

# Machine Learning Engineer Nanodegree

## Capstone Project

---

Saeed Rahman  
October 8th, 2017

## I. Definition

---

### Project Overview

The stock market is a big, confusing beast. With multiple indexes, stock types and categories, it can be overwhelming for the eager investor. But by understanding how the different markets interact with each other, the bigger picture can become much clearer. Observing the relationship between commodities, bond prices, stocks and currencies can lead to smarter trades.

Multiple research ([1], [2], [3], [4], [5]) have shown that the stock markets around the globe has complex interactions between them. Most evident of them is the influence of developed markets (especially US) on others (including emerging markets). Or putting it in layman's words, if the S&P500 Index (US Equity market index) has a rough day (high drawdown), then that would be reflected in the other global Equity markets.

The efficient market hypothesis (EMH) is an investment theory that states it is impossible to "beat the market" because stock market efficiency causes existing share prices to always incorporate and reflect all relevant information. According to EMH this would mean that the market prices at any point of the day would have already accounted for the news, fundamentals of the economy and the sentiments of the market participants at that point.

### Problem Statement

The goal of this project is to apply machine learning principles to predict the Opening (9:30 AM EST) price of the S&P500 Index using information from the prices of other Global markets equity indexes like FTSE100 (United Kingdom), Nikkei225 (Japan), etc. Knowing the Opening price can help us make better pre-market trade decisions.

*Note: A stock index or stock market index is a measurement of the value of a section of the stock market. It is computed from the prices of selected stocks (typically a weighted average). It is a tool used by investors and financial managers to describe the market, and to compare the return on specific investments.*

The solution to the proposed problem is to create custom feature for each index that would incorporate that date's price information till 9:30 AM EST (US market opening time). Then use those features to predict the opening of the S&P500 Index using a regression model.

## Metrics

The project is framed as a regression problem and therefore  $R^2$  (coefficient of determination) and MSE (Mean Squared Error) would be the choices of evaluation metrics available.

$$R^2 = 1 - \frac{SSE}{SST}$$

where SSE is the sum of squared error (residuals or deviations from the regression line) and SST is the sum of squared deviations from the dependent's Y mean

$MSE = \frac{SSE}{n-m}$ , where n is the sample size and m is the number of parameters in the model

$R^2$  is a standardized measure of degree of predictiveness, or fit, in the sample whereas MSE is the estimate of variance of residuals, or non-fit, in the population. The two measures are clearly related and for our purpose, since we are interested in finding the best model that fits our data, we will choose  $R^2$  as our evaluation metric.

$R^2$  can take on any value between 0 and 1, with a value closer to 1 indicating that a greater proportion of variance is accounted for by the model. For example, an R-square value of 0.8234 means that the fit explains 82.34% of the total variation in the data about the average.

Predicting the direction is as important as predicting the exact opening price of the market since there are strategies (involving equities and derivative securities) lets you bet on the market direction. Therefore, results can also be framed into a binary classification problem since the market can Open only higher or lower than the previous Close and hence a fuller analysis of a confusion matrix including false negatives and false positives, and consideration of metrics such as precision, recall, and the combined F1 measure can be obtained.

To measure performance of the models, we will use a novel validation technique called the k-fold sequential cross validation (k-SCV). In this method, we train on all days up to a specific day and test for the next k days. The direct k-fold cross validation method is not applicable in this context as the stock data is a time series unlike other scenarios where the data is available as a set. Therefore, it is meaningless to analyze past stock data after training on future values.

## II. Analysis

---

### Datasets and Inputs

The two main datasets that will be used in the project are:

- 1) The S&P500 daily price. (Open, High, Low, Close)
- 2) 15 Minute intraday prices of Global Equity Indexes (Eastern Standard Time format).
  - FTSE100 – UK
  - DAX – Germany
  - CAC40 – France
  - IBEX35 – Spain
  - SHSZ300 – China
  - OMX30 – Sweden
  - Nikkei225 – Japan
  - SMI – Swiss
  - HSI – Hong Kong

**Features:** Price of the day (till 9:30 AM EST) of major world indexes and previous day return of S&P500

**Target Variable:** Opening Price of S&P500

**Range:** 3/1/2012 to 9/20/2017

**Number of days of data:** 1176

**Source:** Bloomberg

# Data Exploration and Visualization

SPX

	Open	High	Low	Close
Date				
2017-09-19	2506.29	2507.84	2503.19	2506.65
2017-09-18	2502.51	2508.32	2499.92	2503.87
2017-09-15	2495.67	2500.23	2493.16	2500.23
2017-09-14	2494.56	2498.43	2491.35	2495.62
2017-09-13	2493.89	2498.37	2492.14	2498.37

S&P500 Daily Chart



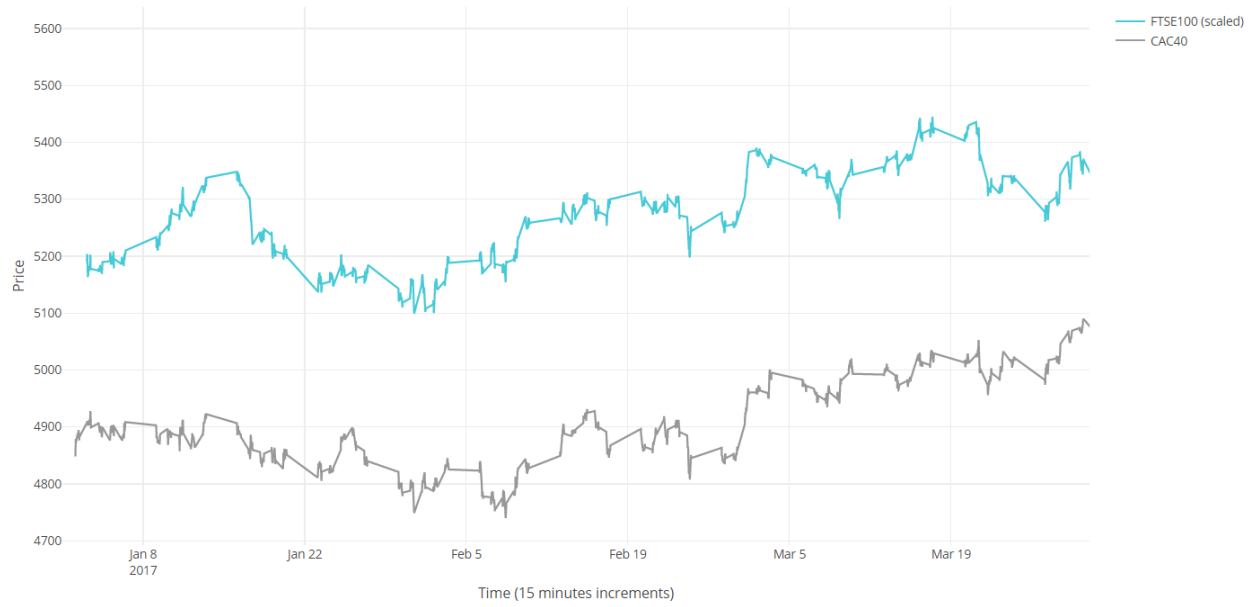
FTSE

	Date	Last Price
0	2017-09-21 11:30:00	7263.90
1	2017-09-21 11:15:00	7266.87
2	2017-09-21 11:00:00	7267.28
3	2017-09-21 10:45:00	7262.96
4	2017-09-21 10:30:00	7268.27

