Machine Learning Methods for Stock Price Prediction in Investment Risk Strategies

INTRODUCTION

OHLC stock data refers to "Open, High, Low, Close" market price points over a trading period (usually one day) [1]. Close price is often considered the most important of these, as it is the final valuation by the market of the stock for that day and is used as a benchmark for the next day's trading [2]. Lots of investment models make use of OHLC data and are commonly used by traders and investment strategists for risk assessment and tolerance evaluation [3]. OHLC data is great for these tasks but not quite enough to obtain truly competitive insights. It is a difficult problem for investors to predict future closing prices based on historical data. There is an active field of research dedicated to this idea, with some approaches using state of the art deep learning methods (such as the following paper by Shen, Shafiq [4]).

This investigation will aim to aid investment strategists and make use of open source historical stock data to create a short-term risk assessment tool (using machine learning to make price related predictions and use them to indicate risk/sell signals). Several datasets were obtained from Kaggle with daily time-series data ranging from 2007 to 2021 [5]. The specific data we will be working with is on stocks from Nasdaq-100 companies with OHLC data, technical indiactors (calculated from OHLC data) and market index data allowing us to extend modelling and analysis to lots of interesting features beyond simple OHLC values [5]. Particularly for a given day (given historical data and some "intra-day" metrics available such as open price) we want to make predctions about the closing price and assess risk of investment accordingly. [6] The intention is for predictive modelling to be used for short-term trading in combination with domain specific market tools/knowledge to implement a risk management strategy for "stop-loss" and "take-profit" [7].

In the next section we give an overview of the data used along with some descriptions and visualisations. We will then perform Exploratory Data Analysis (EDA) and statistical testing related to our stocks to get an initial understanding of the data. This will allow us to make informed decisions about which models we should use as well as obtain some information that can be added to our risk management strategy. We will then take forward the findings into a machine learning experiment section where we discuss approaches and strategies. We will construct 3 inherently different types of model for 3 different types of prediction relating to the close price. Our outputs will be used in a final ensemble model that will serve as a daily risk assessment mechanism to add insight to a user's investment strategy.

Overview of Stock Data

This section aims to give the reader an introduction to stock data within the specific context of the datasets used throughout this investigation. Before further exploration, let us identify "Close(t)" as the close price response variable that we wish to make predictions for. The investigation notebook (provided in the appendix) contains more detailed information about the features in the data. The reader is invited to refer to the Feature Glossary to familiarise themselves with OHLC, volume, main price features, lagged features, technical indicators, and market index data. OHLC data points are commonly used to create "Candlestick" charts, which are a popular tool among traders for analyzing price movements and patterns over time [8]. To aid with our understanding of OHLC data (the primary stock price features), a subsection of time-series Facebook data is charted in FIG 1.

FIG. 1. OHLC Candlestick Chart. Each candlestick represents a sampled monthly trading period. The top and bottom of the candles on each candlestick show the opening and closing price respectively. Where close price is higher than open price, the candlestick is coloured green on the chart, and red for the inverse situation where close price is lower than open price. The thin vertical lines (wicks) extend from the rectangle to the high and low prices (top is high, bottom is low). Volume is displayed as bars on the bottom of the chart. [1]

In the context of our risk assessment strategy, we are most interested in large red candles that indicate strong selling pressure (particularly within the context of the wider market ie. red candles present in a general upward trend are important indicators of risk) [8]. For the rest of the investigation we will focus on using historical data from 4 stocks (Microsoft, IBM, Google, Amazon), and much of our experimentation will focus on predictions for IBM only (to illustrate modelling on one specific stock). However, the notebook source code is set up in such a way that allows us to easily perform investigations on any stock of our choosing (assuming we have the relevant data required).

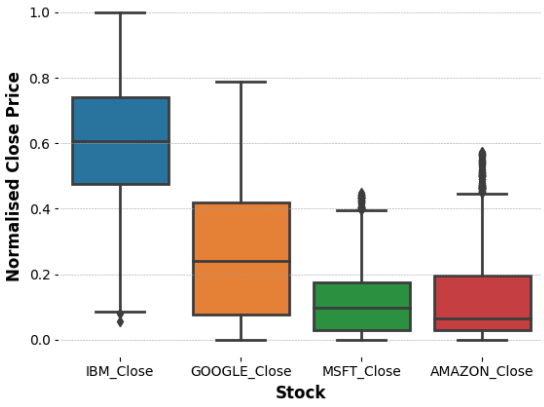


FIG. 3. Box and Whisker plot displaying the distribution of normalised close prices for 4 stocks over a 2000 day period. The IBM data appears significantly larger in magnitude in comparison to the other stocks with a higher box and median. IBM has long whiskers indicating high range with google having the largest IQR and therefore relative variability. Microsoft and Amazon display some outlying data points.

Exploratory Data Analysis (EDA)

FIG 2 and FIG 3 display visual comparisons of close price trends and relative price magnitudes for the stocks under investigation.

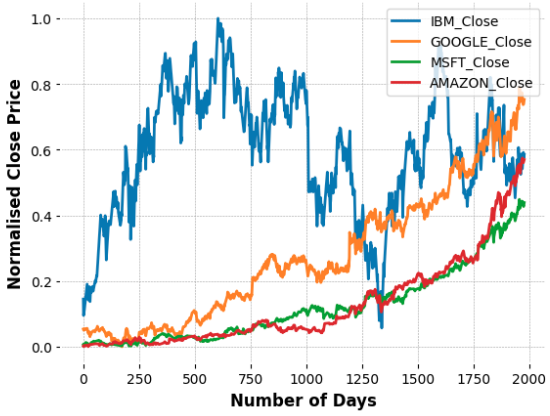


FIG. 2. Line plot of normalised close prices for 4 stocks over a 2000 day period. The IBM data appears significantly varied in comparison to the other price distributions. Amazon and Microsoft data appears to be visually similar and loosely quadratic in trend. Google data shows a slightly different trend but much closer to Amazon/Microsoft then IBM.

IBM is the stock with the highest relative price value, and it displays a large range and varied trend so it makes sense to focus on this one for our investigation. We wish to merge the stock datasets later and a combined market entity could benefit from including a distribution less similar to the others. The notebook provides a more rigorous series of similarity tests using correlation along with other more sophisticated time-series techniques like Dynamic Time Warping (DTW) to quantify relative distances between normalised time-series distributions [9]. IBM was confirmed to be least correlated with the other stocks and market indices. Another section considers the possibility of data leakage/look-ahead bias in the time-series data [10],[11]. Without knowing anything about how the technical indicators have been calculated, we try looking at linear correlation of technical indicators and close price. Extreme linear correlation (suspiciously high) may indicate that the technical indicator was calculated using some future information (we only want historical data up to current day to use in predictions). High correlation can also cause issues in some machine learning algorithms eg. multi-collinearity issues in regression models [12]. We do not want to use these features in predictive modelling. As outlined in the notebook, a portion of indicator features were identified as having over 99% correlation (using Pearson's correlation coefficient) or above and were therefore removed. The notebook contains an outlier analysis section using a Z-score technique and confirms that there are outliers are present in close price data for Amazon and Microsoft which is useful information for the risk plan (for example, highlighting extremes in price transitions). These were chosen not to be removed from the dataset due to the volatile and unpredictable nature of stock movements (we cannot be sure if they are true outliers in context of stocks). Finally, the

notebook also examines lagged close prices and considers linear correlation of each stock with the 3 market indices present in the data. IBM is deemed to be the only one that is not particularly correlated.

Experimental Design

Our EDA section set the foundational framework from which we will conduct further experiments to build machine learning models. We begin our experimental discussion with some notes on pre-processing. The data undergoes some general pre-processing prior to each section before undergoing more model specific preparation. FIG 4 gives details on what our general pre-processing pipeline looks like. We have also mentioned other considerations in the EDA phase around cleaning, and have mentioned that the data already includes extracted features and categorical encoded features. The data is already quite clean (no missing values, some outliers as mentioned but we cannot be sure if these were important due to stocks being volatile).

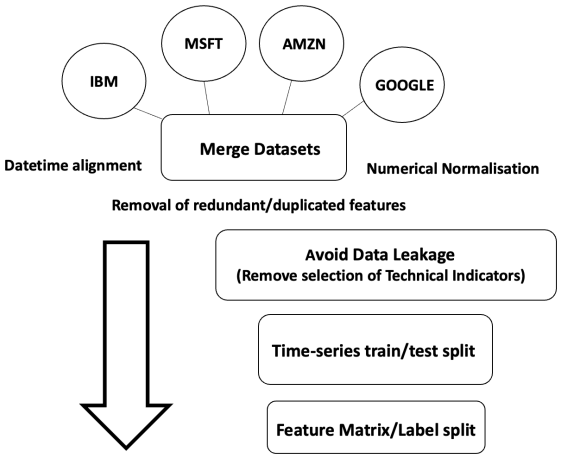
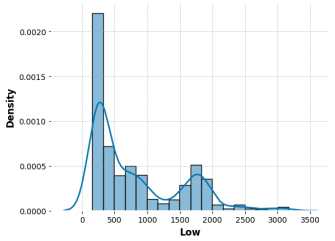
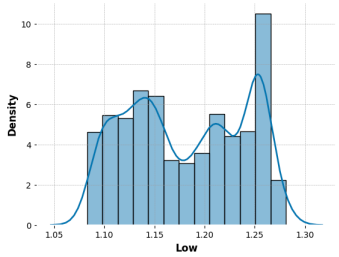


FIG. 4. Outline of a general pre-processing pipeline prior to model consideration. The models we will use later will involve the majority of the steps outlined here. This involves aligning/concatenating standardised stock data, applying initial feature selection, a time-series train/test split and feature/label splitting.

One of the important steps for our data pre-processing is standardisation (or normalisation) - favoured by some of the algorithms/models that we will use. FIG 5 gives a transformation example applied to Amazon Low Price data. Another pre-processing step is feature selection. As the data obtained is extremely complex with lots of features it is hard to know what is relevant without extensive domain knowledge. We have already looked at the potential issues around look ahead bias and data leakage. This allowed us to drop some features immediately.



(a) Amazon Low Price distribution before transformation



(b) Amazon Low Price distribution after transformation

FIG. 5. The density curve for the price distribution in (a) shows 2 bumps with the left one relatively larger. We can see in (b) that the density curve for the data is more balanced (which is expected of the Normal distribution). A Shapiro-Wilk statistical test for normality was conducted in the notebook and confirmed that the data adheres to a normal distribution after transformation.

We will see some more strategies to deal with feature selection later.

Now let us consider the 4 data sets as one stock market entity (after merging) and use collated stock information for predictions about one specific stock at a time (see notebook). It was decided that merged stock data may provide more powerful predictive insights (especially for new stocks that models have not been trained on). We do run the risk however, of creating a lot of extra complexity making models more prone to overfitting. This stock market dataset will be used to train 3 machine learning models of varying type (regression, classification, clustering) in 3 different experiments. For each experiment we will first pre-process the data and employ feature engineering. Then we will choose 2 main types of algorithm to try out. After defining a range of hyperparameters to test and suitable performance metrics, we will implement time-series cross validation to evaluate performance and choose the best model. For optimisation of the chosen model, we will use grid search to search hyperparameter space for the optimal hyperparameters (to improve metrics obtained through cross validation). Finally we test and evaluate the final fine-tuned model on a test dataset. It is to be noted that this is not a strictly sequential process and is very much iterative (things may be discovered and documented along the way). The performances and visualisations of the final models for each type of method

are included later in the report. These will be critically evaluated as well as compared to the model that performed worse. We will now discuss each of the model types individually in detail before the penultimate section of the investigation where we combine the findings into an ensemble method. This will serve as our final risk assessment model for short-term investments.

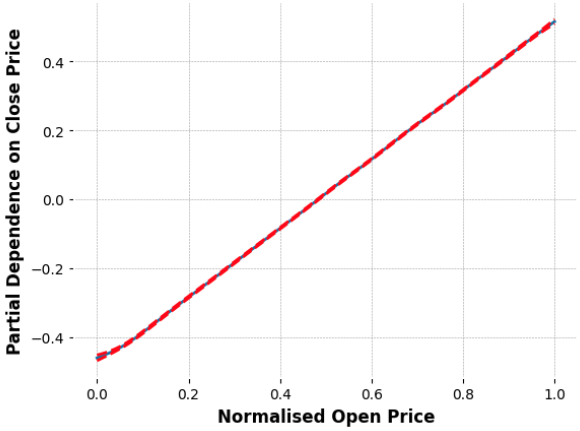


FIG. 6. Partial dependence of normalised Open Price on Close Price for IBM Stock. A very strong linear trend is observed indicating that a linear term may suffice for this feature instead of a spline term.

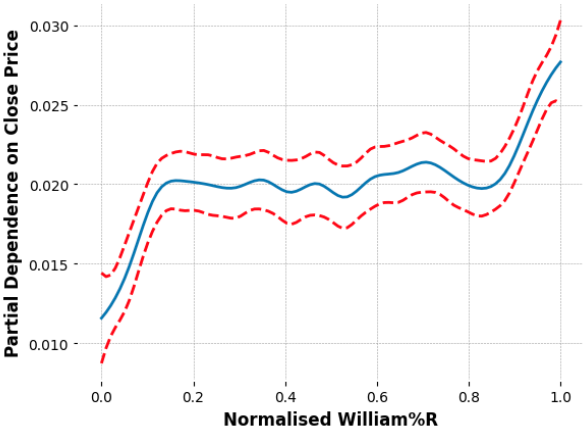


FIG. 7. Partial dependence of normalised William%R indicator on Close Price for IBM Stock. A higher degree polynomial curve is observed with linear trends at the extremes. This suggests that a Natural Cubic spline term may be best to apply to this feature with several knots.

Regression

The data pre-processing for the regression study follows the steps lined out in previous experimental section. We begin this section by assuming that our close price data is linear in nature and form a simple baseline model from which we can obtain a benchmark performance measure to later improve on. Details on this model's implementation and performance can be found in the notebook. This model has high bias and is too simple to capture the complexity of the stock data. The next steps involve extending to handle non-linearity by using a Generalised Additive Model (GAM) which has the flexibility to detect more varied patterns in data [13]. Stock data is volatile, infamously unpredictable and inherently non-linear so a GAM seems a logical next step. At this stage it is noted that the dataset still contains a high number of features and is prone to issues with dimensionality causing unnecessary noise and overfitting training data [14]. To combat this, a feature selection technique called forward selection is used to iteratively select features to add to the model based on statistical significance [15]. In the notebook, an interesting thing to be observed is that Microsoft's 1 day Force Index indicator is selected as an important feature for the prediction of IBM closing price. This could be an example of detection of a complex underlying pattern in the data to be examined further with statistical techniques. A potential issue to be aware of is that of multicollinearity where features are strongly linearly correlated with each other [12]. Correlation analysis can indicate the need for removal (or extraction into combined features). For the GAM implementation, the pyGAM package was used and offers some hyperparameter options to control spline degree and spline number (number of knots in a spline term) [16]. Firstly, partial dependence of features are considered against the target close price variable to help identify sensible options for spline terms. In FIG 6 and FIG 7 we can see the partial dependencies of the first two selected features on the IBM Close Price.

After using the partial dependence plots to set splines, we use grid search to fine-tune the model in combination with a special type of cross validation known as time-series cross-validation to find optimal spline number and regularisation values [17]. In the context of our close price prediction, we wish to capture information about both the magnitude and direction of price transitions. A combination of R^2 score and RMSE are chosen as the performance metrics for evaluating the models to give a balanced view - R^2 gives a measure of explained variance and RMSE gives a measure of absolute error. The final results for the baseline model and the GAM can be found in TABLE I (uncertainty values obtained from 95% confidence intervals of metrics and averaged over

cross validation folds).

TABLE I. Regression Performance Metrics

Model	R^2 Score	RMSE
Linear Regression	-0.749 ±0.470	0.089 ±0.170
Fine-tuned GAM	0.923 ±0.031	0.048 ±0.013

It can be seen from TABLE I that the GAM did indeed perform better in terms of these metrics than the baseline linear model. FIG 8 gives a visual of the predicted data (from the fine-tuned GAM) sequentially extended past the training time-series. Additionally, FIG 9 gives a smaller portion of the predicted data along with the test data for easier visualisation.

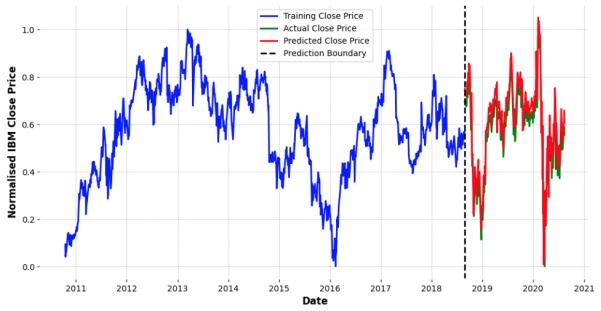


FIG. 8. Line plot of IBM Normalised Close Price data from approximately 2011 until 2021. A prediction boundary is placed at the date chosen to slice the data during the time-series train/test split. Test set data (actual close price over this period) is plot in green and predicted close price in red.

