31005 Assignment 2 Practical Machine Learning Project

September 25, 2019

TIME SERIES FORECASTING ON STOCK PRICE INDEX
Keyu Chen 13261021 Xu Jing 13309618
GitHub:https://github.com/XuJing98/UTS_ML2019_Main/blob/master/Machine_learning_Ass2.ipynb
Video pitch:https://youtu.be/yJqgI6FcOus

1 Introduction

Stock price prediction has become a controversial topic, successful predictions could yield notable profit. However, there are many difficulties in predicting stocks future price. Since the price of each stock is volatile and is heavily influenced by physical and physiological factors, economic recession or inflation, rational and irrational behaviours etc, it is arduous to obtain high accuracy with respect to all complex aspects in future price prediction. The intervention of machine learning algorithms have played a critical role in the financial time-series forecasting. Machine Learning techniques applied in financial fields including stock market, sale analysis have effectively simulated the interests of many researchers and investors. By implementing machine learning algorithms, we could disregard irrational human behaviour and therefore possibly discover insights and patterns that fundamental and technical analyses fail to conclude intuitively.

The goal of the study is to implement machine learning algorithms to predict future stock price, and further develop understandings on time series forecasting. The original data that we have experimented is a set of observations on the values that a variable takes at different times, the stock price index data is collected at regular time intervals and hence it is identified as Time Series data. By evaluating the pattern in the time series data using Deep Learning algorithm LSTM (Long Short-term memory) and time series forecasting model of ARIMA, we are able to predict future price index based on the existing past values. Despite the random and interrelated factors that make stock price almost impossible to predict, time series predictions using machine learning techniques are still valid for short-term forecasting. With the rapid growth in the field of machine learning, it is feasible to implement algorithms for financial analysis with high accuracy.

The following report will extensively describe key challenges, methodologies of the two experimented algorithms, output evaluation of comparative studies and ethical aspects of this research.

2 Exploration

```
In [0]: # import libraries
        import pandas_datareader as web
        import datetime
```

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from statsmodels.tsa.arima_model import ARIMA
%matplotlib inline
```

```
In [13]: # Load data to pandas DataFrame
         data = web.data.get_data_yahoo('oil', start='2018-01-02', end='2019-09-02')
         data.head()
```

Out[13]:

Date	High	Low	Open	Close	Volume	Adj Close
2018-01-02	11.830	11.830	11.830	11.830	0	11.830
2018-01-03	11.830	11.830	11.830	11.830	0	11.830
2018-01-04	11.830	11.830	11.830	11.830	0	11.830
2018-01-05	12.088	12.088	12.088	12.088	500	12.088
2018-01-08	12.088	12.088	12.088	12.088	0	12.088

By importing pandas.datareader library, we extract historical data of Stock Price Index from Yahoo data API. The model implementation is achieved using the cloud based platform Colaboratory. The Stock data experimented is the daily past stock price of oil. The data obtained six attributes: High Price, Low Price, Open Price, Close Price, Volume and Adj Close Price. The research will focus on Adj Close Price as it is the most presentative of stock price index in a trading day.

