LSTM Recurrent Neural Networks to Predict Google Stock Prices

This project involves the use of Recurrent Neural Networks along with Long Short-Term Memory (LSTM) to predict the stock prices of Google.

About RNN

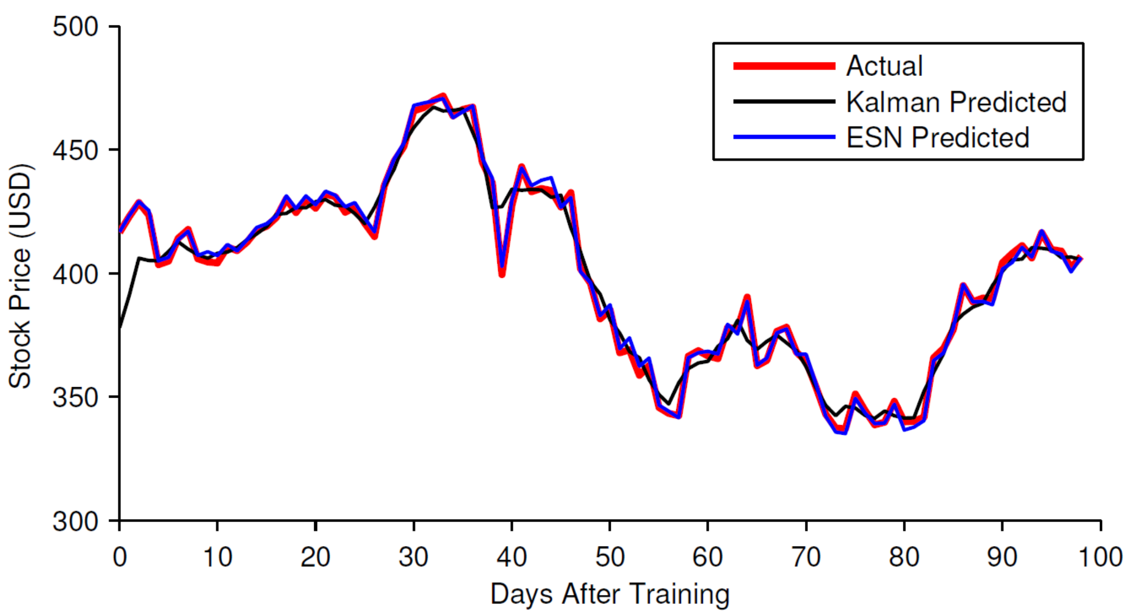
Recurrent Neural Network (RNN) was designed, like any other Neural Network, to function like a specific part of the human brain. When looking at our brain's cerebrum, we can divide it into the Temporal, Parietal, Occipital and Frontal lobe. Out of these, the Frontal lobe is the part which deals with short-term memory and remembers what happened in the immediate present and use it for decision making in the near future. It is this Frontal lobe that the RNN tries to replicate.

About LSTM

The Long Short-Term Memory (LSTM) is a variation of the RNN which solves the Vanishing Gradient Problem, which leads to the decrease in gradient for each level in a regular RNN because of the Recurring Weight being close to 0. This leads to a difficulty in the manipulation of weights for a layer farther away from the present layer and thus makes predictions inaccurate. We would not go into the details of this phenomenon.

About the Dataset

We are using Google's Stock price from 5 years till now from a financial website (Yahoo! Finance). The idea of this project was based on a project by students from Stanford (Financial Market Time Series Prediction with Recurrent Neural Networks - Bernal, Fok, Pidaparthi). The team used an Echo State Network instead of an LSTM. We will use their findings as a comparison to how our LSTM performs. The team trained their model from late 2004 to early 2009 with data from Yahoo! Finance. They created a visualization comparing their predictions with the actual data.

