Investment Strategy using Machine Learning and Technical Indicators

The following paper aims to introduce some basic machine learning models to identify buying and selling conditions for financial assets. In particular, the S&P500 index will be considered, with technical indicator features being extracted from historical price data. We will also consider techniques to identify and address overfitting, a condition where the model fails to generalise well to new data. We will then optimise the resultant model by tuning its hyperparameters to better fit the data. Our final model has an accuracy of 68% and could be used as part of a investment strategy to identify buying and selling conditions in stock indexes.

Contents

1. Preprocessing and Data Analysis

1.1. Data Source and Feature Extraction

Data for this project was sourced from Yahoo finance. In particular, the dataset used was the S&P500 from 1980 through to the beginning of 2024. The S&P500 consists of the 500 largest companies in the United States measured by market capitalisation. Initial features included the volume of trades made, the opening and closing prices alongside the lowest and highest trading price for each day.

For each trading day buying and selling conditions were calculated. These were determined by comparing the current price of the asset to its price in one months time (20 trading days):

- A rise of 2% or more was labelled as 'Strong Buy'.
- A rise of 1% up to 2% was labelled as 'Buy'.
- A price change between -1% and 1% was labelled as 'Hold'.
- A decline of between -1% and -2% was labelled as 'Sell'.
- A decline of more than -2% was labelled as 'Strong Sell'.

From this data source several technical indicators were calculated. Technical indicators are calculations based on historical price, volume, or open interest data of financial assets. These indicators are primarily used in technical analysis to aid traders and investors in making informed decisions about buying or selling these assets. These indicators help visualise and interpret market trends, momentum, volatility, and other relevant information. In total, this work makes use of eight technical indicators, which are used as features for our machine learning models. Some of the most significant technical indicators used in this work are listed below:

ROC The Rate of Change (ROC) is a technical indicator used to measure the percentage change in the current price of a financial asset relative to its price a certain number of periods ago. It is often used to assess the momentum or velocity of a price trend.

CCI The Commodity Channel Index (CCI) is a technical indicator designed to identify cyclical trends in financial markets, specifically focusing on detecting potential overbought or oversold conditions. Developed by Donald Lambert, CCI measures the relationship between the current price of an asset and its historical average.

RSI The Relative Strength Index (RSI) is a popular momentum oscillator used in technical analysis to assess the speed and change of price movements in a financial market. The RSI is based on the concept of relative strength, comparing the magnitude of recent gains to recent losses over a specified time period.

As all features (technical indicators) are numerical there is no preprocessing required to appropriately handle categorical data. If we had categorical features we would need to employ techniques, such as one-hot encoding, to appropriately format these features.

Technical indicators will be used to classify buy and sell signal for the underlying asset. We use basic machine learning techniques, such as logistic regression, alongside some more advanced approaches, such as neural networks, to identify these conditions. We will begin by assessing the feature importance of these technical indicators to obtain a better understanding of the significance of each indicator in determining buying or selling conditions.

1.2. Assessing Feature Importance with Random Forests

We want to be able to understand which features are important and contribute most significantly to the classification task. A useful approach for selecting relevant features is by using random forests. Random forests are an ensemble machine learning technique with the goal of averaging over multiple decision trees to build a more robust model that is less susceptible to overfitting. They also have the advantage of being highly interpretable as we can visualise the decision-making process over the entire tree.

Traditional decision trees are a hierarchical data structure used for decision making in classification tasks. Decision trees consists of nodes that represent decisions to be made on specific features. Based on the outcome of these decisions, algorithm moves to the next node in the tree which consists of another decision to be made or a classification for the object considered. In machine learning we typically only consider binary decision trees where there are only two branches at every node in the diagram.

Random forests are similar to decision trees; however, several decision trees are created, with each decision tree trained on a subset of the training data and a handful of the total features. This creates a 'forest' of decision trees averaged over subsets of the training data. The full process for creating a random forest can be summarised as follows:

1. Randomly choose n samples from the training dataset with replacement.
2. Train a decision tree from the sample. At each node randomly select f features with replacement. Train the decision tree (typically done by maximising the information gain, such as the Gini impurity).
3. Repeat steps 1-2 k times until we have trained k decision trees.
4. Aggregate the prediction over each tree and classify objects based on the most popular classification. This combines the predictions of all trees in the forest to determine the final prediction. (The final prediction is made via majority voting, where the most popular prediction amongst the trees becomes the final prediction).

Using a random forest, it is possible to measure the feature importance by evaluating the contribution of each feature to the overall predictive performance. During the construction of each

decision tree in the random forest, the algorithm measures the decrease in impurity (e.g. Gini impurity or entropy) resulting from splits based on different features. The importance of a feature is then calculated by averaging these impurity decreases across all trees in the forest. Features that consistently lead to more significant reductions in impurity are assigned higher importance scores.