```
In [15]: data.shape
         #419 rows and 6 columns
```

```
Out[15]: (419, 6)
```

We focus on one-step-ahead prediction, which is using 7-day stock market prices (for example, xt-6, . . . , xt) to predict the future stock market prices(for example, xt+1). To generate accurate one-step-ahead stock price prediction, we have compared the performance of different two models fitted on the same dataset, which are LSTM model and ARIMA model. The following figure shows the flowchart of this process. Also, the implementation of both algorithms is shown separately as follows.

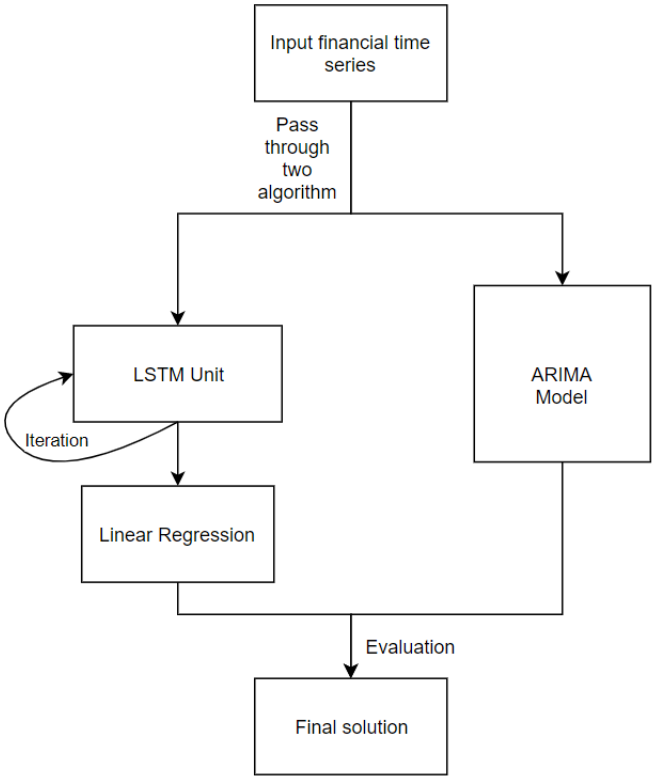
3 Methodology

3.1 ARIMA

3.1.1 Introduction to ARIMA

ARIMA is a time seires forecasting algorithm based on a class of models to predict future values based on the previous data using the lags and the lags or error. The model was first introduced by Box and Jenkins in 1970, the model performs a linear admixture of past values and lag errors and then produce the future value of a variable.

ARIMA models can be treated as a generalization of an autoregressive moving average model which can be applied in time series forecasting and deliver significant insights. In statistics,


Input financial time series
Pass through two algorithm
LSTM Unit
Iteration
ARIMA Model
Linear Regression
Evaluation
Final solution

Flowchart

ARIMA models are applied in predicting confidence intervals if the residuals are not correlated or normally distributed. The model itself comprises of AR (autoregression), I (integration) and MA (moving average) which determine the correlation between an observation and the number of lagged observations, the number of differentiation used on the raw data and the relationship between a value and the residual error of the applied moving average model to those number of lagged observations respectively. The use of models are identified using the of p, d and q parameters, by assigning different number on the parameters, the complexity and the structure of the model are constructed.

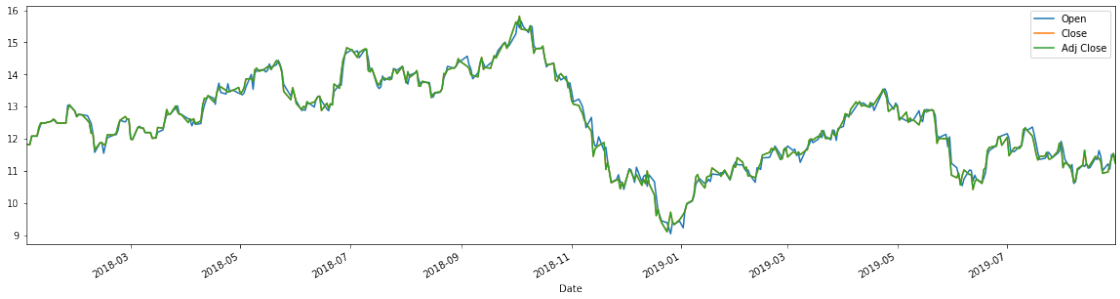
However, to make good use of ARIMA model, the time series must be strictly stationary, which indicates that the mean, variance and covariance of the series are invariant with respect to time. To ensure the series are stationary, elimination of trend and seasonality is required.

The methodology of model identification, estimation of parameters and diagnostic checking is discussed in the following section.

3.1.2 Data Visualisation