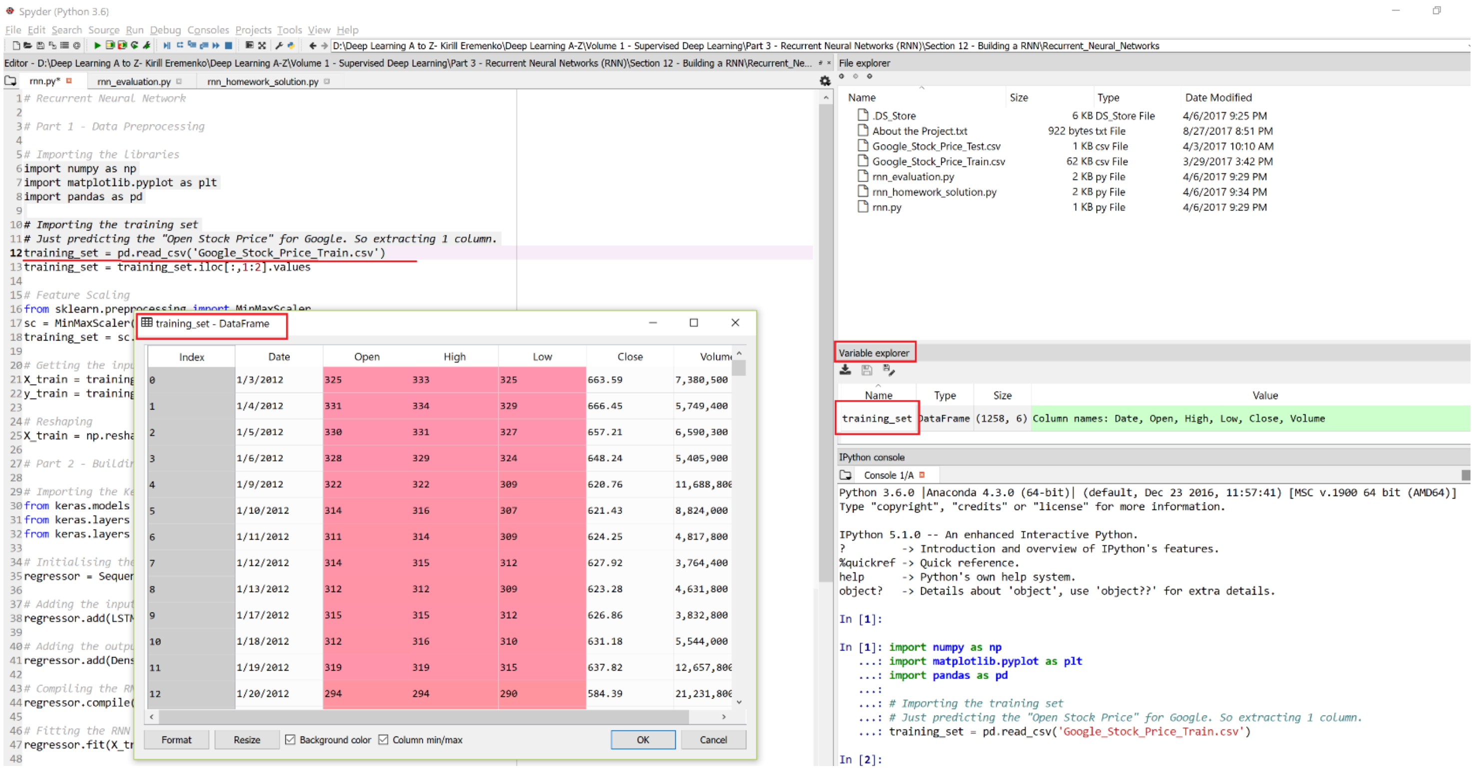


We will also train our LSTM on 5 years of data. We can see that their predictions are quite close to the actual Stock Price. We can try to get the same accuracy from our model as well.

We assume that the present day is January 01, 2017. We will then get the Google Stock price for the previous 5 years. Once we train our LSTM, we will try to predict the stock price for the month of January 2017.

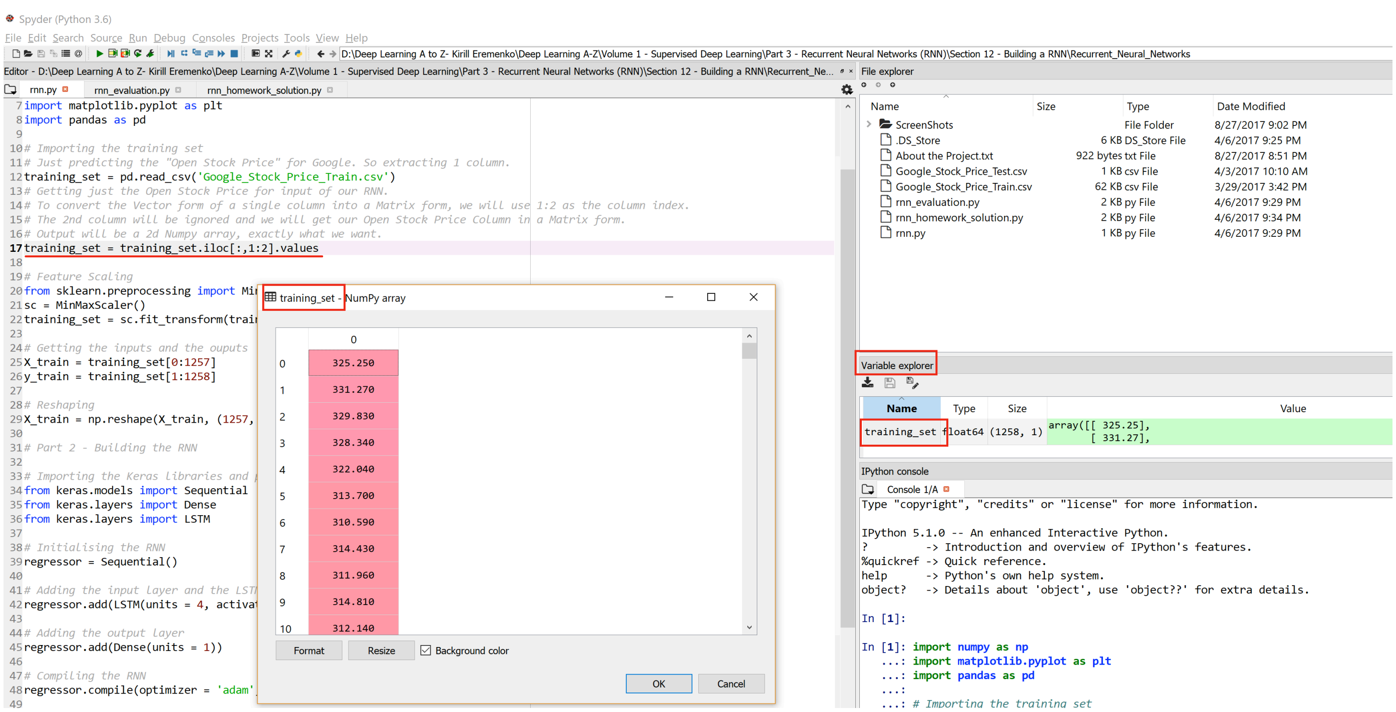
Data Preprocessing

We have the training set of 5 years of Google Stock price. The test set contains the stock price for January 2017. We will first import it. As we can see in the dataframe, the dates range from 2012 to 2016.