Once we've trained the random forest classifier, we can list the most relevant features from the `feature_importances_` variable.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load data into a pandas dataframe
df = pd.read_csv(filename)
target = list(df['Ratings'])
columns_to_drop = [0,1,2,3,4,5,-1]
data = df.drop(df.columns[columns_to_drop], axis=1)
feat_labels = data.columns

# Split data into training and testing datasets
X_train, X_test, Y_train, Y_test = train_test_split(data, target, test_size
    =0.2)

# standardise dataset
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)

# Build random forest
clf = RandomForestClassifier()
clf.fit(X_train_std, Y_train)
importances = clf.feature_importances_

# Plot Feature importances
feature_plot(importances, X_train_std, feat_labels)
```

Fig.1 shows the rating of the most important features in our training dataset.

The ROC technical indicator is the most significant feature in this problem. This analysis could be used to remove less significant features from our model; that is, removing features that do not contribute significantly to the model predictions. This can reduce the computational overhead of the model and also decrease the complexity of the model.

However, if two or more features are highly correlated, one feature may be ranked highly while

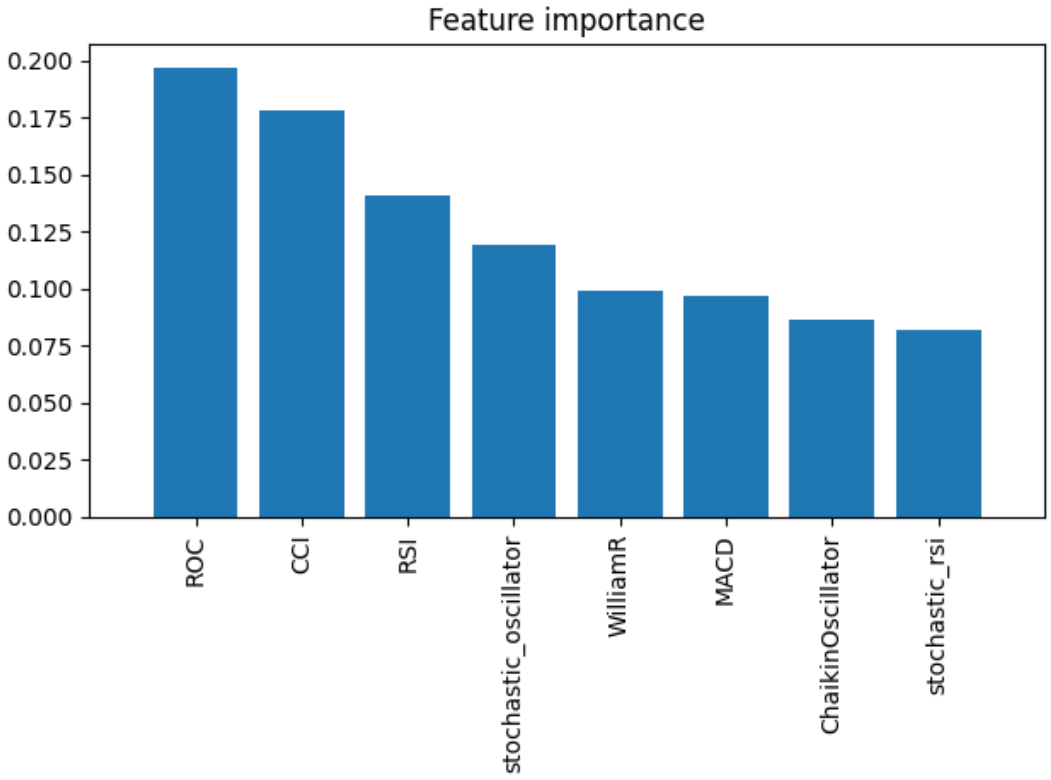

FIG. 1: The most significant technical indicators according to the random forest classifier model. The most significant feature is the ROC technical indicator. This ranking could be used to reduce the feature subspace of the problem by selecting the k most significant features.

the contribution of the other feature(s) may not be captured. In this sense one of the features dominates while the other appears not to contribute to the model. This does not impact the predictive performance of our model but does reduce its interpretability.

1.3. Dimensionality Reduction using Principal Component Analysis

Principal Component Analysis (PCA) helps identify patterns based on correlations between features. PCA transforms the original dataset features into a new set of uncorrelated (orthogonal) features known as principal components. PCA reduces the dimensionality of the dataset we are analysing as the projected subspace has a smaller dimension.

PCA analysis works in the following way:

- Standardise the d-dimensional dataset by transforming dataset features to be between zero and one (alternatively a transform using a normal distribution could be used).
- Construct a covariance matrix. The covariance matrix quantifies the correlation between features in our dataset.
- Decompose the covariance matrix into its eigenvectors and eigenvalues.

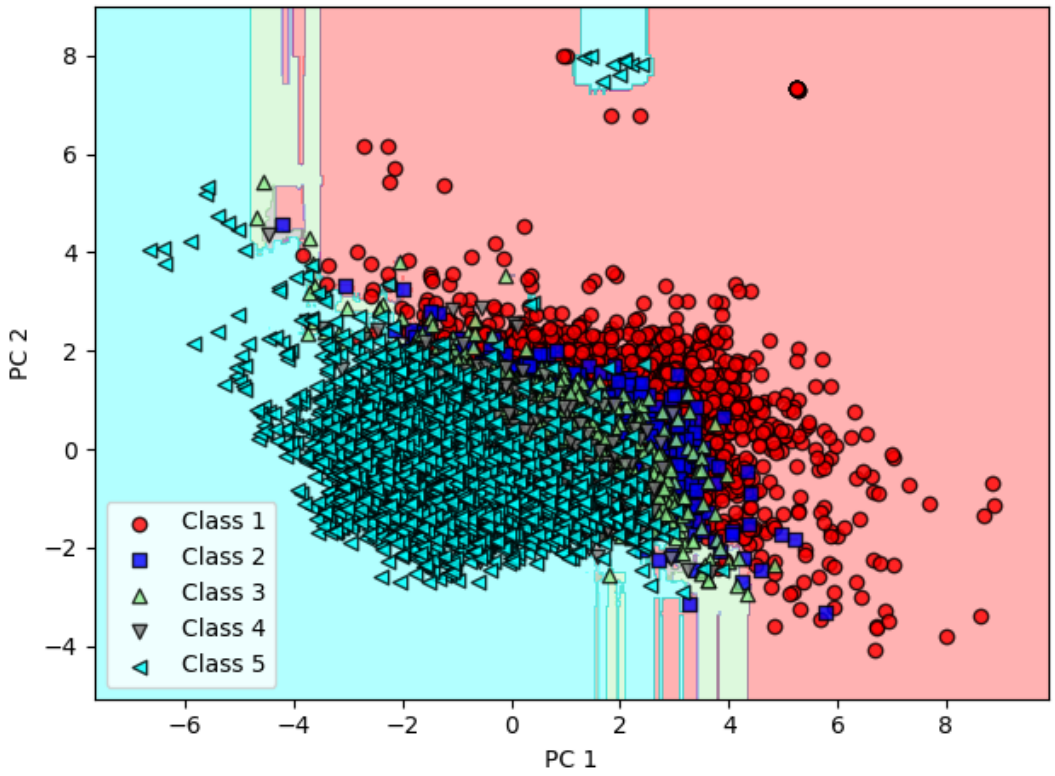

FIG. 2: PCA applied to our eight dimensional input features reducing the dimensionality of our dataset to two dimensions. We've then trained a random forest classifier to group data points into buying and selling conditions. Note that the complicated decision boundaries could be an indication of overfitting.

- Sort the eigenvalues by decreasing value. This ranks the importance of each eigenvector.
- Select k as the dimensionality of the projected feature subspace. Select the k most significant eigenvectors.
- Construct a projection between the eigenvectors and the new feature subspace.
- Transform the original dataset using the projection matrix. This will create a the k dimensional projected subspace.

Once we have the new projected feature subspace we can train our models on the reduced dataset. For example, we could train a random forest classifier on a two-dimensional subspace. Fig.2 shows the results of projecting our eight input features onto a two-dimensional subspace. We then train a random forest classifier on this dataset to identify buying and selling conditions.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

