An Artificial Intelligence approach to Financial Market Prediction

Hanna Carucci Viterbi, Miro Confalone, Cosimo Poccianti, Lorenzo Antolini

May 15, 2022

Contents

Hanna Carucci Viterbi,Miro Confalone, Cosimo Poccianti, Lorenzo Antolini
Euklid Project

1 Introduction

In the world in which we live today, the ability to read and work with data is crucial to improve the decision-making process in complex systems. Thanks to the development of technologies related to AI and machine learning, it is now possible to build increasingly reliable models to improve on pre-existing standards.
In the last decades, the financial sector has increasingly used these techniques to interpret data flows from the stock markets in order to increase the performance of their portfolios.
This project aims to analyse different machine learning models that can be used in stock markets, compare them with each other and find the best one in terms of accuracy.
The work is carried out on several layers.
In the first, the data are collected, manipulated and cleaned to obtain a complete and ordered dataset. Although this first step may seem simple, it must be borne in mind that the performance of a machine learning model is strongly linked to the quality of the data with which it is fed.
The second step of the work consists of creating rules to translate the information of the numerous calculated indicators into useful data for machine learning models. A detailed explanation of the application of the various algorithms used for this purpose can be found in Chapter 3.2.
In the last part of the project, which is also the most significant for answering our initial question, eleven different machine learning models were applied. There are detailed descriptions of the models used including their application to the case study. The results are finally reported in detail in the final part of the paper.

2 Data Collection

To collect the necessary information Data has been gathered from two main sources: Euklid and Yahoo Finance. Data display four financial historical series (IBM, NASDAQ, GOLD, WTI) and for each one of them, the following features representing the starting columns of the datasets:

1. **adjclose**: the daily closing price after adjustments for all applicable splits and dividend distributions.[Euklid]
2. **avg10**: the ten day rolling average of the adjusted clsoe [Euklid]
3. **avg50** : the fifty day rolling average of the adjusted close [Euklid]
4. **volume**: the amount of an asset or security that changes hands over the course of the trading day [YahooFinance]
5. **open**: the opening price of the trading day [YahooFinance]
6. **close**: the closing price of the trading day [YahooFinance]
7. **high**: the highest price of the trading day [YahooFinance]
8. **low**: the lowest price of the trading dat [YahooFinance]

3 Data Manipulation and Cleaning

Before the analysis new variables have been created and the missing values substituted. This process is fundamental in order to create new useful data and to have them ready and cleaned to feed the machine learning models in the next step of the project.

1. The variables have been merged in a unique dataset.
2. For each trading day have been computed the mean respectively of the previous 7,100,200 and 253 day [avg7, avg100, avg200, avg253].
3. For each trading day the difference between the open and close price has been computed [SPD_OPEN/CLOSE].
4. For each trading day the difference between the high and low price has been calculated [SPD_HIGH/LOW].
5. The percentage variation of the adjusted close has been determined [%varAdjClose].
6. Calculated the positive and negative returns for each historical series.
7. Added a binary column where 0 is for negative returns and 1 for positive returns [pos_string].
8. Missing values have been substituted with zero.

3.1 Feature Extraction

The Technical Analysis Library is widely used by trading software developers requiring to perform technical analysis of financial market data. It includes 200 indicators such as ADX, MACD, RSI, Stochastic, Bollinger Bands etc. Momentum Indicators (technical analysis tools used to determine the strength of weakness of a stock's price), Volume indicators and Statistic Functions have been imported for the analysis and different financial indicators have been computed on the times series. For each time series the following indicator have been computed:
MACD daily: classic moving average convergence/divergence. It is a Momentum Indicator.
MACD 9 days: moving average convergence/divergence with signal set on 9 day. Momentum Indicator.
RSI : relative strength index. Momentum Indicator.
OBV: On balance Volume. Volume Indicator.
TSF: Time series forecast. It is a Statistic Function that predict the estimate stock value at 14 day.
WILLR: Williams' %R. Momentum Indicator.
STOCH: Stochastic oscillator slow. Momentum Indicator

3.2 Signal generation

In the dataset there are now the values of the indicators for each daily record of the stocks. After having computed the indicators, the results have been transformed with for loops into value that can be better interpreted by our model.

MACD: A positive MACD value, created when the short-term average is above the longer-term average, is used to signal increasing upward momentum, so it will be a buy signal (1). On the other hand, falling negative MACD values suggest that the downtrend is getting stronger, and that it may not be the best time to buy. In this case the signal is -1.

```
for i in range(len(macd)):
    if macd[i-1] <0 and  macd[i] > 0:
        MACD_DAILY.append(1)
    elif macd[i-1] > 0 and macd[i] < 0:
        MACD_DAILY.append(-1)
    else:
        MACD_DAILY.append(0)
```

RSI: if the value of RSI is higher than 70, the stock is probably 'overbought' in the market, meaning that it is likely that the price of stock lower in the near future. On the other side a RSI lower than 30 is highlighting an "oversold" in the exchange of that specific stock, suggesting that the price is lower and that a bullish push is near.

```
for i in NASDAQ_final["RSI"]:
    if i > 70:
        RSI_NASDAQ.append(-1)
    elif i> 30 and i< 70:
        RSI_NASDAQ.append(0)
    elif i < 30:
        RSI_NASDAQ.append(1)
```

OBV: being a Volume indicator, the OBV only tell if the market is bullish or bearish in that moment so the output of a loop algorithm can be only two. This indicator can reinforce investment decisions by analysing volume trends in the exchange of a particular stock.

```
for i in range(len(obv)):
    if obv[i] > obv_ema[i]:
        strategy = 1
    else:
        strategy = -1
    OBV_strategy.append(strategy)
```

TSF: the time series forecaster is a statistic function that shows the trend of a security's price over a specified time period. The indicator is based upon a regression-based forecast model. The algorithm used for the interpretation of the indicator is as follows:

```
for i in range(len(tsf)):
    if tsf[i] >  (close_IBM[i]*1.05):
        TSF.append(1)
    elif tsf[i] <  (close_IBM[i]*1.05) and tsf[i] >= (close_IBM[i]*0.95):
        TSF.append(0)
    elif tsf[i] <  (close_IBM[i]*0.95):
        TSF.append(-1)
```

The logic is that if the TSF predicted is grater than the actual price the signal generated will be equal to 1. On the other end a TSF minor then the actual stock value is classified as a sell signal so the ouput will be -1. A confidence interval of 0.05 is been created in order to assign 0 value when the TSF is too closer to the actual price.
STOCH: Stochastics is an oscillator used to show when a stock has moved into an overbought or oversold position. It is measured with two line, called K and D. Talib STOCH fuction return two indicator built on these two line as value: STOCHD and STOCHK. The difference between them is that while the K line is build on a time period of 14 days, the D line only takes in consideration the last three trading sessions. The investor needs to watch as the D line and the price of the issue begin to change and move into either the overbought (over the 80 line) or the oversold (under the 20 line) positions. The investor needs to consider selling the stock when the indicator moves above the 80 levels. Conversely, the investor needs to consider buying an issue that is below the 20 line and is starting to move up with increased volume. As

for the algorithms already explained an oversold situation will set the signal on 1 while an overbought trend will be see as a sell signal = -1.

```
IBM_stoch_k = []
IBM_stoch_d = []

for x in stoch_K:
    if x >= 80:
        IBM_stoch_k.append(-1)
    elif x <= 20:
        IBM_stoch_k.append(1)
    else:
        IBM_stoch_k.append(0)

IBM_final["STOCH_K_strategy"] = IBM_stoch_k

for x in stoch_D:
    if x >= 80:
        IBM_stoch_d.append(-1)
    elif x <= 20:
        IBM_stoch_d.append(1)
    else:
        IBM_stoch_d.append(0)

IBM_final["STOCH_D_strategy"] = IBM_stoch_d
```

WILLR: The Williams R is similar to an unsmoothed Stochastic K. The values range from zero to 100, and are charted on an inverted scale, that is, with zero at the top and 100 at the bottom. Values below 20 indicate an overbought condition and a sell signal(-1) is generated when it crosses the 20 line. Values over 80 indicate an oversold condition and a buy signal(1) is generated when it crosses the 80 line.

```
WILLR = []

for i in IBM_final["WILLR"]:
    if i >= -20:
        WILLR.append(-1)
    elif i> -80 and i< -20:
        WILLR.append(0)
    elif i <= -80:
        WILLR.append(1)
```

4 Algorithms

Before computing the algorithms, data has been split through a proportion of 70% for train set and 30% for test set. The target variable is always the *Direction*, precisely the signal of Log Returns (it can be either 1 or -1).

4.1 Simple Moving Averages

According to the Moving Average Rule buy and sell signals generated by two moving averages of the level of the index - a long-period average (avg100) and a short-period average (avg7). In its simplest form, this strategy is expressed as buying (or selling) when the short-period moving averages rises above (or falls below) the long-period moving averages. The idea behind computing moving averages it to smooth out an otherwise volatile series. When the short-period moving

average penetrates the long-period moving average, a trend is considered to be initiated.
The trading rules are

- Go *long* (=+1) when the shorter SMA is above the longer SMA.
- Go *short* (=-1) when the shorter SMA is below the longer SMA.

An example of the position between avg7 and avg100 can be visualized in Figure 1. The trading strategy implicitly derived here only leads to a few trades per se: only when the position value changes (i.e., a crossover happens) does a trade take place. Including opening and closing trades, this would add up to just six trades in total.