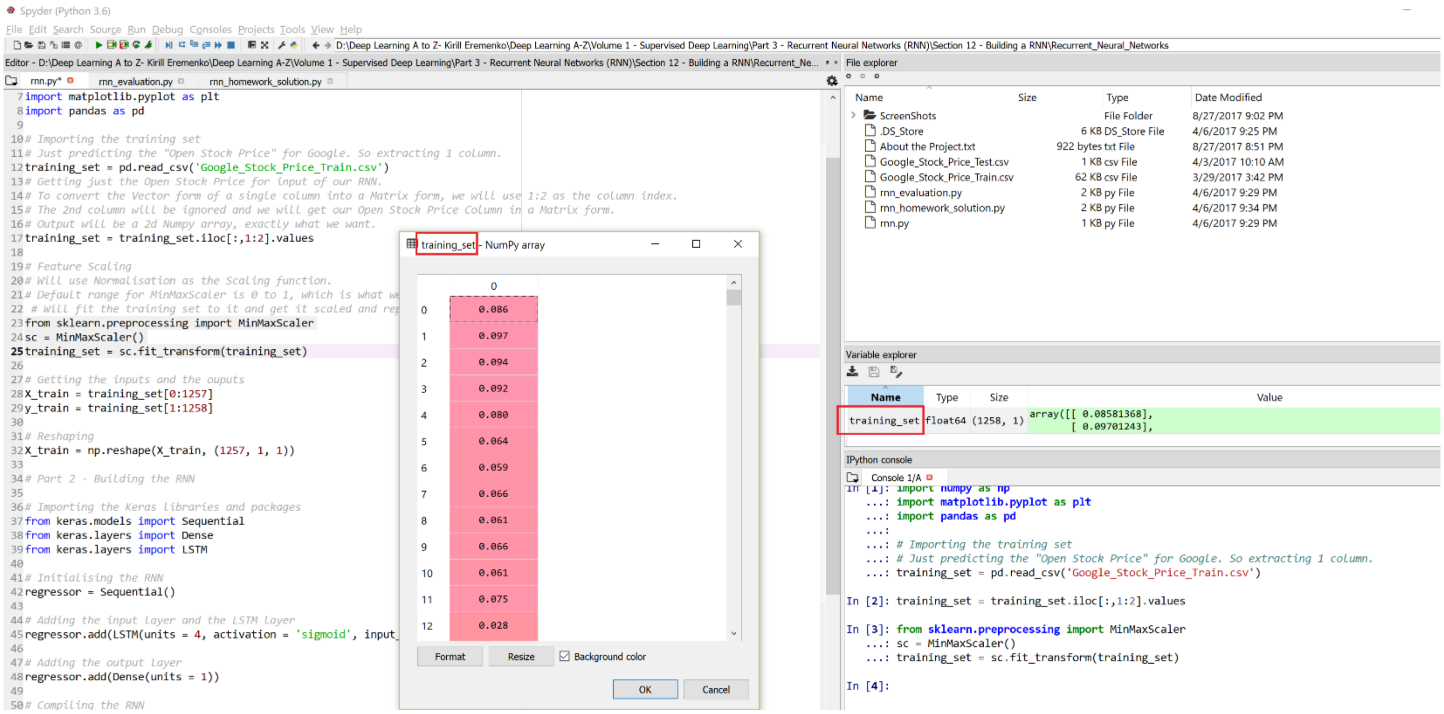


We can see that there are 3 types of stocks - Open, High, Low and Close. There is also a Volume column which contains the Volume of stocks for Google. We will focus on the Open stock price and will predict this price for January 2017.

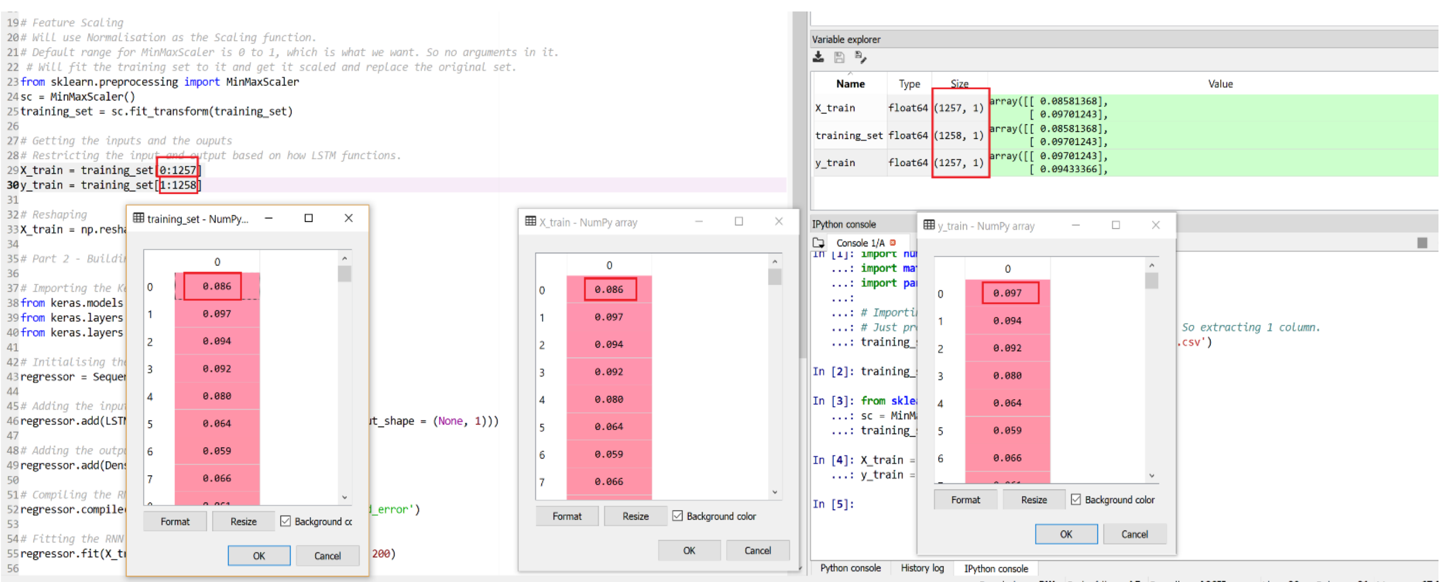
The input for RNN will NOT be Date and Open Stock Price, it will just be the Open Stock Price for different time frames. We will get this by using just the Open Stock Price form our Dataframe using iloc.



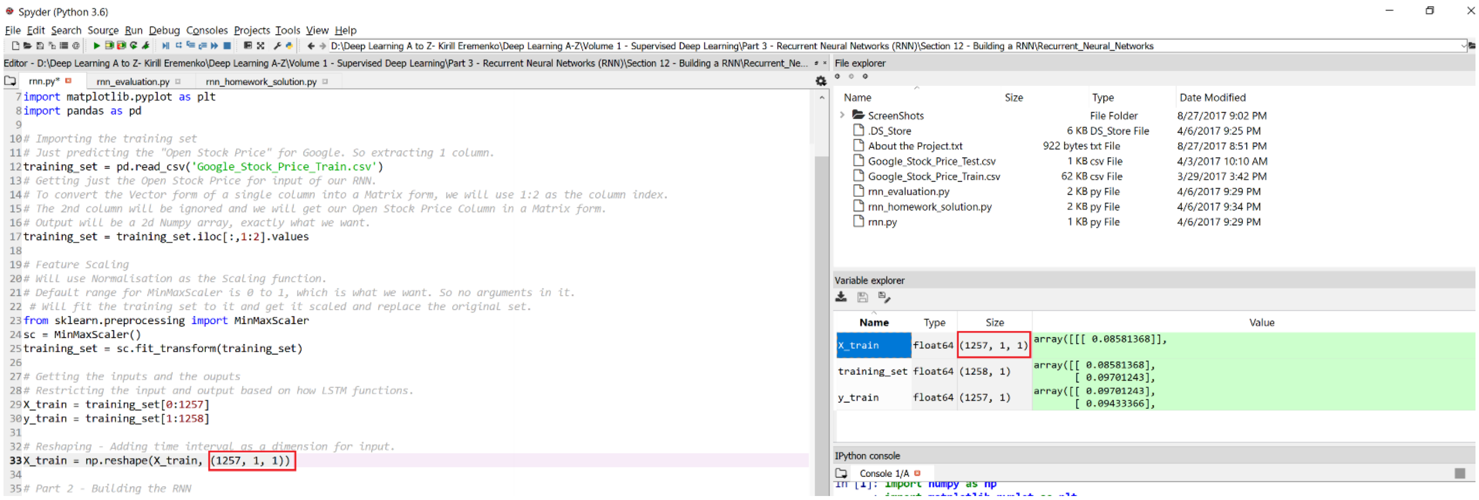
For feature scaling, we have 2 options - Standardisation and Normalisation. Since LSTMs use many sigmoid functions, which work in 0s and 1s, it also makes sense to use Normalisation, that converts the data between 0 and 1. But you can check the results of both the methods and choose for yourself.