```
In [16]: #plot the data
         data[['Open', 'Close','Adj Close']].plot(figsize=(20,5))
```

```
Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x7f73d7511ef0>
```

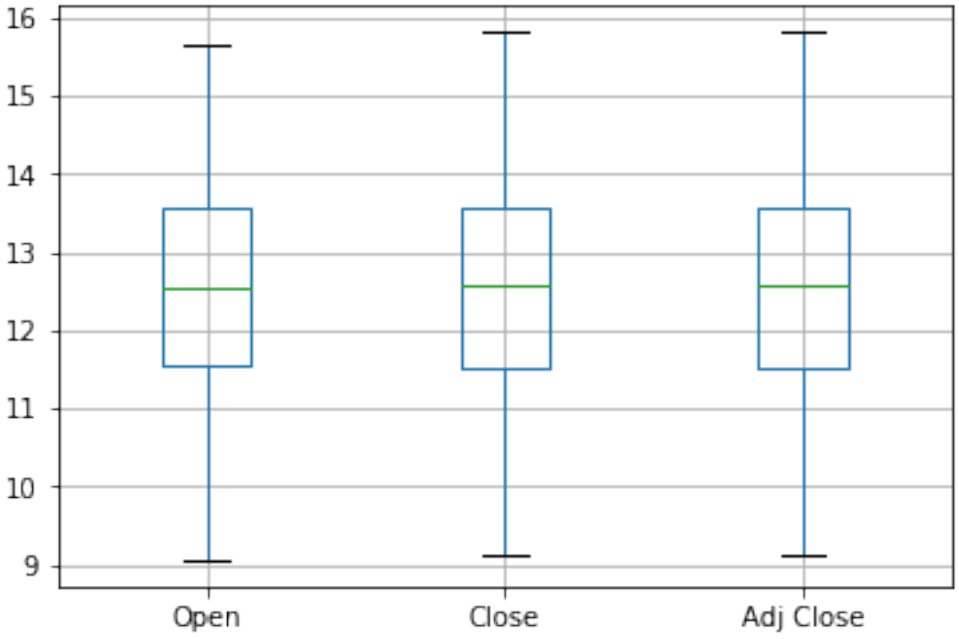


The diagram presents a clear upward trend from 2018-01 to 2018-10, and the price then dramatically dropped from 2018-10 to 2019-01. The price is relatively stable from 2019-01 to 2019-09. In overal, we can not identify trends or seasonality from this visualization, more tools are required.