```

```
# Load data into a pandas dataframe
df = pd.read_csv(filename)
target = list(df['Ratings'])
columns_to_drop = [0,1,2,3,4,5,-1]
data = df.drop(df.columns[columns_to_drop], axis=1)
feat_labels = data.columns

# Split data into training and testing datasets
X_train, X_test, Y_train, Y_test = train_test_split(data, target, test_size
    =0.2)

# standardise dataset
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)

# Principal component analysis
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)

# Build random forest
clf = RandomForestClassifier()
clf.fit(X_train_pca, Y_train)
```

The decision regions for the training data are reduced to two component axes. The complicated decision boundaries between the two classes could be an indicator of overfitting. Our basic model reduced to two dimensions has an accuracy of 62%. Increasing the dimensionality of the PCA subspace may improve the performance of the model. For example, increasing the PCA subspace dimension to five increases the accuracy to 67%.

2. Model Selection

Now that we've appropriately formatted our training data its time to select an algorithm to use for supervised classification. Supervised classification is a machine learning technique where a model is trained on a labeled dataset to predict the categorical class of new unseen instances based on learned patterns from the training data. We'll examine four models that can be used for this task: logistic regression, support vector machines, random forests and feed-forward neural networks. We'll describe the basic mechanics behind these models and discuss the accuracy of the models when applied to our dataset.

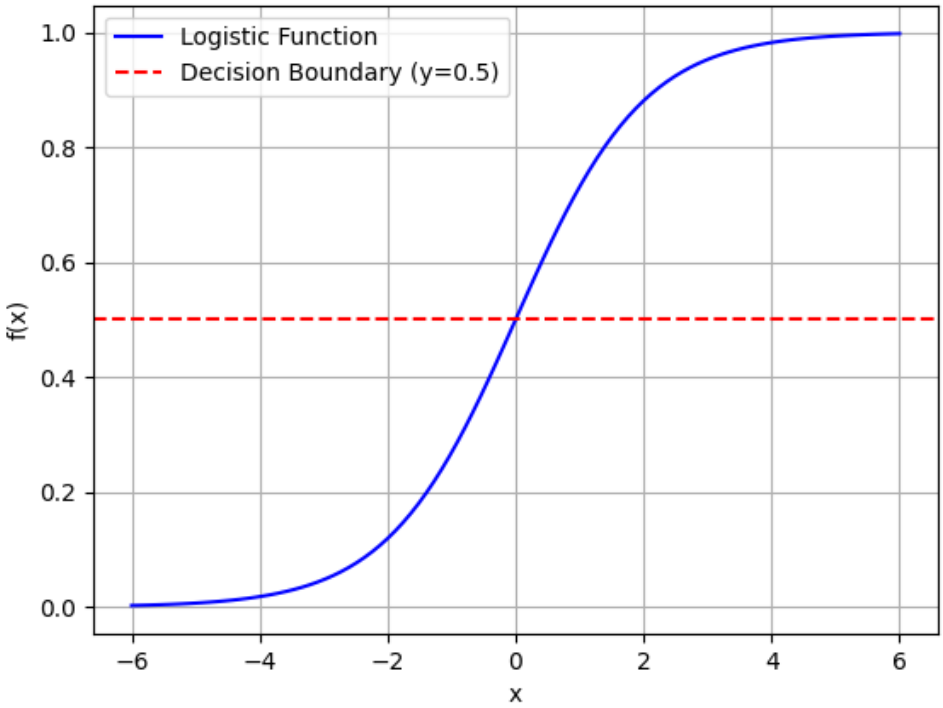

FIG. 3: A plot showing the logistic function. The function maps values to between zero and one. The output of this function can therefore be interpreted as the probability that an input value belongs to a particular class.