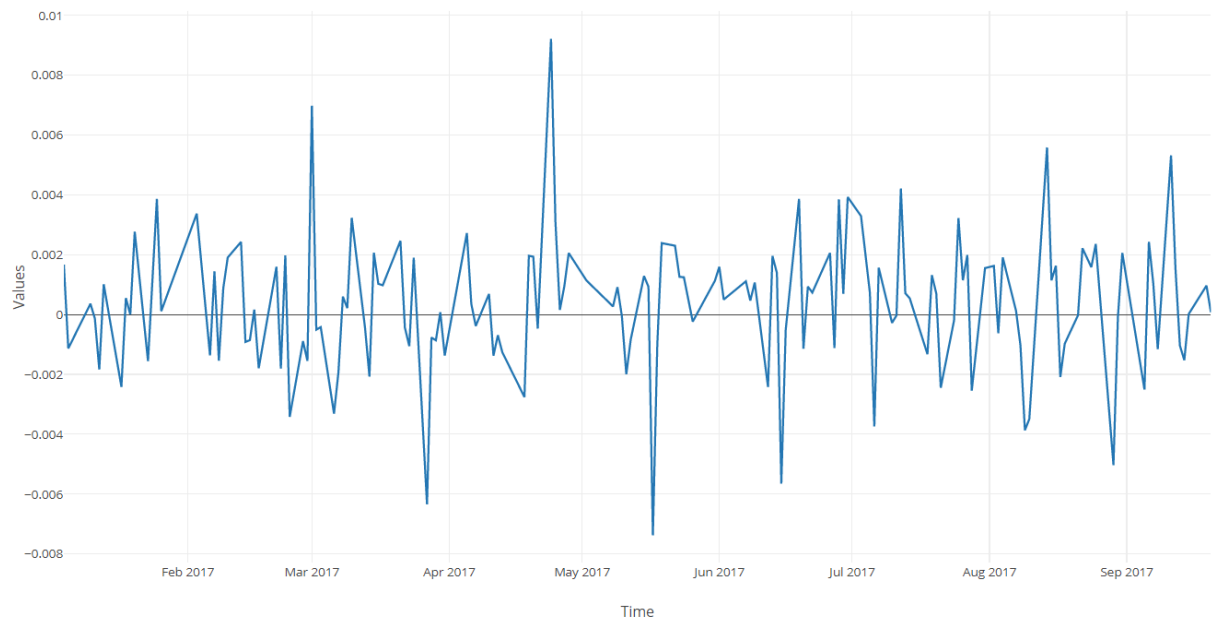
CAC

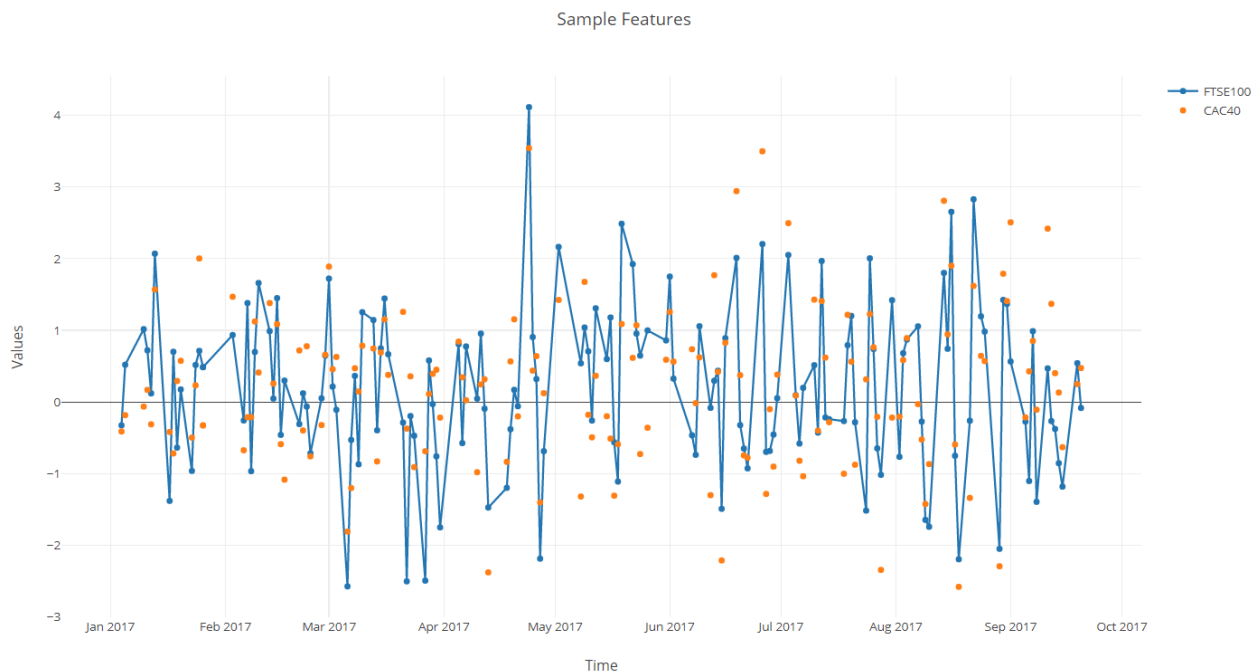
	Date	Last Price
0	2017-09-21 11:30:00	5267.29
1	2017-09-21 11:15:00	5272.70
2	2017-09-21 11:00:00	5269.41
3	2017-09-21 10:45:00	5270.58
4	2017-09-21 10:30:00	5270.84

15 Minute Price Data of UK and French Equity Indices



Target Variables





### *Test for stationarity*

Many statistical tests, deep down in the fine print of their assumptions, require that the data being tested are stationary. Also, if you naively use certain statistics on a non-stationary data set, you will get garbage results.

Stationarity is a one type of dependence structure. Suppose we have a data  $X_1, \dots, X_n$ . The most basic assumption is that  $X_i$  are independent, i.e. we have a sample. The independence is a nice property, since using it we can derive a lot of useful results. The problem is that sometimes (or frequently, depending on the view) this property does not hold.

Now independence is a unique property, two random variables can be independent only in one way, but they can be dependent in various ways. So, stationarity is one way of modeling the dependence structure. It turns out that a lot of nice results which holds for independent random variables (law of large numbers, central limit theorem to name a few) hold for stationary random variables (we should strictly say sequences). And of course, it turns out that a lot of data can be considered stationary, so the concept of stationarity is very important in modeling non-independent data.

Now again the same story holds as with independence and dependence. Stationarity is defined uniquely, i.e. data is either stationary or not, so there is only way for data to be stationary, but lots of ways for it to be non-stationary. Again, it turns out that a lot of

data becomes stationary after certain transformation. ARIMA model is one model for non-stationarity. It assumes that the data becomes stationary after differencing.

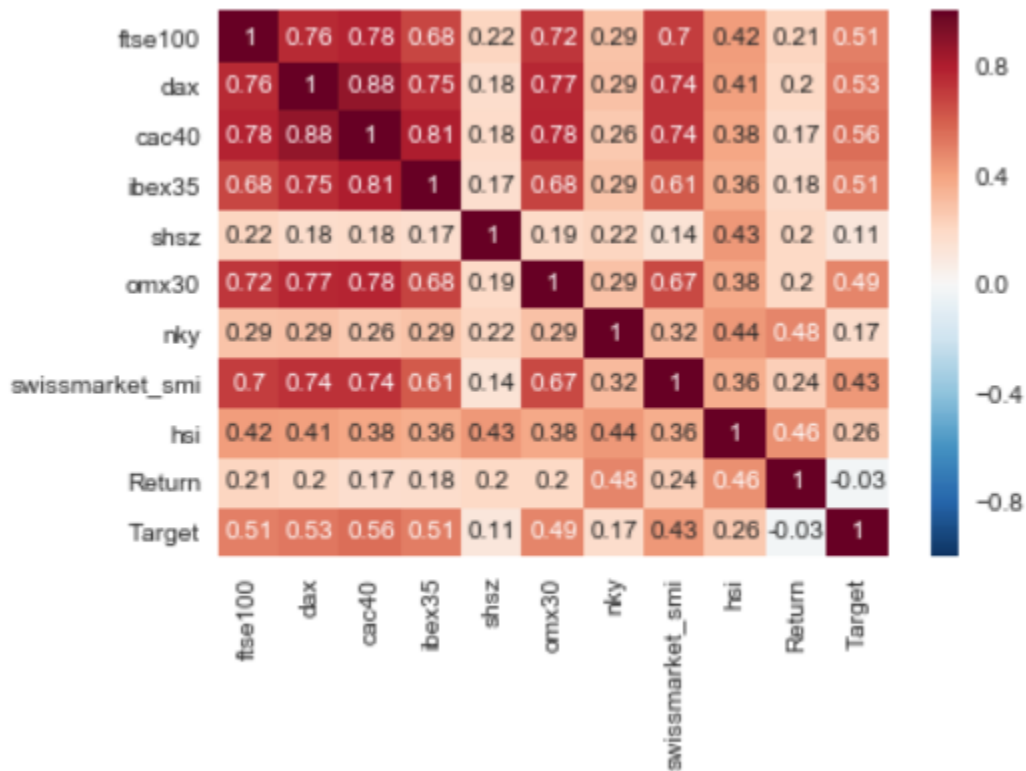
In the regression context, the stationarity is important since the same results which apply for independent data holds if the data is stationary.

Therefore, to avoid chances of spurious regression in our problem we will test for stationarity for our features and target using Augmented Dickey-Fuller test.

```
The series ftse100 is likely stationary.  
The series dax is likely stationary.  
The series cac40 is likely stationary.  
The series ibex35 is likely stationary.  
The series shsz is likely stationary.  
The series omx30 is likely stationary.  
The series nky is likely stationary.  
The series swissmarket_smi is likely stationary.  
The series hsi is likely stationary.  
The series Return is likely stationary.  
The series Target is likely stationary.
```



## Correlation



Looking at the correlation heat map of the feature variables and target, we can make a few observations:

- The 'Target' (S&P500 Open) and the 'Return' (S&P500 yesterday's return) are uncorrelated.
- The 'Target' and the other global equity indices also seems to have no direct linear relation between them.
- We can observe the high correlation between the European equity indices and the low correlations between the Asian and European markets.

## Algorithms and Techniques

### Base Model -LSTM

Time series prediction problems are a difficult type of predictive modeling problem. Unlike regression predictive modeling, time series also adds the complexity of a sequence dependence among the input variables.

A powerful type of neural network designed to handle sequence dependence is called recurrent neural networks. The Long Short-Term Memory network or LSTM network is a

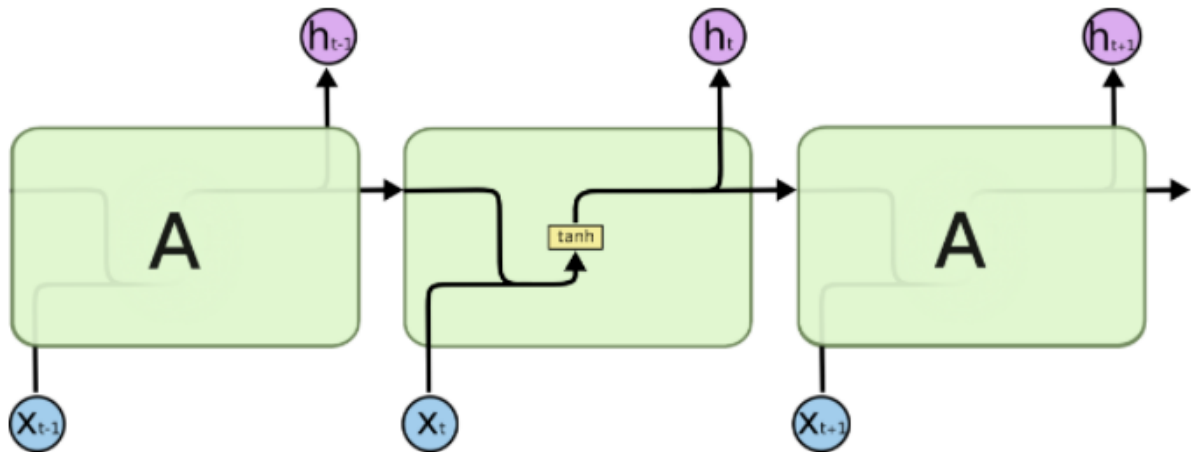
type of recurrent neural network used in deep learning because they provide long lasting memory of previous data and very large architectures can be successfully trained.

