Percentage_Change': predicted_ma20})

        # Calculate error metrics
    mae = mean_absolute_error(results_cluster['Actual_Percentage_Change'],
results_cluster['Predicted_Percentage_Change'])
    mse = mean_squared_error(results_cluster['Actual_Percentage_Change'],
results_cluster['Predicted_Percentage_Change'])
    rmse = np.sqrt(mse)



    print(f"Mean Absolute Error (MAE): {mae}")
```

```
    print(f"Mean Squared Error (MSE): {mse}")
    print(f"Root Mean Squared Error (RMSE): {rmse}")



    return mae, mse, rmse
```

```
# In[677]:
```

```
for k in range(2, 11):  # Example range from 2 to 9 clusters
    mae_k, mse_k, rmse_k = k_value_test(X_train_cluster_pca_reduced, y_train_clust,
X_test_cluster_pca_reduced, y_test_clust, k)
    print(f"Results for k={k}: MAE = {mae_k}, MSE = {mse_k}, RMSE = {rmse_k}")
```