We will next try to determine what the input (X_train) and output (y_train) will be. The input will be the value that changes with time i.e. the current open stock price (time t). The output would obviously be the future value of the same i.e. the near future open stock price (time t+1). The trick behind choosing the ranges would be that the prediction would be a day after the current value. The training set contains 1258 values. The input should be therefore restricted to 1257. The output on the other hand, cannot contain the 0th days prediction, so it will start from 1 and end at 1258.



The use of LSTM involves the prediction along the time of a particular value. The input that we have is currently 2-dimensional - we have 1257 rows and 1 column. We need to add another dimension to the input in account of time. This process is called reshaping. This format of input is required by Keras and the arguments have to be in the order of batch_size(number of rows), timesteps(the number of time intervals or days between any 2 rows, in this case it will be 1) and input_dim (number of columns). These 3 arguments are encapsulated together and come after the original data as the argument of Numpy's reshape function.



Building the RNN

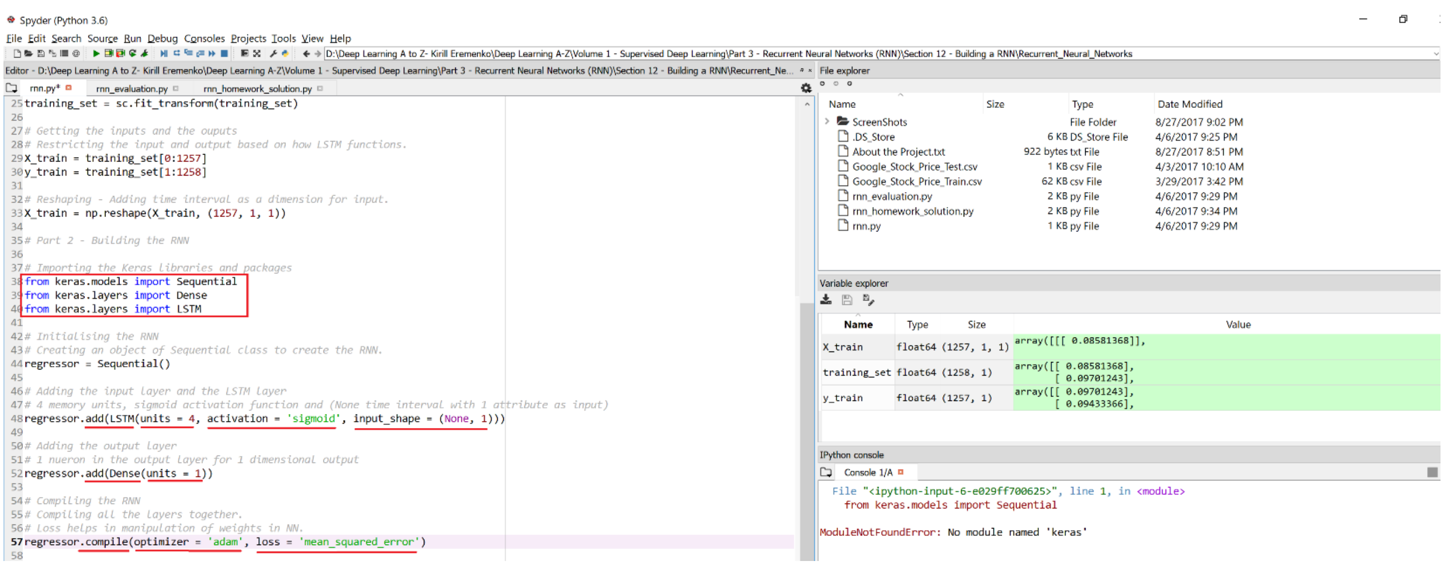
We will first import 3 classes. The Sequential class that will initialise our RNN. The Dense class will create the output layer of our RNN. And finally, the LSTM class which will make our RNN have "Long Memory".

The method to create the RNN will be the same as the one used to create ANN and CNN. We will then use other methods of the Dense and LSTM classes to add the layers and compile it, and eventually fit it.