### Recurrent neural network

The basic idea is that recurrent networks have loops. These loops allow the network to use information from previous passes, which acts as memory. The length of this memory depends on many factors but it is important to note that it is not indefinite. You can think of the memory as degrading, with older information being less and less usable.

For example, let's say we just want the network to do one thing: Remember whether an input from earlier was 1, or 0. It's not difficult to imagine a network which just continually passes the 1 around in a loop. However, every time you send in a 0, the output going into the loop gets a little lower (This is a simplification, but displays the idea). After some number of passes the loop input will be arbitrarily low, making the output of the network 0. As you are aware, the vanishing gradient problem is essentially the same, but in reverse.

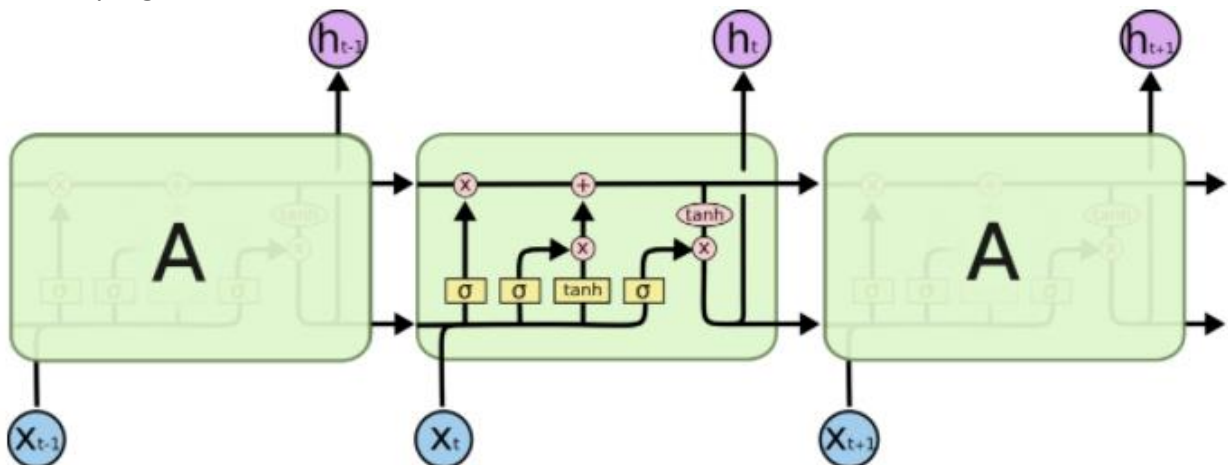
Why not just use a window of time inputs? As plausible solution that might strike is to use a window of time inputs: A sliding window of past inputs being provided as current inputs. That's is not a bad idea, but consider this: While the RNN may have eroded over time, you will always lose the entirety of your time information after you window ends. And while you would remove the vanishing gradient problem, you would have to increase the number of weights of your network by several times. Having to train all those additional weights will hurt you just as badly as (if not worse than) vanishing gradient.



The repeating module in a standard RNN contains a single layer.

LSTM.

You can think of LSTM as a special type of RNN. The difference is that LSTM is able to actively maintain self-connecting loops without them degrading. This is accomplished through a somewhat fancy activation, involving an additional "memory" output for the self-looping



The repeating module in an LSTM contains four interacting layers.

connection. The network must then be trained to select what data gets put onto this bus. By training the network to explicit select what to remember, we don't have to worry about new inputs destroying important information, and the vanishing gradient doesn't affect the information we decided to keep.

### *Comparison Models*

To compare the effectiveness of the LSTM model, we will also run 3 other machine learning regression models on the same problem including 2 different ensemble models.

- 1) Support Vector Regressor
- 2) Random Forest Regressor (Bagging Ensemble model)
- 3) Gradient Boosting Regressor (Boosting Ensemble model)

Support Vector Machines (SVMs) work by finding a maximum margin hyperplane which maximizes separation between data classes. The training samples that are closest to this hyperplane are called the support vectors. Support Vector Regression builds upon this and use kernels to specify a margin of tolerance (epsilon) to provide real number prediction output.

Random Forest (RF) regressors are estimators that fit many classifying decision trees on various subsamples of the dataset. Then they use averaging to improve the prediction accuracy and control overfitting to the data. "RandomForestRegressor": the scikit-learn version of this model combines classifiers by averaging their probabilistic prediction, instead of letting each classifier vote for a single class.

Gradient Boosting Regressor is described as a stage-wise additive model. This is because one new weak learner is added at a time and existing weak learners in the model are frozen and left unchanged, therefore it builds one weak learner at a time, where each new learner helps to correct errors made by previously trained learner. With each learner added, the model becomes even more expressive. It's powerful in capturing complex relationships in data automatically.

### *Optimization:*

The un-tuned versions of these models will use the default parameters provided in the Keras and Scikitlearn packages. Parameter tuning was then used to better the model's performance. Scikit-learn's GridSearchCV module performed the tuning by taking a dictionary of parameters and their values, and testing all their possible combinations. The parameters that produced the best results on the training data was then passed to final model for prediction.

To measure accuracy and hyper-parameter optimization we will use a novel validation technique called the k-fold sequential cross validation (k-SCV). In this method, we train on all days up to a specific day and test for the next k days. The direct k-fold cross validation method is not applicable in this context as the stock data is a time series

unlike other scenarios where the data is available as a set. Therefore, it is meaningless to analyze past stock data after training on future values.

## **Benchmark**

This is not the kind of project for which one can find an obvious benchmark model to assess performance. The main reason being the prediction from this project can only be used in pre-market session and we don't have access to that data and there are lot of caveats trading in the pre-market from liquidity problems, slippage and other transparency issues. So, our best option would be a low but data-specific benchmark is simply whether it is more accurate than a random (naïve) choice.

The desired outcome of my project will be whether my model can correctly identify the Opening direction of the S&P500 – beating a random guess by say 10% would in my view be a reasonable outcome given the degree of complexity of the problem.

Using a naïve predictor, we can be right or wrong 50% of the time and assume that our profit or loss every time we predict the direction is +1\$ and -1\$ (1:1 Risk Reward Ratio). Then our naïve predictor's expected return would be 0\$.

Whereas if we have a slight edge and say we were able to predict the market directions correctly 60% of the time, our expected returns with the above Risk: Reward ratio can be \$ 0.2.

So, choosing a margin of performance (10% in this case) is based on the risk profile or risk appetite of the user of our model. And this risk profile again depends on numerous other factors like type of security being traded, type of trading strategies, etc.. which is clearly out of scope of this project.

To make an apple to apple comparison of my Deep learning model, I would also be modelling 3 other regression algorithms. Two of them being ensemble learners (both boosting and bagging). This would serve as a better benchmark than a naïve predictor. Moreover, the results from all the 4 models can justify my hypothesis; whether there are interactions between equity market around the world and alpha (profit) can be obtained from modelling these relations.