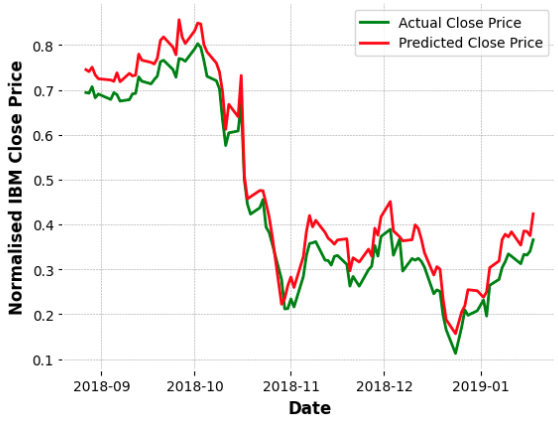


FIG. 9. Line plot of IBM Normalised Close Price data predictions and test data over a 5 month period. The separation in lines makes it clear that our model predictions often overshoot actual price values.

It can be observed in both FIG 8 and FIG 9 that in particular the predicted data does not reach the lower extremes of the test data which is a concern for risk management. In future iterations, we could try oversampling the lower extreme data or add higher weighting to lower price values during model training.

From the analysis, we show that the linear baseline is too simple a model and appears to underfit the data. It is deduced that since the GAM better captures non-linear trends, it is therefore a more powerful and suitable model to use for our business scenario (at the cost of some loss of interpretability due to splining, along with increased computational intensity). With reference to the notebook it is to be noted however, that the GAM performs worse on the test set than the cross validation folds. This indicates overfitting (where the model is too complex and places importance on noise variance present in the training data). Therefore, as an evaluation of the final GAM model, we determine that it is good enough to be used in practice but moving forward we should address the overfitting issue. If more time was available, we could use less splines and more general trends (giving less variability and increasing the bias to allow it to generalise better to unseen data).

Classification

For the Classification study, we begin with some extra data pre-processing (involving the following steps): Define Threshold for Price Stability, Create target labels, Encode target labels (one hot encoding), Drop some redundant date related categories (test set has no leap year), One hot encode remaining categorical features, Drop the Close Price Column (as we are trying to predict classes derived from this), Up-sample under-represented class (to counter class imbalance). Details for this process can be found in the notebook. Feature engineering is used to extract 3 classes from the close price column. We first define a percentage threshold that classifies a price transition form one day to the next as an "Increase", "Decrease" or "Stable" (if deemed not a significantly large transition). The Decrease class is highlighted as the most important for our investment risk model as we want to indicate sell signals from predicted price drops. On examining the class value counts for the new Decrease class, it is observed that there is a large class imbalance with an especially low number of members in the positive class which is particularly inconvenient for our interests. An up-sampling algorithm called SMOTE (Synthetic Minority Over-sampling Technique) based on K-nearest neighbour clustering can be used to generate new representative data points of the minority class to boost the class numbers [18]. After preparing the data for classifier models, we spot check both a simple and complex method. A Support Vector Classifier (SVC) is used under the assumption that we can justify no rigorous dimensionality reduction due to its capabilities in handling data with many features [19]. We also use a simpler extension of regression to the binary

classification problem called Logistic Regression [20]. It is deduced that the SVC is too complex as the logistic regression model achieves much better performance through classification reporting in cross validation folds (see notebook). Precision, recall and F1 score are used as the chosen performance metrics for the classification models with fine-tuning via hyperparameter search across time-series cross validation folds [21]. The results for the final optimised model against the test data can be viewed in TABLE II. This gives us a balanced and comprehensive view of how the model performs against each target class that "accuracy" can't quite cover. Precision gives a measure of reliability of positive predictions whilst Recall measures the ability to detect all instances of the positive class. Although F1 score is used in optimisation, recall is particularly important for a risk management strategy - we want to ensure that we catch all instances of the positive class (it could be costly to miss a downward trend). Further considerations would be to work to improve this metric specifically.

TABLE II. Logistic Regression Performance Metrics

Predicted Decrease	Precision	Recall	F1 Score
No	0.95	0.98	0.97
Yes	0.91	0.83	0.87

A visualisation of the prediction performance via Confusion Matrix can be viewed in FIG 10.

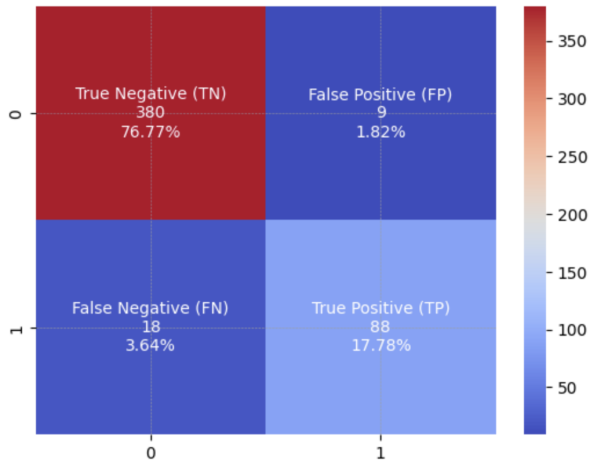


FIG. 10. Confusion Matrix for the Price Decrease class predictions. An ideal model would place all predicted items along the leading diagonal of the matrix. In this case, approximately 5% of the predictions are mis-classified. The density colouring also makes it clear that there are more items in the negative class in the test set displaying class imbalance that is also present in the training set.

The Precision/Recall curve is given in FIG 11.

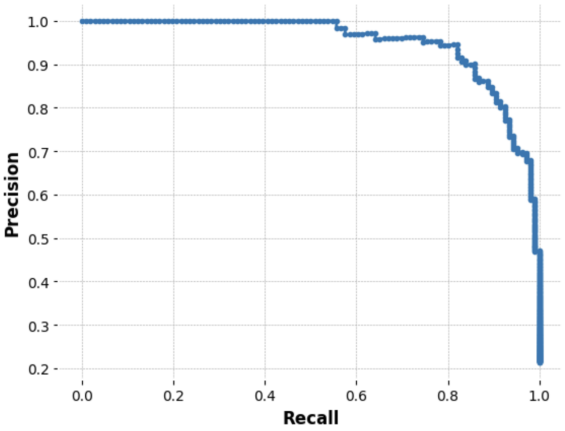


FIG. 11. Precision/Recall Curve for Downward Price Prediction. We can see that precision demonstrates a steeper decrease at a recall of higher than 80%. We would therefore want to aim for a recall of approximately 80% to get the best trade-off.

FIG 12 displays the ROC curve for the data.

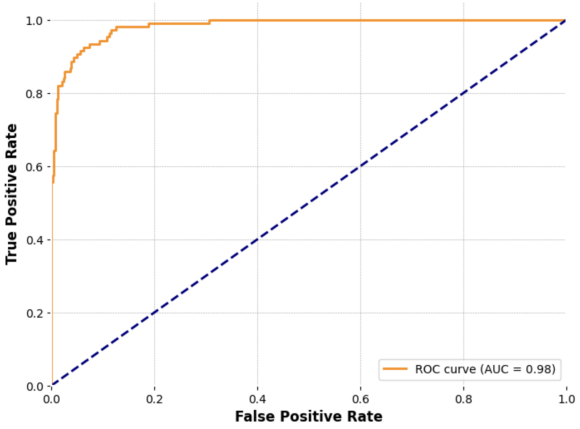


FIG. 12. Receiver Operating Characteristics (ROC) Curve and Area Under Curve (AUC). The dotted line represents a classifier that is truly random. An ROC curve closer to the top left corner with a higher AUC is preferred - this is a plot of the True Positive Rate (Recall) against 1 - True Negative rate (Specificity). In this case, the performance is satisfactory. [22]

The resulting performance is suitable for our strategy but we must be cautious enough to focus on Type II Error Reduction (False Negative) in our risk assessment. This relates to the items in the bottom right cell of the confusion matrix in FIG 10 ie. the price decreases that are missed by the classification model. One thing we could do to improve is simply reduce the price change classification threshold to a smaller percentage difference but in this context the focus should be squeezing the most out of our current threshold. If more time were

available, we could try more rigorous feature selection or use a dimensionality reduction technique/regularisation to improve the model instead.

Clustering

The pre-processing for our Clustering experiment is similar to the regression process but we initially retain more features as we will be performing a dimensionality reduction technique on the data (we have already mentioned why high dimensionality can be an issue) [14]. Following the one-hot encoding of categorical date-related variables, the dataset reaches almost 650 features (see notebook). The main pre-processing we will discuss here is therefore a technique to reduce this high dimensionality. Principle Component Analysis (PCA) is used to identify hyper-planes within feature space that correspond to the directions of highest variance of the data - these are known as Principle Components (PCs) [23]. PCA is highly sensitive to the scale of features and we take steps to ensure that the data is normalised first. After applying PCA, we project the original dataset onto a subset of PCs that explain the majority of the variance in the data. FIG 13 allows us to select an appropriate amount of components to take forward to our clustering mdoel.

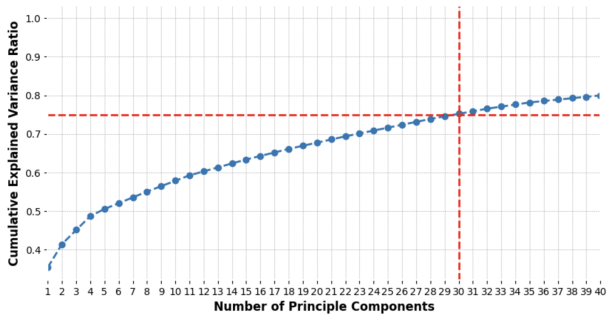


FIG. 13. Cumulative Explained Variance Ratio plot against increasing number of Principle Components. We use this to achieve a balance between dimensionality and level of description of variance in the data. Although 95% is commonly cited in literature, a value of 75% is chosen to strike an appropriate balance in the context of our data [24]. Further experimentation can be done via data cross-validation at a later stage. The axis lines determine that 30 Principle Components are to be used.

In the notebook, the first PC is identified as particularly important (relative to the others) as it contributes around 35% of the explained variance of the full dataset. Using feature loading, it can be determined that technical indicators relating to moving averages are the original features that formed this PC [25]. This is a useful piece of knowledge for our investment strategy. Before feeding the PCA-reduced data to a clustering model, we define the steps taken to achieve a prediction for closing price. First, the clustering model will group similar data points into clusters in an unsupervised manner based on patterns and similarities in features. Next, we will create an extracted feature on the clustered dataset for the next day's close price by introducing a temporal shift to the close price column. We then consider each cluster and average the next day close prices for all data points in the cluster. This value will serve as our current prediction. Each time a new unseen data point is added to a cluster, its next day close price prediction will be predicted by averaging across the other points in the cluster.

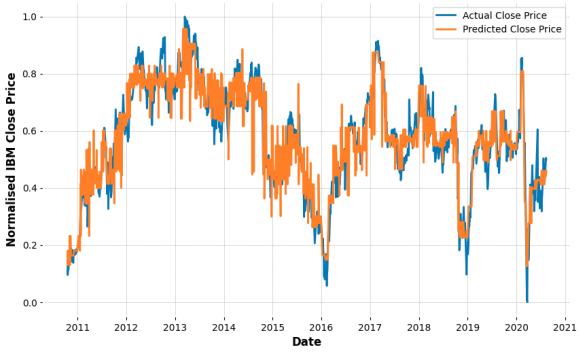
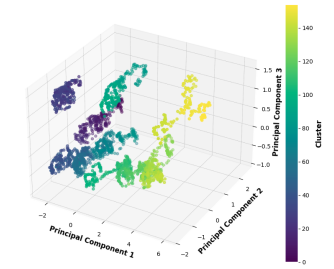


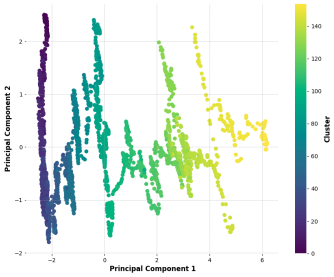
FIG. 14. Line plot of IBM Normalised Close Price prediction data from approximately 2011 until 2021. Includes predictions powered by Affinity Propagation plot with actual stock Close Price data. It can be seen that the predictive model performs worse at the extremes of the data (like the GAM) - we may wish to employ a sampling/weighting technique as mentioned in the Regression section.

First, a KMeans model with varying numbers of predefined clusters is used. This achieves very poor performance and is only able to capture extremely general stepwise trends (refer to notebook). Affinity Propagation is selected as a much better model to use that does not require the number of clusters to be set beforehand [26]. For clustering optimisation and performance metrics, a modified "silhouette score" function is used in combination with a hyperparameter time-series cross validation grid search [27]. Details can be found in the notebook. We can visualise the final predicted Close Price data in FIG 14.

Another process worth doing to gain more insight into clustering is a scatter visualisation in lower dimensions. We can visualise lower dimensional projections of the clustered data onto the first few PCs in FIG 15. Two things can be concluded from examining the clustered data in FIG 15. There are a high number of clusters indicating complex data with lots of distinct groupings, and the second is that the clusters themselves are spread across many dimensions (data is very high dimensional and does not separate entirely well in low dimensions) [28]. Like our regression scenario, as we are predicting a continuous numerical value, the R^2 and RMSE scores are chosen as final prediction performance metrics (along with MSE for complementary insight into outliers) and



(a) 3 Dimensional Projection of Cluster Data using first 3 Principle Components



(b) 2 Dimensional Projection of Cluster Data using first 2 Principle Components

FIG. 15. In (a), we can view some distinct clusters of clusters in 3D. In (b) we can see that there appears to be some separation between data clustered in vertical lines suggesting that the first principle component (along the x axis) separates the data well.

can be viewed in TABLE III.

We can speculate that Affinity Propagation performed better than KMeans at this task due to the nature of the underlying algorithm. We do not have to define cluster number prior to training Affinity Propagation as it compares similarity pairwise between data points [26]. This naturally handles more complex data well. Although Affinity Propagation performed quite well, it is very computationally intensive and can become an issue with increasing dataset size [26]. There is also a lack of interpretability of clusters in high dimensions. Both of these issues could be rectified using further dimensionality reduction and more rigorous feature selection prior to PCA.

TABLE III. Affinity Propagation Performance Metrics

R^2 Score	MSE	RMSE
0.891	0.003	0.061

Ensemble Method & Conclusion

3 model types were used to make predictions about stock close price throughout the investigation. Regression forecasts continuous numerical values which can we quantify price transitions. Clustering is unsupervised so does not use labelled data to group similar data points to find similar price transition patterns. The regression performed better than clustering at the same task (refer to metrics in TABLE I and TABLE III) but clustering could still be useful to support regression predictions or identify hidden groups in high dimensional feature space that otherwise would not be apparent. It could also prove particularly useful for tasks like anomaly detection which would be applicable to stock market data and vital in a risk assessment model [22]. Classification is also deemed suitable for tasks like decision making based on decrease price class. This is also easy for investment strategists to interpret and use. All 3 models are different in nature but useful and there is no reason why they cannot be combined and used together. A final recommendation for tech stock investors is a combined model to assess risk and provide sell signals. The notebook contains information on the implementation of a combined model. Some pseudo-code outlining the process is provided here:

```
drop_threshold = 0.05  # 5% drop
sell_signal =
last_close_price*(1-drop_threshold)

if
    classification_pred == 1
    and regression_pred < sell_signal:
    return 'Strong Sell Signal'

elif
    regression_pred
    < last_close_price
    or clustering_pred < last_close_price:
    return 'Medium Sell Signal'

else:

    return 'Hold: Low Risk of Price Drop'
```

In order to improve on our results in the future we want to regularly update the composite models as new data is collected, adding to the dataset as well as implement monitoring to understand performance on new data [30]. If given more time, an interesting piece of work could extend predictions further in the future without relying on current day price data. We could use a more advanced method known as Walk Forward Optimisation to predict future information some days in advance (which would allow for longer-term investment decisions)[29].

WORD COUNT = 2993

APPENDIX

REFERENCES

[1] AnyChart Documentation, OHLC Chart: Basic Charts, Sound credit risk assessment and valuation for loans,[https://docs.anychart.com/Basic_Charts/OHLC_Chart#:~:text=An%20open%2Dhigh%2Dlow%2D,one%20day%20or%20one%20hour.] accessed 20/11/2023.

[2] Investopedia, What Is Closing Price? Definition, How It's Used, and Example, [https://www.investopedia.com/terms/c/closingprice.asp#:~:text=The%20closing%20price%20is%20considered,market%20sentiment%20toward%20that%20stock.] accessed 20/11/2023.

[3] Nataliia Kuznietsova1, Eduard Bateiko1, Analysis and Development of Mathematical Models for Assessing Investment Risks in Financial Markets, 1 National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", ave. Peremohy 37, Kyiv, 03056, Ukraine.