```
In [17]: data[['Open', 'Close','Adj Close']].boxplot()
         #description on the aquired data
         data.describe()
```

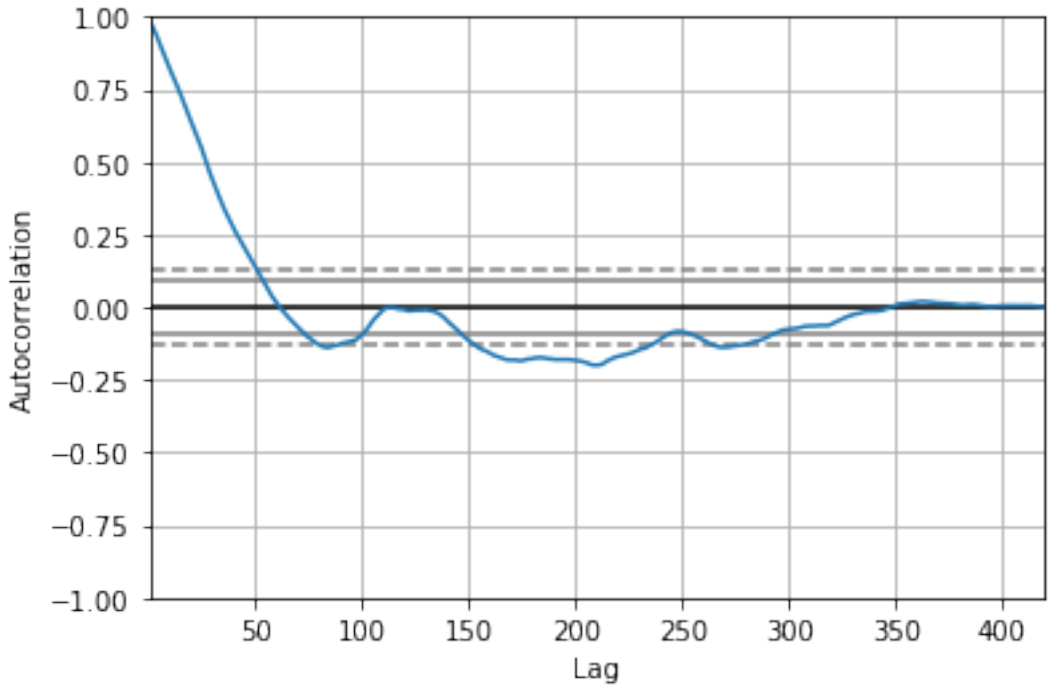
```
Out[17]:              High         Low  ...         Volume   Adj Close
         count  419.000000  419.000000  ...     419.000000  419.000000
         mean    12.625369   12.453980  ...   65384.248210   12.538449
         std      1.324000    1.367548  ...   95893.726514    1.345337
         min      9.400000    9.056000  ...       0.000000    9.115000
         25%     11.636100   11.393600  ...    9000.000000   11.516300
         50%     12.596000   12.498000  ...   32500.000000   12.564000
         75%     13.701000   13.508700  ...   94500.000000   13.570900
```

```
max     15.844000   15.550000  ...  920000.000000   15.814000

[8 rows x 6 columns]
```



The box plot demonstrates a normal distribution as the interquartile range box is located in the center.

```
In [18]: from pandas.plotting import autocorrelation_plot
         data_adj = data['Adj Close']
         autocorrelation_plot(data_adj)
         plt.show()
```



Since the data is known as a time series, an autocorrelation can help us understand whether the observations of a time series are positively correlated or negatively correlated efficiently. As shown above, the Adj Close prices are highly correlated with their previous values, the observations are statistically significant for lags up to 50, which further implies there the trends and seasonality in this time series.

3.1.3 Stationarity Transformation

As described in the previous section, the implementation of ARIMA forecasting model is based on the assumption of data being stationary. While obvious trends and seasonal effects can be discovered in data visualisation, it is essential to collect evidence of stationarity of the observations before we fit them into our model. On the other hand, if non-stationarity of the time series data is explored, we then need to transform the original data to obtain consistent mean and variance.

To check the stationarity of the time series, we have implemented Dickey-Fuller test to statistically summarize the behaviour of the data. By making strong assumptions on the data, Dickey-Fuller test can provide us the confidence level to determine whether or not the null hypothesis should be rejected.

As shown below, the null hypothesis of the test indicates that the time series is not stationary, while the alternate hypothesis indicates that the time series is stationary.

H0: not stationary, if failed to be rejected, the time series are suggested to be non-stationary, which means they are time dependent.

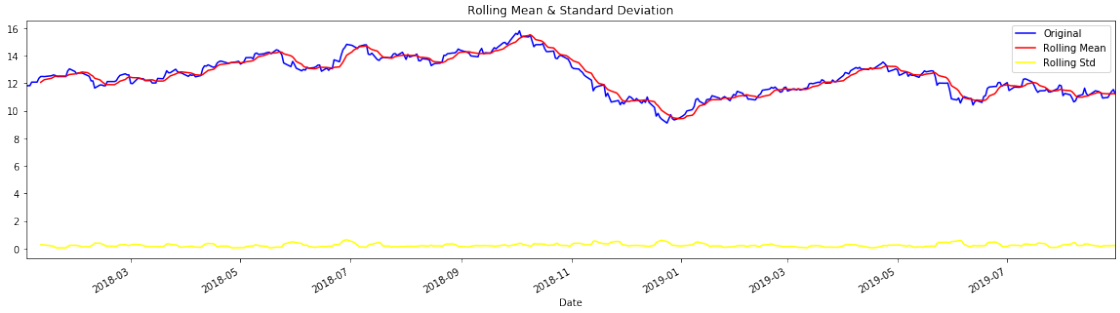
Ha: stationary, the null hypothesis is rejected, we have enough evidence to conclude that the time series are not time-dependent.

The results of Dickey-Fuller test can be interpreted using the p-value. As shown below, if the p-value is less than the critical values of 5% or 1%, the null hypothesis is likely to be rejected, or vice versa.

P-value > 0.05: failed to reject H0, the time series are suggested to be non-stationary
P-value <= 0.05: reject H0, we are 95% confident that the time series are stationary