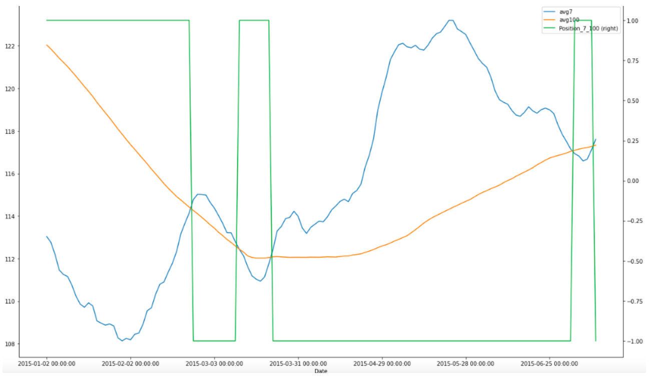


Figure 1: $Position\_7\_100$

The Positions were built both on the train and test set.

```
train['Position_7_10'] = np.where(train["avg7"]>train["avg10"],1,-1)
train['Position_7_50'] = np.where(train["avg7"]>train["avg50"],1,-1)
train['Position_10_50'] = np.where(train["avg10"]>train["avg50"],1,-1)
train['Position_7_200'] = np.where(train["avg7"]>train["avg200"],1,-1)
train['Position_10_200'] = np.where(train["avg10"]>train["avg200"],1,-1)
train['Position_50_200'] = np.where(train["avg50"]>train["avg200"],1,-1)
train['Position_7_100'] = np.where(train["avg7"]>train["avg100"],1,-1)
train['Position_10_100'] = np.where(train["avg10"]>train["avg100"],1,-1)
train['Position_50_100'] = np.where(train["avg50"]>train["avg100"],1,-1)
train['Position_100_200'] =np.where(train["avg100"]>train["avg200"],1,-1)
```

As a result, 10 strategies have been computed. Since all of them could not be put in the ensemble model as it would have resulted in highly correlations, a Random Forest was performed to get a unique position, as the ensemble of all the upper ones seemed to be the best solution. The y was the direction and the X all the positions.

```
X_train_IBM = train[['Position_7_50', 'Position_10_50', 'Position_7_200', 'Position_10_200', '
    Position_50_200','Position_7_100','Position_10_100','Position_50_100','Position_100_200']]
y_train_IBM = train['direction']
X_valid_IBM = test[['Position_7_50', 'Position_10_50', 'Position_7_200', 'Position_10_200', '
    Position_50_200','Position_7_100','Position_10_100','Position_50_100','Position_100_200']]
y_valid_IBM = test['direction']
```

And then the Random Forest was performed:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier

rf_IBM = RandomForestClassifier(n_estimators=100,
                            n_jobs=-1,
                            random_state=42)
```

```

rf_IBM.fit(X_train_IBM, y_train_IBM)
y_pred_IBM = rf_IBM.predict(X_valid_IBM)
y_pred1_IBM = rf_IBM.predict(X_train_IBM)

print('Score Train: ', rf_IBM.score(X_train_IBM, y_train_IBM))
print('Score Test: ', rf_IBM.score(X_valid_IBM, y_valid_IBM))
```

```
Score Train:  0.5365538620511495
Score Test:  0.5204819277108433
```

The same passages were done for NASDAQ, GOLD and WTI. The results for the Accuracy on the Train and Test set were, respectively for
NASDAQ:

```
Score Train:  0.5528287264272798
Score Test:  0.5524096385542169
```

GOLD:

```
Score Train:  0.5584275713516424
Score Test:  0.4908976773383553
```

WTI

```
Score Train:  0.5454787943451587
Score Test:  0.5286069651741293
```

4.2 Frequency Approach

Beyond more sophisticated algorithms and techniques, one might come up with the idea of just implementing a frequency approach to predict directional movements in financial markets. To this end, one might transform the two real-valued features to binary ones and assess the probability of an upward and a downward movement, respectively, from the historical observations of such movements, given the four possible combinations for the two binary features ((0, 0), (0, 1), (1, 0), (1, 1)).

First 2 lags are defined (Log Returns shifted respectively by one and two days)

```
lags = 2
def create_lags(data):
    global cols
    cols = []
    for lag in range(1, lags + 1):
        col = 'lag_{}'.format(lag)
        data[col] = data['Log_Returns'].shift(lag)
        cols.append(col)
```

and then **Bins** defines the signal of the lags (0 if it's negative, 1 if it's positive)

```
def create_bins(data, bins=[0]):
    global cols_bin
    cols_bin = []
    for col in cols:
        col_bin = col + '_bin'
        data[col_bin] = np.digitize(data[col], bins=bins)
        cols_bin.append(col_bin)
```

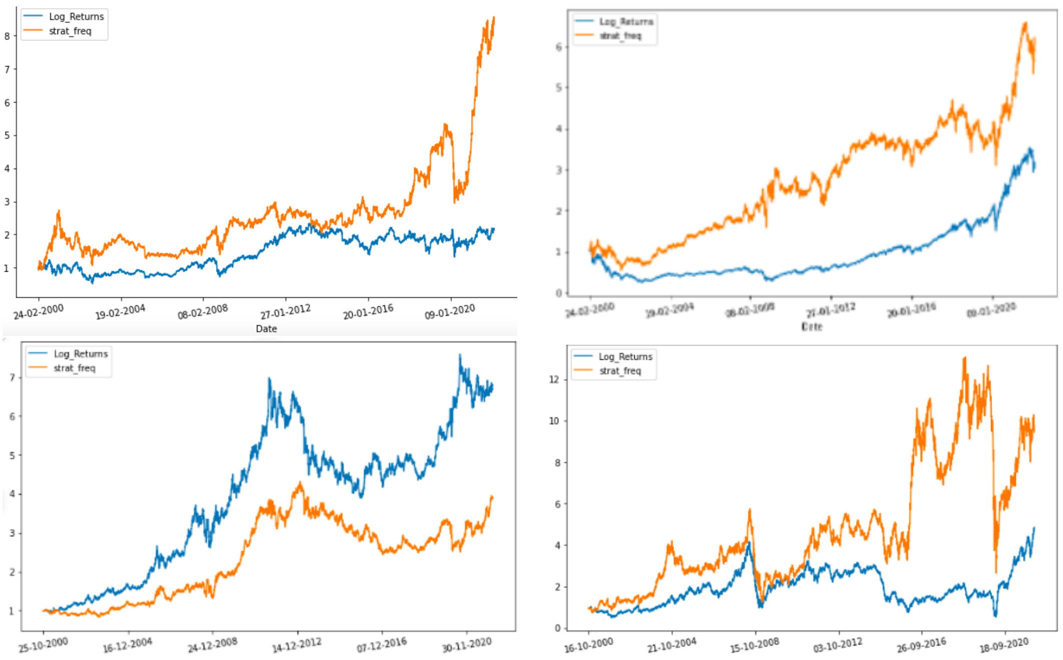
The position is defined as following: if the sum of the two lags is equal to 2, so if bins is 1 both for lag1 and for lag2, then give -1, otherwise 1. It is expressed as selling when the Log returns were positive both yesterday and the day before.

```
IBM['pos_freq'] = np.where(IBM[cols_bin].sum(axis=1) == 2, -1, 1)
```

Finally, the strategy (Orange Line), built as following;

```
IBM['strat_freq'] = IBM['pos_freq'] * IBM['Log_Returns']
```

has been compared to the Actual Log Returns (Blue Line) in a vectorized backtesting:



In the figure it is possible to see that the strategy doesn't perform well for GOLD.

4.3 Bolinger Bands

Another strategy has been built using the Bolinger Bands. An empty column has been created as placeholder for the position signals. Then, the newly created position column was filled - set to sell (-1) when the price hit the upper band, and set to buy (1) when it hits the lower band. Forward, filled the position column to replace the "None" values with the correct long/short positions to represent the "holding" of the position. Finally,the daily market return were calculated and multiplied by the position to determine strategy returns

```
train['PositionBB'] = None
for row in range(len(train)):
    if (train['AdjClose'].iloc[row] > train ['upperband'].iloc[row]) and (train ['AdjClose'].iloc[
    row-1] < train ['upperband'].iloc[row-1]):
        train['PositionBB'].iloc[row] = -1

    if (train ['AdjClose'].iloc[row] < train ['lowerband'].iloc[row]) and (train ['AdjClose'].iloc[
    row-1] > train ['lowerband'].iloc[row-1]):
        train ['PositionBB'].iloc[row] = 1

train['PositionBB'].fillna(method='ffill',inplace=True)
#Calculate the daily market return and multiply that by the position to determine strategy returns
train['Strategy_Return_BB'] = train ['Log_Returns'] * train ['PositionBB'].shift(1)
train['Strategy_Return_BB'].cumsum().plot()
```

4.4 K-Means Clustering

This section applies k-means clustering to financial time series data to automatically come up with clusters that are used to formulate a trading strategy. The idea is that the algorithm identifies two clusters of feature values that predict either an upward movement or a downward movement. However, the resulting trading strategy shows a slight outperformance at the end compared to the benchmark passive investment. It is noteworthy that no guidance (supervision) is given and that the hit ratio— i.e., the number of correct predictions in relationship to all predictions made is less than 50%. The K-Means Clustering was run on the signal of 2 lags (Log Returns shifted respectively by one and two days)