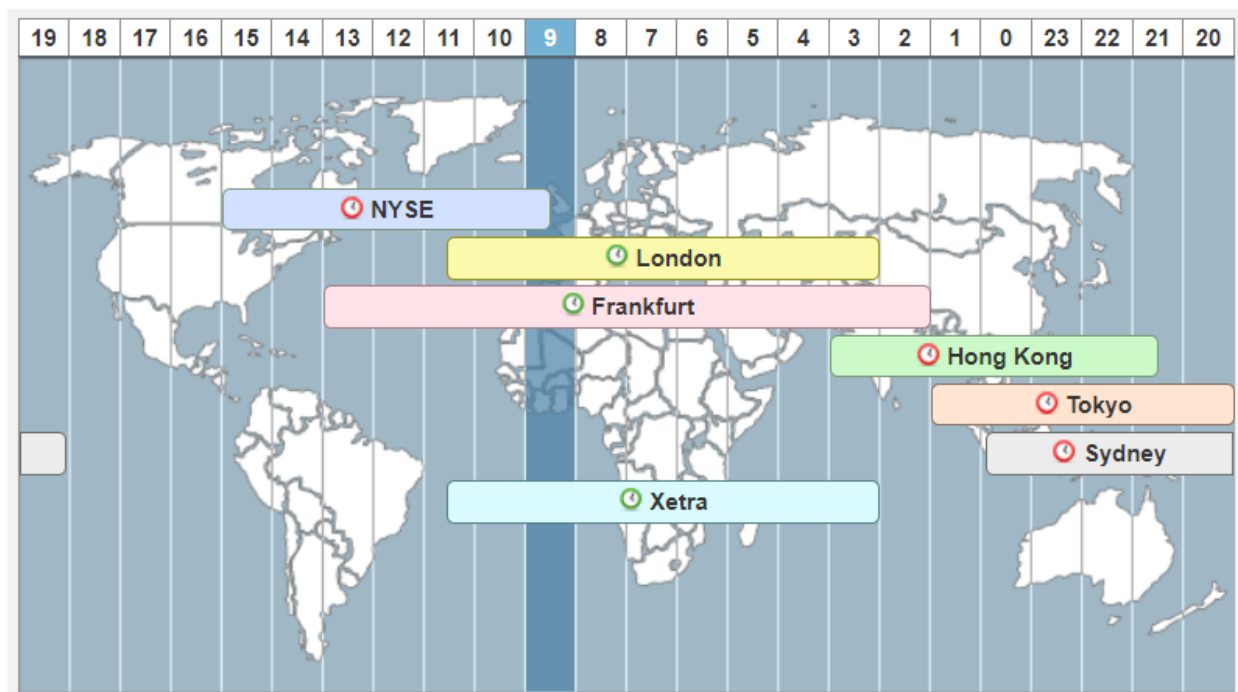
## **III. Methodology**

### **Data Preprocessing**

### Data Cleaning:

1. The first step in my workflow would be to pre-process the data. The various steps involved would
  - Cluster the data-points of each index with only 15 minutes difference to segregate the intraday prices into each trading day.
  - While doing so, care should be given as certain Asian markets have 1 hours breaks and similar intraday breaks, so that is taken into consideration.
  - Also, errors and missing data can generate multiple set of data-points for each date, so those errors and discontinuity should be dealt with.

### Feature Engineering:



Let's take an example of United Kingdom Equity Index FTSE100 at 10/4/2017

#### United Kingdom

Trading Hours	08:00 AM – 04:30 PM (+1 Hours GMT)	03:00 AM – 11:30 AM ET
---------------	------------------------------------	------------------------

To create the custom feature, let's define a few variables

Open: Opening price (3:00 AM EST)

Current: Price at 9:30 AM EST

High: Highest price between 3:00 AM EST and 9:30 AM EST

Low: Lowest price between 3:00 AM EST and 9:30 AM EST

$$\text{Custom Feature} = \frac{\text{Current} - \text{Open}}{\text{High} - \text{Low}}$$

The benefits of the above feature are:

- It has encapsulated the price information from the opening of the UK market till the opening of the US market.
- Has captured the volatility of the price during that range
- It is analogous to Sharpe-ratio, which is a performance metrics used to measure the mean-variance ratio of a portfolio.
- It is normalized
- It is stationary and this would help avoid spurious regression.

The above process is repeated for other indices under study.

The Target or the variable that we are predicting can be taken as modified returns of the S&P500 price.

$$\text{Target} = \frac{\text{Opening}(\text{today}) - \text{Close}(\text{yesterday})}{\text{Close}(\text{yesterday})}$$

We would also use the previous day returns of S&P500 as one of the feature along with the above set of custom features.

Final Features-Target Data Frame

Date	ftse100	dax	cac40	ibex35	shsz	omx30	nky	swissmarket_smi	hsi	Return	Target
2013-05-14	0.760000	0.442971	0.143895	-0.201361	-0.814936	-0.270364	-0.343043	0.571928	-0.004122	0.000043	-0.000012
2013-05-15	0.292977	-0.488402	-0.291413	0.408870	0.936595	0.518349	4.403992	2.050314	0.005401	0.010142	-0.000733
2013-05-16	-0.382740	-0.163429	-0.800112	-0.658849	0.714419	0.596471	-0.257411	-0.642584	0.003431	0.005114	-0.000428
2013-05-21	0.921569	0.358141	0.158036	-0.600000	0.307873	0.255066	0.209040	0.631762	-0.003917	-0.000708	-0.000054
2013-05-22	1.012598	0.859114	-0.334518	-0.324747	0.190078	0.601179	1.533346	1.248234	-0.538089	0.001722	0.000138

*Feature Scaling:*

*Train - Test - Validation dataset*

**Training set:** A set of examples used for learning, that is to fit the parameters [i.e., weights] of the model.

**Validation set:** A set of examples used to tune the parameters [i.e., architecture, not weights] of a classifier, for example to choose the number of hidden units in a neural network.

**Test set:** A set of examples used only to assess the performance [generalization] of a fully specified classifier.

In our problem:

We have totally 1175 days of data.

Where we would use 1000 days for training and 175 for testing.

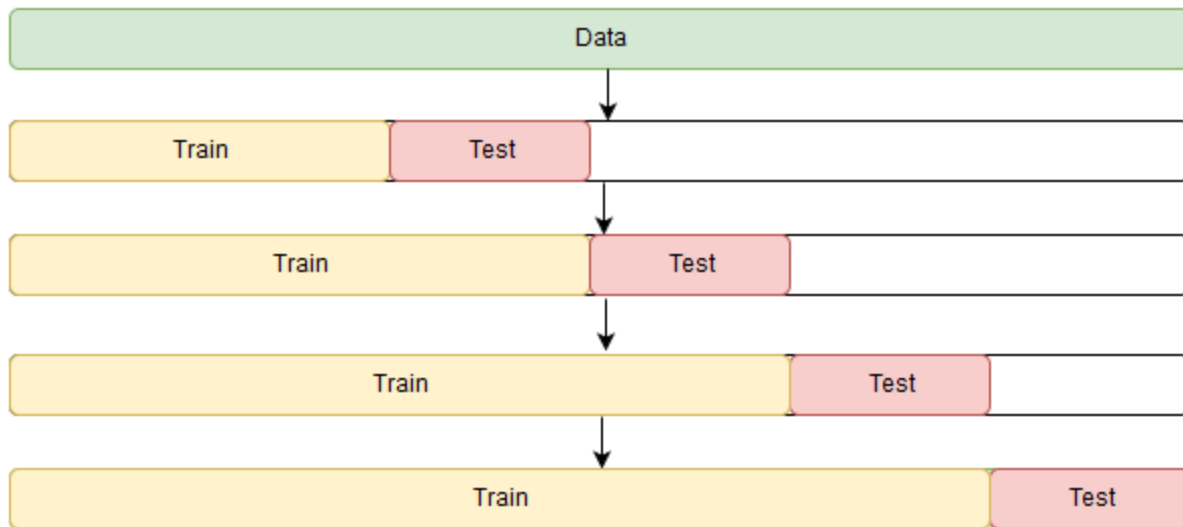
Since our LSTM model has a 60-day look back window. This comes down to 940 and 115 days.

We would be using the same train-test split for both the Deep Neural Network model and the comparison models.

For hyper-parameter tuning and reducing overfitting, we need to use a validation set that is derived from the training set for validating the performance of the model after iterating through each grids of available parameter space. A common method employed for this training-validation split is the K-fold cross validation. But since we are dealing with time-series data, normal K-fold method would introduce look-ahead bias into our modeling. Therefore we use a sequential cross validation (roll-forward CV) technique implemented using the 'sklearn.model\_selection.TimeSeriesSplit'

Start with a small subset of data for training purpose, forecast for the later data points and then checking the accuracy for the forecasted data points. The same forecasted data points are then included as part of the next training dataset and subsequent data points are forecasted.





## Implementation

### Look Back Window:

For our problem, we are using a 60 day look back window for training the modeling. Which means that we would feeding the past 60 days of data for predicting tomorrows price, this approximation was arbitrary based on the intuition that shocks in the market would have a 3 month affects. This combined with the long-memory of the network would give an ideal result.

### Input Shape:

LSTM networks in Keras package requires a specific input structure for our features.

**[number of training points, window size, number of features]**

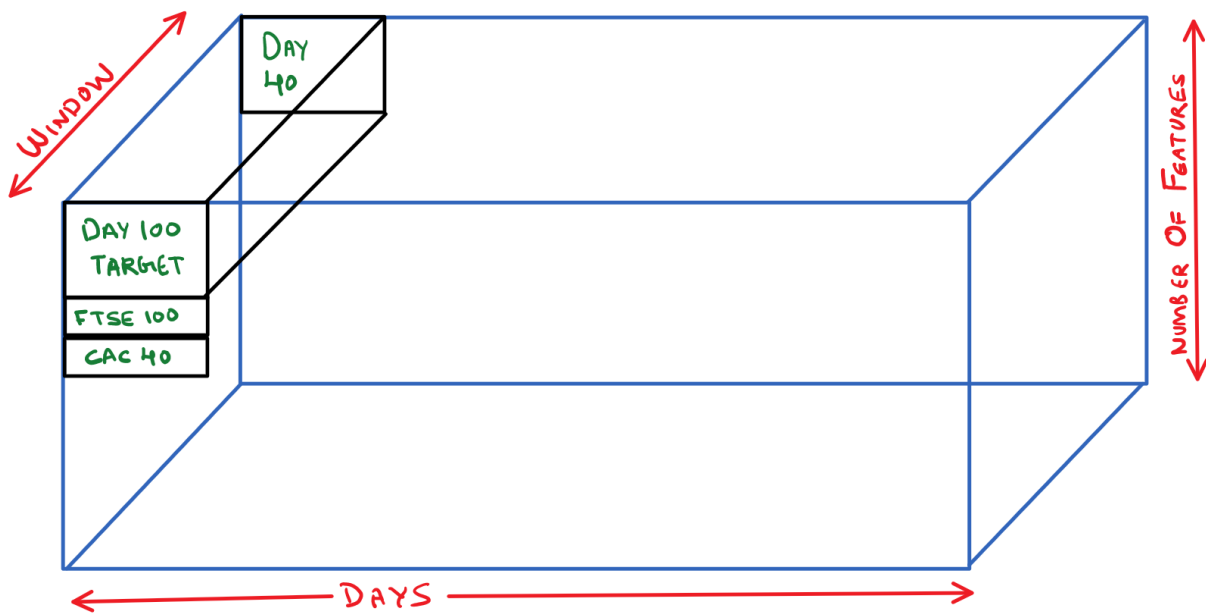
In our case, the training feature matrix has a shape

**[940, 60, 10]**

940 – represents the number of days of data in our training. The number of datasets chosen for training was 1000, but our first day would only begin on day 61 for we 60 days of data to predict day 61.