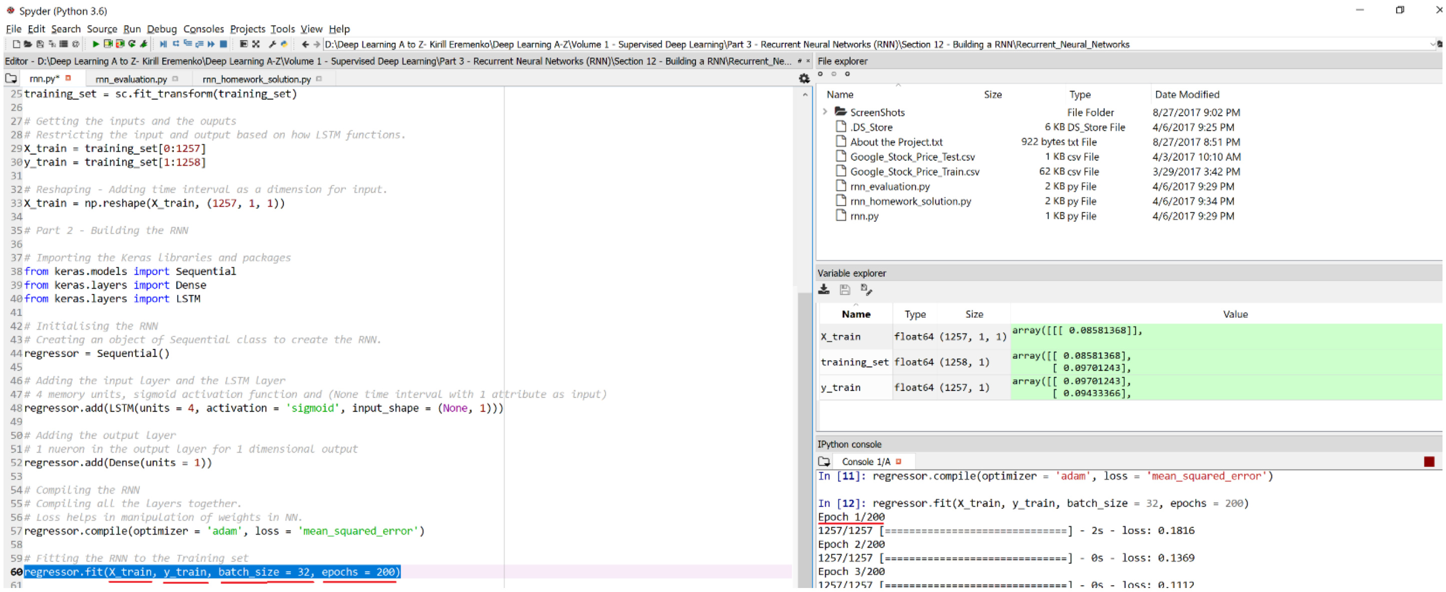
We are predicting a continuous variable and are hence using a regression model instead of a classification model and call our object 'regressor'. To this regressor object, we will add the LSTM layer, which in itself will take the input layer as input. The arguments we add to the LSTM layer are: units - number of memory units, activation function - can be tanh or sigmoid. Other arguments will be default. But there will be an additional argument - the input shape argument to specify the format of our input layer. This argument would be none and 1 - none to specify that model can expect any time step and 1 because we have just 1 column of input. The optimal number of memory units that we can use is 4, the activation function is sigmoid, and the input_shape would be (None, 1).

The next layer that we will add is the Output layer. We will use the Dense class, with the argument being units, everything else being default. The units are the number of neurons that should be present in the output layer, which is dependent on the dimensions of the output. So, our units argument for Dense class will have value 1.

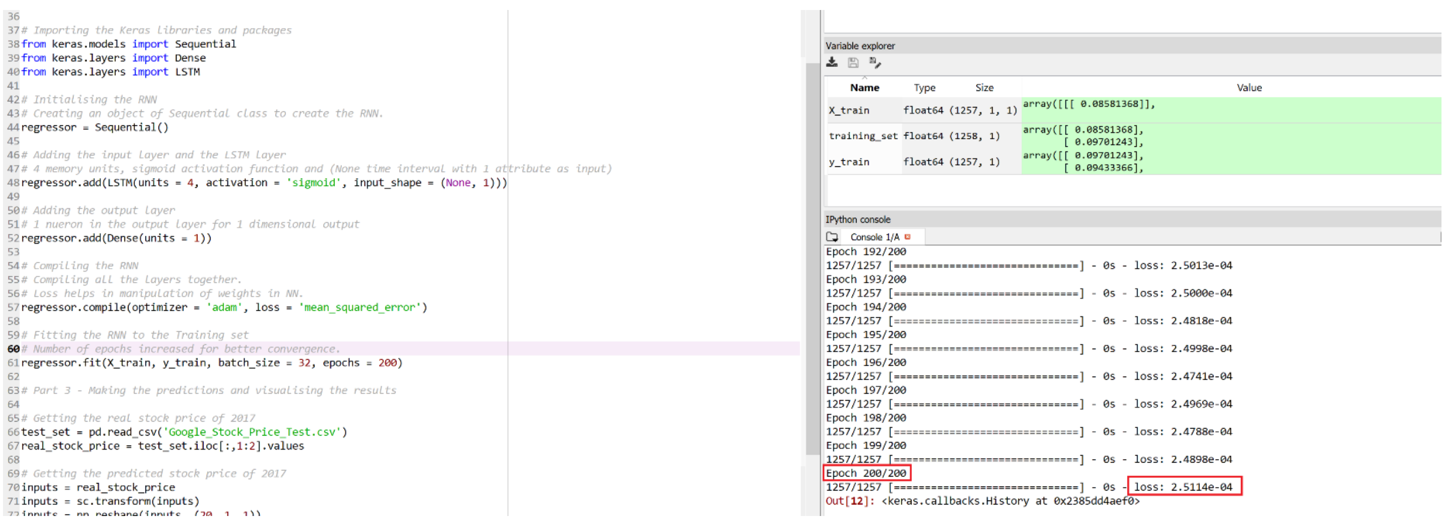
To compile all the layers into a single system, we will use the compile function along with its arguments. Optimizer can be RMSprop or Adam. Both the optimizers gave similar results but RMS was memory heavy, so we went forwarded with Adam. But usually, RMSprop is recommended in Keras documentation. The Loss argument decides the manipulation of weights, so for this we should be using Mean Squared Error for continuous variable. For test set, we might use Root Mean Square Error in its place. Other arguments will be default.



Now we will fit this regressor to the training dataset. We will use the fit method for this. The important arguments include the input, output, batch_size and epochs. We will keep the default batch size of 32 but will change epochs to 200 for better convergence.