```
lags = 2
def create_lags(data):
  global cols
  cols = []
  for lag in range(1, lags + 1):
    col = 'lag_{}'.format(lag)
    data[col] = data['Log_Returns'].shift(lag)
    cols.append(col)
```

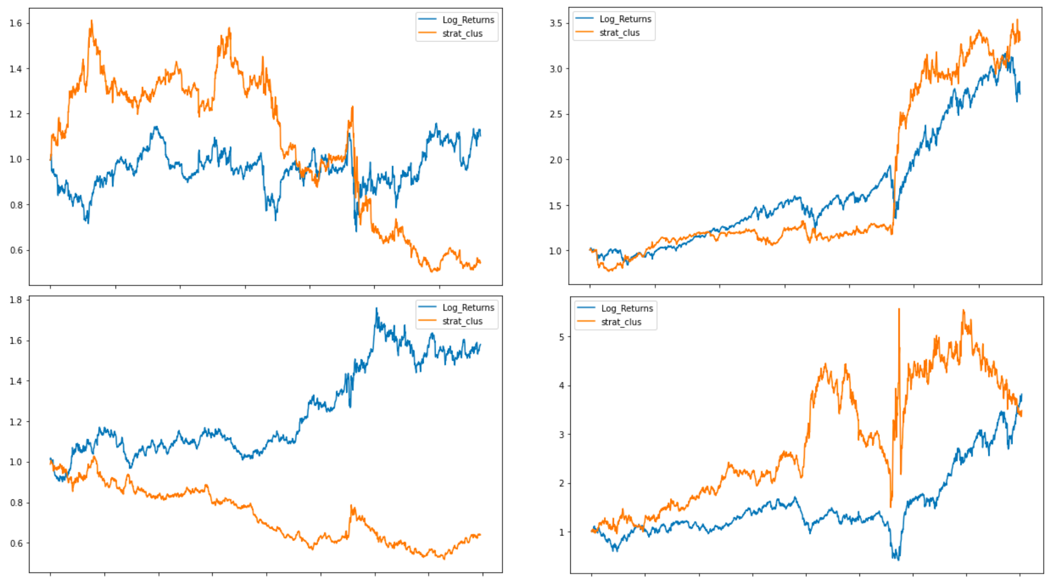
Then the model was implmented with a k-means++ initialization

```
model = KMeans(n_clusters=2, random_state=0)
model.fit(train[cols])
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
n_clusters=2, n_init=10,
random_state=0, tol=0.0001, verbose=0)
```

The number of clusters is equal to 2, sell or buy. From the following graphs it is possible to see the performance of the strategy, which was built as follows

```
test['pos_clus'] = model.predict(test[cols])
test['strat_clus'] = test['pos_clus'] * test['Log_Returns']
```

As we can see from the figures below [top left IBM; thend NASDAQ, bottom left GOLD and then WTI] the strategy for gold did not perform well, while it did for WTI.



4.5 Logistic Regression

The Logistic Regression was used as our baseline model and was implemented on the following three different inputs:

1. **Two Binary Features**: The signal of Two Lags (the signal shifted respectively by one and two days)
2. **Five Binary Features**: In an attempt to improve the strategies' performance, the algorithm was implemented on five lags instead of two
3. **Five Digitized Features**: It uses the first and second moment of the historical log returns to digitize the features data, allowing for more possible feature value combinations

In the following table, it is possible to see the results about the Performances on the test set of the above mentioned input. As we can see, the performances are almost the same for all, and relatively low (0.5 on average)

Logistic Regression Performances			
Security	Accuracy TwoBin	Accuracy FiveBin	Accuracy FiveDig
IBM	0.532	0.518	0.526
NASDAQ	0.566	0.566	0.567
GOLD	0.532	0.528	0.517
WTI	0.517	0.524	0.513

4.6 Naive Bayes Classifier

On the same inputs was implemented also the Naive Bayer Classifier. Also here, performances on the test set are highly similar to those obtained by the implementation of the Logistic Regression: again, on average the metrics is mildly low, around 0.5:

Naive Bayes Classifier Performances			
Security	Accuracy TwoBin	Accuracy FiveBin	Accuracy FiveDig
IBM	0.533	0.519	0.529
NASDAQ	0.567	0.567	0.532
GOLD	0.533	0.529	0.517
WTI	0.517	0.525	0.511

4.7 Support Vector Machine

A support vector machine algorithm was implemented on the same three input classes, using a value of c equal to 1, as well as for logistic regression, chosen using an heuristic approach to find the lowest possible level of misclassification. The results do not show significant improvements compared with the previous algorithms.

Support Vector Machine classifier Performances			
Security	Accuracy TwoBin	Accuracy FiveBin	Accuracy FiveDig
IBM	0.533	0.497	0.497
NASDAQ	0.566	0.541	0.549
GOLD	0.533	0.517	0.529
WTI	0.517	0.520	0.534

In the figure 2 the comparison between logistic regression, Naive Bayes and Support Vector Machine strategy performances is shown, using a vectorised backtesting.

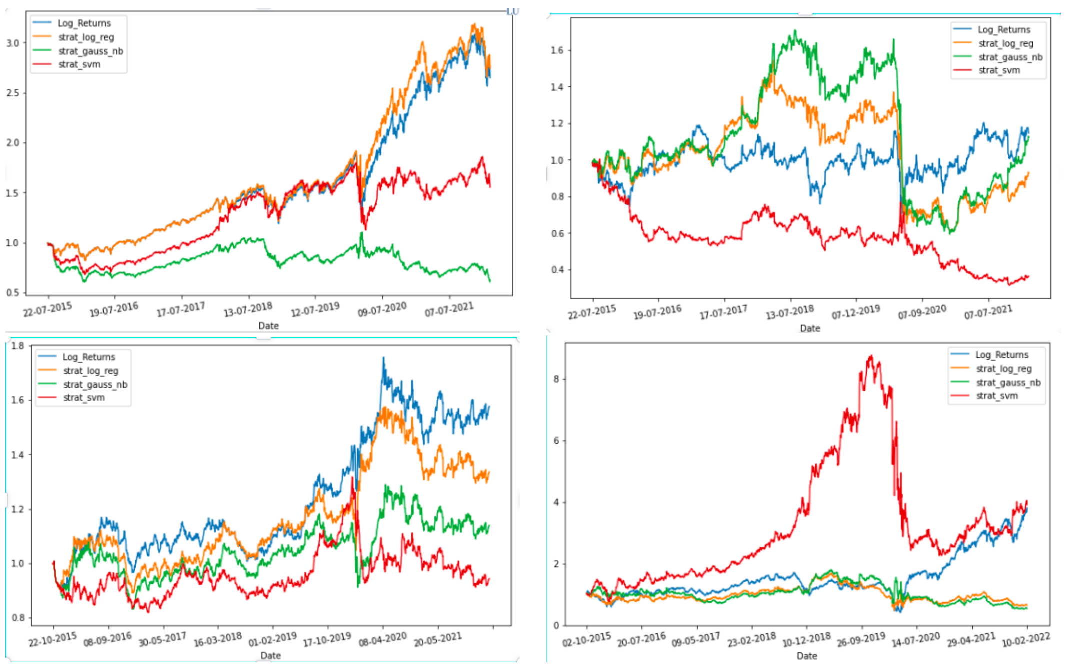


Figure 2: From the Figures Above, the strategies for the test set of each specific model, using Five Digitized Features as inputs, are compared . Starting from the top right, there is IBM, NASDAQ, legt right GOLD and then WTI. While SVM's strategy performs well for WTI (red line), it doesn't for the others. Conversely, the strategy of the Naive Bayer Classifier (green line), and that of the logistic regression (yellow line) perform better for IBM and NASDAQ. For Gold, the best strategy is still the one of the Log Returns.

4.8 Deep Neural Networks

Since the performance obtained with the previous algorithms was not successful, the approach was changed, aiming to utilise the computational power of deep neural networks. Two of them were used to try out two different logics, so that the benefits of both could be reaped. The first neural network was implemented using the Scikit-Learn package, while the second uses TensorFlow. Five lag of the signal - i.e. if the log is positive set +1, if negative -1 - of the logarithmic returns, are used as input for the neural networks.

4.8.1 Scikit-Learn

Using the Scikit-Learn package a deep neural network for classification was created with two layers of 250 neurons each and Limited-memory BFGS as solver.

```
model = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=2 * [250], random_state=1)
```

The improvement in accuracy in predicting the direction of the series is not yet sufficient, as the following table shows, showing the accuracies for the test set.

DNN with Scikit-Learn	
Security	Accuracy on test
IBM	0.498
NASDAQ	0.519
GOLD	0.542
WTI	0.509

4.8.2 TensorFlow

The other deep neural network was implemented using TensorFlow and designed with 3 layers of 500 neurons each.