2.1. Logistic Regression

Logistic regression is used for binary classification − meaning the algorithm predicts the probability of an instance belonging to one of two outcomes. In our case we'll use the extension of logistic regression which predicts the probability of an instance belonging to multiple outcomes. This uses the one-versus-all (OvA) method, which is a multiclass classification approach where a separate binary classifier is trained for each class, treating it as the positive class while considering the other classes as the negative class.

In logistic regression, the model calculates a weighted sum of the input features and applies a logistic function to the sum. The model is trained by minimising the logistic loss, adjusting the weights assigned to each feature through an optimisation algorithm (e.g., gradient descent) to maximise the likelihood of the observed class labels. The logistic function maps all numbers to a value between zero and one meaning the output can be interpreted as the probability that an instance belongs to a particular class. A plot of the logistic function can be seen in Fig.3.

Logistic regression models are trained on labelled data. Weights (and the bias) are updated during training to minimise the difference between the models predictions and the training data outcomes. This means adjusting the weights to minimise the loss function associated with

logistic regression. The most commonly used loss function in logistic regression is the logistic loss, also known as cross-entropy loss or log loss. The process of training a logistic regression model is given below:

Initalise Parameters Start with initial values for the weights and bias.

Forward Pass For each training instance, compute the predicted probability for the instance belonging to a particular class. This is done using the weights and bias of the model and the logistic function.

Compute Loss Use the logistic loss function to calculate the loss for the current model parameters.

Compute Gradients Calculate the gradients of the loss with respect to the model parameters (weights and bias).

Update Parameters Adjust the parameters in the opposite direction of the gradients to minimise the loss.

Repeat Iterate through steps 2-5 for a predetermined number of iterations.

The logistic regression model is similar to the original perceptron model except the stepwise activation function is replaced with the logistic function. A logistic regression model can be trained using scikit-learn:

```
from sklearn.linear_model import LogisticRegression

# Load data into a pandas dataframe
df = pd.read_csv(filename)
target = list(df['Ratings'])
columns_to_drop = [0,1,2,3,4,5,-1]
data = df.drop(df.columns[columns_to_drop], axis=1)
feat_labels = data.columns

# Split data into training and testing datasets
X_train, X_test, Y_train, Y_test = train_test_split(data, target, test_size
    =0.2)

# Standardise dataset
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)

# Train logistic regression model
lr = LogisticRegression(C=100.0, solver='lbfgs', multi_class='ovr')
```

```
lr.fit(X_train_std, Y_train)

# Test accuracy
Y_pred = lr.predict(X_test_std)
accuracy = accuracy_score(Y_test, Y_pred)
print("Accuracy:", accuracy)
```

The $C = 100$ parameter refers to the regularisation strength. Regularisation is designed to prevent the model from becoming too complex and overfitting the data. Regularisation adds another term to the loss function designed to prevent the model from assigning too much importance to one feature. C is inversely proportional to the regularisation strength; lower values of C correspond to stronger regularisation.

Our logistic regression model has a final accuracy of 66%.

2.2. Support Vector Machines

Support Vector Machines (SVM) are a type of machine learning algorithm used for classification tasks. SVMs work by finding the best possible decision boundaries that separate classes in the training data. Suppose you have points on a plot, with each point belonging to one of two categories. SVMs aim to to find a line (or hyperplane in larger dimensions) that maximally separates these points into classes.

SVMs identify support vectors, which are the data points closest to the decision boundary. SVMs then maximise the margin between decision boundaries, which is the distance between support vectors of the two classes. By maximising the margin between support vectors, the algorithm maximises the distance between the decision boundaries of the two classes.

If a linear decision boundary does not fit the data well, SVMs can employ a technique known as the kernel trick. This maps the data onto a higher-dimensional space where linear separation is possible. The most popular kernel to achieve this it the radial bias function (RBF) kernel. This allows the SVM model to generalise to non-liner problems, making it an attractive model to chose whilst still being relatively lightweight.

```
from sklearn.svm import SVC

# Train SVM model
svm = SVC(kernel='linear', C=1.0, random_state=1)
svm.fit(X_train_std, Y_train)

# Test accuracy
Y_pred = svm.predict(X_test_std)
accuracy = accuracy_score(Y_test, Y_pred)
print("SVM Accuracy:", accuracy)
```

```
return
```

Training this model gives an accuracy of 68%. Normally SVMs and logistic regression models give similar results, hence the similar accuracy.

2.3. Random Forests

We've already described the random forest classifier so we will not revisit it here. A random forest classifier can be implemented in the following way using scikit-learn:

```
from sklearn.ensemble import RandomForestClassifier

# Define model
clf = RandomForestClassifier()
clf.fit(X_train_std, Y_train)

# Accuracy Test
Y_pred = clf.predict(X_test_std)
accuracy = accuracy_score(Y_test, Y_pred)
print("Accuracy:", accuracy)
```

This model had an accuracy of 69%.