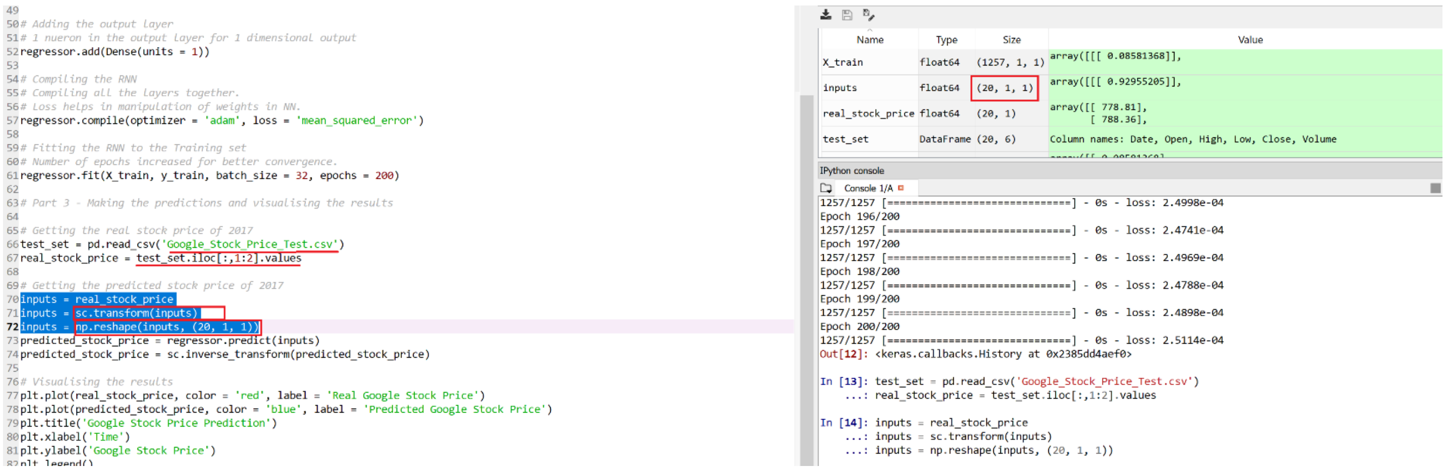
With the passage of fitting, we see that the loss keeps on decreasing. But we will get accurate results only if we have the same loss in the test set.



Making Predictions and Visualising the Result

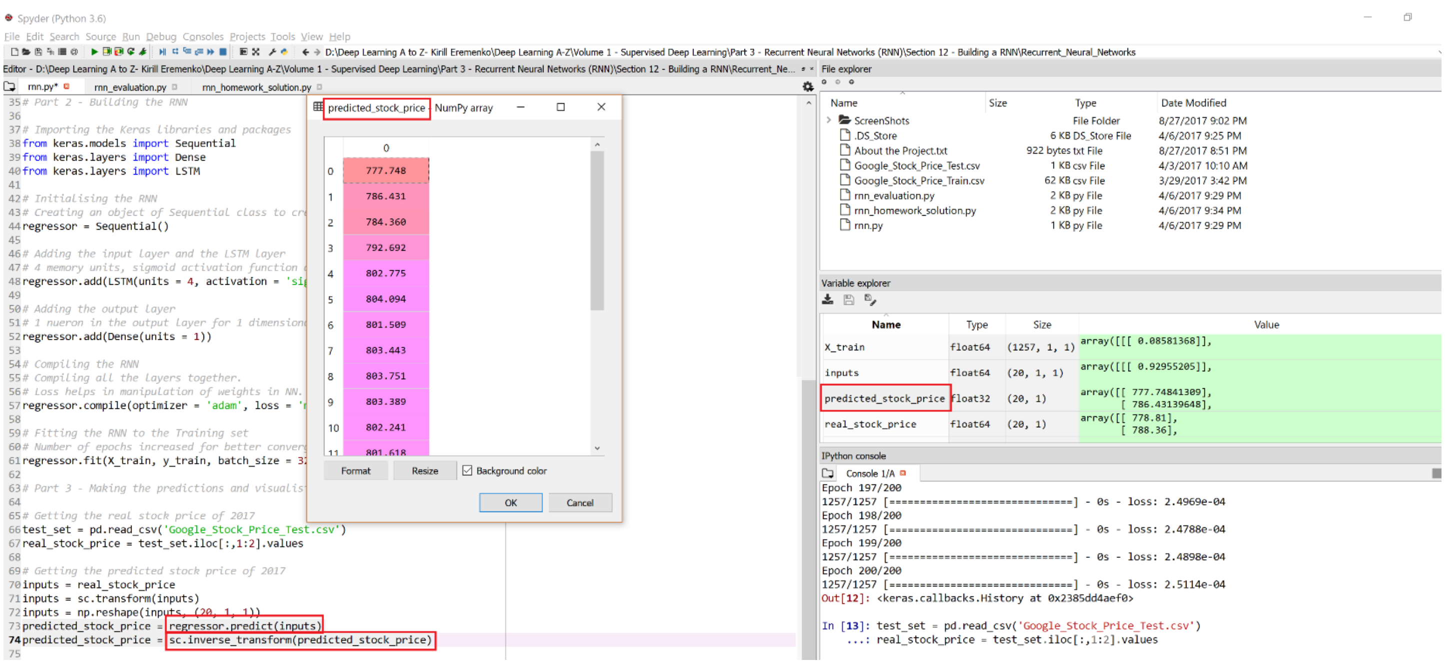
The methods of getting the Test Dataset is same as that Training Dataset. We will just rename it to real_stock_price so that we can distinguish between prediction and actual values.

Next, we use our model to make predictions on the test dataset. But we should keep in mind that every prediction is for the next day and not the present. The input will be the real_stock_price. The model that we have made is on scaled values. When used as it is, it will give incorrect predictions. So we will convert the input using the same “sc” object used for scaling the training data. We will also have to reshape the data according to the format expected by the predict method in a 3d format.