```
In [0]: from statsmodels.tsa.stattools import adfuller
        ## Dicky-Fuller test to test stationarity
        def test_stationarity(data_adj):

            #Determing rolling statistics
            rolmean = data_adj.rolling(window=7).mean()
            rolstd = data_adj.rolling(window=7).std()

            #Plot rolling statistics:
            orig = data_adj.plot(color='blue',label='Original', figsize=(20,5))
            mean = rolmean.plot(color='red', label='Rolling Mean', figsize=(20,5))
            std = rolstd.plot(color='yellow', label = 'Rolling Std', figsize=(20,5))
            plt.legend(loc='best')
            plt.title('Rolling Mean & Standard Deviation')
            plt.show(block=False)

            #Perform Dickey-Fuller test:
            print('Results of Dickey-Fuller Test:')
            dftest = adfuller(data_adj, autolag='AIC')
            dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','
            for key,value in dftest[4].items():
                dfoutput['Critical Value (%s)'%key] = value
            print(dfoutput)
```

```
In [20]: test_stationarity(data_adj)
```




```
Results of Dickey-Fuller Test:
Test Statistic                  -1.650533
p-value                          0.456763
#Lags Used                       0.000000
Number of Observations Used    418.000000
Critical Value (1%)             -3.446091
Critical Value (5%)             -2.868479
```

```
Critical Value (10%)             -2.570466
dtype: float64
```

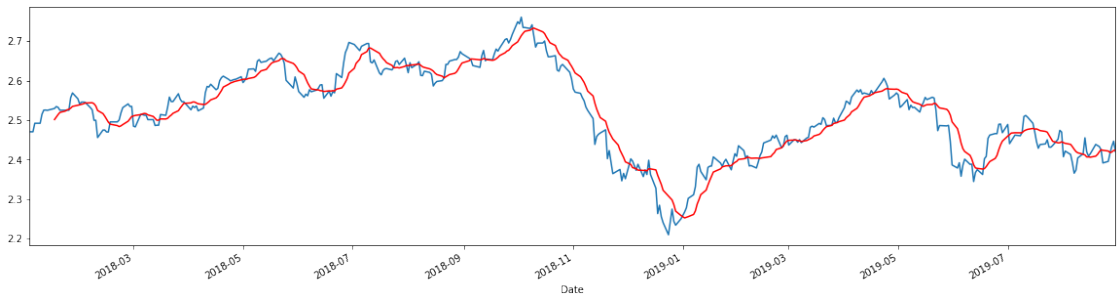
After performing Dickey-Fuller test on the time series, we observe that the p-value is 0.46 which is larger than 0.05. We do not have enough evidence to reject the null hypothesis therefore we fail to conclude that the time series is stationary.

To eliminate the trend in the time series, we decided to use **log transformation** to penalise high values more than lower values.

We then applied **smoothing** on the transformed data to estimate the trend by taking the rolling average.

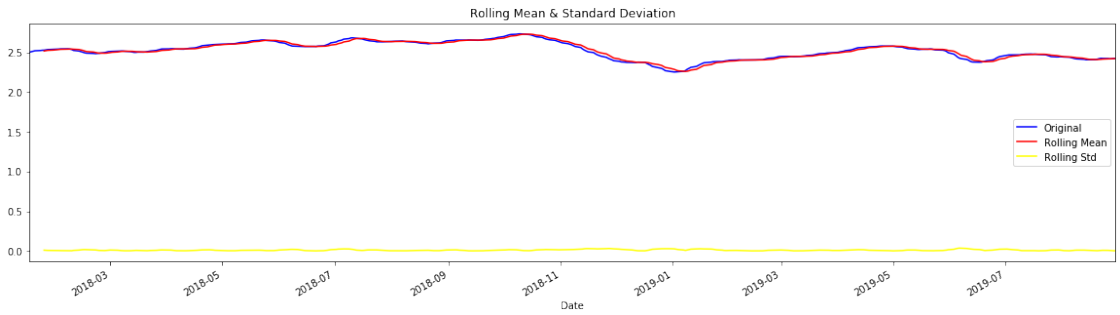
```
In [21]: #log transformation
         data_log = np.log(data_adj)
         #plot moving average
         mv_avg = data_log.rolling(window=10).mean()
         data_log.plot(figsize=(20,5))
         mv_avg.plot(color='red',figsize=(20,5))
```