[4] Shen, J., Shafiq, M.O. Short-term stock market price trend prediction using a comprehensive deep learning system. J Big Data 7, 66 (2020).

[5] Kaggle, US Stock Market Data & Technical Indicators, [https://www.kaggle.com/datasets/nikhilkohli/us-stock-market-data-60-extracted-features/] accessed 20/11/2023.

[6] Investopedia, Intraday: Definition, Intraday Trading, and Intraday Strategies [https://www.investopedia.com/terms/i/intraday.asp] accessed 20/11/2023.

[7] Rundo, F.; Trenta, F.; di Stallo, A.L.; Battiato, S. Grid Trading System Robot (GTSbot): A Novel Mathematical Algorithm for Trading FX Market. Appl. Sci. 2019, 9, 1796.

[8] Candlestick Patterns: How to Read Candlestick Charts — CMC Markets, Candlestick patterns: a guide to candlestick charts, [https://www.cmcmarkets.com/en-gb/trading-guides/candlestick-charts] accessed 20/11/2023.

[9] Gold, Omer; Sharir, Micha (2018). "Dynamic Time Warping and Geometric Edit Distance: Breaking the Quadratic Barrier". ACM Transactions on Algorithms. 14 (4).

[10] Shachar Kaufman; Saharon Rosset; Claudia Perlich (January 2011). "Leakage in data mining: Formulation, detection, and avoidance". Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining. Vol. 6. pp. 556–563.

[11] Investopedia, Look-Ahead Bias: What it Means, How it Works, [https://www.investopedia.com/terms/l/lookaheadbias.asp#:~:text=Look%2Dahead%20bias%20is%20when,not%20show%20an%20accurate%20result.] accessed 20/11/2023.

[12] Chan JY-L, Leow SMH, Bea KT, Cheng WK, Phoong SW, Hong Z-W, Chen Y-L. Mitigating the Multicollinearity Problem and Its Machine Learning Approach: A Review. Mathematics. 2022; 10(8):1283.

[13] Hastie, T. J.; Tibshirani, R. J. (1990). Generalized Additive Models. Chapman & Hall/CRC. ISBN 978-0-412-34390-2.

[14] Bellman, Richard Ernest; Rand Corporation (1957). Dynamic programming. Princeton University Press. p.

[15] Hocking, R. R. (1976) "The Analysis and Selection of Variables in Linear Regression," Biometrics, 32.

[16] A Tour of pyGAM - pyGAM documentation, A Tour of pyGAM, [https://pygam.readthedocs.io/en/latest/notebooks/tour_of_pygam.html] accessed 20/11/2023.

[17] scikit, 4.1. Partial Dependence and Individual Conditional Expectation plots, [https://scikit-learn.org/stable/modules/partial_dependence.html] accessed 20/11/2023.

[18] Chawla, N. V.; Bowyer, K. W.; Hall, L. O.; Kegelmeyer, W. P. (2002-06-01). "SMOTE: Synthetic Minority Oversampling Technique". Journal of Artificial Intelligence Research.

[19] scikit, sklearn.svm.SVC [https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html] accessed 20/11/2023.

[20] Tolles, Juliana; Meurer, William J (2016). "Logistic Regression Relating Patient Characteristics to Outcomes".

[21] scikit, sklearn.model_selection.TimeSeriesSplit [https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html] accessed 20/11/2023.

[22] Geron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems (2nd ed.). O'Reilly.

[23] Jolliffe, Ian T.; Cadima, Jorge (2016-04-13). "Principal component analysis: a review and recent developments". Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences. 374 (2065): 20150202.

[24] Dekking, Frederik Michel; Kraaikamp, Cornelis; Lopuhaä, Hendrik Paul; Meester, Ludolf Erwin (2005). "A Modern Introduction to Probability and Statistics". Springer Texts in Statistics.

[25] Andrecut, M. (2009). "Parallel GPU Implementation of Iterative PCA Algorithms". Journal of Computational Biology. 16 (11): 1593–1599.

[26] scikit, sklearn.cluster.AffinityPropagation [https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AffinityPropagation.html] accessed 20/11/2023.

[27] Peter J. Rousseeuw, Silhouettes: A graphical aid to the interpretation and validation of cluster analysis, Journal of Computational and Applied Mathematics, Volume 20, 1987, Pages 53-65, ISSN 0377-0427.

[28] Galbraith S, Daniel JA, Vissel B. A study of clustered data and approaches to its analysis. J Neurosci. 2010 Aug 11;30(32):10601-8.

[29] Pardo, Robert E. (1992). Design, Testing and Optimiza-

tion of Trading Systems (1st ed.). US: J. Wiley. pp. 108–119. ISBN 0-471-55446-4.

[30] Guangyue Zhang, Hilal Atasoy, Miklos A. Vasarhelyi, Continuous monitoring with machine learning and interactive data visualization: An application to a healthcare payroll process, International Journal of Accounting Information Systems, Volume 46, 2022, 100570, ISSN 1467-0895.

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

# # Instructions before running the Notebook
#
# Please run the imports below and install any missing packages
through pip https://pip.pypa.io/en/stable/cli/pip_install/ (a quick
google of the documentation should point you in the right direction)
#
# Specifically have a look at the following:
#
# * imblearn
#
# * fastdtw
#
# * mplfinance
#
# * mpl_toolkits
#
# Please set the 5 file path variables (in the next cell) to
locations of the 5 dataset csv files on your local machine.
#
# Finally, understand that this Notebook is designed to be a
supplementary appendix to the report (containing source code). Main
discussion and interpretations, reasoning etc. will be found in the
report. Additionally, a lot of extra trial and error, data
exploration etc. has been removed from the notebook in the interest
of keeping a more concise comprehensive narrative of the most
iportant findings.
#

# In[1]:


# Set file paths for the datasets here
fb_path = ''
ibm_path = ''
amzn_path = ''
msft_path = ''
google_path = ''


# In[2]:


# Import all necessary libraries
from sklearn.model_selection import TimeSeriesSplit
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import silhouette_score, make_scorer
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import precision_score, recall_score, f1_score
```

```
from fastdtw import fastdtw
from scipy.spatial.distance import euclidean
from sklearn.metrics import confusion_matrix
from scipy.stats import shapiro
from scipy.stats import entropy
from sklearn.svm import SVC
import seaborn as sns
from scipy.stats import boxcox
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.metrics import roc_curve, auc
import pandas as pd
from sklearn.decomposition import PCA
from pygam import LinearGAM, s, f
import pandas as pd
import mplfinance as mpf
from sklearn.preprocessing import OrdinalEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import TimeSeriesSplit,
RandomizedSearchCV
from sklearn.metrics import precision_recall_curve
from mpl_toolkits.mplot3d import Axes3D
from sklearn.cluster import KMeans
from sklearn.cluster import AffinityPropagation

# Set random seed for randomised operations
import random
random.seed(42)


# # Introduction to Stock data, charting and description of our Data
#
# ## Note: Most of the main discussions, reasoning, decisions behind
methods used and their results/comparisons will be found in the
report (not inlcuded here to allow the notebook to be more concise
and contain only general ideas and code)
#
#
# #### The Data we are working with
#
# Link to source - contains the 5 stocks we have: https://
www.kaggle.com/datasets/nikhilkohli/us-stock-market-data-60-
extracted-features/
#
# The following descriptions are taken from the dataset source:
#
# * Datasets are on stocks from Nasdaq-100 companies
#
# * OHLC data, technical indicators calculated from OHLC data, and
```

external stock market indices
#
* This datasets have around 64 features which include features extracted from OHLC, other index prices such as QQQ(Nasdaq-100 ETF) & S&P 500, (stock market index, is an index that measures the performance of a stock market, or of a subset of a stock market. It helps investors compare current stock price levels with past prices to calculate market performance. - from wikipedia) technical indicators such as Bollinger bands, EMA(Exponential Moving Averages), Stochastic %K oscillator, RSI, etc.
#
* Furthermore, It has lagged features from previous day price data as we know previous day prices affect the future stock price.
#
* Then, the data has date features which specify, if it's a leap year, if its month start or end, Quarter start or end, etc.
#
* All of these features have something to offer for forecasting. Some tell us about the trend, some give us a signal if the stock is overbought or oversold, some portrays the strength of the price trend.
#

Initial Data Exploration and Overview
#
NB: Intially, all 5 datasets were quickly examined but initial exploration of 4 of them have been removed in order to streamline the notebook. Some references to observations made may still be included.
#
Let us consider one dataset first to get an understanding of the data.
#

In[3]:

# Load in some data and view head
import pandas as pd
fb_df = pd.read_csv(fb_path)
fb_df.head()

In[4]:

# Quick info on data attributes (columns)
fb_df.info()

Quick Observations
#
All 5 datasets have the same set of features - some simply have more rows (more time series data collected over wider time frame -

```
but the nature of the attributes etc. is the same), lots of
confusing column names present that we wish to understand.
#
# Let's construct a quick glossary to help us keep track of the
meaninig of the columns...

# # Feature Glossary
#
# * Define feature as: attribute + its values
#
# Following obtained from online searches for terms (and assumptions
where necessary):
#
# #### Main Data of Interest
#
# * Date: Date in YYYY-MM-DD format
#
# * Open,High,Low,Close(t): Price points of stock over one trading
day (t)
#
# * Volume: Total number of shares traded for stock within trading
day
#
# * Lagged Closes: Extracted features consisting of close data from
(t - n) days
#
# * Close Forecast: Extracted feature consisting of close data from
day (t + 1)
#
# #### Technical Indicators
#
# * Upper_Band, Lower_Band: Upper/Lower Bollinger bands
#
# * SD20 - 20 day Standard Deviation
#
# * STD5 - 5 day Standard Deviation
#
# * MA: Moving average (over an n day period where features include
various n values)
#
# * EMA: The exponential moving average (over an n day period where
features include various n values)
#
# * MACD: Moving Average Convergence/Divergence
#
# * ATR: Average True Rate
#
# * ADX: Adams diversifid equity fund (market index)
#
# * CCI: Commodity Channel Index
#
# * ROC: Rate of Change
#
# * RSI: Relative Strength Index
```

```
#
# * William%R: Williams Percent Range
#
# * SO%K: Stochastic Oscillator value
#
# * ForceIndex (calculated over 1 or 20 day period)
#
# #### Market Indices
#
# * QQQ: Invesco QQQ ETF - exchange-traded fund (ETF) tracking the
Nasdaq 100 Index
#
# * DJIA: Dow Jones Industrial Average
#
# * SnP: S&P 500
#

# ### "Technical Indicator" and "Market Index"
#
# Sources for definintons/descriptions:
#
# https://www.investopedia.com/terms/t/
technicalindicator.asp#:~:text=Technical%20indicators%20are%20heuris
tic%20or%20mathematical%20calculations%20based%20on%20the,and%20exit
%20points%20for%20trades.
#
# https://www.investopedia.com/terms/m/marketindex.asp
#
# * Technical Indicators: mathematical calculations based on the
price, volume, or open interest of a security or contract. Used in
technical analysis, these indicators aim to predict future market
trends or to provide insights into the current state of the market.
They are primarily used by traders and analysts to help make
informed decisions about when to buy, sell, or hold a security.
#
# * Market Index: statistical measure that represents the value of a
group of stocks from a particular segment of the stock market. It
acts as a benchmark to track the performance and health of specific
sectors or the entire market. An index is composed of selected
stocks, often grouped based on certain criteria like market
capitalization, industry sector, or company size.

# We will not spend too much time understanding the means by which
the technical indicators were calculated but can proceed under the
assumption that they may prove to be useful to us for predictions -
soon we will explore a few of them in more depth (and visualise
them)

# ### Dates
# Two object columns relating to Date (we know these must be strings
as data came from a csv file)

# In[5]:
```

```
fb_df['Date'].head()


# In[6]:


fb_df['Date_col'].head()


# Date looks to be in String YYYY-MM-DD format - we can easily
convert this to datetime datatype to work with date alignment later
#
# The dates also look to be in order from earliest (lowest dataframe
index) to latest (highest dataframe index) which makes the data a
bit easier to work with
#
# They look to be duplicate columns so we can perhaps get rid of one
later (Date_col = date collected? - this is only other object
attribute we have).
#
# Let's see if the columns are duplicates:

# In[7]:


# Testing for duplicate columns

# Extract dates from the df
dates = fb_df[['Date', 'Date_col']]

# Return True if columns are identical
print(fb_df['Date'].equals(fb_df['Date_col']))


# Yes, they are duplicates so we can safely drop one later...

# ## Integer Data
#
# All of the integer data appears to be encoded categorical data
(apart from volume which follows a discrete numerical distribution)

# In[8]:


# Integer data
non_ints = fb_df.select_dtypes(exclude=['int64']).columns
int_data = fb_df.drop(non_ints, axis=1)
int_data.head()


# ### (True) Numerical Integer

# In[9]:
```

```
# Volume hist/density plot
sns.histplot(fb_df['Volume'], stat='density', kde=True,
kde_kws={"cut": 3})


# ### Categorical Integers
#
# We can identify some encoded categorical data here (not
necessarily numeric in nature)

# The following features are multi-category (multi-label) - eg.
Month ranges from 1 --> 12, Day from 1 --> 31 etc. :
#
#         Day
#         Year
#         Month
#         DayofWeek
#         DayofYear
#         Week
#
# And the remaining categorical features are binary:
#
#         Is_month_end
#         Is_month_start
#         Is_quarter_end
#         Is_quarter_start
#         Is_year_end
#         Is_year_start
#         Is_leap_year
#
# #### NB: Some of the category features may be treated as simply
numeric by us dependeing on the context of our later analysis/
modelling tecniques
#

# ### Float Data
#
# Majority of the data
#
# Looks to be the OHLC + volume data + all other technical
indicators and market indices
#
# This is all continuous numerical data

# ### Overview Summary
#
# Now that we have a feel for the attributes and datatypes, let's
analyse the OHLC (and Volume) data more closely, including some
visualisations, and epxloration of technical indicators + market
indices.
#
```

```
# # Business Context and Setting up Problem & Solution
#
# ### Report includes overview of our aims
#
# Summary of Goal:
#
# Form an investment risk strategy for tech stocks.
#
# We will do this by building 3 models for predicting close price
one day forward:
#
# * Regression
#
# * Classification
#
# * Clustering
#
# -------
#
# * Identify Close(t) price as the main target variable of interest

# # Candlestick Charting - EDA and Understanding Stock Data
#
# NB: Some additional EDA work has been removed from the notebook to
help the reader focus on the most important visualisations and
findings.
#
# Candlestick charting and corresponding interpretations will be a
useful tool to add to the risk management strategy as well as help
us to understand the meaning behind the features we are working with
for prediction models
#

# ### Mplfinance stock analysis

# We will be making use of the following library to aid with
visualisation and understanding of our financial data: https://
github.com/matplotlib/mplfinance/tree/master - "contains a new
matplotlib finance API that makes it easier to create financial
plots. It interfaces nicely with Pandas DataFrames."
#
# Data visualisation library designed for financial data (good for
visualising stock market data).

# In[10]:


# New temporary dataframe corresponding to only the relevant OHLC
data for one stock for now - eg. facebook
data = pd.DataFrame({
    'Open': np.asarray(fb_df['Open']),
    'High': np.asarray(fb_df['High']),
    'Low': np.asarray(fb_df['Low']),
    'Close': np.asarray(fb_df['Close(t)']),
```

```
    'Volume': np.asarray(fb_df['Volume'])
}, index=pd.to_datetime(np.asarray(fb_df['Date'])))

# Re-sample the data to monthly data to give better visualisation on
chart
monthly_data = data.resample('M').agg({
    'Open': 'first',
    'High': 'max',
    'Low': 'min',
    'Close': 'last',
    'Volume': 'sum'
})

# Plot candlestick diagram
mpf.plot(monthly_data, type='candle', style='charles',
         volume=True)  # Include Volume data on the chart


# ### Interpretation
#
# Interpret this plot a little bit - this will aid with our
understanding of OHLC data - we choose monthly trading periods for
beter visulisation - just aiming to get a feel for stock data and
the different features on offer at this stage (inital EDA).
#
# * Each candlestick represents one trading period (we have chosen
to sample monthly but this could be done by day, week etc.)
#
# * The top and bottom of the thick rectangles (candles) on each
candlestick show the opening and closing price respectively
#
# * If close price is higher than open price, the candlestick is
coloured green on the chart, and red for the inverse situation where
close price is lower than open price
#
# * The thin vertical lines (wicks) extend from the rectangle to the
high and low prices (top is high, bottom is low)
#
#     ##### VOLUME is displayed on the bottom
#
# #### Insight
#
# * Large green candles indicate strong buying pressure
#
# * Large red candles indicate strong selling pressure
#
# * Long upper wick - buyers push price up but sellers win out by
reducing by end of trading period
#
# * Long lower wick - sellers push price down but buyers win out by
increasing by end of trading period
#

# ## Explore Lagged Close Data
```

```
# ### Lagged Data (and forecasted) - common and useful for time-
series forecasting
#
# Lagging time series (info source): https://www.kaggle.com/code/
ryanholbrook/time-series-as-features
#
# "Lagging a time series means to shift its values forward one or
more time steps, or equivalently, to shift the times in its index
backward one or more steps. In either case, the effect is that the
observations in the lagged series will appear to have happened later
in time." - common in feature extraction for this kind of data
#
# #### Illustration
#
# The Close(t-1), t-2 etc. are lagged features ie. t - n days
#
# The forecast column looks to simply be data for the close price at
current day and shifted 1 day forward ie. t + 1 day
#
# The following plot illustrate this visually at a given instance in
time
#

# In[11]:


# Close, lagged close features + close forecast feature
fb_lagged_close = fb_df[['Close(t)', 'S_Close(t-1)', 'S_Close(t-2)',
'S_Close(t-3)', 'S_Close(t-5)', 'Close_forcast']]

# Plot the lines for a small snapshot of time-series data to view
lag
plt.figure(figsize=(10, 6))
fb_lagged_close[1200:].plot(kind='line')
plt.show()


# ## Explore Market Indices
#
# Market indices (info source): https://www.investopedia.com/nasdaq-
vs-sp500-vs-dow-5213252
#
# --> The most well-known market indices, such as the S&P 500, the
Dow Jones Industrial Average (DJIA), and the NASDAQ Composite (which
we have in our datasets), are used to gauge the overall market trend
and as a benchmark against which the performance of individual
stocks or investment portfolios can be measured.