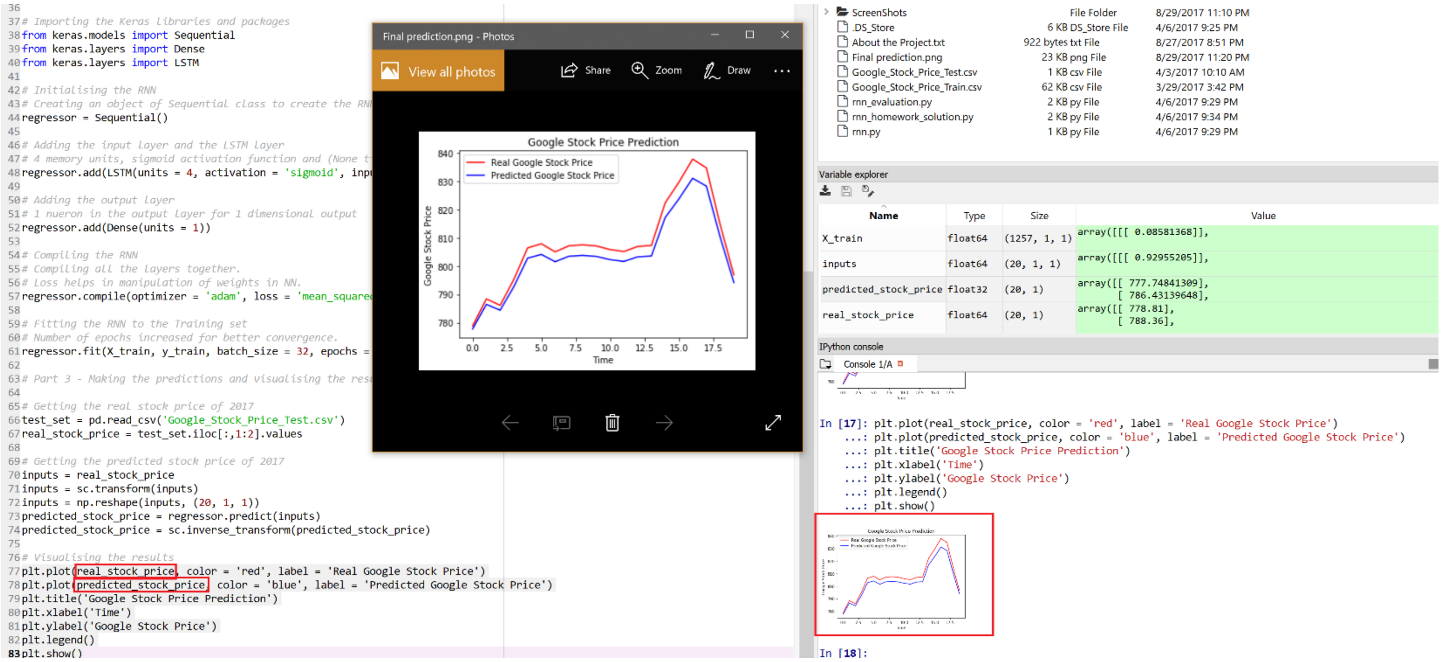


We will now use the regressor model to make predictions on the input and store it in the predicted_stock_price. The argument would obviously be the input. We now have the predicted stock price for the January 2017.

But this output will be scaled. We will have to use the inverse transform method of the same “sc” object we had used to scale the data to get the proper predicted values. This is the final prediction.



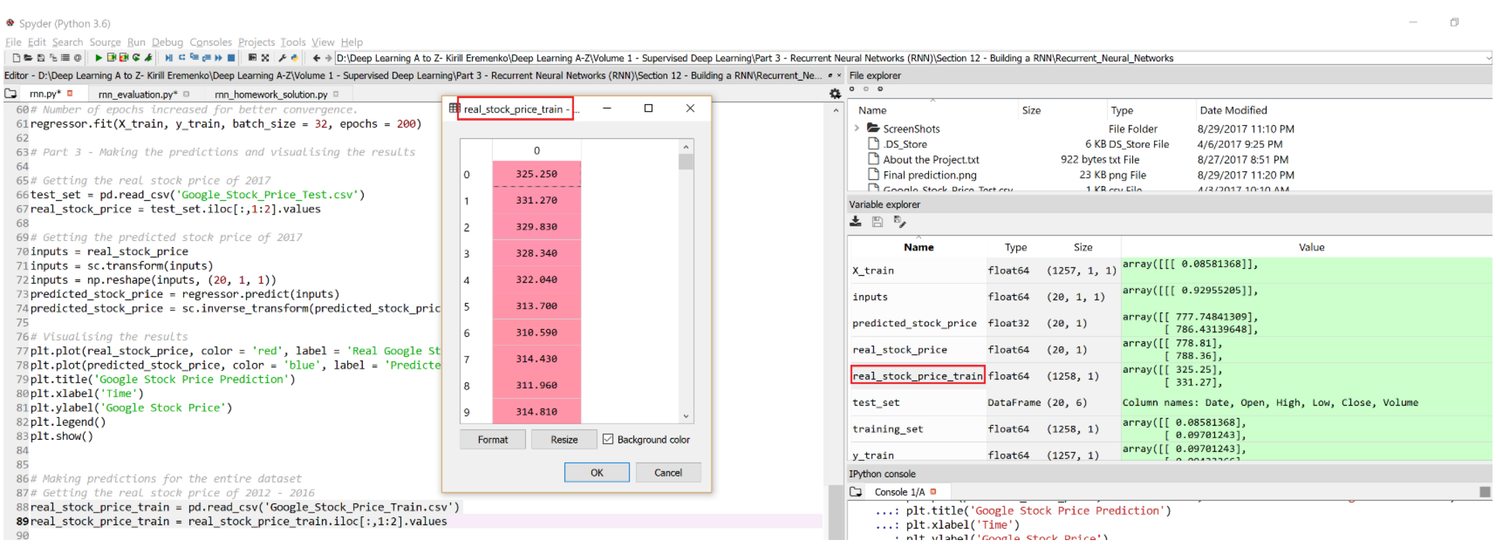
Now we visualize our predictions with the actual stock prices of Google. For this we use the pyplot module. We have some arguments with pyplot, like the use of color. For real stock prices, we will use Red. We will also include a label mentioning the real stock price. We will keep blue the color for Predicted Stock price and change the label as well. We will also add the axis labels and title and display it.



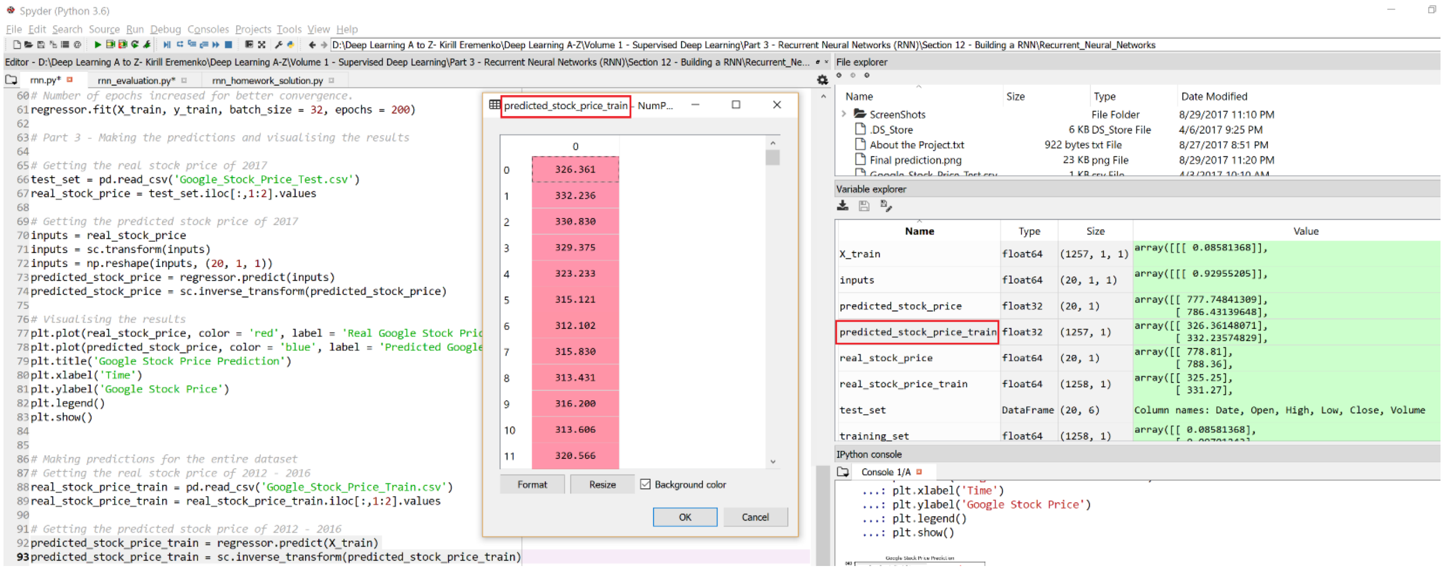
What's important to note is that this is a 1 time-step prediction i.e. input is of time t and prediction is of time t+1. We should note that we were able to make these predictions for 20 days only because we had the stock price for 20 days. This would not have been possible if we had the stock price for just 1 day. It would be wonderful to make predictions for a long future, but we would hardly get such amazing predictions. In finance, there is the Browning Motion, which makes future values of stock prices independent of the past, so it would be impossible to make long term predictions for stock price.