60 – Look-back widow

10 – Number of features.



Activation functions used – relu

Number of epochs

### LSTM Structure

```
regressor_multi=Sequential()
regressor_multi.add(LSTM(units = 128, input_shape = (60, 10),return_sequences=True))
regressor_multi.add(LSTM(units = 64))
regressor_multi.add(Dense(units = 32))
regressor_multi.add(Dense(units = 16))
regressor_multi.add(Dense(units = 1))
regressor_multi.compile(optimizer = 'RMSprop', loss = 'mean_squared_error',metrics=['mse'])
regressor_multi.summary()
```

Layer (type)	Output Shape	Param #
lstm_1 (LSTM)	(None, 60, 128)	71168
lstm_2 (LSTM)	(None, 64)	49408
dense_1 (Dense)	(None, 32)	2080
dense_2 (Dense)	(None, 16)	528
dense_3 (Dense)	(None, 1)	17
Total params: 123,201		
Trainable params: 123,201		
Non-trainable params: 0		

**Explanation of layers:**

The model we have chosen here has 2 LSTM layers with 128 nodes and 64 nodes each. After that we have two fully connected layers with 32 and 16 nodes and finally we have one output node.

The reasons why I have selected an architecture like this are:

- The very complicated nature of the problem indicates that a simple linear relationship is not going to solve the problem. Say we use only one hidden layer, then it will only capture the linear relationship between the inputs and output. The more hidden layers we have, the more nonlinearity between the features and output can be captured.
- LSTM layers have a memory of the previous data-points it has trained on, therefore my intuition behind putting LSTM layers in the beginning is so that the time dependencies between the different indices can be modelled first. And two LSTM is again as suggested above is to get the non-linear interactions between these different indices. Moreover, two LSTM provided better results on our train – validation dataset than one LSTM layer with more nodes.
- The addition of 2 fully connected layer is again from testing out different architectures and selecting the one given the best performance on the train-validation set.
- Note: The validation set has been selected as 10% of the training set without shuffling the data. I have only used validation set to select the model and not used the test set for model selection. This ensures that I am not introducing look-ahead bias into the model selection procedure.

#### *Parameters:*

The different parameters that I would be using during the implementation and optimization of Deep Neural Network architecture are:

#### **Optimizer:**

Optimization algorithms helps us to minimize (or maximize) a Loss function (another name for Error function)  $E(x)$  which is simply a mathematical function dependent on the Model's internal learnable parameters which are used in computing the target values( $Y$ ) from the set of predictors( $X$ ) used in the model. For example — we call the Weights( $W$ ) and the Bias( $b$ ) values of the neural network as its internal learnable parameters which are used in computing the output values and are learned and updated in the direction of optimal solution i.e minimizing the Loss by the network's training process and also play a major role in the training process of the Neural Network Model .

## **Dropout**

It is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data.

Dropout is a technique where randomly selected neurons are ignored during training. They are “dropped-out” randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

As a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features providing some specialization. Neighboring neurons become to rely on this specialization, which if taken too far can result in a fragile model too specialized to the training data. This reliant on context for a neuron during training is referred to complex co-adaptations.

You can imagine that if neurons are randomly dropped out of the network during training, that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.

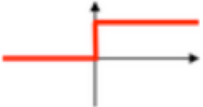

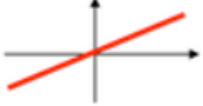
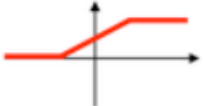
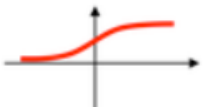
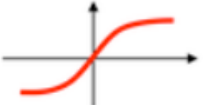
The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to overfit the training data.

### **Activation Function:**

Activation functions are meant to introduce non-linear behavior into the network. A network without any activations is completely linear and this way it cannot learn many interesting problems.

The basic idea of activations in neural networks is that of function composition, each layer has one activation and this "stacks" as the network gets deeper, increasing the complexity of the functions that the network can represent.

Few activation functions with their corresponding function and response are:

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer NN	

### Validation\_split:

If you set the validation\_split argument in model.fit to e.g. 0.1, then the validation data used will be the last 10% of the data. If you set it to 0.25, it will be the last 25% of the data, etc. Note that the data isn't shuffled before extracting the validation split, so the validation is literally just the last x% of samples in the input you passed.

**Shuffle:** Whether to shuffle the samples at each epoch, since we are using time-series data, we set it to false.

### Epoch and Batch\_size:

This comes in the context of training a neural network with gradient descent. Since we usually train NNs using stochastic or mini-batch gradient descent, not all training data is used at each iterative step. Stochastic and mini-batch gradient descent use a batch\_size number of training examples at each iteration, so at some point you will have used all data to train and can start over from the beginning of the dataset.

Considering that, one epoch is one complete pass through the whole training set, means it is multiple iterations of gradient descent updates until you show all the data to the NN, and then start again.

### ModelCheckpoint

After every epoch (one feedforward and back-propagation cycle), validation set (a small part of the training set) is used and the RMSE (root-mean-square deviation) of the model is calculated. If it is better than the previous epoch, then that model would be saved. A ModelCheckpoint is a functionality offered by Keras to save models with good validation scores during training.

### Comparison models

The regression models we are considering for comparison are:

- 1) Support Vector Regression
- 2) Random Forest Regression
- 3) Gradient Boosting Regression

Each of these models are trained-tested and optimized using a pipeline framework. The steps involved in this framework:

- Split the training dataset into 10%, 50% and 100% sized buckets of the original training dataset.
- Set the different parameter grid space for each of the above models
- Then each of these samples, testing datasets and parameter grids are given into the ML-pipeline, in which
  - 1) Scaling is performed if required
  - 2) Set the time-series cross validation object
  - 3) Grid Search Optimization is carried out with the given parameter space and the sequential cross validation object.

Model	Parameter	Grid
SVR	Kernel	'poly', 'rbf', 'sigmoid'
	Degree	3,5,7
RandomForest Regressor	Number of estimators	50,100,250,500
GradientBoosting Regressor	Learning Rate	.1,1,1.5
	Number of estimators	50,100,250,500

- 4) The best model and parameters are selected based on the results from cross-validation performance.
- 5) R-squared, which is the performance metric used to evaluate regression models are calculated for training and testing.
- 6) Then the predicted values and the true target variables are transformed into 0's and 1's based on the direction of the market opening.  
Eg: If today's Open is greater than yesterdays Close then the target would be 1, otherwise 0.
- 7) Then the Accuracy and F-Score of the transformed results are calculated.
- 8) Finally, all the results including the confusion matrix are plotted.

#### *Challenges during model implementation:*

- One coding section where I spend a considerable amount of time is getting the training and test dataset in the right shape as required by Keras.
- Previously I have worked with Tensorflow in designing a Convolution Neural Network and Tensorflow was my first choice, but one of the main decision to switch to Keras has been the abstraction and especially setting the dimension of the dataset in each layer.
- Other challenge that I faced, is sometimes the results using the same model are not reproducible since the weights are initially set randomly, this confused me a little in the beginning even after setting the initial seed inside keras models, but then setting the seed outside solved the problem.
- The biggest trouble that I faced is hyper-parameter tuning. 'KerasRegressor' which is the wrapper for sklearn's GridSearchCV has an option to run epochs parallelly, but unfortunately this wasn't working in my Linux VPS as there was possible memory leak in GPU during thread initialization, I spent quite some time figuring it out but to no avail.
- But that problem actually helped me to look more at the hyper-parameters and understand its working in the model and shortlisting the parameters to a smaller grid for optimization.

## **Refinement**

To tune the hyper parameter in the algorithms used in the Deep learning regression, I used an exhaustive grid search. In fact, it was too exhaustive. After training more than 36 hours on Google's Cloud GPU VPS, I decided instead of reducing the parameter space, I got rid of the exhaustive grid search method as the estimate for completing the grid search was 3 days. Instead I optimized 1 parameter at a time keeping all the other parameters as default values. This is definitely not the best method compared to exhaustive grid search, but when the minimum charges for a GPU cloud instance is 1\$ per hour, this seemed to be the best option.

<b>Optimizer</b>	'SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam'
<b>Batch Size</b>	10, 20, 40, 60
<b>Activation for Output Node</b>	'sigmoid', 'linear', 'relu'

The model was then tuned using scikit-learn's GridSearchCV by wrapping the Keras model using 'KerasRegressor'. This is an approach to parameter tuning that builds and evaluates models for each combination of algorithm parameters specified in the parameter space above. Also for avoiding over fitting and for cross validation, we have used a sequential cross-validation of 3.