# In[12]:


# Close and market index features for Facebook
fb_close_data = fb_df[['Close(t)', 'QQQ_Close', 'SnP_Close',
```

```
'DJIA_Close']]


# In[13]:


# Must normalise the data so we can compare trends at similar scale
scaler = MinMaxScaler()
norm_fb_close_data =
pd.DataFrame(scaler.fit_transform(fb_close_data),
columns=fb_close_data .columns)


# In[14]:


# Lets look at the close features on same plot
plt.figure(figsize=(10, 6))
norm_fb_close_data.plot(kind='line')
plt.show()


# #### Observation
#
# As we can see here, the facebook close price data seems to be
quite highly correlated with the market index data over this time
period. This is not true for all of the stocks we are provided with.
#
# Eg: Compare market index close prices with IBM close price
data ...

# In[15]:


# Same process as above but for IBM data
ibm_df = pd.read_csv(ibm_path)
ibm_close_data = ibm_df[['Close(t)', 'QQQ_Close', 'SnP_Close',
'DJIA_Close']]

scaler = MinMaxScaler()
norm_ibm_close_data =
pd.DataFrame(scaler.fit_transform(ibm_close_data),
columns=ibm_close_data.columns)

# Lets look at the close features
plt.figure(figsize=(10, 6))  # Set the figure size (optional)
norm_ibm_close_data.plot(kind='line')  # Plot the lines
plt.show()


# # Interpretation
#
# * FB has similar trend to the market indices, same dips and rises
#
```

```
# * Can add this information to our risk model
#
# Stock close price trend looks to be correlated with the market
trend - useful info
#
# -------
#
# This section prompts statistical similarity testing to confirm if
this is actually the case

# # Similarity testing
#
# We will consider 2 stocks now (amazon and IBM) then move to
insights and discuss how we can use this information going forward
#
# Form statistical hypotheses:
#
# * Null Hypothesis: The stocks have a statistically similar close
price distribtion
#
# * Alternative Hypothesis: The stock close price distribtions are
statistically different.
#

# #### Tests:
#
#
# * Correlation Analysis: Compute correlation coefficients to look
at linear relationships between stock close prices
#
# * (Time Series Specific) Similarity: Dynamic Time Warping (DTW)
can be used to measure similarity between two temporal sequences,
which may vary in speed [Source: https://
medium.datadriveninvestor.com/dynamic-time-warping-dtw-d51d1a1e4afc]
#
#

# ### Correlation Testing
#
# * Examine Linear correlation between Amazon and all 3 market
indices
#
# * Examine Linear correlation between IBM and all 3 market indices

# In[16]:


# Correlation testing Amazon with the Market Indices
amzn_df = pd.read_csv(amzn_path)
amzn_close_data = amzn_df[['Close(t)', 'QQQ_Close', 'SnP_Close',
'DJIA_Close']]

scaler = MinMaxScaler()
norm_amzn_close_data =
```

```
pd.DataFrame(scaler.fit_transform(amzn_close_data),
columns=amzn_close_data.columns)
corr_matrix = norm_amzn_close_data.corr()

# Heatmap so it is easier to visualise correlation
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', vmin=-1,
vmax=1, square=True, linewidths=0.5)


# In[17]:


# Correlation testing IBM with the Market Indices

ibm_df = pd.read_csv(ibm_path)
ibm_close_data = ibm_df[['Close(t)', 'QQQ_Close', 'SnP_Close',
'DJIA_Close']]

scaler = MinMaxScaler()
norm_ibm_close_data =
pd.DataFrame(scaler.fit_transform(ibm_close_data),
columns=ibm_close_data.columns)
corr_matrix = norm_ibm_close_data.corr()

# Heatmap so it is easier to visualise correlation
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', vmin=-1,
vmax=1, square=True, linewidths=0.5)


# #### Interpretation
#
# We are interested in the first row (or column) of the covariance
matrix
#
# The strength of the linear correlation can be viewed visually as a
deeper red (magnitude closer to 1) meaning stronger correlation.
#
# * Amazon is much more strongly correlated with each of the market
indices than IBM
#
# * IBM still shows some amount of correlation (above 65% for all
market indices)

# ## Time-series similarity
#
# Use Dynamic Time Warping
#
# * Do amazon vs all 3 market indices
#
# * Do IBM vs all 3 market indices

#
# * We need to first normalise the data - bring stocks close prices
to same scale before using DTW
```

```
#
# DTW sources:
#
# [https://kritjunsree.medium.com/stock-prediction-masterclass-
unleashing-the-power-of-dynamic-time-
warping-639bc4100a12#:~:text=The%20Power%20Behind%20DTW,patterns%20o
ccurring%20at%20different%20times.]
#
# [Source: https://medium.datadriveninvestor.com/dynamic-time-
warping-dtw-d51d1a1e4afc]
#
# [https://medium.com/@shachiakyaagba_41915/dynamic-time-warping-
with-time-series-1f5c05fb8950]
#
# #### Compare all ---> lowest distance metric is the most similar
#

# In[18]:


# Check similarity based on close metric

# Compute DTW distance
qqq_dist, path = fastdtw(norm_amzn_close_data['Close(t)'],
norm_amzn_close_data['QQQ_Close'], dist=2)
snp_dist, path = fastdtw(norm_amzn_close_data['Close(t)'],
norm_amzn_close_data['SnP_Close'], dist=2)
djia_dist, path = fastdtw(norm_amzn_close_data['Close(t)'],
norm_amzn_close_data['DJIA_Close'], dist=2)

print("QQQ - DTW distance between stocks:", qqq_dist)
print("SnP - DTW distance between stocks:", snp_dist)
print("DJIA - DTW distance between stocks:", djia_dist)


# In[19]:


# Check similarity based on close metric

# Compute DTW distance
qqq_dist, path = fastdtw(norm_ibm_close_data['Close(t)'],
norm_ibm_close_data['QQQ_Close'], dist=2)
snp_dist, path = fastdtw(norm_ibm_close_data['Close(t)'],
norm_ibm_close_data['SnP_Close'], dist=2)
djia_dist, path = fastdtw(norm_ibm_close_data['Close(t)'],
norm_ibm_close_data['DJIA_Close'], dist=2)

print("QQQ - DTW distance between stocks:", qqq_dist)
print("SnP - DTW distance between stocks:", snp_dist)
print("DJIA - DTW distance between stocks:", djia_dist)


# ### Insights
```

```
#
# * Amazon more strongly similar to the market indices than IBM (as
was the case with linear correlation)

# # Explore Technical Indicators

# #### Technical Indicators
#
# Common ones used (from various online sources) that we have
avilable in our data are:
#
# * ATR
#
# * Several MA (especially - 20,50,200)
#
# * RSI
#
# I will not explain here what each one represents or how it is
calculated - refer to quantitative finance docs online for better
understanding.

# In[20]:


# Using Facebook data again
data = pd.DataFrame({
    'Open': np.asarray(fb_df['Open']),
    'High': np.asarray(fb_df['High']),
    'Low': np.asarray(fb_df['Low']),
    'Close': np.asarray(fb_df['Close(t)']),
    'Volume': np.asarray(fb_df['Volume'])
}, index=pd.to_datetime(np.asarray(fb_df['Date'])))

# Indicator columns
ma_50 = fb_df['MA50']
atr = fb_df['ATR']
rsi = fb_df['RSI']

# Add extra plots for the indicators
apdict = mpf.make_addplot(ma_50[1200:], color='black')
apdict2 = mpf.make_addplot(atr[1200:], color='blue')
apdict3 = mpf.make_addplot(rsi[1200:], panel=1, color='red',
ylabel='RSI')

# Plot the candlestick diagram - a little less than a month of data
for easier visualisation
mpf.plot(data[1200:], type='candle', style='charles',
         volume=True,
         addplot=[apdict, apdict2, apdict3])


# #### Interpretation
#
# * Black line is MA50 - if price is above the line, it’s generally
```

```
considered in an upward trend
#
# * Blue line is ATR - High ATR indicates high volatility and large
movements (larger candles) are more likely
#
# * Red line is RSI -
#
# Value above 70 indicates stock could be overbought = sell signal
# Value below 30 indicates stock could be oversold = buy signal
#
# Both of these situations can be viewed on the chart with uptrend/
downtrend following the 30 and 70 RSI respectively
#
#
# These indicators are useful for risk management

# # Avoiding Potential Data Leakage
#
# * Lets look at correlation between technical indicators and close
price on day t
#
# * Extremely high correlation might indicate that the technical
indicator was calculate using future information (we only want
historical data up to day t)

# In[21]:


ibm_df = pd.read_csv(ibm_path)
amzn_df = pd.read_csv(amzn_path)


# In[22]:


correlation_matrix = ibm_df.corr(numeric_only=True)

corr_threshold = 0.98
close_price_column = 'Close(t)'

for col in correlation_matrix.columns:

    # Avoid duplicate correlation pairs
    if col != close_price_column:

        # Obtain pearson correlation coefficent
        p_correlation = correlation_matrix.at[close_price_column,
col]

        # Check if magnitude of coefficient is above specified
threshold
        if abs(p_correlation) > corr_threshold:

            # Print column pairs and correlation values
```

```
            print(close_price_column, col, p_correlation)


# Potential causes of issues in predictive modelling - we will want
to get rid of these during pre-processing

# # Further EDA
#
# * Extending analysis to comparing data from several stocks
# * Descriptive statistics
# * Outlier Analysis
#

# ## Stock Comparison

# Note the start and end date of all 5 datasets
#
# * Facebook: ( 2015-10-16 ) --> ( 2020-08-13 )
#
# * Amazon: ( 2010-10-18 ) --> ( 2020-08-13 )
#
# * Microsoft: ( 2005-10-17 ) --> ( 2020-08-13 )
#
# * Google: ( 2007-10-17 ) --> ( 2020-08-13 )
#
# * IBM: ( 2000-10-16 ) --> ( 2020-08-13 )
#
# Choose approx 10 years (length of amazon data)
#

# #### Remove facebook
#
# Facebook only has 5 years worth of data... do not use this one,
only consider the other 4 and use 10 years worth of data on them
#
# Also convenient that we have decided to not use the FB dataframe
as we can avoid any biases that may have formed from the initial EDA
we have performed earlier. We still have gained an understanding of
the features and datatypes and even visualised some prices and
indicators

# In[23]:


# Load fresh datasets (exclude facebook)
ibm_df = pd.read_csv(ibm_path)
google_df = pd.read_csv(google_path)
msft_df = pd.read_csv(msft_path)
amzn_df = pd.read_csv(amzn_path)


# #### Align dates, create new combined dataframe and normalise

# In[24]:
```

```
# Convert date col in each to a datetime
ibm_df['Date'] = pd.to_datetime(ibm_df['Date'])
google_df['Date'] = pd.to_datetime(google_df['Date'])
msft_df['Date'] = pd.to_datetime(msft_df['Date'])
amzn_df['Date'] = pd.to_datetime(amzn_df['Date'])

# Drop data on other 3 dataframes - so only has where datetime >=
amzn earliest datetime
cutoff_date = pd.to_datetime('2010-10-18') # earliest entry from the
amazon data

ibm_df = ibm_df[ibm_df['Date'] >= cutoff_date]
msft_df = msft_df[msft_df['Date'] >= cutoff_date]
google_df = google_df[google_df['Date'] >= cutoff_date]

# Make dataframe with Date, 4 types of close
combined_data = pd.DataFrame({
    'IBM_Close': np.asarray(ibm_df['Close(t)']),
    'GOOGLE_Close': np.asarray(google_df['Close(t)']),
    'MSFT_Close': np.asarray(msft_df['Close(t)']),
    'AMAZON_Close': np.asarray(amzn_df['Close(t)'])
}, index=pd.to_datetime(google_df['Date']))


# MinMax scale the close columns for comparison
scaler = MinMaxScaler()
norm_combined_data =
pd.DataFrame(scaler.fit_transform(combined_data),
columns=combined_data.columns)


# ## Sample this data before comparing and further EDA
# Sampling portions of the data before more in-depth EDA will help
us to avoid 'data snooping bias' (reserving a set to remain unlooked
at)
#
# We want to cut off the end of the dataset

# In[25]:


# Cut off the last 20% of rows (this is time-series aligned data)
split_index = int(len(norm_combined_data) * 0.8)

# Split the df at index
split_norm_combined_data = norm_combined_data.iloc[:split_index]


# ### Lets now compare each of the 5 stocks of interest visually
using a line plot
#
```

```
# In[26]:


# One plot for all stock closing prices here
plt.figure(figsize=(10, 6))
split_norm_combined_data.plot(kind='line')
plt.xlabel('Number of Days')
plt.ylabel('Normalised Close Price')
plt.show()


# #### Observations
#
# * IBM has a very different trend to the others
#
# * MSFT and AMAZON look very similar (trend lines look loosely
quadratic, disregarding the variability)
#
# * Google looks a bit more linear but still similar (most similar
to amazon)
#
# We will take a look at some more statistical tests to exmaine
relationships between these stocks soon.
#
# NB: All of them generally increase over time apart from IBM (which
declined significantly at one point). There may have been some
market incident or external factor that caused this.
#

# ## Descriptive Statistics
#
#

# In[27]:


# Quick descriptive statistics on a dataset - Choose Google
google_df = pd.read_csv(google_path)
google_df.describe()


# #### Descriptive stats
#
# Can look at similar descriptions for each of the other
datasets ... (won't include here)
#

# In[28]:


# Reshape data for plotting
melted_df = split_norm_combined_data.melt(value_vars=['IBM_Close',
'GOOGLE_Close', 'MSFT_Close', 'AMAZON_Close'], var_name='Category',
value_name='Value')
```

```
# Create boxplot
sns.boxplot(x='Category', y='Value', data=melted_df)
plt.xlabel("Stock")
plt.ylabel("Normalised Close Price")


# # Outlier analysis
#
# Visual inspection of the box plots lead us to believe that we have
outlying data-points
#
# ### Don't remove the outliers but still useful (can add
information to our risk management model)
#
#
# Lets use a statistical method to properly identify potential
outliers

# In[29]:


# Start with fresh datasets again
# Load fresh datasets (exclude facebook)
ibm_df = pd.read_csv(ibm_path)
google_df = pd.read_csv(google_path)
msft_df = pd.read_csv(msft_path)
amzn_df = pd.read_csv(amzn_path)


# In[30]:


# Outlier code - Z score analysis - look at outliers where datapoint
lies outside 3 std deviations from mean
def find_outliers(dataframe, column):

    # Calculate Z score for all data points in column
    Z_scores = (column - column.mean()) / column.std()

    # Create another dataframe consisting of rows with outliers
    outliers = dataframe[(Z_scores > 3.0) | (Z_scores < -3.0)]

    return outliers


# In[31]:


find_outliers(google_df, google_df['Close(t)'])


# In[32]:
```

```
find_outliers(ibm_df, ibm_df['Close(t)'])


# In[33]:


find_outliers(amzn_df, amzn_df['Close(t)'])


# In[34]:


find_outliers(msft_df, msft_df['Close(t)'])


# ### Results
#
# * Amazon and Microsoft have potential outliers (IBM and Google
don't)
#
# * We can simply keep this information in mind as part of our risk
model - we will not remove these during pre-processing for reasons
discussed above

#
# ## More Similarity  analysis for stocks against each other
#
# DTW testing -  Amzn and msft definitiely correlated, google a
little bit ... can we check these against IBM?
#
# Same as before let us do correlation and DTW
#
# * Do correlation for amazon vs all 3 others
#
# * Do similarity for amazon vs all 3 others
#
# * Do similarity for IBM vs all 3 others
#
# * Insights and how we can use this information going forward (IBM
is good for diversification)
#
#
#