2.4. Neural Networks

Neural networks are a relatively complicated topic and we will not go over them in depth here but instead aim to give an overview of the topic. Interested readers can find a more thorough treatment here: `https://bluehood.github.io/research/benh_machine-learning-intrusion-detection_2024.pdf`.

Neural networks (NN) are computational models inspired by the human brain's structure and functioning. Comprising of interconnected nodes, or artificial neurons, organised into layers, these networks process information through a series of operations. Each connection between neurons is associated with a weight that adjusts during training, allowing the network to learn patterns and relationships in the data. The input layer receives raw data, such as pixel values in an image or feature vectors in natural language processing or, in our case, a set of technical indicators. The information then propagates through hidden layers, where weighted sums and activation functions transform the input. Finally, the output layer produces the network's prediction or classification. During training, the network adjusts its weights based on the error between predicted and actual outputs, minimising this error through backpropagation and optimisation algorithms like gradient descent.

NN are known to be able to approximate arbitrary functions, which makes them a strong candidate as a machine learning model. Their ability to learn complex patterns in data makes them

versatile for tasks such as image recognition and language translation. We will apply a NN to our technical indicators in order to classify buying and selling conditions.

Our NN consists of an input layer, with eight input values corresponding to our eight technical indicators, mapping to 256 nodes. There are then two further layers, one with 256 input nodes outputting to 128 nodes and one with 128 input nodes mapping to 5 output features. The Relu activation function is applied to the input layer and the hidden layers. The output layer maps to five output neurons corresponding to our possible output buying/selling conditions. Cross Entropy Loss was defined for the loss function and the popular Adam optimiser was used. A learning rate of 0.01 was defined and the network was trained for 20 epochs. The model was defined in the following class:

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader

class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, 256)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(256, 128)
        self.l3 = nn.Linear(128, num_classes)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.l3(out)
        return out
```

The NN had a final accuracy of 69.6%.

3. Evaluation and Hyperparameter Tuning

After collecting the dataset, performing dimensionality reduction and training some base models we can start evaluating the model and fine-tuning their hyperparameters. In this section, we'll use the random forest classifier, as this was one of the best performing models.

3.1. K-Fold Cross Validation

K-Fold Cross Validation is a technique to assess how well a machine learning model generalises to unseen data. Data is divided into k folds (subsets) of equal length. The model is then trained and evaluated k times, using one fold as the testing dataset and the remaining folds as the training set. This ensures that the model is evaluated over different parts of the dataset, resulting a more robust overview of its performance.

We can implement K-fold Cross Validation using scikit-learn:

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(estimator=clf, X=X_train_std, y=Y_train, cv=10,
    n_jobs=1)
print(f'CV accuracy scores: {scores}')
print(f'CV accuracy: {np.mean(scores):.3f} +/- {np.std(scores):.3f}')
```

This resulted an accuracy of 67.6% over all folds, plus or minus 2.2%. This low level of variance suggests that the model will generalise well unseen data.

3.2. Learning Curves

Learning curves are visual representations of how a model performs and how this performance changes as it learns from the training data. Learning curves are designed to assess how well a model is learning and the models dependence on the size of data provided or the number of epochs that the model is trained for. For example, learning curves can show if a model is overfitting; that is, performing well on the training data but poorly on unseen data.

Learning curves can help engineers make decisions about how to adjust the model's complexity, whether regularisation techniques should be implemented to reduce overfitting, or whether more training data should be collected.

Fig.4 shows the learning curve for a random forest classifier applied to our technical indicator data. There is a large discrepancy between the accuracy of the training and validation sets. In this case there is a difference of roughly 35% between the two datasets. This large difference suggests that the random forest classifier model is overfitting and will not generalise well to new data.

One method for dealing with overfitting is to tune the model's hyperparameters. By selecting certain values for the models parameters we can reduce the complexity of the model, which in turn will reduce the amount of overfitting. For example, we can set the number of estimators and tree depth in our model to reduce its complexity:

```
from sklearn.pipeline import make_pipeline