```
model = tf.contrib.learn.DNNClassifier(hidden_units=3 * [500], n_classes=len(bins) + 1,
    feature_columns=fc)
```

However, even in this case, a good compromise in the optimisation of the model could not be found, and the results are worse than those obtained with Scikit-Learn, as the following table shows.

DNN with Tensorflow	
Security	Accuracy on test
IBM	0.494
NASDAQ	0.432
GOLD	0.466
WTI	0.533

4.9 Extreme learning machine

Driven by the poor previous results and the need to find methods that were not too computationally intensive, it was decided to implement an extreme learning machine, which is a single hidden layer feedforward neural network. ELM assigns random values to the weights between input and hidden layer, frozen during training. The implementation followed the path taken by Simerjot Kaur in *Algorithmic Trading using Sentiment Analysis and Reinforcement Learning*. The advantages in using an Extreme Learning Machine are the higher speed in converging that it has, compared with a usual deep neural network, and also, due to its randomisation component, it is more likely to find the global minimum of the loss function.

4.9.1 Data Preprocessing

The features used as input are, accordingly with the research of Simerjot Kaur:

- Simple Moving Average (10 days)
- Moving Average Convergence and Divergence
- Stochastic K
- Stochastic D
- Relative Strength Index
- Larry William's R

The inputs are then scaled and normalised, and the dataset is split into train e test sets, with the same ratio 70%-30% as before.

4.9.2 The model

In order to choose the right number of neurons to put in the hidden layer, a for loop was created in which the model was trained with several hidden layers, starting with five neurons and adding another five neurons at each iteration. The best model was chosen based on accuracy, being optimal with 100 neurons for IBM, 21 for Nasdaq (on which, due to poor performance, the for loop had a step of 1 and not 5), 90 for WTI and 140 for Gold. This was possible due to the extreme

learning machine's convergence speed; the loops took less than 5 minutes to execute. The activation function is, in all cases, a sigmoid; the output is therefore between 0 and 1. As output signal 1 was assigned for values above the average of the output as a whole, -1 otherwise.
An example for IBM:

```
elm = HPELM(X_train.shape[1], 1,'c', batch = 24)
elm.add_neurons(100, "sigm")
elm.train(X_train,y_train)
```

Hpelm differs from elm just for a computational reason; indeed, it uses all the cores of the machine.

4.9.3 Results

The results, in this case, are significantly better in terms of both accuracy and performance, as the table below and the graphs in Figure 3 show.
The accuracy has significantly increased for all the security, even if it is not yet really good for the Nasdaq index. In the graphs are shown the performances for all the securities in terms of percentage returns, exhibiting a significant improvement if compared with the performance of the algorithms previously computed.

Extreme Learning Machine	
Security	Accuracy on test
IBM	0.837
NASDAQ	0.623
GOLD	0.805
WTI	0.826

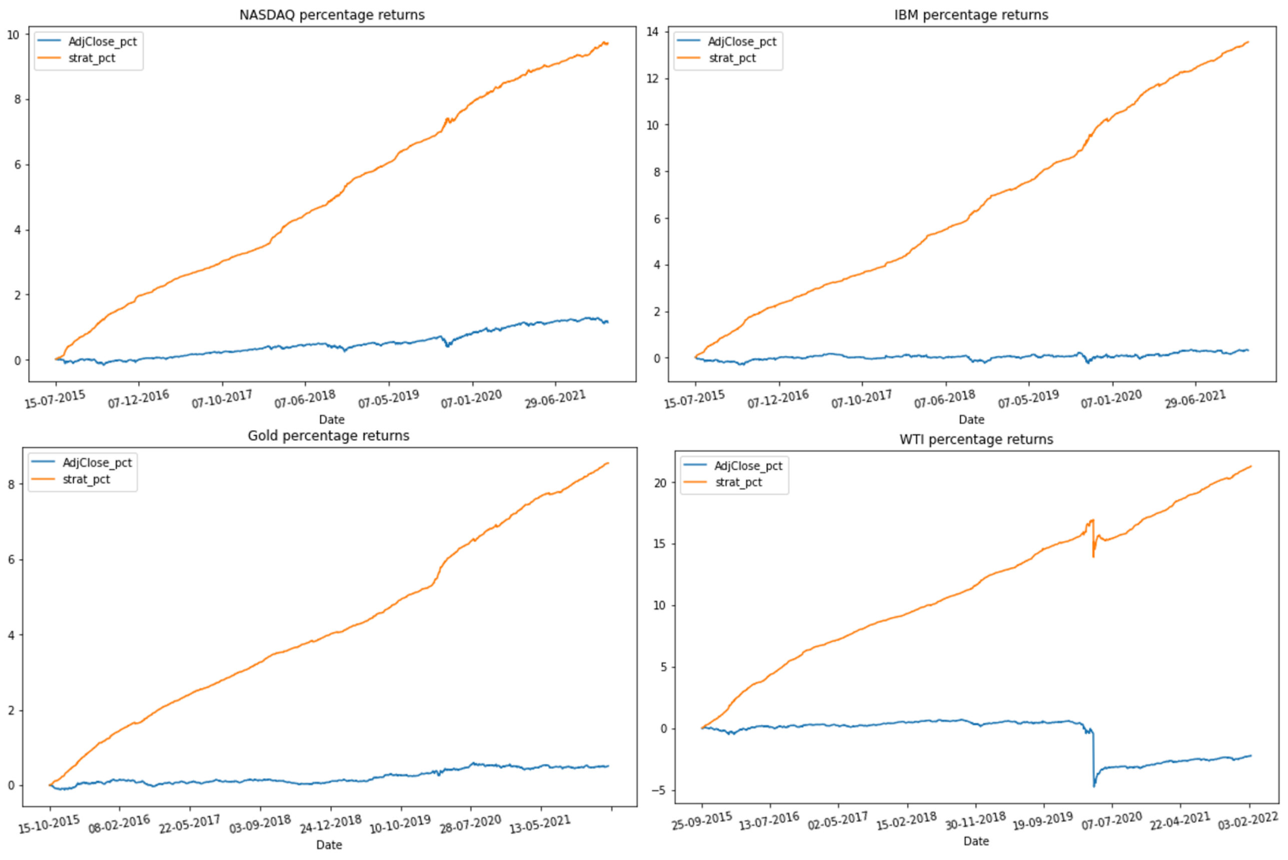


Figure 3: Percentage returns for all the securities, for the five year of the test set. Comparision between passive strategy and ELM strategy.

5 Ensemble Model

In order to collate all the results obtained and to take advantage of their individual positive aspects, all output signals of each strategy were collected, i.e. all times a strategy dictates to go long (+1) or short (-1) for each specific date in the time series, and were put together in a dataset, along with logarithmic returns, the values of the adjusted close, and the direction signal (i.e. +1 when returns are positive, -1 otherwise). The second-level algorithm chosen to process all strategies and, in turn, create a buy or sell signal is a random forest with 100 trees.

5.1 Correlations

Before feeding the random forest, all correlations between the various outputs of each algorithm for every security were assessed. For every pair of strategies with at least 40% correlation, one was removed. As expected, the strategies with the highest correlation are those that were calculated on the same algorithms, albeit with different inputs -i.e. two binary features, five binary features and five digitised features.-

The final strategies removed from the feeding of the random forest are:

- For **IBM**: Two Binary Log reg, Two Binary Gauss_naive Bayes, Five Binary Log Reg, Five Binary SVM, Five digitized Log Reg.

- For **NASDAQ**: Two Binary Log Reg, Two Binary Naive Bayer, Two Binary SVM, Five Binary Log reg, Five Binary Gauss Naive Bayes, Five digitized Gauss Naive Bayes.
- For **Gold**: Two Binary Log Reg, Two Binary Naive Bayer, Two Binary SVM, Five Binary Log reg, Five digitized Gauss Naive Bayes, Five digitized Gauss Naive Bayes.
- For **WTI**: Two Binary Log Reg, Two Binary Naive Bayer, Five Binary Log reg, Five digitized Gauss Naive Bayes, Five digitized Gauss Naive Bayes, Five digitized svm

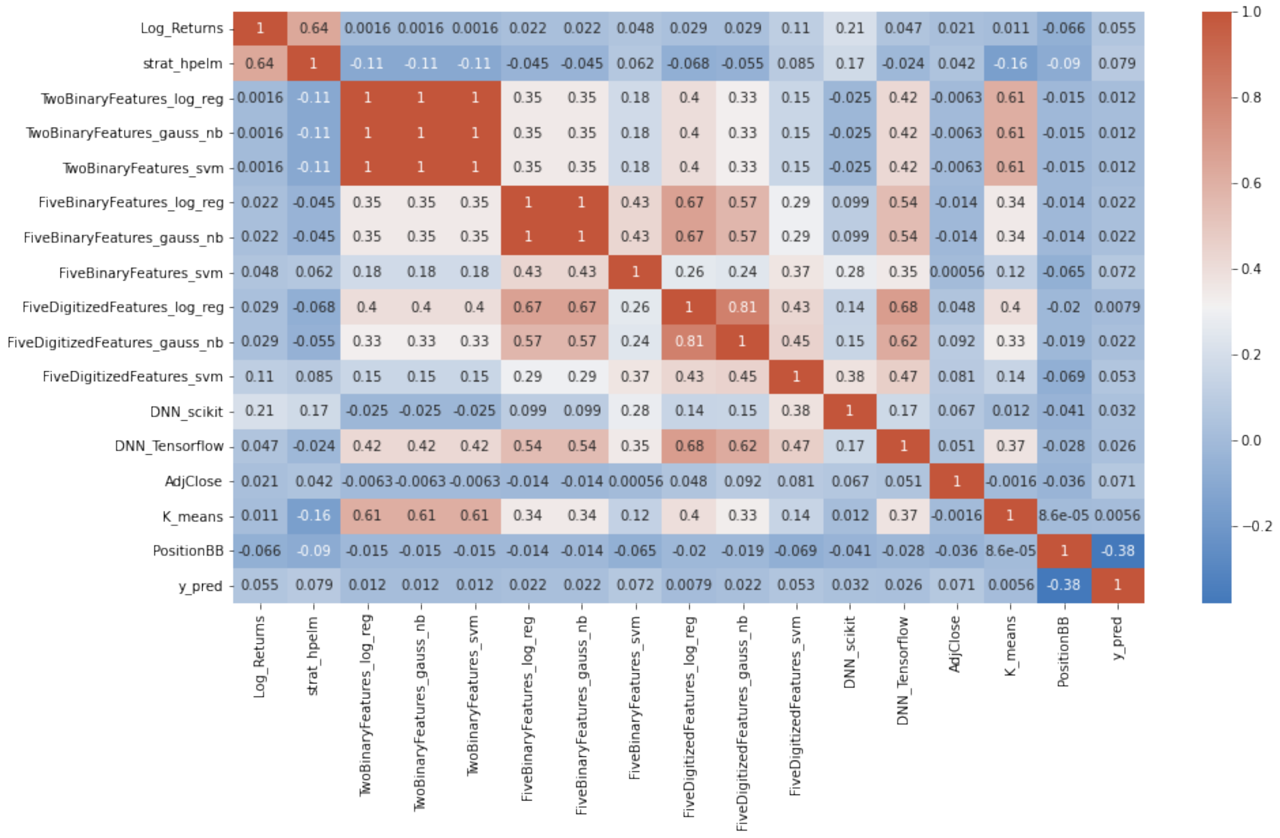


Figure 4: Example of the correlations between the strategies created for WTI

5.2 Results

The results, shown in the table below, are significantly better than the algorithms alone, except for the extreme learning machine, which is still the best in both performance and accuracy. However, the random forest results are very similar.

Ensemble Model	
Security	Accuracy on test
IBM	0.798
NASDAQ	0.596
GOLD	0.796
WTI	0.814

6 LSTM

The final model that was applied to the four time series uses an Artificial Neural Network, in the form of a Recurrent Neural Network with an Long-Short Term Memory Architecture (LSTM), the advantage of using Recurrent Neural Network in Time Series is that this model is able to predict future outcomes based on previous time frames. In this case, to predict the next day's direction, the model considered the previous 14 days.

6.1 Data preprocessing

The data were all normalized and scaled to bring in the range 0-1 and with mean zero and standard deviation 1, then the LSTM model requires that the data have 3D shape (batch_size, num_timesteps, num_features), so the data are reshaped thanks to the NumPy package. Data were split in train and test set with a ratio of 70/30, the data have been also reshaped to accomodate the requirement of Recurrent Neural network that requires that the previous states need to be fed into the model, in this case the data have been reshaped into a NumPy array with shape (days before, number of features, number of target variable). In our case it will be produced a 3d array with one target variables, the number of feauture that will be considered and the previous state based on the previous 14 days. This transformation has been applied both on the train and the test set

6.2 First Execution

Using the Tensorflow library the first four LSTM models were built and run using all the features, this didn't bring any improvement and the performance of the LSTM were bad, with a lot of underfitting and the train accuracy wasn't going above 50 percent even changing parameters, number of layers, optimizer, activation function, etc.

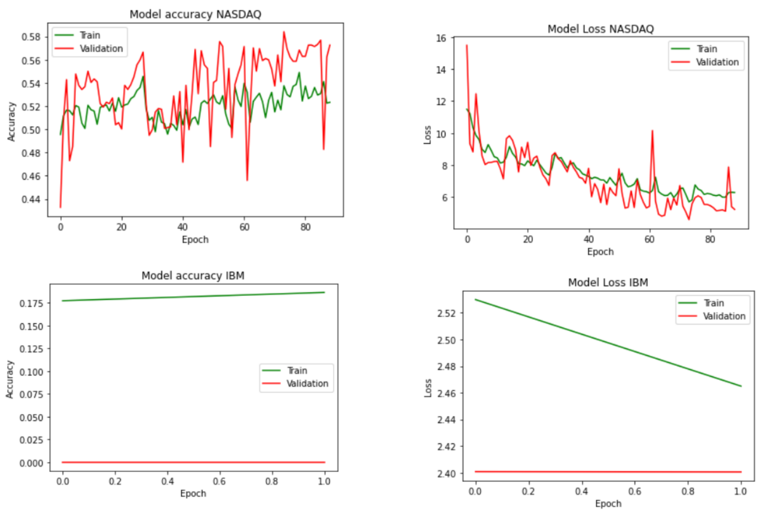


Figure 5: The figures above show the bad performances that our LSTM was achieving, here IBM and NASDAQ are brought as an example, but the same results were obtained for GOLD and WTI

6.3 Model optimization

To optimize our model a Random Forest was applied for features selection, the Adjusted Close variable, the Log Returns and the Returns were removed to prevent leakage from the target variable since they were really similar to it. Depending on the importance given by the Random Forest Classifier to the various variable the most important ones were selected and used to be fed into the LSTM model, but we added back the Adjusted Close variable.

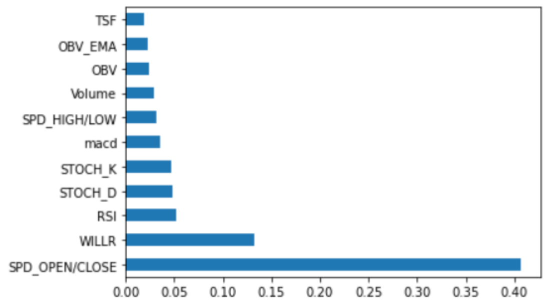


Figure 6: IBM

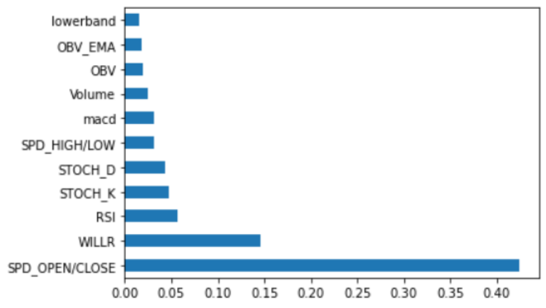


Figure 7: NASDAQ

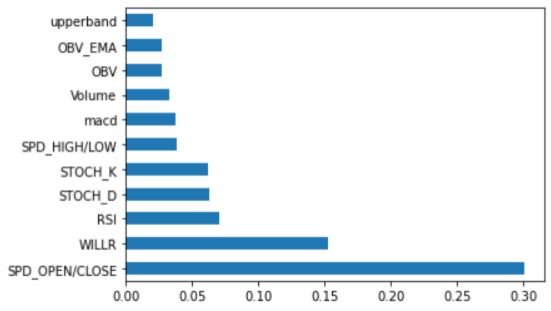


Figure 8: GOLD

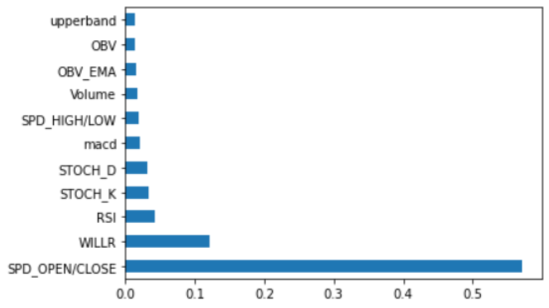


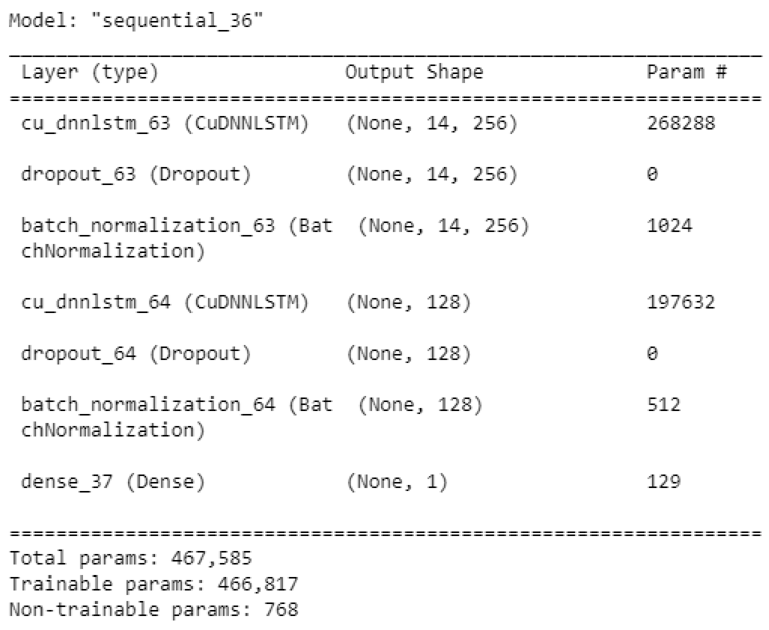
Figure 9: WTI

Figure 10: The most important variables found by the Random Forest Classifier are shown for each security

The spread between Open and Close *SPD_OPEN/CLOSE* was the most important variable in all the 4 securities, same for WILLR and RSI even if the latter doesn't have a great weight in WTI.

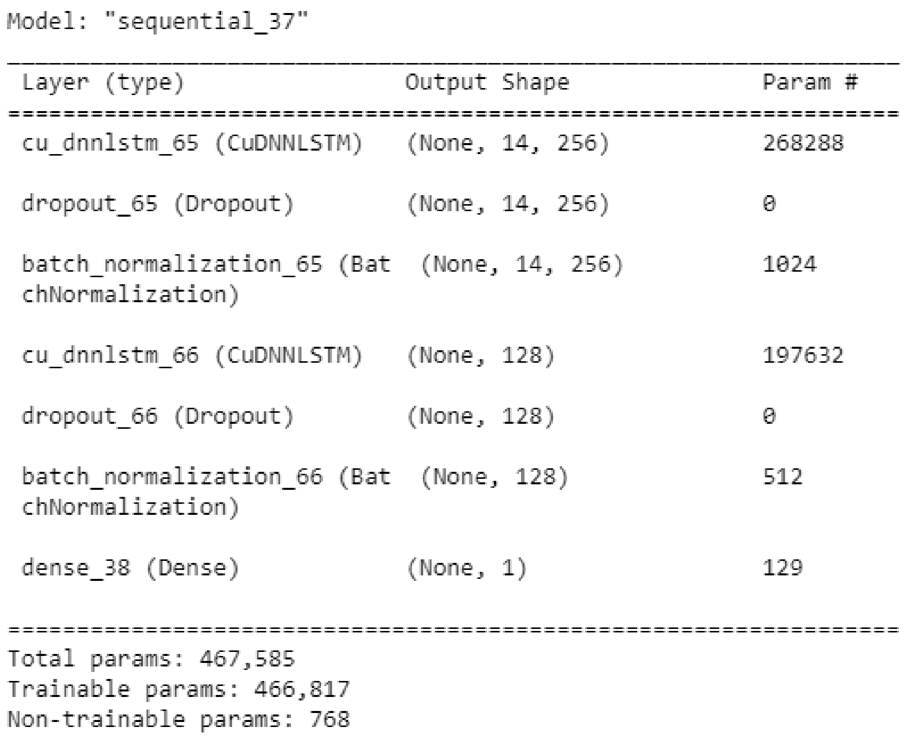
6.4 Final LSTM models

So, after the data were passed through a Random Forest Classifier for feature selection, four new LSTM models were built, one for each security. For IBM, NASDAQ, and GOLD thie following architecture has been used, adjusted close has always been added back:



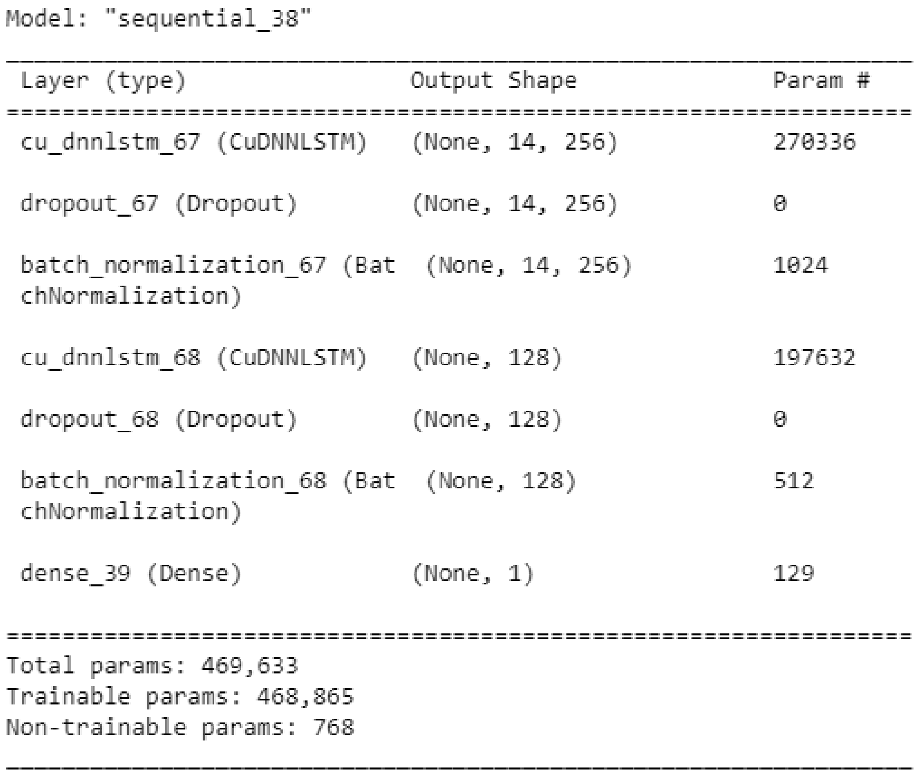
```
Model: "sequential_36"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 cu_dnnlstm_63 (CuDNNLSTM)   (None, 14, 256)           268288