# In[35]:


# Correlation Matrix for Stocks
corr_matrix = split_norm_combined_data.corr()

# Heatmap so it is easier to visualise correlation
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', vmin=-1,
vmax=1, square=True, linewidths=0.5)
```

```
# ### Insights
#
# * Almost zero correlation between IBM and the others
# * Amazon, google, microsoft are strongly correlated with each
other
# * Strongest correlation between microsoft and amazon

# In[36]:


# Amazon Similarities

# Compute DTW distance
ibm_dist, path = fastdtw(split_norm_combined_data['AMAZON_Close'],
split_norm_combined_data['IBM_Close'], dist=2)
msft_dist, path = fastdtw(split_norm_combined_data['AMAZON_Close'],
split_norm_combined_data['MSFT_Close'], dist=2)
google_dist, path =
fastdtw(split_norm_combined_data['AMAZON_Close'],
split_norm_combined_data['GOOGLE_Close'], dist=2)

print("IBM - DTW distance between stocks:", ibm_dist)
print("MSFT - DTW distance between stocks:", msft_dist)
print("GOOGLE - DTW distance between stocks:", google_dist)


# In[37]:


# IBM Similarities

# Compute DTW distance
amzn_dist, path = fastdtw(split_norm_combined_data['IBM_Close'],
split_norm_combined_data['AMAZON_Close'], dist=2)
msft_dist, path = fastdtw(split_norm_combined_data['IBM_Close'],
split_norm_combined_data['MSFT_Close'], dist=2)
google_dist, path = fastdtw(split_norm_combined_data['IBM_Close'],
split_norm_combined_data['GOOGLE_Close'], dist=2)

print("AMZN - DTW distance between the two time series:", ibm_dist)
print("MSFT - DTW distance between the two time series:", msft_dist)
print("GOOGLE - DTW distance between the two time series:",
google_dist)


# ### Results
#
# * Same observations as with the linear correlation

# # DATA PREPARATION OPTIMISATION - General pre-processing ideas
#
# General Pre-prcoessing Notes
```

```
#
# * Choosing to merge data from the different stocks into one
Dataset (alignment and concatenation)
#
# * We want to retain lots of information before considering each
model sperately and beginning proper feature selection etc.
#
# * Later, for each model type we will do more refined data
preprocessing, feature engineering, model selection
#
# For each model we want to ensure the following steps are covered:
#
# * Split
# * Clean
# * Feature Engineer
# * Encode
# * Transform
# * Standardise + Normalise (as scaling)
#
# We will cover the pre-processing individually for each model but
will move into a quick discussion on a few common key points

# ## Resampling - Splitting Data
#
# Do this before any feature enigneering --> don't want to leak
information from the test set into the training set
#
# #### How to split???? Time series data:
#
# * As mentioned, we will use a time-series split to split the data
at a certain specified date index to reserve the latest data for
testing
#
# * Later we will also employ some techniques in combination with
cross validation

# ## Boxcox Normalisation
#
# Common transformation procedure that we can make use of during
pre-processing
#
# Use boxcox method

# In[38]:


ibm_df = pd.read_csv(ibm_path)
google_df = pd.read_csv(google_path)
msft_df = pd.read_csv(msft_path)
amzn_df = pd.read_csv(amzn_path)


# In[39]:
```

```
fig = plt.figure(figsize=(15,15))
ax = fig.gca()

# Examine numerical distributions more closely - eg. for just amazon
now
OHL_amzn_df = amzn_df[['Open','High','Low']]

for row, column in enumerate(OHL_amzn_df.columns):
    plt.figure(row)
    sns.histplot(OHL_amzn_df[column], stat='density', kde=True,
kde_kws={"cut": 3})


# In[40]:


# Box cox implementation to make distrubutions more normal -
# this helps to reduce impact of outliers (eg. which pca can be
sensitive to)
def boxcox_transformation(df, col_name):
    transformed_data, lambda_value = boxcox(df[col_name])
    df[col_name] = transformed_data
    return df


# In[41]:


transformed_amzn_low = boxcox_transformation(amzn_df,'Low')


# In[42]:


sns.histplot(transformed_amzn_low['Low'], stat='density', kde=True,
kde_kws={"cut": 3})


# This looks much more like a normal distribution

# #### Shapiro-Wilk Normality test
#
# * Quick statistical test to prove the data is now normally
distributed
#
# * Set p vlaue to 0.05

# In[43]:


# P value choose 0.05
# SW Statistic closer to 1 = normal data
def shapiro_wilk(df,col_name):
```

```
    return shapiro(df[col_name])


# In[44]:


shapiro_wilk(transformed_amzn_low,'Low')


# Data now has a better normal approximation - statistic and p value
support this

# #### More notes on transformations
#
# * We will also make use of Sklearn MinMax scaling to squeeze
values between 0 and 1 (good for preserving patterns in the closing
prices as relative distances are maintained): https://scikit-
learn.org/stable/modules/generated/
sklearn.preprocessing.MinMaxScaler.html
#

# ## Note: Feature Engineering (Extraction)
#
# The datasets already have a lot of extracted features: Derived
Features (tecnical indicators), lag features and extenral data like
the market index data

# ----------------------
# ----------------------
# ----------------------
# ----------------------

# # Regression
#

# ### Pre-processing Steps
#
# Note that this is an iterative process, extra steps may be added
when the model selection/evaluation stage is reached. Let's start
with a general outline leading up to our first model evaluation:
#
# * Load datasets
#
# * Convert Dates to datetime datatype and align datasets by date
(via cutoff)
#
# * Drop duplicate date columns
#
# * Normalise numerical data (such that features retain
distributions and are squeezed between 0 and 1)
#
# * Drop duplicate columns that are equivalent in all datasets (eg.
market index data)
#
```

```
# * Drop categorical data - these features were deemed to not have
particularly statistically significant or meaningful relationships
for predicting exact continuous numerical close price. Incluing too
many features like this may increase the risk of overfitting by
introducing noise. One hot encoding can also lead to extremely high
dimensional data which is not preferred by a lot of models (both in
terms of computational complexity and performance)
#
# * Drop technical indicators that we identified as potential
sources of data leakage earlier
#
# * Drop Low and High price data - Our approach to the problem aims
to avoid using all price data for a given day. This means that given
only the open price on a given day we can make predictions about the
close price without using any high or low data - creates a more
powerful risk assessment model
#
# * Time-series split the data by cutting off the latest portion of
data to reserve for testing
#
# Data is then ready to be fed to regression models.

# In[45]:


def general_regression_preprocessing():

    # Load in fresh datasets
    ibm_df = pd.read_csv(ibm_path)
    google_df = pd.read_csv(google_path)
    msft_df = pd.read_csv(msft_path)
    amzn_df = pd.read_csv(amzn_path)

    # Convert date col in each to a datetime
    ibm_df['Date'] = pd.to_datetime(ibm_df['Date'])
    google_df['Date'] = pd.to_datetime(google_df['Date'])
    msft_df['Date'] = pd.to_datetime(msft_df['Date'])
    amzn_df['Date'] = pd.to_datetime(amzn_df['Date'])

    # drop data on other 3 so only has where datetime >= amzn
earliest datetime
    cutoff_date = pd.to_datetime('2010-10-18') # earliest entry from
the amazon data
    ibm_df = ibm_df[ibm_df['Date'] >= cutoff_date]
    msft_df = msft_df[msft_df['Date'] >= cutoff_date]
    google_df = google_df[google_df['Date'] >= cutoff_date]

    # Need to reset and align indices of each dataframe now
    ibm_df = ibm_df.reset_index(drop=True)
    msft_df = msft_df.reset_index(drop=True)
    google_df = google_df.reset_index(drop=True)

    # Code for normalising - make 4 normalised datasets with dates
aligned
```

```
    # drop date col
    ibm_df = ibm_df.drop('Date_col',axis=1)
    google_df = google_df.drop('Date_col',axis=1)
    amzn_df = amzn_df.drop('Date_col',axis=1)
    msft_df = msft_df.drop('Date_col',axis=1)

    # convert volume to float64
    ibm_df['Volume'] = ibm_df['Volume'].astype('float64')
    google_df['Volume'] = google_df['Volume'].astype('float64')
    amzn_df['Volume'] = amzn_df['Volume'].astype('float64')
    msft_df['Volume'] = msft_df['Volume'].astype('float64')

    # Code for getting rid of columns that are common across all of
the datasets (we will keep them on IBM)
    # Drop all int64 columns on every dataset apart from one
    int_cols = amzn_df.select_dtypes(include=['int64']).columns

    # Drop int64 columns
    amzn_df = amzn_df.drop(int_cols, axis=1)
    google_df = google_df.drop(int_cols, axis=1)
    msft_df = msft_df.drop(int_cols, axis=1)

    # Also drop date columns from these 3
    amzn_df = amzn_df.drop('Date', axis=1)
    google_df = google_df.drop('Date', axis=1)
    msft_df = msft_df.drop('Date', axis=1)

    # Drop all columns beginning with QQQ, SnP and DJIA from these 3
(as these will also be common across all datasets)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('QQQ')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('SnP')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('DJIA')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in google_df.columns if
col.startswith('QQQ')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('SnP')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('DJIA')]
    google_df = google_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in msft_df.columns if
col.startswith('QQQ')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in msft_df.columns if
```

```
col.startswith('SnP')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('DJIA')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)

    # Drop moving averages and band data as concern around data
leakage
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('MA')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('EMA')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('Upper')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('Lower')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in google_df.columns if
col.startswith('MA')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('EMA')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('Upper')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('Lower')]
    google_df = google_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in msft_df.columns if
col.startswith('MA')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('EMA')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('Upper')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('Lower')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)

    # Also include IBM here
    columns_to_drop = [col for col in ibm_df.columns if
col.startswith('MA')]
    ibm_df = ibm_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in ibm_df.columns if
col.startswith('EMA')]
    ibm_df = ibm_df.drop(columns_to_drop, axis=1)
```

```
    columns_to_drop = [col for col in ibm_df.columns if
col.startswith('Upper')]
    ibm_df = ibm_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in ibm_df.columns if
col.startswith('Lower')]
    ibm_df = ibm_df.drop(columns_to_drop, axis=1)

    # Rename all float64 type columns in each dataset by prefixing
with stock

    amzn_df.rename(columns={col: "amzn_" + col for col in
amzn_df.columns if amzn_df[col].dtype == 'float64'}, inplace=True)
    google_df.rename(columns={col: "google_" + col for col in
google_df.columns if google_df[col].dtype == 'float64'},
inplace=True)
    msft_df.rename(columns={col: "msft_" + col for col in
msft_df.columns if msft_df[col].dtype == 'float64'}, inplace=True)
    ibm_df.rename(columns={col: "ibm_" + col for col in
ibm_df.columns if ibm_df[col].dtype == 'float64'}, inplace=True)

    # Code for concatenating the datasets
    stock_df = pd.concat([ibm_df, amzn_df, msft_df, google_df],
axis=1)

    # Drop all lagged Close features (and forecast) from entire
dataset

    columns_to_drop = [col for col in stock_df.columns if
col.endswith('S_Close(t-1)')]
    stock_df = stock_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in stock_df.columns if
col.endswith('S_Close(t-2)')]
    stock_df = stock_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in stock_df.columns if
col.endswith('S_Close(t-3)')]
    stock_df = stock_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in stock_df.columns if
col.endswith('S_Close(t-5)')]
    stock_df = stock_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in stock_df.columns if
col.endswith('Open(t-1)')]
    stock_df = stock_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in stock_df.columns if
col.endswith('Close_forcast')]
    stock_df = stock_df.drop(columns_to_drop, axis=1)

    # Drop Low and High data as well

    columns_to_drop = [col for col in stock_df.columns if
```

```
col.endswith('Low')]
    stock_df = stock_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in stock_df.columns if
col.endswith('High')]
    stock_df = stock_df.drop(columns_to_drop, axis=1)

    return stock_df


# In[46]:


# Utility functions for splitting data and ensuring sets are
normalised
def split_stock_df(stock_df,percent):

    # Code for splitting the merged dataset into train and test
    # find split point
    split_index = int(len(stock_df) * percent)

    # Split the DataFrame
    train_df = stock_df.iloc[:split_index+1]
    test_df = stock_df.iloc[split_index:]

    # Normalise train_df float columns
    float_cols = train_df.select_dtypes(include=['float64']).columns
    scaler = MinMaxScaler()
    norm_train_df = train_df.copy()
    norm_train_df[float_cols] =
scaler.fit_transform(train_df[float_cols])

    # Normalise test_df float columns
    float_cols = test_df.select_dtypes(include=['float64']).columns
    scaler = MinMaxScaler()
    norm_test_df = test_df.copy()
    norm_test_df[float_cols] =
scaler.fit_transform(test_df[float_cols])

    return norm_train_df, norm_test_df

# Split a train or test df into X and multivariate y
def data_label_split_multi(df):

    # split training df into X and y
    y =
df[['ibm_Close(t)','amzn_Close(t)','google_Close(t)','msft_Close(t)'
]]
    X = df.drop(y,axis=1)

    # Re-index by Date
    X  = X.set_index('Date', drop=True)

    return X,y
```

```
# Split a train or test df into X and univariate y
def data_label_split_single(df,col_name):

    # Split training df into X and y
    y = df[[col_name]]

    # Still do not want to include the other close prices for
prediction
    X =
df.drop(['ibm_Close(t)','amzn_Close(t)','google_Close(t)','msft_Clos
e(t)'],axis=1)

    # Re-index by Date
    X  = X.set_index('Date', drop=True)

    return X,y


# In[47]:


stock_df = general_regression_preprocessing() # perform general pre-
processng

# Split data - for now lets leave most recent 20% for testing - can
return to this later during cross validation
train_df, test_df = split_stock_df(stock_df,0.8)
X,y = data_label_split_multi(train_df)


# ## Cross validation - non-trivial as we are dealing with time-
series data.
#
# Can't use normal cross fold validation - the time-series data is
sequential so we cannot break up the sequence.
#
# Want to use cross validation combined with a time series split
(described by the following article):
# https://medium.com/@Stan_DS/timeseries-split-with-sklearn-
tips-8162c83612b9
#
# SKlearn documentation: https://scikit-learn.org/stable/modules/
generated/sklearn.model_selection.TimeSeriesSplit.html
#
# This method gradually increases the size of the training set over
time, keeping the test set fixed
#
# -----
#
# #### From docs (see above source):
#
# Time Series cross-validator
# Provides train/test indices to split time series data samples that
```

```
are observed at fixed time intervals, in train/test sets. In each
split, test indices must be higher than before, and thus shuffling
in cross validator is inappropriate.
#
# This cross-validation object is a variation of KFold. In the kth
split, it returns first k folds as train set and the (k+1)th fold as
test set.
#
# Note that unlike standard cross-validation methods, successive
training sets are supersets of those that come before them.Time
Series Split maintains the order of data, gradually adding more data
to the training set in each fold.

# In[48]:


# Takes in a regression model instance, X feature matrix and y
target labels
def regression_cross_val(model, X, y):

    tscv = TimeSeriesSplit(n_splits=5) # Splits for validation sets

    # Create arrays to store performance metric scores across sets
    rmse_scores = []
    r2_scores = []

    # Loop through data in each split
    for i, (train_index, test_index) in enumerate(tscv.split(X)):

        # Time-series train/test split
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        # Fit model
        model.fit(X_train, y_train)

        # Predict y labels
        predictions = model.predict(X_test)

        # Evaluate rmse and r2, then append to arrays
        rmse = mean_squared_error(y_test, predictions,
squared=False)
        rmse_scores.append(rmse)
        r2 = r2_score(y_test, predictions)
        r2_scores.append(r2)

    # Calculate average rmse across all of the validation folds
    average_rmse = np.mean(rmse_scores)
    print(f"Average RMSE across all folds: {average_rmse}")

    # Calculate average r squared across all of the validation folds
    average_r2 = np.mean(r2_scores)
    print(f"Average R2 across all folds: {average_r2}")
```

```
    # Prediction Confidence Intervals
    rmse_prediction_intervals = np.percentile(rmse_scores, [2.5,
97.5], axis=0)
    r2_prediction_intervals = np.percentile(r2_scores, [2.5, 97.5],
axis=0)

    # Obtain width of intervals
    rmse_interval_widths = rmse_prediction_intervals[1] -
rmse_prediction_intervals[0]
    r2_interval_widths = r2_prediction_intervals[1] -
r2_prediction_intervals[0]


    # Average out the uncertainty across all predictions
    rmse_mean_uncertainty = np.mean(rmse_interval_widths)
    r2_mean_uncertainty = np.mean(r2_interval_widths)

    print(f"Average RMSE Uncertainty: {rmse_mean_uncertainty}")
    print(f"Average R Squared Uncertainty: {r2_mean_uncertainty}")




# In[49]:


# Multivariate Linear Regression (4 response variables in y)
regression_cross_val(LinearRegression(), X, y)


# This example was a multivariate example, predicting all 4 stock
close prices - the performance metrics gave ok results but do not
mean much here as they are averages across all folds and all 4
stocks.
#
# It would be more useful for us to consider honing in on one stock.
This is what we will do for the majority of the investigation.