## IV. Results

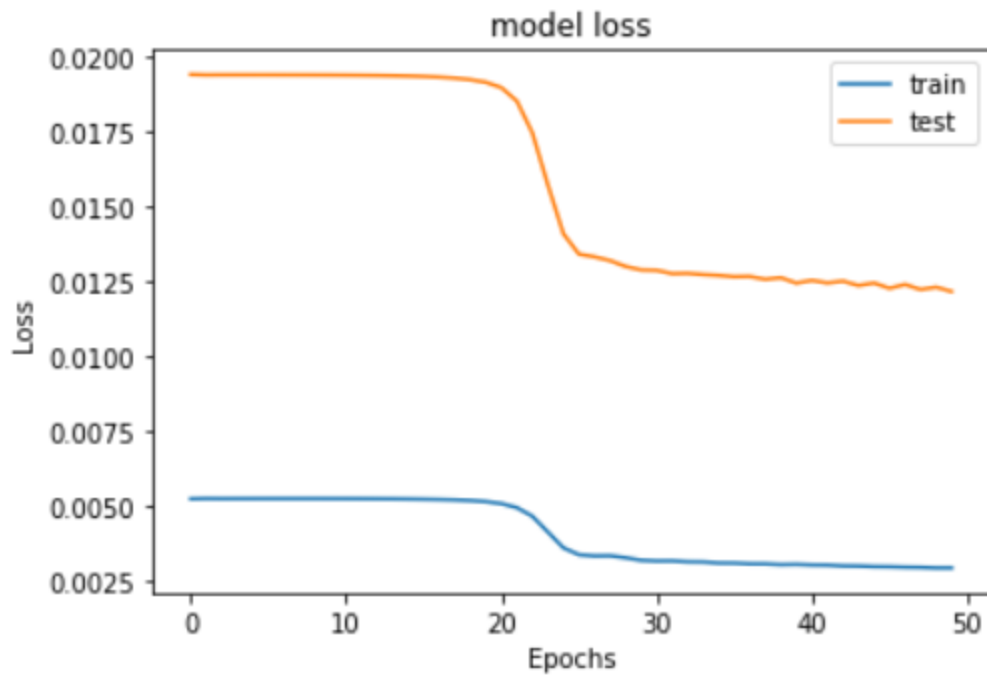
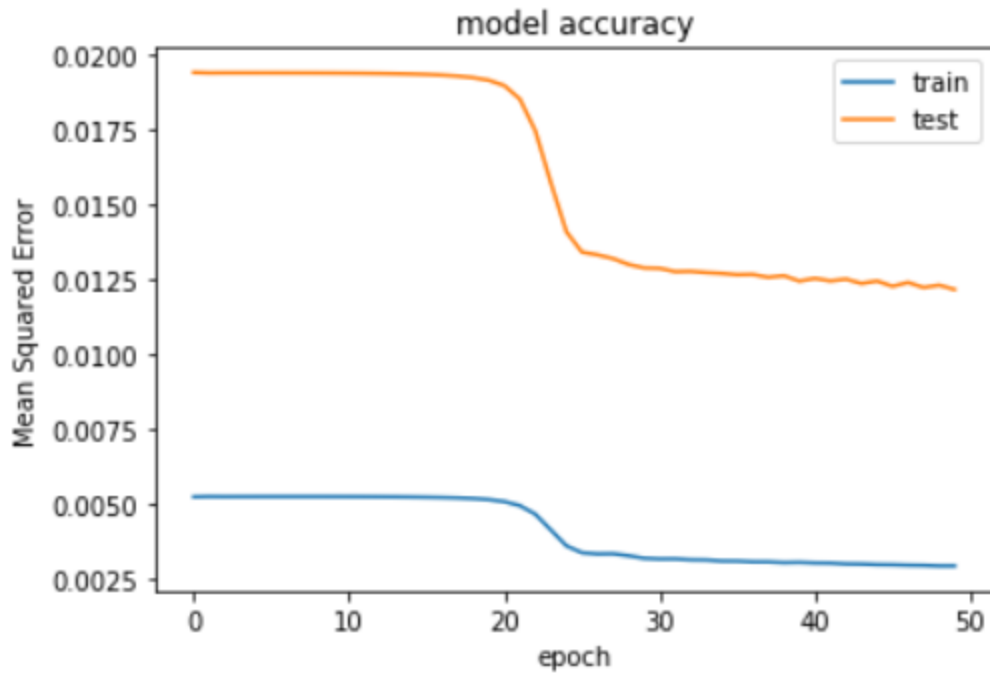
---

### Model Evaluation and Validation

#### *Untuned Model Results*

<b>Metrics</b>	<b>Train</b>	<b>Test</b>
<b>R<sup>2</sup> Score</b>	0.43028	0.364937
<b>Accuracy</b>	0.7793	0.739
<b>F Score</b>	0.79507	0.78947





The training set in the above graph indicates the validation split on the training set. We can clearly see that the model improves from around 15-30 epochs and then it fails to reduce the error more. So, our objective in optimization will be to effectively find the

hyper-parameter that would fit the data without compromising the bias-variance tradeoff.

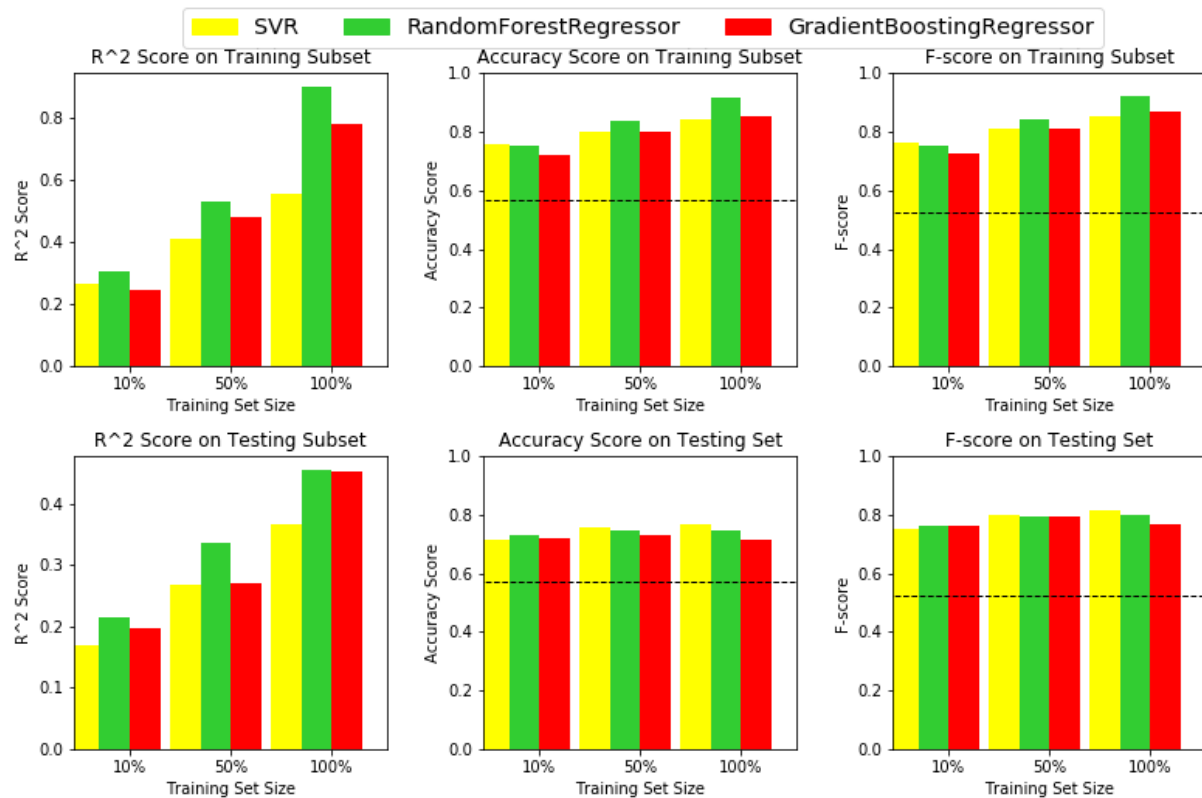
```
from keras.callbacks import ModelCheckpoint
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.from_scratch_1.hdf5',
                               verbose=1, save_best_only=True)

start_time=datetime.datetime.now()
history=regressor_multi.fit(X_train,Y_train,validation_split=0.1,batch_size=20,epochs=50,shuffle=False,\
                             callbacks=[checkpointer],verbose=1)

end_time=datetime.datetime.now()
print("Time Taken",end_time-start_time)
```

### Comparison Model Results

Performance Metrics for Three Supervised Learning Models



The dotted line on the Training and Testing result shows the results from the naïve estimator.

The results for naïve predictor:

Accuracy	0.5234
----------	--------

<b>F-score</b>	0.5683
----------------	--------

<b>Training</b>			
<b>Metrics</b>	<b>SVR</b>	<b>RandomForest</b>	<b>GradientBoosting</b>
<b>R^2 Score</b>	0.5550	0.8993	0.77985
<b>Accuracy</b>	0.8406	0.91718	0.8562
<b>F Score</b>	0.8519	0.92157	0.87199
<b>Testing</b>			
<b>R^2 Score</b>	0.3670	0.45554	0.45230
<b>Accuracy</b>	0.765	0.74782	0.76923
<b>F Score</b>	0.8136	0.7988	0.71304

From the results of the comparison model, it is evident that our initial deep learning model falls short in accurately predicting the price and the direction of S&P500. Moreover, implementation time and training can be performed much quicker on our comparison model.

So, now we'll hyper-parameter tuning on our deep learning model to see how much improvement we can make by optimizing the parameters using GridSearch Cross-Validation.