```
Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x7f73d7322160>
```



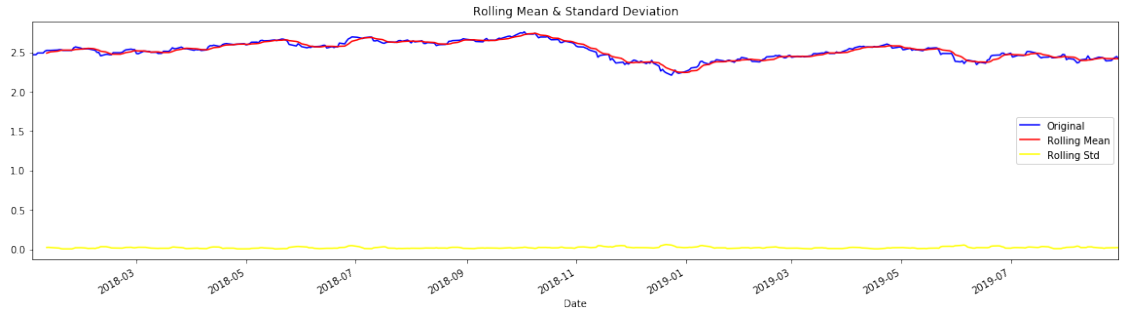
Note that we are taking the last 10 values to define the the moving average, the first 9 values of moving average are not defined. Therefore We need to drop the NaN values to perform a Dickey-Fuller test.