# ## Single target predictions

# In[50]:


# Split data into fresh train and test sets again
train_df, test_df = split_stock_df(stock_df,0.8)

# Create X and y for each stock individually
X_ibm,y_ibm = data_label_split_single(train_df,'ibm_Close(t)')
X_amzn,y_amzn = data_label_split_single(train_df,'amzn_Close(t)')
X_google,y_google =
data_label_split_single(train_df,'google_Close(t)')
X_msft,y_msft = data_label_split_single(train_df,'msft_Close(t)')

# Single variable Linear Regression (1 response variable in y)
```

```
print("IBM: \n")
regression_cross_val(LinearRegression(),X_ibm,y_ibm)
print("\n")
print("AMZN: \n")
regression_cross_val(LinearRegression(),X_amzn,y_amzn)
print("\n")
print("GOOGLE: \n")
regression_cross_val(LinearRegression(),X_google,y_google)
print("\n")
print("MSFT: \n")
regression_cross_val(LinearRegression(),X_msft,y_msft)


# Model performance isn't quite as good for IBM ... This makes sense
referring back to the time-series visualisation. IBM was quite
varied.Tthe linear Model may be too simple to capture the non-
linearity - we need to increase variance - make the model more
compex! However, this gives us a baseline model - This gives us some
benchmark performance metrics to improve on
#

# ### Feature Importance ---> Feature Selection
# Feature selection through forward selection can help you identify
the most informative features, reducing model complexity and
potential overfitting.

# ### Hone in on IBM - Look at predicting only  one stock
#
# * Let's take a quick look at further feature selection to help us
obtain some important features that we can use to take forward for
future modelling

# In[51]:


train_df, test_df = split_stock_df(stock_df,0.8)
X,y = data_label_split_single(train_df,'ibm_Close(t)')


# In[52]:


# Forward selection (Feature Selection)
def forward_selection(model,X,y):

    # Instantiate the tscv object
    tscv = TimeSeriesSplit(n_splits=5)

    # Create list of all features in X
    full_feats = X.columns.tolist()

    # Create mutable feature list (we will be removing features from
this)
    remaining_feats = full_feats.copy()
```

```
    # Array to store chosen features via forward selection
    selected_feats = []

    # Set initial best rmse (approx. - obtained from previous run of
linreg model)
    best_rmse = 0.05

    # Enter Loop
    while remaining_feats:

        # Array for scoring average rmse for feature combination
        rmse_scores_per_feat = []

        # Iterate through remaining features
        for feat in remaining_feats:

            # Current feature list
            current_feats = selected_feats + [feat]

            # Array for scoring rmse
            rmse_scores = []

            # Timeseries cross validation implementation that we
have seen before
            for i, (train_index, test_index) in
enumerate(tscv.split(X)):
                X_train, X_test =
X[current_feats].iloc[train_index],
X[current_feats].iloc[test_index]
                y_train, y_test = y.iloc[train_index],
y.iloc[test_index]

                model.fit(X_train, y_train)
                predictions = model.predict(X_test)
                rmse = mean_squared_error(y_test, predictions,
squared=False)
                rmse_scores.append(rmse)

            average_rmse = np.mean(rmse_scores)
            rmse_scores_per_feat.append(average_rmse)

        # Look for feature that had highest imporvement in
performance
        # Find best - which minimised the rmse the most
        best_feature_rmse = min(rmse_scores_per_feat)
        best_feature =
remaining_feats[rmse_scores_per_feat.index(best_feature_rmse)]

        # Check condition if our best feature rmse is below
specified threshold
        if best_feature_rmse < best_rmse:

            best_rmse = best_feature_rmse
```

```
            # Add feature to selected list and remove from remaining
list
            selected_feats.append(best_feature)
            remaining_feats.remove(best_feature)

            # Print out information on feature and corresponding
rmse
            print(f"Chose {best_feature} with corresponding RMSE
score of: {best_rmse}")

            # Threshold rmse that we want to get a score below
            if best_feature_rmse < 0.015:
                break
        else:
            break

    # Print and return final selected feature list
    print("Final chosen features:", selected_feats)

    return selected_feats


# In[53]:


selected_features = forward_selection(LinearRegression(),X,y)


# #### Interesting that a microsoft feature was chosen !

# # Build GAM Regression Model - preprocess, feature engineer, fine-
tune/cross-val, final performance metrics

# #### More advanced Regression (GAM)
#
# More advanced parametric model
#
# The actual stock data is not really linear - so how can we
improve?
#
# * Try generalized additive model to offer flexibility, without
losing the ease of interpretability of linear models
#
# For more information: https://en.wikipedia.org/wiki/
Generalized_additive_model
#

# ## GAM
#
# We will make heavy use of the pyGAM library in this section.
#
# Please refer to documentation: https://pygam.readthedocs.io/en/
latest/notebooks/tour_of_pygam.html
```

```
# In[54]:


train_df, test_df = split_stock_df(stock_df,0.70)
X,y = data_label_split_single(train_df,'ibm_Close(t)')


# In[55]:


# Reduce X dataset to only the features with highest importance we
found using forward selection
X = X[selected_features]

# Regression using basic Linear regrssion vs. Linear GAM
print("Linear Regression: \n")
regression_cross_val(LinearRegression(),X,y)
print("\n")
print("Linear GAM: \n")
regression_cross_val(LinearGAM(), X, y)


# ### Use GAM

#
# GAM is slightly better than normal Linear Regression
#
# Using a GAM allows you to capture non-linear relationships in the
data, which is beneficial for our complex stock price prediction.

# In[56]:


# Extending GAM to use splines
gam = LinearGAM(s(0) + s(1) + s(2) + s(3) + s(4))  # 's' refers to
spline term (one for each feature)
gam.fit(X,y)


# ## Look at feature dependence and examine trends
#
# * We don't know what spline terms to use so this will help
#
# The documentation epxlains how to do this. The following
implementation is taken largely form documentation - refer to this
above for further explanation

# In[57]:


# Loop through terms in fitted GAM model
for i, term in enumerate(gam.terms):
```

```
    if term.isintercept:
        continue

    # Obtain partial dependencies
    XX = gam.generate_X_grid(term=i)
    pdep, confi = gam.partial_dependence(term=i, X=XX, width=0.95)

    plt.figure()
    plt.plot(XX[:, i], pdep)
    plt.plot(XX[:, i], confi, c='r', ls='--')

    plt.title(f'Partial dependence for feature {i}')


# ## Findings
#
# * Choose Linear term for first feature
#
# * Choose Natural cubic spline for second (linear at the extremes)
- Using pygam, the s() default is natural cubic
#
# * Not sure on number of splines hyperparam for features - can try
a grid search to find this
#

# ## Hyperparam and fine-tuning

#
# * Manual grid search
#
# * This implementation uses a further modified version of the tscv
method we have been using thus far
#
# * As before, see docs for more information on gam and hyperparams
eg. lam is regularisation term

# In[58]:


# Takes in a range of search values for n_splines and lam
hyperparameters - this is effectively our search grid
def gam_cross_val_grid_search(X,y,n_splines_range,lambda_range):

    tscv = TimeSeriesSplit(n_splits=3)

    # Initialise the "current best" rmse and r2
    best_rmse = 0.1
    best_r2 = 0.85

    # Set up empty dictionary to store optimal hyperparams
    optimal_params = {}

    for n_splines in n_splines_range:
        for lam in lambda_range:
```

```
            rmse_scores = []
            r2_scores = []

            # tscv method from before
            for train_index, test_index in tscv.split(X):
                X_train, X_test = X.iloc[train_index],
X.iloc[test_index]
                y_train, y_test = y.iloc[train_index],
y.iloc[test_index]

                # Define the model with hyperparameters
                # lam is a regularisation term - see docs
                model = LinearGAM(s(0) + s(1, n_splines=n_splines) +
s(2, n_splines=n_splines), lam=lam)

                # Fit and evaluate the model as before
                model.fit(X_train, y_train)
                predictions = model.predict(X_test)
                rmse_scores.append(mean_squared_error(y_test,
predictions, squared=False))
                r2_scores.append(r2_score(y_test, predictions))

            # Calculate average RMSE and R2 for the current
hyperparameters
            average_rmse = np.mean(rmse_scores)
            average_r2 = np.mean(r2_scores)

            # Update the best hyperparameters based on RMSE
            if average_rmse < best_rmse:

                best_rmse = average_rmse
                best_r2 = average_r2

                # Set current best hyperparameters
                optimal_params = {'n_splines': n_splines, 'lambda':
lam}

    # Print the best hyperparameters
    print("Optimised Hyperparameters...\n")
    print("Optimised number of splines:",
optimal_params['n_splines'])
    print("Optimised regulrisation term:", optimal_params['lambda'])
    print("Optimised RMSE score:", best_rmse)
    print("Optimised R2 score:", best_r2)


# In[59]:


# Define the hyperparameter search grid - some trial and error here
n_splines_range = [5, 8, 12, 15, 20]
lambda_range = np.logspace(-5, 3, 11)
```

```
# Run the search using our function
gam_cross_val_grid_search(X,y,n_splines_range,lambda_range)


# ## Final Model

# In[60]:


# Final Model Evaluation against test data
train_df, test_df = split_stock_df(stock_df,0.8)

# Label splits and reduce the X sets via feature selection
X_train, y_train = data_label_split_single(train_df,'ibm_Close(t)')
X_test, y_test = data_label_split_single(test_df,'ibm_Close(t)')
X_train = X_train[selected_features]
X_test = X_test[selected_features]

n_splines=8 # Optimum splines

# Chosen fine-tuned model: instantiate, fit and predict
model = LinearGAM(s(0) + s(1, n_splines=n_splines) + s(2,
n_splines=n_splines), lam=0.1)
model.fit(X_train,y_train)
predictions = model.predict(X_test)

# Evaluate performance
rmse = mean_squared_error(y_test, predictions, squared=False)
r2 = r2_score(y_test, predictions)

print(f"Final RMSE: {rmse}")
print(f"Final R2 score: {r2}")


# In[61]:


# Visualise the predicted data
predictions = pd.DataFrame(predictions)

# Obtain train/test dates from original stock dataframe
train = stock_df.iloc[:split_index+1]
test = stock_df.iloc[split_index:]
train_dates = train['Date']
test_dates = test['Date']
train_dates = pd.to_datetime(train_dates)
test_dates = pd.to_datetime(test_dates)

# Plot train data against train dates
plt.figure(figsize=(12, 6))
plt.plot(train_dates, y_train, label='Training Close Price',
color='blue')

# Plot actual test set data against test dates
```

```
plt.plot(test_dates, y_test, label='Actual Close Price',
color='green')

# Plot predicted data
plt.plot(test_dates, predictions, label='Predicted Close Price',
color='red')

# Prediction boundary
plt.axvline(x=train_dates[-1:], color='black', linestyle='--',
label='Prediction Boundary')

# Setting plot labels, titles and legend
plt.xlabel('Date')
plt.ylabel('Normalised IBM Close Price')
plt.legend()


# In[62]:


# Comparing Predicted vs. Actual (Snapshot of approx. 5 months)

# Plot testing data
plt.plot(test_dates[:100], y_test[:100], label='Actual Close Price',
color='green')

# Plot predicted data
plt.plot(test_dates[:100], predictions[:100], label='Predicted Close
Price', color='red')

plt.xlabel('Date')
plt.ylabel('Normalised IBM Close Price')
plt.legend()


# Performance metrics are worse on test set
#
# Looks like the model is overfitting the training set a bit - if we
had more time then would go back and reduce the complexity of the
model

# In[63]:


# We will use this later in ensemble decision method
regression_predictions = predictions


# ----------
# ----------
# ----------
# ----------

# # Classification
```

```
# ## Preprocessing Steps
#
# A lot of the steps are similar to the regression example so will
not go into too much detail
#
# * Load in datasets
# * Align datasets via date
# * Normalise Data
# * Initial feature selection
# * Time series split data
# * Define Threshold for Price Stability
# * Create target labels
# * Encode target labels (one hot encoding)
# * Drop some redundant date related categories (test set has no
leap year)
# * One hot encode remaining categorical features
# * Drop the Close Price Column (as we are trying to predict classes
derived from this)
# * Upsample under-represented class (to counter class imbalance)
# * X,y data split
#
# Data is now ready to be fed to classifiers.

# In[64]:


# Classification general preprocessor
def general_classification_preprocessing():
    # Load in fresh datasets
    ibm_df = pd.read_csv(ibm_path)
    google_df = pd.read_csv(google_path)
    msft_df = pd.read_csv(msft_path)
    amzn_df = pd.read_csv(amzn_path)

    # Convert date col in each to a datetime
    ibm_df['Date'] = pd.to_datetime(ibm_df['Date'])
    google_df['Date'] = pd.to_datetime(google_df['Date'])
    msft_df['Date'] = pd.to_datetime(msft_df['Date'])
    amzn_df['Date'] = pd.to_datetime(amzn_df['Date'])

    # drop data on other 3 so only has where datetime >= amzn
earliest datetime
    cutoff_date = pd.to_datetime('2010-10-18') # earliest entry from
the amazon data
    ibm_df = ibm_df[ibm_df['Date'] >= cutoff_date]
    msft_df = msft_df[msft_df['Date'] >= cutoff_date]
    google_df = google_df[google_df['Date'] >= cutoff_date]

    # Need to reset and align indices of each dataframe now
    ibm_df = ibm_df.reset_index(drop=True)
    msft_df = msft_df.reset_index(drop=True)
    google_df = google_df.reset_index(drop=True)
```

```
    # Code for normalising - make 4 normalised datasets with dates
aligned

    # drop date col
    ibm_df = ibm_df.drop('Date_col',axis=1)
    google_df = google_df.drop('Date_col',axis=1)
    amzn_df = amzn_df.drop('Date_col',axis=1)
    msft_df = msft_df.drop('Date_col',axis=1)

    # convert volume to float64
    ibm_df['Volume'] = ibm_df['Volume'].astype('float64')
    google_df['Volume'] = google_df['Volume'].astype('float64')
    amzn_df['Volume'] = amzn_df['Volume'].astype('float64')
    msft_df['Volume'] = msft_df['Volume'].astype('float64')

    # Code for getting rid of columns that are common across all of
the datasets (we will keep them on IBM)
    # Drop all int64 columns on every dataset apart from one
    int_cols = amzn_df.select_dtypes(include=['int64']).columns

    # Drop int64 columns
    amzn_df = amzn_df.drop(int_cols, axis=1)
    google_df = google_df.drop(int_cols, axis=1)
    msft_df = msft_df.drop(int_cols, axis=1)

    # Also drop date columns from these 3
    amzn_df = amzn_df.drop('Date', axis=1)
    google_df = google_df.drop('Date', axis=1)
    msft_df = msft_df.drop('Date', axis=1)

     # Drop all columns beginning with QQQ, SnP and DJIA from these
3 (as these will also be common across all datasets)

    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('QQQ')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('SnP')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('DJIA')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in google_df.columns if
col.startswith('QQQ')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('SnP')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('DJIA')]
    google_df = google_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in msft_df.columns if
```

```
col.startswith('QQQ')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('SnP')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('DJIA')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)


    # Drop moving averages and band data as concern around data
leakage

    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('MA')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('EMA')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('Upper')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('Lower')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('Volume')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in google_df.columns if
col.startswith('MA')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('EMA')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('Upper')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('Lower')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('Volume')]
    google_df = google_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in msft_df.columns if
col.startswith('MA')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('EMA')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('Upper')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
```

```
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('Lower')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('Volume')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)

    # Also include IBM here
    columns_to_drop = [col for col in ibm_df.columns if
col.startswith('MA')]
    ibm_df = ibm_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in ibm_df.columns if
col.startswith('EMA')]
    ibm_df = ibm_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in ibm_df.columns if
col.startswith('Upper')]
    ibm_df = ibm_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in ibm_df.columns if
col.startswith('Lower')]
    ibm_df = ibm_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in ibm_df.columns if
col.startswith('Volume')]
    ibm_df = ibm_df.drop(columns_to_drop, axis=1)

    # Rename all float64 type columns in each dataset by prefixing
with stock

    amzn_df.rename(columns={col: "amzn_" + col for col in
amzn_df.columns if amzn_df[col].dtype == 'float64'}, inplace=True)
    google_df.rename(columns={col: "google_" + col for col in
google_df.columns if google_df[col].dtype == 'float64'},
inplace=True)
    msft_df.rename(columns={col: "msft_" + col for col in
msft_df.columns if msft_df[col].dtype == 'float64'}, inplace=True)
    ibm_df.rename(columns={col: "ibm_" + col for col in
ibm_df.columns if ibm_df[col].dtype == 'float64'}, inplace=True)