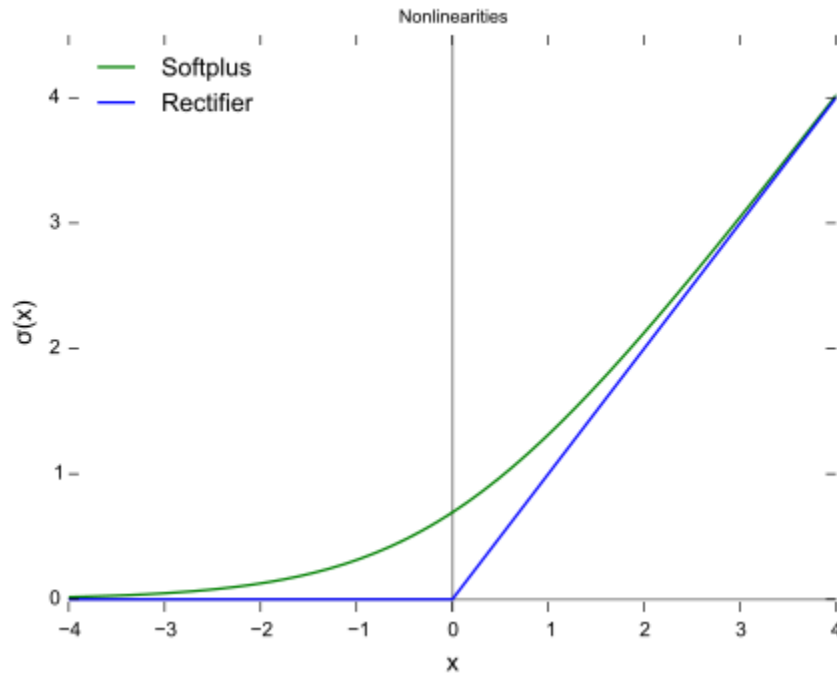
### *Optimized LSTM Results*

Since the Exhaustive grid-search method took more 3 days to train over the parameter grid – space. I resorted to a faster parameter tuning method. Which is as follows

- First, I tuned an appropriate 'optimization algorithm' from a list of optimization algorithms
- Then using the best optimization algorithm, I tuned for an appropriate batch size
- Then using the previous parameters I tuned for the activation function that needs to be used for the Fully Connected and the Output layer.

This is the final structure of the tuned LSTM-Fully Connected model. Our Grid-Search results suggested that the best:

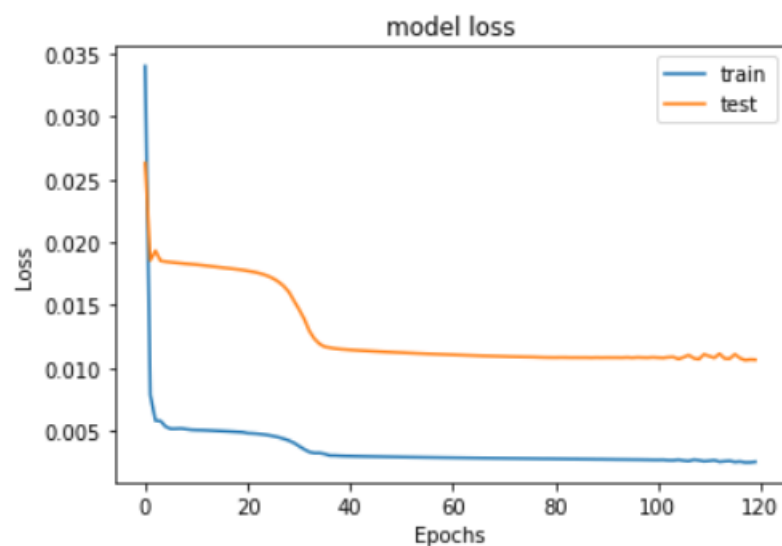
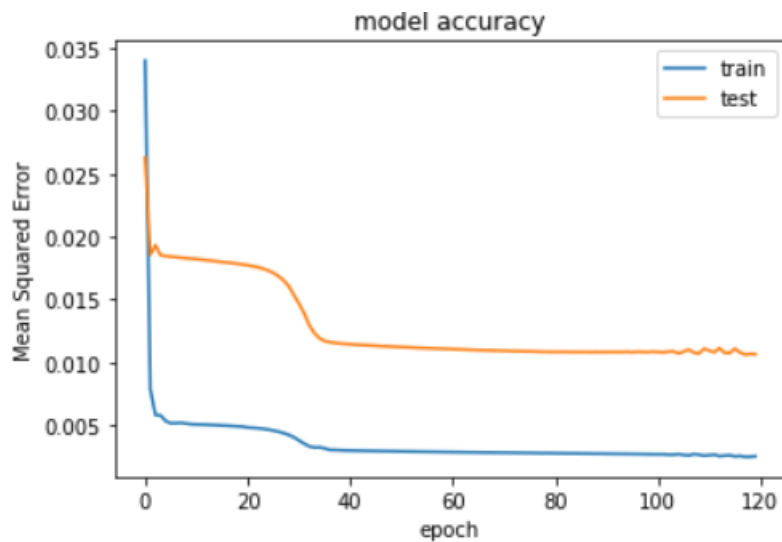
- Optimization algorithm is 'Adam', which is a stochastic optimization method.
- Activation layer for the fully connected layer: 'relu' or rectifier function



- Batch Size = 60

```
regressor_optimized=Sequential()
regressor_optimized.add(LSTM(units = 128, input_shape = (60, 10),return_sequences=True))
regressor_optimized.add(LSTM(units = 64))
regressor_optimized.add(Dense(units = 32,activation = 'relu'))
regressor_optimized.add(Dense(units = 16,activation = 'relu'))
regressor_optimized.add(Dense(units = 1))
regressor_optimized.compile(optimizer = 'Adam', loss = 'mean_squared_error',metrics=['mse'])
regressor_optimized.summary()
```

Layer (type)	Output Shape	Param #
lstm_7 (LSTM)	(None, 60, 128)	71168
lstm_8 (LSTM)	(None, 64)	49408
dense_9 (Dense)	(None, 32)	2080
dense_10 (Dense)	(None, 16)	528
dense_11 (Dense)	(None, 1)	17
Total params: 123,201		
Trainable params: 123,201		
Non-trainable params: 0		



The above graph shows the Loss reducing as the number of epochs increases. Here 'test' set indicates the validation set which is 10% of the training set.

Looking at the results below of the optimized model, the  $R^2$  value is fairly low for the test set while for training set it has considerably improved. This suggests overfitting of the data to the training set.

To avoid this problem, we'll add Dropout of 10% into the LSTM layers. This ensures that randomly selected 10% of the nodes in the LSTM layer are switched off (removed from the network) during every epoch (forward and back-propagation of the entire training set) and therefore would help reduce overfitting.

```

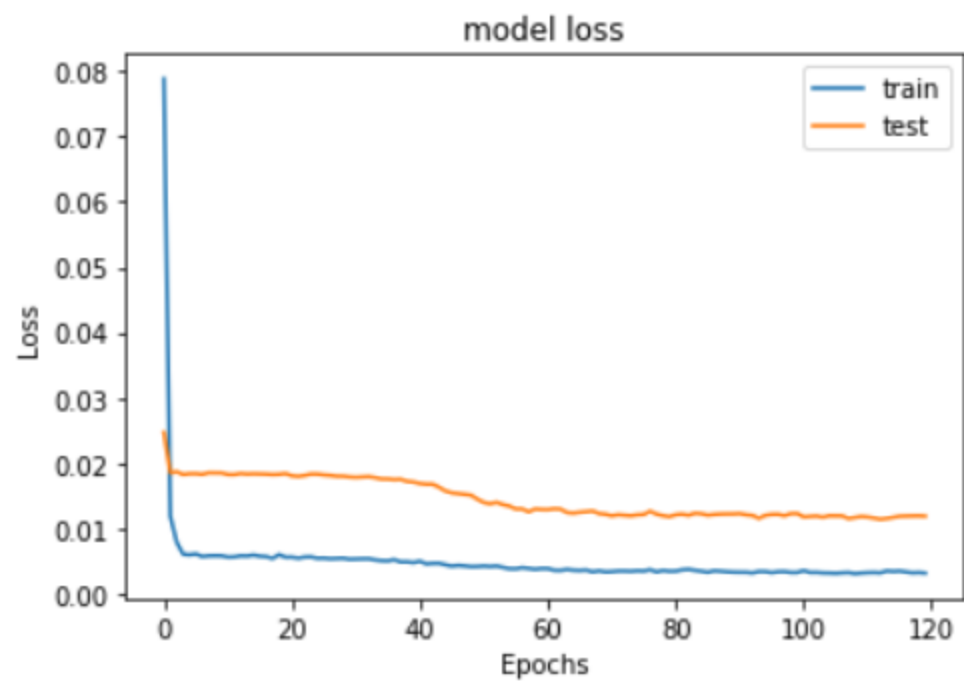
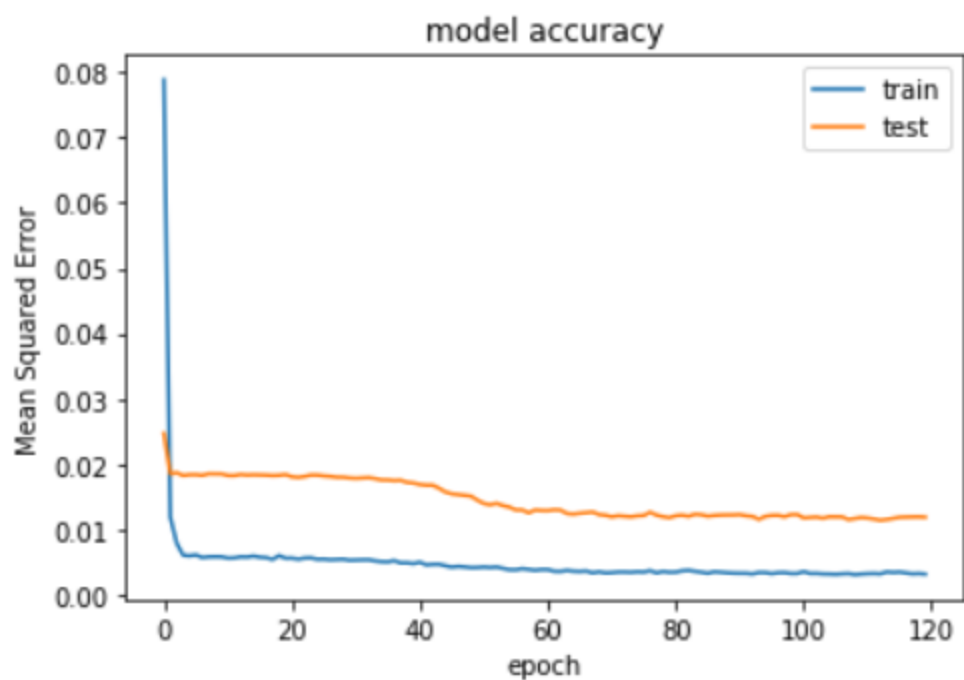
regressor_optimized=Sequential()
regressor_optimized.add(LSTM(units = 128, input_shape = (60, 10),return_sequences=True))
regressor_optimized.add(Dropout(.1))
regressor_optimized.add(LSTM(units = 64))
regressor_optimized.add(Dropout(.1))
regressor_optimized.add(Dense(units = 32,activation = 'relu'))
regressor_optimized.add(Dense(units = 16,activation = 'relu'))
regressor_optimized.add(Dense(units = 1))
regressor_optimized.compile(optimizer = 'Adam', loss = 'mean_squared_error',metrics=['mse'])

regressor_optimized.summary()