```

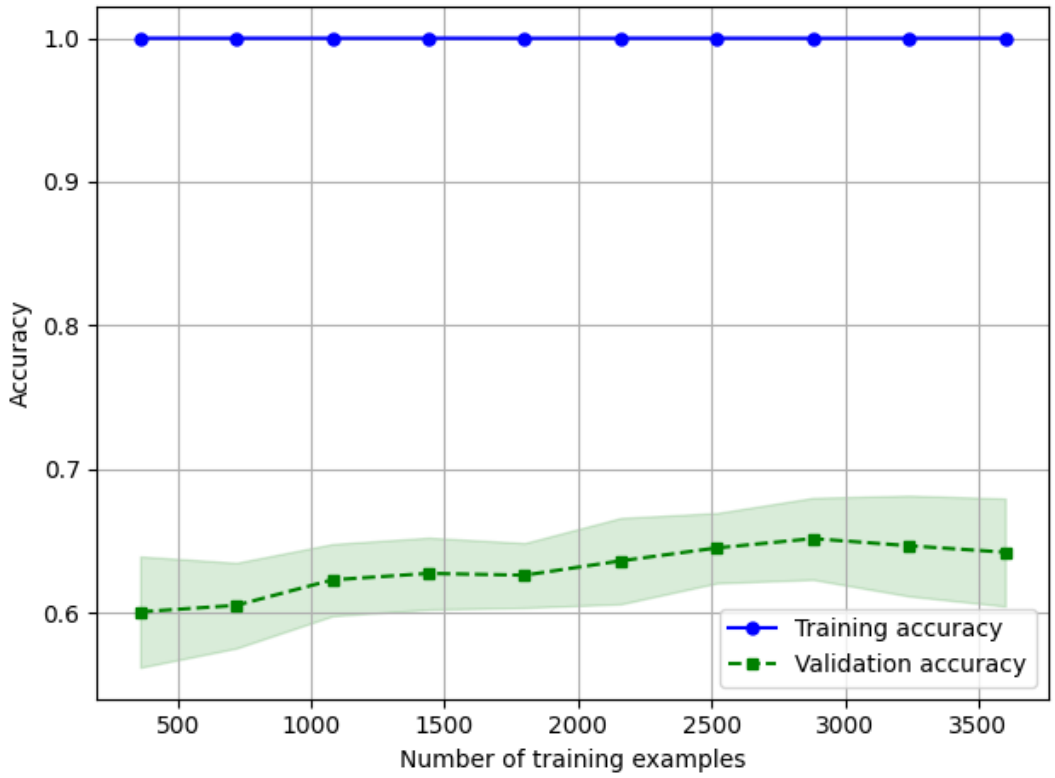

FIG. 4: A learning curve showing the random forest classifier model's accuracy as a function of the size of the training dataset passed to the model. The training accuracy has a constant value of 100% whilst the validation set has an accuracy of roughly 65%. The large discrepancy between the training and validation accuracy suggests that the model is overfitting and will not generalise well to new unseen data. Techniques to deal with overfitting should be implemented, such as hyperparameter tuning.

```
pipe_lr = make_pipeline(StandardScaler(), RandomForestClassifier(
    n_estimators=50, max_depth=5, min_samples_split=5, random_state=42))
```

Fig.5 shows the learning curve for our tuned random forest classifier. Reducing the number of estimators and providing a max depth reduced the difference between the training and validation set's accuracy. This reduced the degree of overfitting in our model, as the training and testing set accuracies converge to roughly the same value. Notice also that increasing the training examples available to the model improves the model's performance and reduces the degree of overfitting. This suggests that collecting more training data could improve the performance of the model.

3.3. Validation Curves

Validation curves show the variance in the performance of the model with respect to variations in a specific hyperparemter. Validation curves typically show the changes in the models performance as a function of a given hyperparameters.

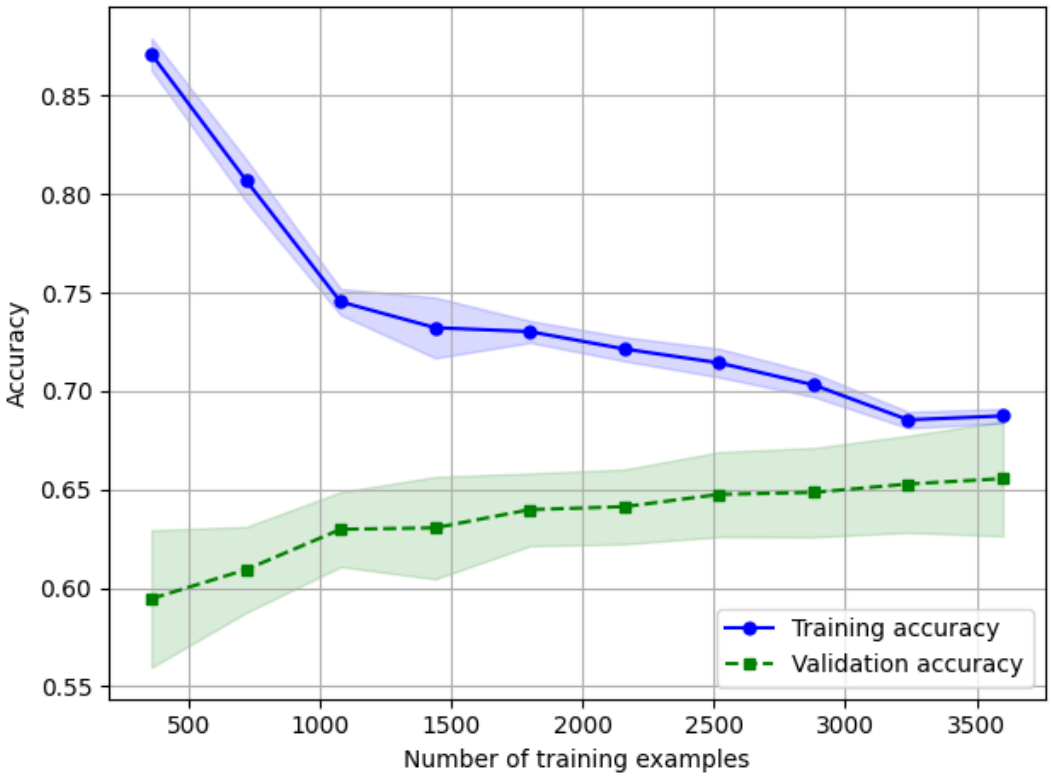

FIG. 5: A learning curve applied to our tuned random forest classifier model. Reducing the number of estimators to 50 and setting the max depth to 5 reduced the complexity of the model. This, in turn, reduced the difference between the training and validation sets accuracy. Our tuned model does not suffer from overfitting to the same extent as our untuned model. Note that the testing and validation accuracy converge as the number of training samples increase. This suggests that collecting more data for the model could increase the model's performance.