 dropout_63 (Dropout)        (None, 14, 256)           0

 batch_normalization_63 (Bat  (None, 14, 256)          1024
 chNormalization)

 cu_dnnlstm_64 (CuDNNLSTM)   (None, 128)               197632

 dropout_64 (Dropout)        (None, 128)               0

 batch_normalization_64 (Bat  (None, 128)              512
 chNormalization)

 dense_37 (Dense)            (None, 1)                 129

=================================================================
Total params: 467,585
Trainable params: 466,817
Non-trainable params: 768
```

Figure 11: IBM model summary



```
Model: "sequential_37"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 cu_dnnlstm_65 (CuDNNLSTM)   (None, 14, 256)           268288

 dropout_65 (Dropout)        (None, 14, 256)           0

 batch_normalization_65 (Bat  (None, 14, 256)          1024
 chNormalization)

 cu_dnnlstm_66 (CuDNNLSTM)   (None, 128)               197632

 dropout_66 (Dropout)        (None, 128)               0

 batch_normalization_66 (Bat  (None, 128)              512
 chNormalization)

 dense_38 (Dense)            (None, 1)                 129

=================================================================
Total params: 467,585
Trainable params: 466,817
Non-trainable params: 768
```

Figure 12: NASDAQ model summary

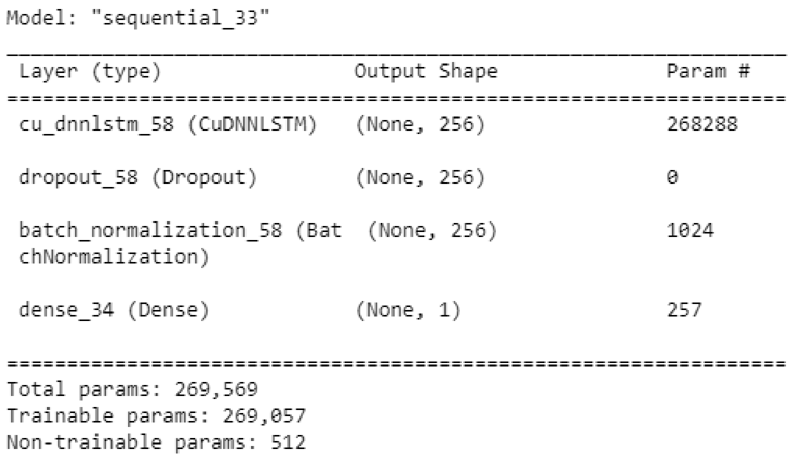


```
Model: "sequential_38"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 cu_dnnlstm_67 (CuDNNLSTM)   (None, 14, 256)           270336

 dropout_67 (Dropout)        (None, 14, 256)           0

 batch_normalization_67 (Bat  (None, 14, 256)          1024
 chNormalization)

 cu_dnnlstm_68 (CuDNNLSTM)   (None, 128)               197632

 dropout_68 (Dropout)        (None, 128)               0

 batch_normalization_68 (Bat  (None, 128)              512
 chNormalization)

 dense_39 (Dense)            (None, 1)                 129

=================================================================
Total params: 469,633
Trainable params: 468,865
Non-trainable params: 768
_________________________________________________________________
```

Figure 13: GOLD model summary

So here we used 2 CuDNNLSTM layers which use the GPU instead of the CPU, the number of neurons were set thanks to a mathematical formula: and has been found that in our case the right number was between 200 and 300 neurons, this for the first hidden layer, while for the second hidden layer the number of neurons were halved. Dropout and Batch Normalization were used to reduce the probability of overfitting, Batch Normalization allows to normalize and standardize the batch at each iteration. A batch size of 24 was used in all the 3 time series, the optimizer used was Adam with a learning rate of 10-□4. The final dense layer used a sigmoid activation function, to used the sigmoid the target variable was first transformed from -1/1 to 0/1.

For WTI the architecture was slightly changed using a unique LSTM layer with a Dense Layer as an output layer, also the learning rate has been changed to 0.01 with respect to 0.04 for the previous models. In all the 4 securities, some regulizers are used, but they differ with respect to each security, regulizers along with dropout and Batch Normalization to stabilize the model and prevent overfitting

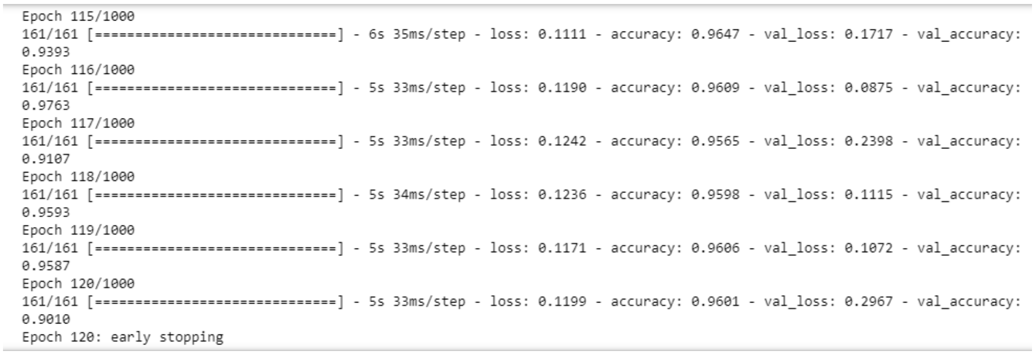