    # Code for concatenating the datasets
    stock_df = pd.concat([ibm_df, amzn_df, msft_df, google_df],
axis=1)

    # Drop Low and High data as well

    columns_to_drop = [col for col in stock_df.columns if
col.endswith('Low')]
    stock_df = stock_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in stock_df.columns if
col.endswith('High')]
    stock_df = stock_df.drop(columns_to_drop, axis=1)

    return stock_df
```

```
# In[65]:


def split_stock_df(stock_df,percent):

    # Code for splitting the merged dataset into train and test
    # Find split point
    split_index = int(len(stock_df) * percent)

    # Split the DataFrame
    train_df = stock_df.iloc[:split_index+1]
    test_df = stock_df.iloc[split_index:]

    return train_df, test_df


# ### Create the new labels - after normalising so we can have a
consistent percentage threshold

# In[66]:


def classify_price_changes(df, price_column, threshold_percent):

    # Calculate the percentage change in price
    price_change_percentage = df[price_column].pct_change() * 100

    # Classify the transition in price into one of 3 categories
    def classify_transition(price_change):

        if price_change > threshold_percent:
            return 'Up' # Price Increase

        elif price_change < -threshold_percent:
            return 'Down' # Price Decrease

        else:
            return 'Stable' # No significant Price Change

    return price_change_percentage.apply(classify_transition)


# ### Encoding Target Categories

# The difference between categories is not uniform (i.e., the step
from 'Down' to 'Stable' is not the same as from 'Stable' to 'Up').
Therefore, ordinal encoding is not the best fit.
#
# (Numerical distance is meaningful in models like logistic
regression, SVMs)
#
# Choose one hot encoding

# In[67]:
```

```
def one_hot_encode_targets(df,cols):

    # Initialise encoder
    one_hot_encoded_df = pd.get_dummies(df, columns=cols)

    return one_hot_encoded_df


# In[68]:


def full_classification_preprocesser(stock_df):

    # DATA PREPARATION - same functions as used in regression before
    train_df, test_df = split_stock_df(stock_df,0.8)

    # Classify price changes
    train_df['ibm_price_change'] = classify_price_changes(train_df,
'ibm_Close(t)', 1.0)
    test_df['ibm_price_change'] = classify_price_changes(test_df,
'ibm_Close(t)', 1.0)

    # One hot encode targets
    train_df = one_hot_encode_targets(train_df,['ibm_price_change'])
    test_df = one_hot_encode_targets(test_df,['ibm_price_change'])

    # Drop the following date columns from both datasets
    # The reason is that these contain class values that do not
exist in the test set
    # due to the time series nature of the split (mostly due to
there not being a leap year in test set)

    train_df = train_df.drop(['Year'],axis=1)
    test_df = test_df.drop(['Year'],axis=1)

    train_df = train_df.drop(['Is_leap_year'],axis=1)
    test_df = test_df.drop(['Is_leap_year'],axis=1)

    train_df = train_df.drop(['DayofYear'],axis=1)
    test_df = test_df.drop(['DayofYear'],axis=1)

    train_df = train_df.drop(['Week'],axis=1)
    test_df = test_df.drop(['Week'],axis=1)

    # Drop Close Price from the model
    train_df = train_df.drop(['ibm_Close(t)'],axis=1)
    test_df = test_df.drop(['ibm_Close(t)'],axis=1)

    # Encode rest of the categories for training data
    # Get remaining categorical columns
    train_int64_columns =
train_df.select_dtypes(include=['int64']).columns
```

```
    # One hot encode columns
    train_df = one_hot_encode_targets(train_df,train_int64_columns)

    # Encode rest of the categories for test data
    # Get remaining categorical columns
    test_int64_columns =
test_df.select_dtypes(include=['int64']).columns
    # One hot encode columns
    test_df = one_hot_encode_targets(test_df,train_int64_columns)

    return train_df, test_df


# ### Do one-hot encodings for rest of the categoricals - (the "Is"
columns are done for us)
#
#
# Day
#
#
# DayofWeek
#
#
# DayofYear
#
#
# Week
#
#
# Year
#
#
# Month
#

# In[69]:


stock_df = general_classification_preprocessing()
train_df, test_df = full_classification_preprocesser(stock_df)


# In[70]:


train_df[['ibm_price_change_Up','ibm_price_change_Stable','ibm_price
_change_Down']].value_counts()


# ### Imbalanced Class distributions - Notice the problem here with
value counts
#
# * Use upsampling method to handle this
#
```

```
# Imbalanced learn documentation - SMOTE (a clustering KNN based
upsampling technique to generate similar data points)
# https://imbalanced-learn.org/stable/references/generated/
imblearn.over_sampling.SMOTE.html
#
# https://medium.com/@corymaklin/synthetic-minority-over-sampling-
technique-smote-7d419696b88c
#
#
# ""
# SMOTE - Synthetic Minority Over-sampling Technique
#
# Actually uses K nearest neighbours clustering algorithm underneath
the hood to generate new synthetic datapoints to create more
representative instances of a minority category (in our case default
= yes)
# ""

# In[71]:


# Want to upsample the underpresented classes
def upsample(X_train, y_train):

    smote = SMOTE(random_state=42)
    X_train_smote, y_train_smote = smote.fit_resample(X_train,
y_train)

    return X_train_smote, y_train_smote


# ## Split data into X and y --> predict binary class outcome for
one specific stock
#
# * We choose "Down" as the most important factor for risk
assessment - so will train a binary classifier
#

# In[72]:


# split a train or test df into X and multivariate y
def data_label_split_multi(df):
    # split training df into X and y
    y = df[['ibm_price_change_Down']]
    X =
df.drop(['ibm_price_change_Down','ibm_price_change_Up','ibm_price_ch
ange_Stable'],axis=1)
    return X,y


# In[73]:
```

```
# Re-index both sets by Date
train_df.set_index('Date', inplace=True)
test_df.set_index('Date', inplace=True)


# In[74]:


X,y = data_label_split_multi(train_df)
X_test,y_test = data_label_split_multi(test_df)


# In[75]:


# Upsample the training data
X,y = upsample(X, y)


# In[76]:


y.value_counts()


# Classes are now balanced !

# #### Data should now be ready to feed to models
#

# ## Models
# Sources:
#
# * SVMs: https://www.sciencedirect.com/topics/nursing-and-health-
professions/support-vector-machine
# * SCV sklearn: https://scikit-learn.org/stable/modules/generated/
sklearn.svm.SVC.html
#
# SVMs are effective in high-dimensional spaces - Even in a space
with many dimensions, SVMs can perform well as long as there is a
clear margin of separation between classes
#
# * This is good as our data is very high dimensional (with manh
features so this justifies not having to do extra dimensioanlit
yreduction pre-processing)
#
# Spot check an SVC

# In[77]:


# Cross validation binary classifier
def binary_classifier(model,X,y):
```

```
    # Very similar time series cross validation implementation to
the regression approach
    tscv = TimeSeriesSplit(n_splits=3)

    # Performance metric arrays
    accuracy_scores = []
    f1_scores = []

    for train_index, test_index in tscv.split(X):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        y_train = y_train.values.ravel()
        y_test = y_test.values.ravel()

        model.fit(X_train, y_train)
        predictions = model.predict(X_test)

        # Calculate Accuracy score
        accuracy = accuracy_score(y_test, predictions)
        accuracy_scores.append(accuracy)

        # Calculate F1-score
        f1 = classification_report(y_test, predictions,
output_dict=True, zero_division=1)['macro avg']['f1-score']
        f1_scores.append(f1)

        # Classification report
        print("Report:\n", classification_report(y_test,
predictions, zero_division=1))

    # Calculate average accuracy across all of the validation folds
    average_accuracy = np.mean(accuracy_scores)
    print(f"Average Accuracy across all folds: {average_accuracy}")

    # Calculate average F1-score across all of the validation folds
    average_f1 = np.mean(f1_scores)
    print(f"Average F1-score across all folds: {average_f1}")


# In[78]:


# This can be run against each of the target classes (Up, Stable or
Down)
binary_classifier(SVC(),X,y)


# #### Also try using a Logistic regressor for comparison

# In[79]:


# This can be run against each of the target classes (Up, Stable or
```

```
Down)
binary_classifier(LogisticRegression(),X,y)


# ### Insights
#
# * SVC possibly too complex a model for this data, choose Logistic
Regression going forward

# # Importance of Close Price --> Down Class
#
# Most interested in flagging the Down class in terms of risk
assessment
#

# ### Hyperparameter fine-tuning
#
# LogisticRegression sklearn: https://scikit-learn.org/stable/
modules/generated/sklearn.linear_model.LogisticRegression.html
#
# Hyperparam interpretation: https://scikit-learn.org/stable/
modules/generated/sklearn.linear_model.LogisticRegressionCV.html
#
# * NB: This process involved quite a lot of trial and error and
iterating through methods
#
# Note the code in the following cell is commented out by default as
it is computantionally intensive to run (you can uncomment and run
if you wish)

# In[80]:


# # Hyperparam fine-tuning for Model

# # Parameter grid (for hyperparam value ranges)
# param_grid = {
#     'C': np.logspace(-4, 4, 20),
#     'penalty': ['l1', 'l2'],
#     'solver': ['liblinear']
# }

# tscv = TimeSeriesSplit(n_splits=3)

# # Create model object
# model = LogisticRegression()

# # Set up random grid search (with cross validation)
# randomized_cv = RandomizedSearchCV(
#     estimator=model,
#     param_distributions=param_grid,
#     n_iter=100,
#     cv=tscv,
#     scoring='f1_macro',  # Choose F1 as scoring metric (most
```

```
important for our case)
#     random_state=42,
#     verbose=1
# )

# # Fit randomised search estimator object to training data
# randomized_cv.fit(X, y.values.ravel())

# # Obtain best model choice from the randomised search estimator
# best_model = randomized_cv.best_estimator_

# # Print optimal parameters and corresponding F1 score
# print("Optimum Parameters:", randomized_cv.best_params_)
# print("Optimum F1 Score:", randomized_cv.best_score_)


# In[81]:


# Final Model Evaluation against test data - using optimum
hyperparams found

# Chosen model with optimal hyperparams
model =
LogisticRegression(solver='liblinear',penalty='l1',C=11.690190732246
,random_state=42)
model.fit(X,y)
predictions = model.predict(X_test)

# Get probability decision scores (probabilities used for the class
predictions)
y_scores = model.decision_function(X_test)

# Classification report
print("Report:\n", classification_report(y_test, predictions,
zero_division=1))


# Not bad results this time!

# In[82]:


# Obtain some other useful metrics
coefficients = model.coef_
feature_names = X.columns
feature_importance = pd.Series(coefficients[0], index=feature_names)
probabilities = model.predict_proba(X_test)
uncertainty = entropy(probabilities.T)


# In[83]:
```

```
# Look at most important features inolved in the predictions
feature_importance.head(2)


# ### The two most important features involved in prediction were
Open Price (as with regression), and SD20 indicator

# ### Confusion matrix - can be related to Recall and Type II error
#
# ref for code implementation: https://github.com/DTrimarchi10/
confusion_matrix/blob/master/cf_matrix.py

# In[84]:


# Confusion matrix
from sklearn.metrics import confusion_matrix

confusion_matrix = confusion_matrix(y_test,predictions)

# Labels for confusion matrix cells
cell_labels = ['True Negative (TN)', 'False Positive (FP)', 'False
Negative (FN)', 'True Positive (TP)']

# Flatten the confusion matrix and combine it with the labels
counts = ["{0:0.0f}".format(value) for value in
confusion_matrix.flatten()]
percentage_counts = ["{0:.2%}".format(value) for value in
confusion_matrix.flatten()/np.sum(confusion_matrix)]

labels = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in zip(cell_labels,
counts, percentage_counts)]
labels = np.asarray(labels).reshape(2,2)

sns.heatmap(confusion_matrix, annot=labels, fmt='', cmap='coolwarm')


# In[85]:


# Get Scores
print("Precision: ",precision_score(y_test,predictions))
print("Recall: ",recall_score(y_test,predictions))
print("F1 Score: ",f1_score(y_test,predictions))


# In[86]:


# Plot Precision-recall curve
precisions, recalls, thresholds, =
precision_recall_curve(y_test,y_scores)
plt.plot(recalls, precisions, marker='.', label='Logistic
Regression')
```

```
plt.xlabel('Recall')
plt.ylabel('Precision')


# Hands on ML
#
# Precision, recall, threshold plot:
# https://rachelchen0104.medium.com/hands-on-machine-learning-with-
scikit-learn-tensorflow-bbb1c91cb128

# In[87]:


# Precision, recall, threshold plot imspired by hands-on ml

# Plot Precision-Recall vs. Threshold
plt.figure(figsize=(8, 6))
plt.plot(thresholds, precisions[:-1], label='Precision')
plt.plot(thresholds, recalls[:-1], label='Recall')
plt.xlabel('Decision Threshold')
plt.ylabel('Precision / Recall')
plt.legend()


# In[88]:


# ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_scores)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC
= {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right")


# In[89]:


# This will be used later in a combined ensemble decision method
classification_predictions = predictions


# ----------
# ----------
# ----------
# ----------
```

```
#
# # Clustering
#
# We will first group similar days (datapoints) into clusters based
on included features (excluding future price information). Once we
have created these clusters, we can use the mean of the historical
"next day" close prices of each cluster to make predictions about
the next day close price for a new unseen datapoint.
#

# ## Preprocessing Steps
#
# This time we will do PCA for dimensionality reduction so we won't
worry about any rigorous feature selection at first
#
# Preprocessing steps are the same as regression but we choose to
keep all technical indicators this time
#

# In[90]:


def general_clustering_preprocessing():

    # Load in fresh datasets
    ibm_df = pd.read_csv(ibm_path)
    google_df = pd.read_csv(google_path)
    msft_df = pd.read_csv(msft_path)
    amzn_df = pd.read_csv(amzn_path)

    # Convert date col in each to a datetime
    ibm_df['Date'] = pd.to_datetime(ibm_df['Date'])
    google_df['Date'] = pd.to_datetime(google_df['Date'])
    msft_df['Date'] = pd.to_datetime(msft_df['Date'])
    amzn_df['Date'] = pd.to_datetime(amzn_df['Date'])

    # drop data on other 3 so only has where datetime >= amzn
earliest datetime
    cutoff_date = pd.to_datetime('2010-10-18') # earliest entry from
the amazon data
    ibm_df = ibm_df[ibm_df['Date'] >= cutoff_date]
    msft_df = msft_df[msft_df['Date'] >= cutoff_date]
    google_df = google_df[google_df['Date'] >= cutoff_date]

    # Need to reset and align indices of each dataframe now
    ibm_df = ibm_df.reset_index(drop=True)
    msft_df = msft_df.reset_index(drop=True)
    google_df = google_df.reset_index(drop=True)

    # Code for normalising - make 4 normalised datasets with dates
aligned

    # drop date col
    ibm_df = ibm_df.drop('Date_col',axis=1)
```

```
    google_df = google_df.drop('Date_col',axis=1)
    amzn_df = amzn_df.drop('Date_col',axis=1)
    msft_df = msft_df.drop('Date_col',axis=1)

    # convert volume to float64
    ibm_df['Volume'] = ibm_df['Volume'].astype('float64')
    google_df['Volume'] = google_df['Volume'].astype('float64')
    amzn_df['Volume'] = amzn_df['Volume'].astype('float64')
    msft_df['Volume'] = msft_df['Volume'].astype('float64')

    # Code for getting rid of columns that are common across all of
the datasets (we will keep them on IBM)
    # Drop all int64 columns on every dataset apart from one
    int_cols = amzn_df.select_dtypes(include=['int64']).columns

    # Drop int64 columns
    amzn_df = amzn_df.drop(int_cols, axis=1)
    google_df = google_df.drop(int_cols, axis=1)
    msft_df = msft_df.drop(int_cols, axis=1)

    # Also drop date columns from these 3
    amzn_df = amzn_df.drop('Date', axis=1)
    google_df = google_df.drop('Date', axis=1)
    msft_df = msft_df.drop('Date', axis=1)

    # Drop all columns beginning with QQQ, SnP and DJIA from these 3
(as these will also be common across all datasets)

    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('QQQ')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('SnP')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in amzn_df.columns if
col.startswith('DJIA')]
    amzn_df = amzn_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in google_df.columns if
col.startswith('QQQ')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('SnP')]
    google_df = google_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in google_df.columns if
col.startswith('DJIA')]
    google_df = google_df.drop(columns_to_drop, axis=1)

    columns_to_drop = [col for col in msft_df.columns if
col.startswith('QQQ')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('SnP')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)
```

```
    columns_to_drop = [col for col in msft_df.columns if
col.startswith('DJIA')]
    msft_df = msft_df.drop(columns_to_drop, axis=1)