Fig.6 shows a validation curve for our random forest classifier applied to the tree max depth parameter. For small values of the max depth parameter the training and validation accuracies align well. Increasing the max depth parameter beyond 5 results in a discrepancy between the training and validation accuracies. In particular, the training accuracy increases to 100% as the decision boundaries of the model become more complex. The validation accuracy remains at around 68% showing that for large values of the max depth parameter the model is overfitting. Furthermore, for small values of the hyperparameter the model underfits. This suggests that the optimal value of this hyperparemter is 5.

We can also apply validation curves to other hyperparameters of the model. For example, Fig.7 shows a validation curve applied to the number of estimators. There is limited dependence of both the training and validation accuracy on this parameter, with each remaining relatively constant with respect to the number of estimators. Moreover, the training and validation accuracy are roughly the same value. This suggests that the value of the hyperparmeter will have little dependence on the model overfitting and will not influence the accuracy either.

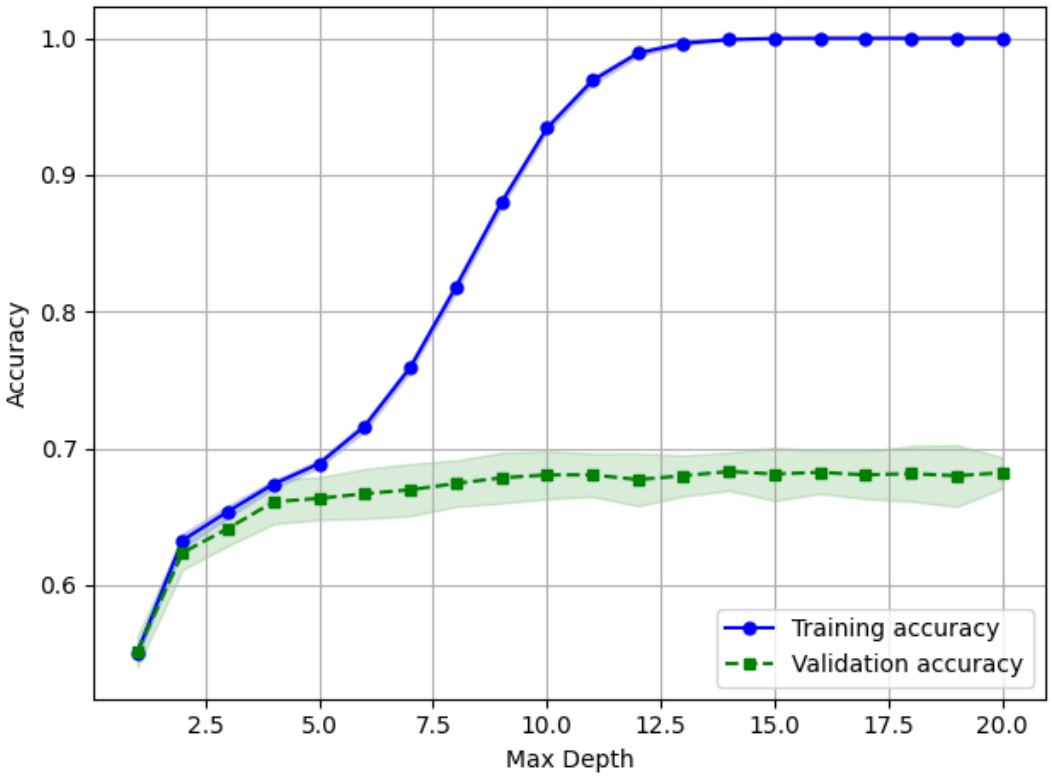

FIG. 6: A validation curve applied to the tree maximum depth parameter for our random forest classifier model. The plot shows both the training and validation accuracy. For small values of the max depth the training and validaiton accuracy are roughly the same value. However, increases the max depth parameter over 5 results in a divergence of the training accuracy from the validation accuracy. Validation accuracy plateaus at roughly 68% whilst the training accuracy increases towards 100%. This is again a consequence of the model overfitting the training data. Morover, as small values of the tree depth the model appears to underfit. This suggests that the optimal value of our hyperparameter is 5.

3.4. Optimising Hyperparameters using Randomised Search

A randomised search explores different combinations of hyperparameter settings for our model. This method uses a randomised search opposed to a simpler grid search which tries every possible combination of hyperparameters in a trial set. Randomised search tries a random subset of hyperparamater combinations to evaluate. For each combination the model is trained and evaluated and the accuracy is recorded. We can then select the hyperparameter values which optimise the performance of the model.

Our randomised search will focus on two hyperparamters: the number of trees (*n_estimators*) and the tree depth (*max_depth*). We will restrict the maximum tree depth to be below 5 to prevent the model from overfitting.

```
1 from sklearn.model_selection import RandomizedSearchCV
2
```