```
Model: "sequential_33"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 cu_dnnlstm_58 (CuDNNLSTM)   (None, 256)               268288

 dropout_58 (Dropout)        (None, 256)               0

 batch_normalization_58 (Bat  (None, 256)              1024
 chNormalization)

 dense_34 (Dense)            (None, 1)                 257

=================================================================
Total params: 269,569
Trainable params: 269,057
Non-trainable params: 512
```

Figure 14: WTI model summary

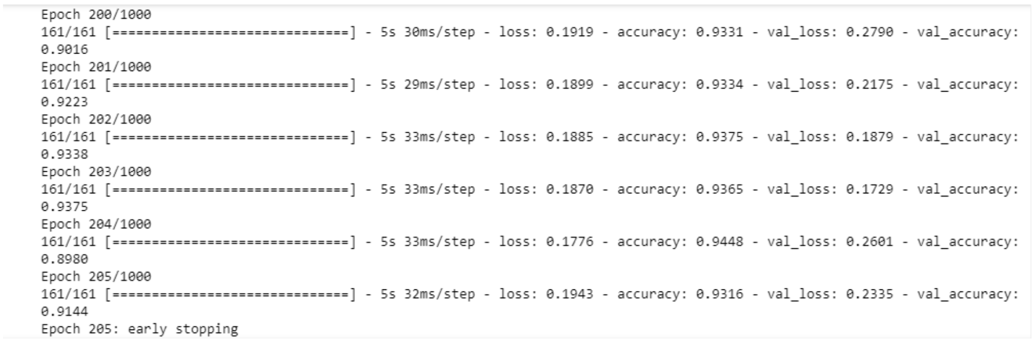
6.5 Model Training and Validation

The model was trained and validated on the test set, the number of epochs was set to 1000 but an Early Stopping callback was used with a patience of 30, meaning that if the validation loss didn't reach a new minimum in 30 steps the model training would stop.



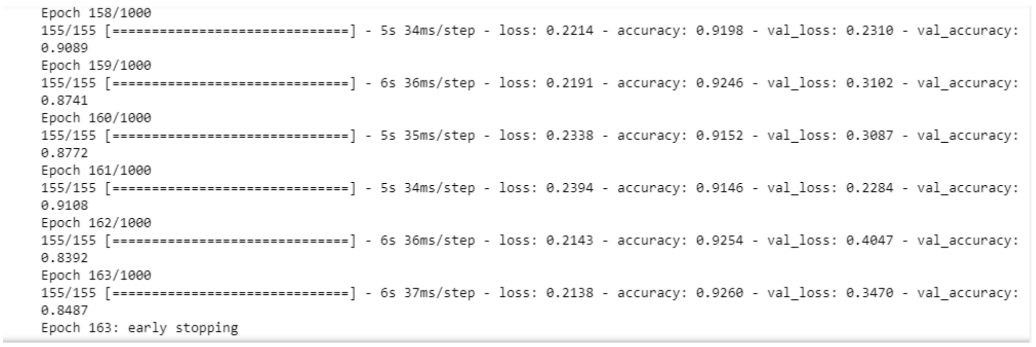
```
Epoch 115/1000
161/161 [==============================] - 6s 35ms/step - loss: 0.1111 - accuracy: 0.9647 - val_loss: 0.1717 - val_accuracy:
0.9393
Epoch 116/1000
161/161 [==============================] - 5s 33ms/step - loss: 0.1190 - accuracy: 0.9609 - val_loss: 0.0875 - val_accuracy:
0.9763
Epoch 117/1000
161/161 [==============================] - 5s 33ms/step - loss: 0.1242 - accuracy: 0.9565 - val_loss: 0.2398 - val_accuracy:
0.9107
Epoch 118/1000
161/161 [==============================] - 5s 34ms/step - loss: 0.1236 - accuracy: 0.9598 - val_loss: 0.1115 - val_accuracy:
0.9593
Epoch 119/1000
161/161 [==============================] - 5s 33ms/step - loss: 0.1171 - accuracy: 0.9606 - val_loss: 0.1072 - val_accuracy:
0.9587
Epoch 120/1000
161/161 [==============================] - 5s 33ms/step - loss: 0.1199 - accuracy: 0.9601 - val_loss: 0.2967 - val_accuracy:
0.9010
Epoch 120: early stopping
```

Figure 15: IBM training and validation



```
Epoch 200/1000
161/161 [==============================] - 5s 30ms/step - loss: 0.1919 - accuracy: 0.9331 - val_loss: 0.2790 - val_accuracy:
0.9016
Epoch 201/1000
161/161 [==============================] - 5s 29ms/step - loss: 0.1899 - accuracy: 0.9334 - val_loss: 0.2175 - val_accuracy:
0.9223
Epoch 202/1000
161/161 [==============================] - 5s 33ms/step - loss: 0.1885 - accuracy: 0.9375 - val_loss: 0.1879 - val_accuracy:
0.9338
Epoch 203/1000
161/161 [==============================] - 5s 33ms/step - loss: 0.1870 - accuracy: 0.9365 - val_loss: 0.1729 - val_accuracy:
0.9375
Epoch 204/1000
161/161 [==============================] - 5s 33ms/step - loss: 0.1776 - accuracy: 0.9448 - val_loss: 0.2601 - val_accuracy:
0.8980
Epoch 205/1000
161/161 [==============================] - 5s 32ms/step - loss: 0.1943 - accuracy: 0.9316 - val_loss: 0.2335 - val_accuracy:
0.9144
Epoch 205: early stopping
```

Figure 16: NASDAQ training and validation



```
Epoch 158/1000
155/155 [==============================] - 5s 34ms/step - loss: 0.2214 - accuracy: 0.9198 - val_loss: 0.2310 - val_accuracy:
0.9089
Epoch 159/1000
155/155 [==============================] - 6s 36ms/step - loss: 0.2191 - accuracy: 0.9246 - val_loss: 0.3102 - val_accuracy:
0.8741
Epoch 160/1000
155/155 [==============================] - 5s 35ms/step - loss: 0.2338 - accuracy: 0.9152 - val_loss: 0.3087 - val_accuracy:
0.8772
Epoch 161/1000
155/155 [==============================] - 5s 34ms/step - loss: 0.2394 - accuracy: 0.9146 - val_loss: 0.2284 - val_accuracy:
0.9108
Epoch 162/1000
155/155 [==============================] - 6s 36ms/step - loss: 0.2143 - accuracy: 0.9254 - val_loss: 0.4047 - val_accuracy:
0.8392
Epoch 163/1000
155/155 [==============================] - 6s 37ms/step - loss: 0.2138 - accuracy: 0.9260 - val_loss: 0.3470 - val_accuracy:
0.8487
Epoch 163: early stopping
```

Figure 17: GOLD training and validation

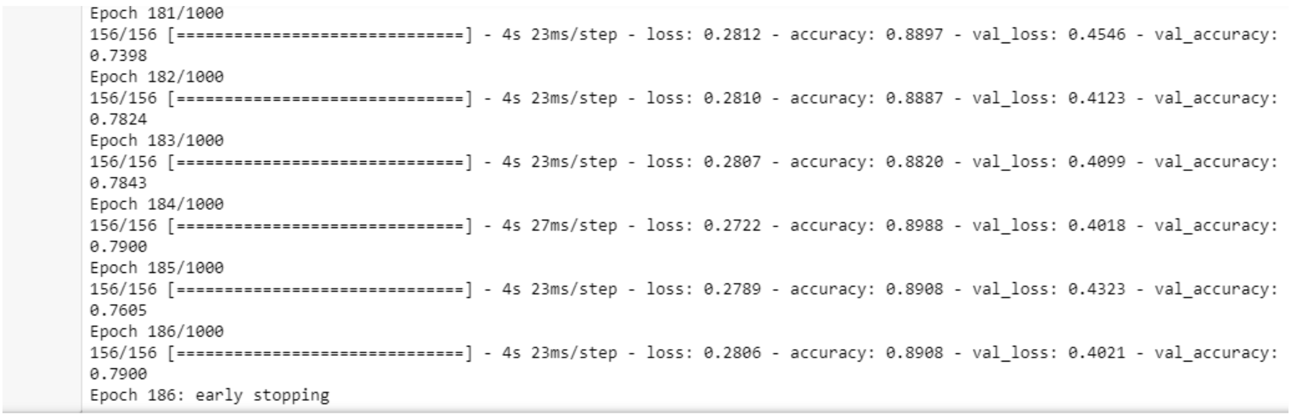