```
In [23]: mv_avg.dropna(inplace=True)
         test_stationarity(mv_avg)
```



```
Results of Dickey-Fuller Test:
Test Statistic                  -2.310523
p-value                          0.168597
#Lags Used                      11.000000
Number of Observations Used    398.000000
Critical Value (1%)             -3.446888
Critical Value (5%)             -2.868829
Critical Value (10%)            -2.570653
dtype: float64
```

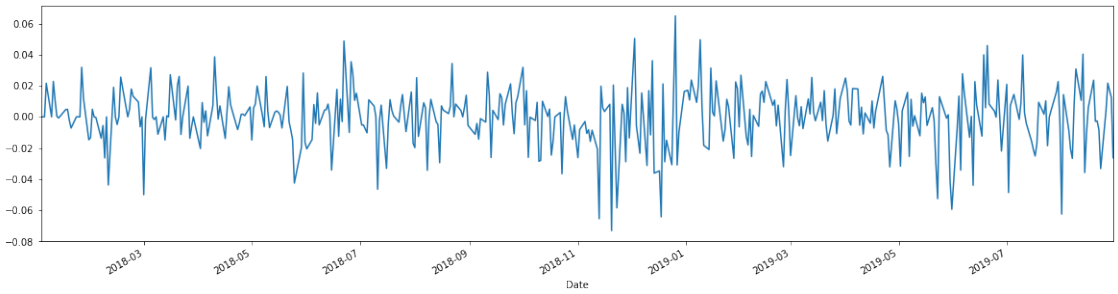
```
In [24]: test_stationarity(data_log)
```



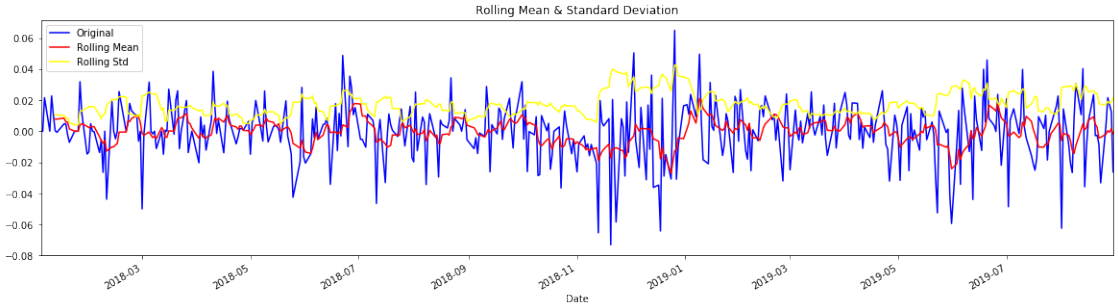

```
Results of Dickey-Fuller Test:
Test Statistic                  -1.700091
p-value                          0.431066
#Lags Used                       0.000000
Number of Observations Used    418.000000
Critical Value (1%)             -3.446091
Critical Value (5%)             -2.868479
Critical Value (10%)            -2.570466
dtype: float64
```

The results show that the p-value has decreased by 0.03 after log transformation.
To further transform the data, we used **differencing** to make the time series stationary.

```
In [25]: #Differencing
         data_log_diff = data_log - data_log.shift()
         data_log_diff.plot(figsize=(20,5))

Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x7f73d71bbc50>
```




```
In [26]: #Drop NaN values and test stationarity
         data_log_diff.dropna(inplace=True)
         test_stationarity(data_log_diff)
```




```
Results of Dickey-Fuller Test:
Test Statistic                  -20.978651
p-value                           0.000000
#Lags Used                        0.000000
Number of Observations Used     417.000000
Critical Value (1%)              -3.446129
Critical Value (5%)              -2.868496
Critical Value (10%)             -2.570475
dtype: float64
```

After performing first order differencing, the p-value became approximately 0. Hence we can conclude that we are confident to reject the null, and the time series is stationary.

3.1.4 Train-Test Spiliting

```
In [0]: X = data_log.values
        size = int(len(X) * 0.8)
        train_A, test_A = X[0:size], X[size:len(X)]
        history = [i for i in train_A]
        predictions = list()
```

3.1.5 Algorithm Implementation

To find the order of the **AR model** and the **MA model**, we need to determin the value of parameter **p** and **q** respectively. Since the p-value is less than 0.05 after the transformation, we reject the null hypothesis and conclude that the time series is stationary. We have used first order differencing technique once, hence the value of parameter **d** is 1.

To investigate the parameter of p and q, we applied ACF and PACF to find out the required lags and the lags of errors.

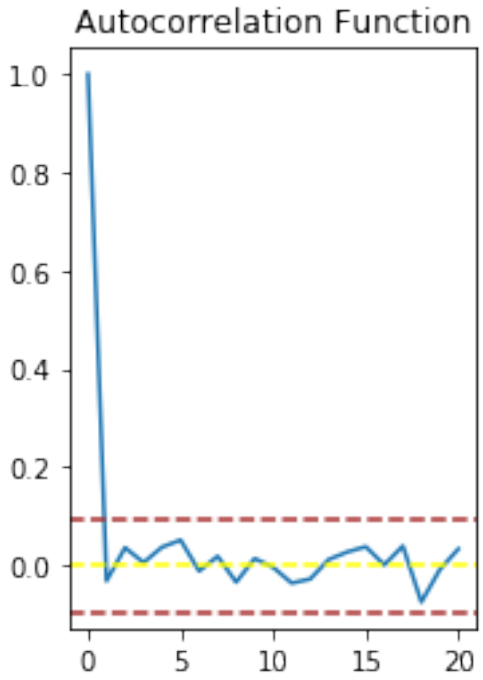
```
In [0]: #ACF and PACF plots:
        from statsmodels.tsa.stattools import acf, pacf
        from sklearn.metrics import mean_squared_error
        from math import sqrt
```

Autocorrelation Function (ACF)