Further Analysis

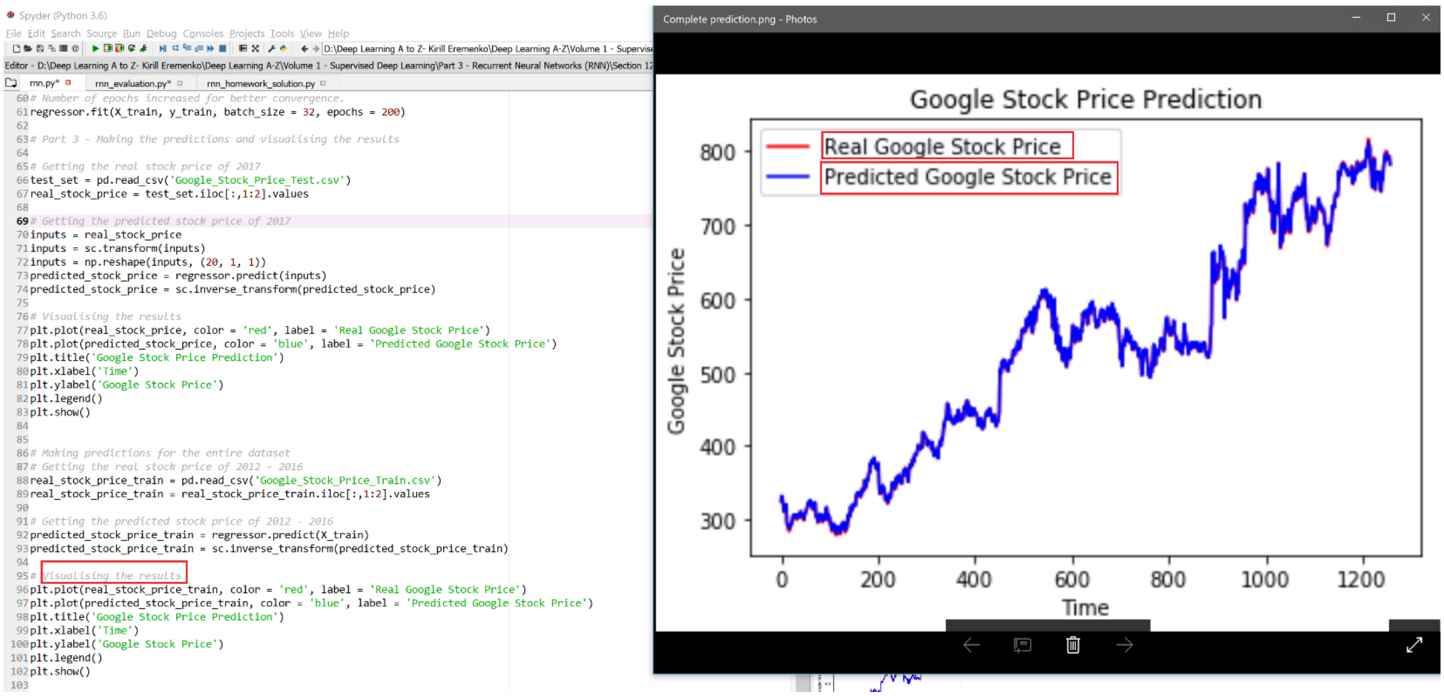
We will take this a little further and make predictions for the stock price from 2012 to 2016. The method of getting the input, scaling and reshaping has not changed.



We already have the regressor and Xtrain from previous work. We just have to make predictions using it. And then we can unscale the predictions to regain the actual values of the stocks.



We compare our predictions with the actual prices through visualisation. At first, we are not able to see the real stock price. But after zooming in, we can see that the LSTM has very accurately predicted the prices and the Blue graph is mostly overlapping the Red one.



Evaluating the RNN

RNNs are evaluated using the Root Mean Square (RMSE) of the test set. We will the root using the math library. The mean squared error function is taken from the scikit-learn's metrics module. These 2 functions when combined, will calculate the rmse for the test sets actual values and the predictions we made. The actual RMSE value does not tell us anything about the size of the test set and how the error correlates to it. We need this in percentage. 800 is the average value of the stock in the test set, so we will divide the RMSE value by 800 to get a percentage value.

We see that the value is close to 0.4% which is a very low error rate and good for our predictions.

Summary

Although the LSTM we have designed predicts quite accurately the stock prices of Google, the reason it is so accurate is because it is learning at time-step of 1. This leads to a reset of hidden layer, and this process goes on and the model is not learning anything useful. This output is not relevant because of this 1 time-step learning.

To make improvements in our model, we need to increase the time-step.

Clarification

The project discussed previously is indeed from Stanford, but it is created by undergraduate students of the CS229 course. The students of that course submitted a final project for evaluation; it was just put online but it is not published anywhere. There is a big difference between reading a Stanford paper published online and reading an assignment written by a group of undergrads.