```

Layer (type)	Output Shape	Param #
lstm_9 (LSTM)	(None, 60, 128)	71168
dropout_3 (Dropout)	(None, 60, 128)	0
lstm_10 (LSTM)	(None, 64)	49408
dropout_4 (Dropout)	(None, 64)	0
dense_12 (Dense)	(None, 32)	2080
dense_13 (Dense)	(None, 16)	528
dense_14 (Dense)	(None, 1)	17

=====  
 Total params: 123,201  
 Trainable params: 123,201  
 Non-trainable params: 0



Training						
Metrics	SVR	Random Forest	Gradient Boosting	LSTM+FC	LSTM+FC Optimized	LSTM+FC Optimized with Dropout
<b>R<sup>2</sup> Score</b>	0.5550	0.8993	0.77985	0.43028	0.43945	0.4994
<b>Accuracy</b>	0.8406	0.91718	0.8562	0.77931	0.77413	0.79310
<b>F Score</b>	0.8519	0.92157	0.87199	0.79507	0.810306	0.8231
Testing						
<b>R<sup>2</sup> Score</b>	0.3670	0.45554	0.45230	0.36493	0.346050	0.43872
<b>Accuracy</b>	0.765	0.74782	0.76923	0.73913	0.74782	0.74782
<b>F Score</b>	0.8136	0.7988	0.71304	0.78947	0.81210	0.80303

We can clearly see that the optimized model after adding the dropout layer have clearly achieved our goal of reducing variance. Not just that, it also improved the predictions on the training set to a greater extend by generalizing.

## Justification

To see how the model holds up, four different regression models were considered (2 of them are ensemble models). All the four models including the deep learning model performed far better than the benchmark model (naïve predictor).

The LSTM model clearly seem to have relatively good R<sup>2</sup> score and F Score. And my intuition of it able to remember the previous market data and long-term dependencies seem to be right.





Legend:

- trace 0 - increasing
- trace 0 - decreasing
- Higher Predicted Opening
- Lower Predicted Opening

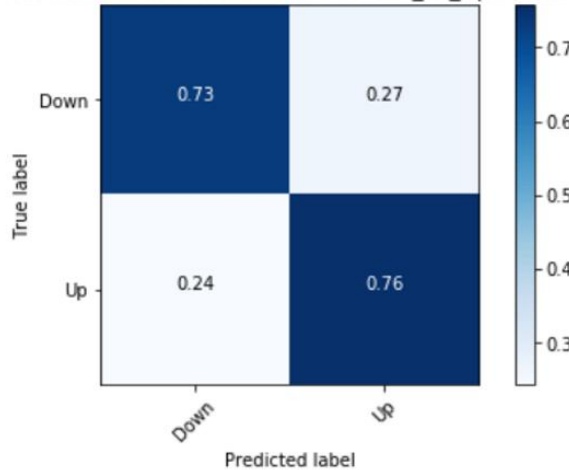
Looking at the above plots of the S&P500 Daily OHLC (Open High Low Close) prices with the opening predictions, it is very evident that our model can predict the direction of the Opening really well and even the price prediction is within good range, there are certainly some days where the prediction are very off the actual, which suggests that there is more room for improvements.

## V. Conclusion

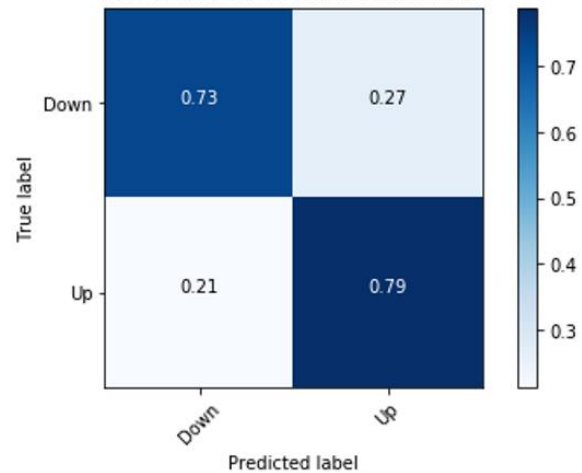
---

### Free – Form Visualization

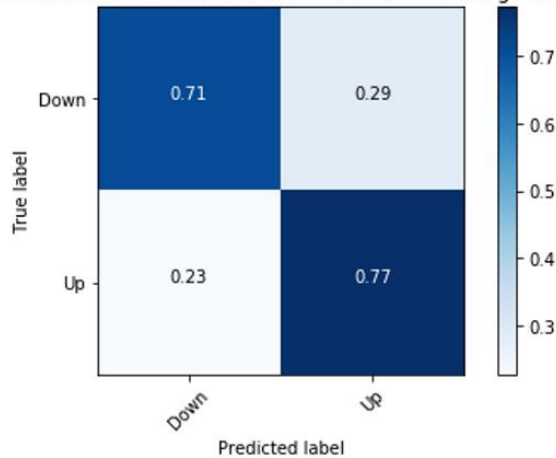
Normalized confusion matrix for LSTM\_FC\_Optimized



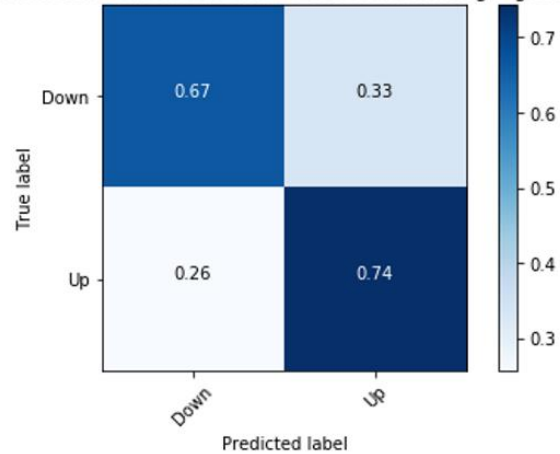
Normalized confusion matrix for SVR



Normalized confusion matrix for RandomForestRegressor



Normalized confusion matrix for GradientBoostingRegressor



The normalized-confusion matrix of all the different models, shows that all of the models clearly has been successful in predicting the direction of the market to a certain extent with less type 1 and type 2 errors.

### Reflection

One of the most difficult part of the project had been the amount of computational resource and time it takes for training the deep learning network. Even on a VPS GPU instance with 24 GB Graphics memory. The estimated time for an exhaustive grid search

was about 3-4 days. This limitation forced me to adapt and search for new ways to tune the hyper-parameters.

One of the reasons why the LSTM model wasn't able to give stellar performance compared to the other 3 regression models can be attributed these reasons.

- The lack of data: In this project, I was using only 4 years of data and comparing that with other deep learning models training data, this is too low.
- Next, the long-term memory of the LSTM can turn out to be curse than a boon sometimes, especially when used alone. Since financial markets tend to change its characteristics very often, which means that the relationship between the indices that existed two years back may have been completely vanished by now.

## **Improvement**

The possible improvements that could be made to improve this project are:

- Collecting more data, at least 10-15 years of data. The more we have, the better it is.
- Having more indices, now we had only 10 different global indices as feature set. We must include all possible markets, even emerging markets data.
- Perform multithreading on different epochs while doing the exhaustive grid-search to speed up the process and utilize the entire capacity of the GPU.
- Modelling more complicated network with other Recurrent Neural Network architectures.
- Adding in market regime switching models. As interdependence and characteristics of market fluctuate very often, so the relationships between different markets might not stay constant. So, we need to incorporate into our model a new feature or another model that we'll be able to understand this changing behavior of the market.