    # Rename all float64 type columns in each dataset by prefixing
with stock

    amzn_df.rename(columns={col: "amzn_" + col for col in
amzn_df.columns if amzn_df[col].dtype == 'float64'}, inplace=True)
    google_df.rename(columns={col: "google_" + col for col in
google_df.columns if google_df[col].dtype == 'float64'},
inplace=True)
    msft_df.rename(columns={col: "msft_" + col for col in
msft_df.columns if msft_df[col].dtype == 'float64'}, inplace=True)
    ibm_df.rename(columns={col: "ibm_" + col for col in
ibm_df.columns if ibm_df[col].dtype == 'float64'}, inplace=True)

    # Code for concatenating the datasets
    stock_df = pd.concat([ibm_df, amzn_df, msft_df, google_df],
axis=1)

    return stock_df


# In[91]:


stock_df = general_clustering_preprocessing()
float_cols = stock_df.select_dtypes(include=['float64']).columns

# MinMax scaling
# Normalise data - MinMax scale all floats (good for PCA) - stops
features with largest scale from dominating
# Must normalise the data so we can compare trends at similar scale
scaler = MinMaxScaler()
stock_df[float_cols] =
pd.DataFrame(scaler.fit_transform(stock_df[float_cols]),
columns=float_cols)

# Re-index via date
stock_df  = stock_df.set_index('Date', drop=True)

# One hot encode all integer columns
integer_columns = stock_df.select_dtypes(include=['int64']).columns
stock_df = one_hot_encode_targets(stock_df,integer_columns)

# Convert to binary
boolean_columns = stock_df.select_dtypes(include=['bool']).columns
stock_df[boolean_columns] = stock_df[boolean_columns].astype(int)


# In[92]:
```

```
stock_df.shape


# In[93]:


# Sklearn pca implementation
pca = PCA()
pca.fit(stock_df)


# Plotting code inspired by Lu Bai lecture notes (used similar
implementation previously)

# In[94]:


# Retrieve the cumulative sum using explained variance ratio
cumsum = np.cumsum(pca.explained_variance_ratio_)

# Number of dimensions required to reach 75% explained variance
dim = np.argmax(cumsum >= 0.75) + 1

print(f"Number of Principle Components required to reach 75%
Explained Variance: {dim}")

# Plot Nums components against cum exp var
plt.figure(figsize=(10, 5))
plt.plot(range(1, stock_df.shape[1]+1), cumsum, marker='o',
linestyle='--')
plt.axvline(x=dim, color='r', linestyle='--', label=f'{dim}
Components')
plt.axhline(y=0.75, color='r', linestyle='--', label='75% Explained
Variance')
plt.xticks(range(1, stock_df.shape[1]+1))  # Set x-axis ticks
plt.xlim(1, dim + 10)  # Adjust the second parameter as needed
plt.xlabel('Number of Principle Components')
plt.ylabel('Cumulative Explained Variance Ratio')
plt.grid(True)

# Visualise explained variance ratios
plt.figure(figsize=(10, 5))
plt.bar(range(1, dim + 1), pca.explained_variance_ratio_[:dim],
align="center")
plt.xlabel("Principal Component")
plt.ylabel("Explained Variance Ratio")
ax.set_xticks(range(13))


# In[95]:


pca = PCA(dim)
pca_df = pca.fit_transform(stock_df)
```

```
# In[96]:


pca_df = pd.DataFrame(pca_df)


# Chose 75% (after some experimentation later with cross validation)
as a trade-off
#
# retaining some complexity whilst still reducing the dimensionality
signifcantly

# ## Feature Loading
#
# A way to retrieve original features that had high impact on
composition of PCAs by computing the correlations between the
original variable and the principal components: https://
scentellegher.github.io/machine-learning/2020/01/27/pca-loadings-
sklearn.html

# Interested in PC1 since it was so large
#
# Seems to have been formed from moving average data mostly

# In[97]:


# Obtain top feature loadings for first principal component (largest
explained variance)
loadings = pd.DataFrame(pca.components_[:1, :].T, columns=['PC1'],
index=stock_df.columns)

# Empty dataframe to store top features
top_10_loadings = pd.DataFrame()

# Iterate through the loadings
for col in loadings.columns:

    # Sort these loadings by absolute value
    sorted_loadings =
loadings[col].abs().sort_values(ascending=False)

    # Find top 10 features with strongest contribution
    strongest_feats = sorted_loadings.head(10)

    # Store top features
    top_10_loadings[col] = strongest_feats

# Display relationships on plot
plt.figure(figsize=(10, 7))
sns.heatmap(top_10_loadings, annot=True, cmap='coolwarm', vmin=-1,
vmax=1)
```

```
plt.title('Feature Loadings')


# ### Data should now be ready to feed to a clustering model

# In[98]:


# Column names are integers -->
# Need to convert to Strings and prefix them for readability (and
required prior to feeding to models)
prefix = "PC_"
pca_df.columns = [prefix + str(col) for col in pca_df.columns]


# In[99]:


# Apply KMEANs clustering

# Fit kmeans model on pca data and predict clusters
kmeans = KMeans()
clusters = kmeans.fit_predict(pca_df)

# Create copy dataframe (extra columns to be added)
df = stock_df.copy()

# Add cluster data to the dataframe and index by Date
df['clusters'] =
pd.DataFrame(clusters,index=pd.Series(stock_df.index))

# Create "next day close" column by temporal shift of the original
dataframe close price rows
df['next_day_close(t)'] = stock_df['ibm_Close(t)'].shift(-1)

# Group the next day close price data by cluster
cluster_averages = df.groupby('clusters')
['next_day_close(t)'].mean()

# Create predicted close prices by taking the mean of cluster
averages
df['predicted_close(t)'] = df['clusters'].map(cluster_averages)

# Print Number of clusters
n_clusters = len(set(kmeans.labels_))
print(f"Number of clusters: {n_clusters}")

# There is no next-day data for the last row in dataframe so remove
this
df = df[:-1]

# Get new date series to store dates
dates = pd.Series(df.index)
```

```
# Calculate performance metrics from actual vs. predicted close
prices
mse = mean_squared_error(df['ibm_Close(t)'],
df['predicted_close(t)'])
rmse = np.sqrt(mse)
r2 = r2_score(df['ibm_Close(t)'], df['predicted_close(t)'])
print(f'MSE: {mse}')
print(f'RMSE: {rmse}')
print(f'R-squared: {r2}')

# Plot actual vs. predicted
plt.figure(figsize=(10, 6))
plt.plot(df['ibm_Close(t)'], label='Actual Close Price')
plt.plot(df['predicted_close(t)'], label='Predicted Close Price')
plt.xlabel('Date')
plt.ylabel('Normalised IBM Close Price')
plt.title('Actual vs Predicted Close Prices')
plt.legend()


# ### KMeans is quite poor
#
# * Only manages to captures extremely general step-like movements
#
# Let us try another model -
#
# ### Affinity Propagation
#
# Affinity Propagation Clustering: https://scikit-learn.org/stable/
modules/generated/sklearn.cluster.AffinityPropagation.html
#
# https://towardsdatascience.com/unsupervised-machine-learning-
affinity-propagation-algorithm-explained-d1fef85f22c8
#
# --> Sends messages between pairs of samples until convergence.
# Does not require the number of clusters to be specified.
# Can be computationally expensive.
# Affinity Propagation do not require pre-specifying the number of
clusters but can be computationally intensive for large datasets
#

# In[100]:


# This code is the same process as above but using affinity
propagation model

# Apply Affinity Propagation clustering
affinity_prop = AffinityPropagation()
clusters = affinity_prop.fit_predict(pca_df)

# Create copy dataframe (extra columns to be added)
df = stock_df.copy()
```

```
# Add cluster data to the dataframe and index by Date
df['clusters'] =
pd.DataFrame(clusters,index=pd.Series(stock_df.index))

# Create "next day close" column by temporal shift of the original
dataframe close price rows
df['next_day_close(t)'] = stock_df['ibm_Close(t)'].shift(-1)

# Group the next day close price data by cluster
cluster_averages = df.groupby('clusters')
['next_day_close(t)'].mean()

# Create predicted close prices by taking the mean of cluster
averages
df['predicted_close(t)'] = df['clusters'].map(cluster_averages)

# Print Number of clusters
n_clusters = len(set(affinity_prop.labels_))
print(f"Number of clusters: {n_clusters}")

# There is no next-day data for the last row in dataframe so remove
this
df = df[:-1]

# Get new date series to store dates
dates = pd.Series(df.index)

# Calculate performance metrics from actual vs. predicted close
prices
mse = mean_squared_error(df['ibm_Close(t)'],
df['predicted_close(t)'])
rmse = np.sqrt(mse)
r2 = r2_score(df['ibm_Close(t)'], df['predicted_close(t)'])
print(f'MSE: {mse}')
print(f'RMSE: {rmse}')
print(f'R-squared: {r2}')

# Plot actual vs. predicted
plt.figure(figsize=(10, 6))
plt.plot(df['ibm_Close(t)'], label='Actual Close Price')
plt.plot(df['predicted_close(t)'], label='Predicted Close Price')
plt.xlabel('Date')
plt.ylabel('Normalised IBM Close Price')
plt.title('Actual vs Predicted Close Prices')
plt.legend()


# Hyperparam tuning is a bit more complicated in this scenario
#
#

# A lot of trial and error was used here
#
# Custom version of the silhouette scorer was implemented - see
```

```
docs/source code
# https://scikit-learn.org/stable/modules/generated/
sklearn.metrics.silhouette_score.html
#
# https://github.com/scikit-learn/scikit-learn/blob/
4f97facc3a992c6e2459c3da86c9d69b0688d5ab/sklearn/metrics/cluster/
_unsupervised.py#L38
#
# --> Measures how 'similar' an object is to its own cluster
compared to other clusters. The silhouette score ranges from -1 to
+1, where a high value indicates that the object is well matched to
its own cluster and poorly matched to neighboring clusters.
#
# (Also see documentation for appropriate hyperparams used)
#
#

# #### The following cell was commented out by default as search is
computationally intensive (can be uncommented)

# In[101]:


# # Define a custom version of the scoring function taken from
sklearn silhouette scoring
# def silhouette_scorer(estimator, X):
#     # Fit estimator model and predict clusters
#     clusters = estimator.fit_predict(X)
#      # Silhouette score is not defined for edge cases ie. 1 or max
clusters
#     if len(set(clusters)) == 1 or len(set(clusters)) == len(X):
#         return -1
#     # Actual silhouette score function
#     return silhouette_score(X, clusters)

# # Set up prmeter grid for search
# param_grid = {
#     'damping': np.linspace(0.5, 1.0, num=12, endpoint=False),
#     'max_iter': [200, 300, 400, 500, 600]
# }

# # RandomizedSearchCV used with the custom silhouette function
# randomized_search = RandomizedSearchCV(
#     # Pass in AffinityPropagation() object
#     AffinityPropagation(), param_grid, n_iter=10, random_state=42,
verbose=1,
#     # Score by silhouette score
#     scoring=make_scorer(silhouette_scorer)
# )

# # Fit the search to PCA data
# randomized_search.fit(pca_df)

# # Obtain the best parameters and best model
```

```
# print("Optimum Parameters:", randomized_search.best_params_)
# best_affinity_propagation = randomized_search.best_estimator_

# # Call predict on the best model to get final clusters predictions
# clusters = best_affinity_propagation.predict(pca_df)



# In[102]:


# Apply Optimised Affinity Propagation clustering obtained from
previous cell

# affinity_prop = AffinityPropagation(max_iter=200, damping=0.5)
# clusters = affinity_prop.fit_predict(pca_df)

# Create copy dataframe (extra columns to be added)
df = stock_df.copy()

# Add cluster data to the dataframe and index by Date
df['clusters'] =
pd.DataFrame(clusters,index=pd.Series(stock_df.index))

# Create "next day close" column by temporal shift of the original
dataframe close price rows
df['next_day_close(t)'] = stock_df['ibm_Close(t)'].shift(-1)

# Group the next day close price data by cluster
cluster_averages = df.groupby('clusters')
['next_day_close(t)'].mean()

# Create predicted close prices by taking the mean of cluster
averages
df['predicted_close(t)'] = df['clusters'].map(cluster_averages)

# Print Number of clusters
n_clusters = len(set(affinity_prop.labels_))
print(f"Number of clusters: {n_clusters}")

# There is no next-day data for the last row in dataframe so remove
this
df = df[:-1]

# Get new date series to store dates
dates = pd.Series(df.index)

# Calculate performance metrics from actual vs. predicted close
prices
mse = mean_squared_error(df['ibm_Close(t)'],
df['predicted_close(t)'])
rmse = np.sqrt(mse)
r2 = r2_score(df['ibm_Close(t)'], df['predicted_close(t)'])
print(f'MSE: {mse}')
```

```
print(f'RMSE: {rmse}')
print(f'R-squared: {r2}')

# Plot actual vs. predicted
plt.figure(figsize=(10, 6))
plt.plot(df['ibm_Close(t)'], label='Actual Close Price')
plt.plot(df['predicted_close(t)'], label='Predicted Close Price')
plt.xlabel('Date')
plt.ylabel('Normalised IBM Close Price')
plt.legend()


# Best results !

# ## Cluster Visualisation
#
# See Matplotlib docs for 3D scatter plots

# In[103]:


# Set up 3D pot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Use top 3 PCs from the PCA reduced data and use cluster labels
from Affinity Propagation model
scatter = ax.scatter(pca_df.iloc[:, 0], pca_df.iloc[:, 1],
pca_df.iloc[:, 2],
                     c=clusters, cmap='viridis', marker='o')

# Set axis labels corresponding to the 3 Principle Components
ax.set_xlabel('Principal Component 1')
ax.set_ylabel('Principal Component 2')
ax.set_zlabel('Principal Component 3')

# Add colourbar
plt.colorbar(scatter, label='Cluster')


# In[104]:


# Same process as above but we will use the first 2 PCs to create a
2D plot

plt.figure(figsize=(10, 8))
# Using first two PCs
scatter = plt.scatter(pca_df.iloc[:, 0], pca_df.iloc[:, 1],
                      c=clusters, cmap='viridis', marker='o')

plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar(scatter, label='Cluster')
```

```
# In[105]:


# These will be used later in the ensemble decision function
clustering_predictions = df['predicted_close(t)']
closes = df['ibm_Close(t)']


# ----------
# ----------
# ----------
# ----------

# # Final Ensemble Decision Making Model
#
# * Construct a function that takes nth day prediction from each
model and outputs a new binary value to invest or to not invest
#
# * How should this actually work? How long-term, over how many days
does user want to decide to invest or not?
#
# * The combination of model outputs is good and worthwhile

# Variables:
#
# * regression_pred: Next day's closing price predicted by the
regression model.
# * clustering_pred: Next day's closing price predicted by the
clustering model.
# * classification_pred: Prediction from the classification model (1
for decrease, 0 for no decrease).
#
# (Each of these are for day n)
#
# Example for choosing decisions:
#
# * Strong Selling Signal: Classification model predicts decrease
AND regression model predicts a significant drop (close prices dips
below a certain user-specified threshold)
#
# * Medium Selling Signal: Either the regression model predicts a
slight close price drop OR clustering model gives a downward trend.
#
# * Hold Signal: No strong indication of close price decrease in any
model

# In[106]:


# Final function for assessment of risk
def investment_risk_assessor(regression_pred, clustering_pred,
classification_pred, last_close_price):
```

```
    # Allow end users to choose suitable threshold - corresponds to
significant drop in close price
    drop_threshold = 0.05  # 5% drop
    sell_signal = last_close_price * (1 - drop_threshold)

    if classification_pred == 1 and regression_pred < sell_signal:
        return 'Strong Sell Signal: High Risk of Price Drop'
    elif regression_pred < last_close_price or clustering_pred <
last_close_price:
        return 'Medium Sell Signal: Moderate Risk of Price Drop'
    else:
        return 'Hold: Low Risk of Price Drop'


# ### Example of how this might work in practice
#
# * On day of prediction we obtain the required feature values for
our models on day (as well as historical data) and get 3 model
outputs
#
# * Feed the decision function the 3 model outputs for day of
prediction, as well as the preceding day's close price

# In[107]:


regression_predictions


# In[108]:


classification_predictions =
pd.DataFrame(classification_predictions)


# In[109]:


clustering_predictions = pd.DataFrame(clustering_predictions)


# In[110]:


closes = pd.DataFrame(closes)


# In[111]:


# Obtain penultimate value in closes array (accounts for the time
shifted data)
```

```
last_close_price = closes.iloc[-2].values[0]

# Obtain final value from the other prediction arrays
regression_pred = regression_predictions.iloc[-1].values[0]
classification_pred = classification_predictions.iloc[-1].values[0]
clustering_pred = clustering_predictions.iloc[-1].values[0]


# In[112]:


# Feed one row to the investment_decision function to get an example
outpupt
decision = investment_risk_assessor(regression_pred,
clustering_pred, classification_pred, last_close_price)


# In[113]:


decision


# In this case, the decision model thinks there is a moderate risk
of the price dropping!
```