```
Epoch 181/1000
156/156 [==============================] - 4s 23ms/step - loss: 0.2812 - accuracy: 0.8897 - val_loss: 0.4546 - val_accuracy:
0.7398
Epoch 182/1000
156/156 [==============================] - 4s 23ms/step - loss: 0.2810 - accuracy: 0.8887 - val_loss: 0.4123 - val_accuracy:
0.7824
Epoch 183/1000
156/156 [==============================] - 4s 23ms/step - loss: 0.2807 - accuracy: 0.8820 - val_loss: 0.4099 - val_accuracy:
0.7843
Epoch 184/1000
156/156 [==============================] - 4s 27ms/step - loss: 0.2722 - accuracy: 0.8988 - val_loss: 0.4018 - val_accuracy:
0.7900
Epoch 185/1000
156/156 [==============================] - 4s 23ms/step - loss: 0.2789 - accuracy: 0.8908 - val_loss: 0.4323 - val_accuracy:
0.7605
Epoch 186/1000
156/156 [==============================] - 4s 23ms/step - loss: 0.2806 - accuracy: 0.8908 - val_loss: 0.4021 - val_accuracy:
0.7900
Epoch 186: early stopping
```

Figure 18: WTI training and validation

6.6 Model Performances

Model Performance				
Security	Train Performance	Test Performance	Loss on Train	Loss on Test
IBM	96.01	90.10	0.119	0.2967
NASDAQ	93.16	91.44	0.194	0.2335
GOLD	92.6	84.87	0.2138	0.347
WTI	93.16	79.00	0.194	0.2335

It can be said that the model doesn't seem to suffer from overfitting and that the performance on the accuracy reflects the fact that the model learned step by step and improved after each epoch, the only security that present some anomaly is the WTI, since the accuracy on the test peak at 90 percent, but then starts to fall, but again it doesn't seem to overfit since the test error doesn't grow when the training error is diminishing, the test loss stays the same.

7 Conclusions

Most models have an accuracy score averaging around 50% on the testset. Nothing quite impressive, taking into account that the accuracy of these models remains low on all four assets with no out liners, thus making their implementation as a strategy unprofitable for all the securities. Three of the models analysed, however, stand out as performing far better than the others. They are respectively: the Extreme learning machine, the Ensamble model and the LSTM.

LSTM turns out to be the most accurate model in interpreting the market. It can predict whether the next day's price will be higher or lower with a test accuracy of 91.44% when applied to the NASDAQ index. Its implementation as a strategy on GOLD and IBM is also the best found for these two securities, with a test score of 84.87% and 90.10%, respectively.
By contrast, the extreme learning machine is the best model to be applied on the WTI, with an accuracy on the test of 82.6%. The performance of the model is also good on IBM and GOLD with accuracy on both assets above 80%.

The Ensamble model with the Random forest classifier also performs very well on various assets. The accuracy results on the test are in fact 79% for IBM and GOLD. Slightly better on WTI where it scores 81.4%.

In conclusion, it can be said that the implementation of machine learning in finance is a winning move, especially if the application of different strategies is accompanied by a thorough study and analysis of the strengths and weaknesses of the various models on different stocks.

References

[1] Yves Hilpisch, *Artificial Intelligence in Finance*, October 2020

[2] Simerjot Kaur, *Algorithmic Trading using Sentiment Analysis and Reinforcement Learning*

[3] Yves Hilpisch, *Python For Finance*, December 2014

[4] Jian Wang, Siyuan Lu, Shui-Hua Wang, Yu-Dong Zhang, *A review on extreme Machine Learning*, 2021

[5] William Brock, *Simple Technical Trading Rules and the Stochastic Properties of Stock Returns*

8 Appendix

8.1 Peer-review

The group worked really well together, equally distributing the work load among the members. Although each member of the group really focused on the whole project, each giving different and useful ideas, some tasks were specifically divided between members. In particular, Lorenzo worked on the Data PreProcessing and Feature Extraction part, Hanna and Cosimo focused on some Algorithms (e.g. HPELM) and Miro worked on the LSTM.

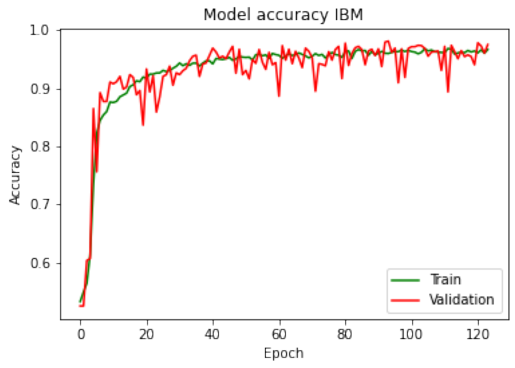


Figure 19: IBM accuracy

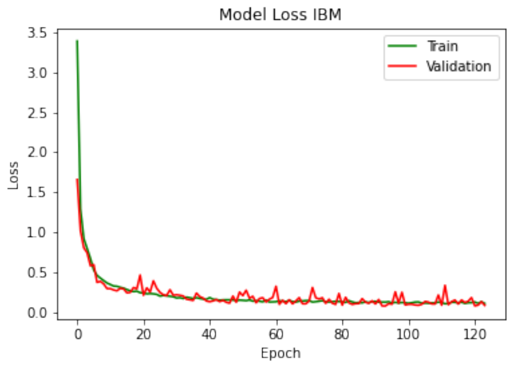


Figure 20: IBM loss

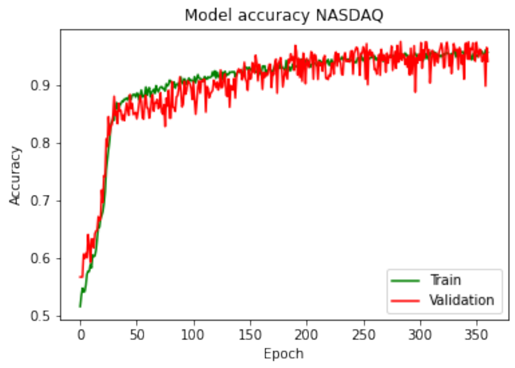


Figure 21: NASDAQ accuracy

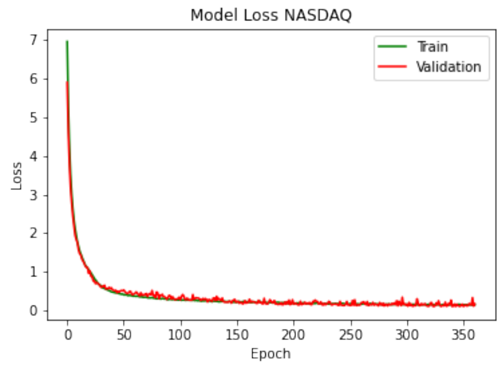


Figure 22: NASDAQ loss

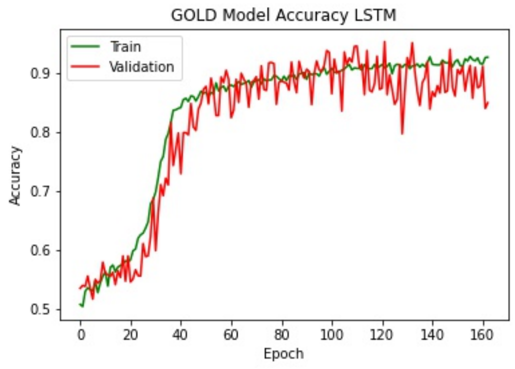


Figure 23: GOLD accuracy

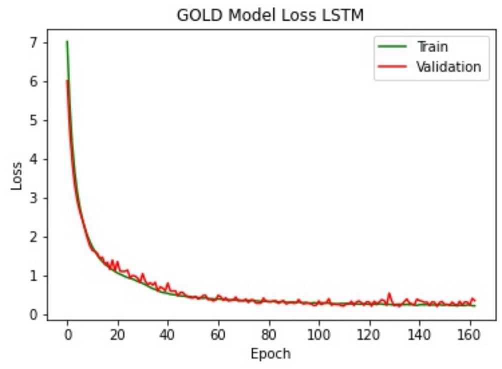
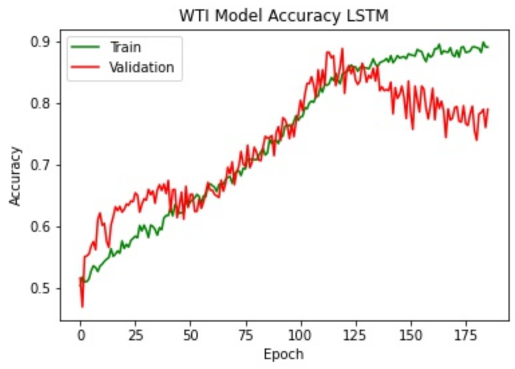


Figure 24: GOLD loss



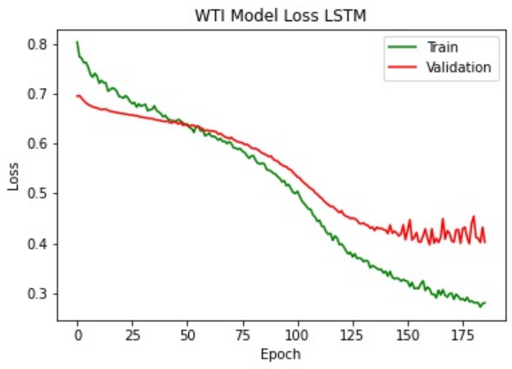


Figure 25: WTI accuracy

Figure 26: WTI Loss