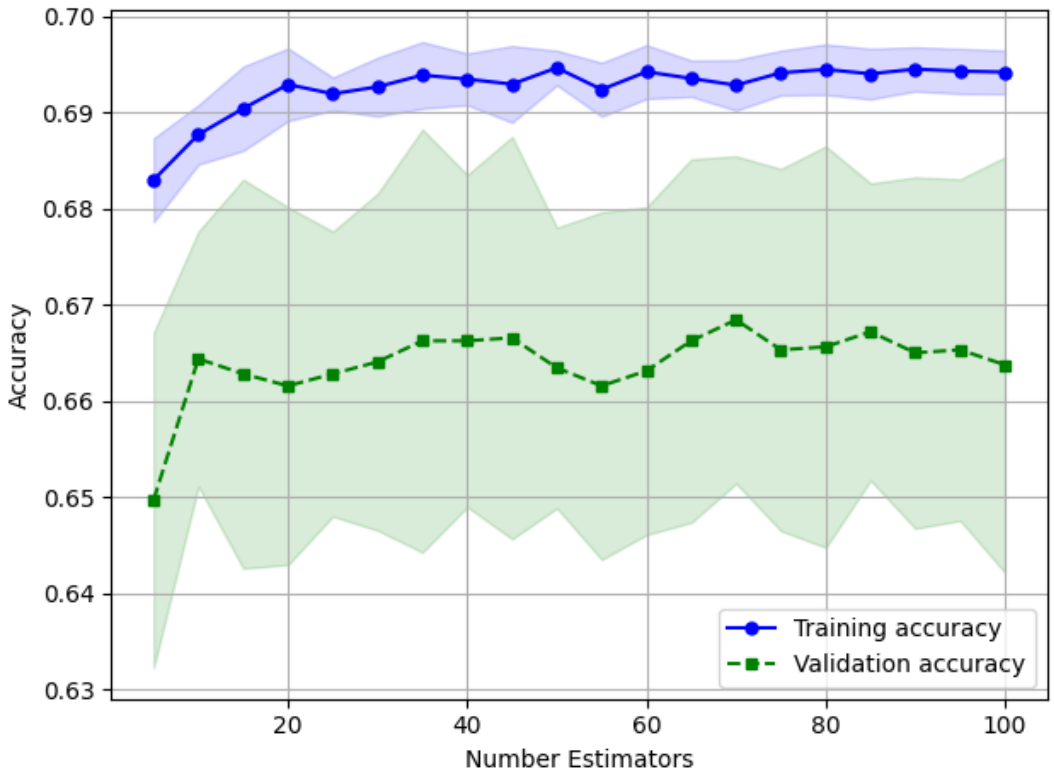

FIG. 7: A validation curve applied to the number of estimators hyperparameter for our random forest classifier model. The plot shows both the training and validation accuracy. There is limited dependence on the hyperparameter on both the training and validation accuracy. The training and validation accuracy are in rough agreement which suggests this hyperparameter will have little effect on the model overfitting and will not influence the model's accuracy.

```
def randomSearch(model, X_train, y_train):
    # Define the hyperparameter search space
    param_dist = {
        'n_estimators': range(10, 200, 10),
        'max_depth': range(1, 6),
    }
    # Create a RandomizedSearchCV object
    random_search = RandomizedSearchCV(
        estimator=model,
        param_distributions=param_dist,
        n_iter=10,  # Number of random combinations to try
        cv=5,       # Number of cross-validation folds
        scoring='accuracy',  # Use accuracy as the evaluation metric
        random_state=1
    )
    random_search.fit(X_train, y_train)
```

```
    print("Best Hyperparameters:", random_search.best_params_)
    print("Best score:", random_search.best_score_)
    return
```

Running this will output the best hyperparameter setting identified by the search, along with the accuracy of the best performing model. In our case the *max_depth* parameter has a value of 5 and the (*n_estimators*) has a value of 20. This model had an accuracy of 67%.

4. Final Model

We have gone through the process of feature extraction from the training dataset, analysed these features and performed feature reduction, analysed candidate models and performed hyperparameter tuning. Our best performant model was subject to overfitting and we noted how this could be addressed via hyperparameter tuning. We can now construct the final model, which will be based on one of our best performant models, namely the random forest classifier.

We will initialise the model with the best performing hyperparameters values.

```
# Split data into training and testing datasets
X_train, X_test, Y_train, Y_test = train_test_split(data, target, test_size
    =0.2)

# Standardise dataset
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)

# Define model
clf = RandomForestClassifier(n_estimators=20, max_depth=5)
clf.fit(X_train_std, Y_train)

# Accuracy Test
Y_pred = clf.predict(X_test_std)
accuracy = accuracy_score(Y_test, Y_pred)
print("Accuracy:", accuracy)
```

Our model has a final accuracy of 68.4%. Our random forest classifier was selected as a trade-off between performance and computational efficiency. The NN performed similarly to the random forest classifier but is more computationally expensive.

5. Conclusions

This research has given an overview of preparing data for supervised machine learning classification tasks. We've extracted relevant technical indicator features from pricing data for the S&P500 index. Feature importance was assessed to determine the most significant features of the dataset. We also gave an overview of principal component analysis as an unsupervised method to reduce the dimensionality of larger datasets.

We introduced several models that can be used for supervised classification and discussed the relative strengths of each model. We trained each model on our dataset and assessed the performance of each model based on its accuracy. We then fine-tuned the best performing model, namely the random forest classifier, and looked at techniques to address the overfitting model.

Our final model had an accuracy of 68%. The model is ready to be implemented in a trading strategy by considering the buy and sell signals generated by the model. Further analysis of the model could consider different performance metrics, such as confusion matrices and the F1 score. Additionally, more complicated models, such as recurrent neural networks, could be considered to model the temporal price and feature changes inherent to asset pricing.

Further analysis would have to be performed to determine whether the model applies to more general stock prices rather than just the S&P500. It is likely that the model will apply well to other indexes, such as the Nasdaq, particularly if the indices are US based. However, further enquiry would be required to verify this.