```
In [28]: lag_acf = acf(data_log_diff, nlags=20)
         #Plot ACF:
         plt.subplot(121)
         plt.plot(lag_acf)
         plt.axhline(y=0,linestyle='--',color='yellow')
         plt.axhline(y=-1.96/np.sqrt(len(data_log_diff)),linestyle='--',color='brown')
         plt.axhline(y=1.96/np.sqrt(len(data_log_diff)),linestyle='--',color='brown')
         plt.title('Autocorrelation Function')
```

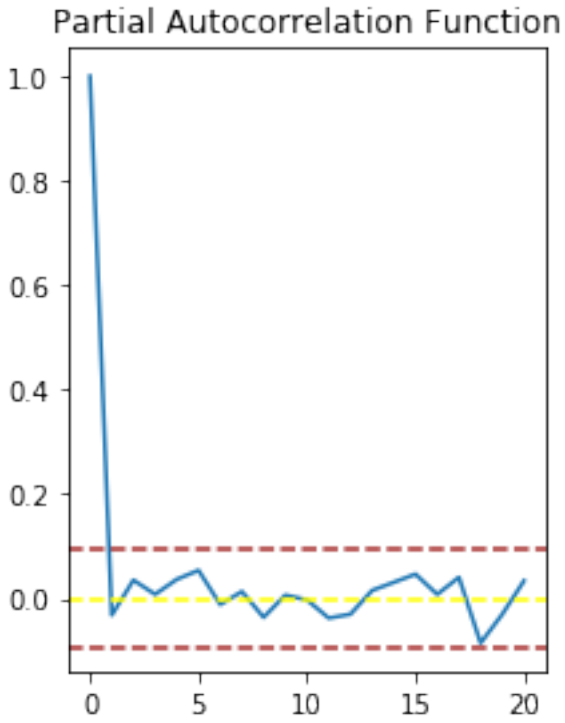
```
/usr/local/lib/python3.6/dist-packages/statsmodels/tsa/stattools.py:541: FutureWarning: fft=Tru
  warnings.warn(msg, FutureWarning)
```

```
Out[28]: Text(0.5, 1.0, 'Autocorrelation Function')
```



Partial Autocorrelation Function (PACF)

```
In [29]: lag_pacf = pacf(data_log_diff, nlags=20, method='ols')
         #Plot PACF:
         plt.subplot(122)
         plt.plot(lag_pacf)
         plt.axhline(y=0,linestyle='--',color='yellow')
         plt.axhline(y=-1.96/np.sqrt(len(data_log_diff)),linestyle='--',color='brown')
         plt.axhline(y=1.96/np.sqrt(len(data_log_diff)),linestyle='--',color='brown')
         plt.title('Partial Autocorrelation Function')
         plt.tight_layout()
```



The ACF and PACF plots for the time series after differencing are shown above, the lines in both plots are crossing the upper confidence interval at the lag value of 2 for the first time. Therefore we can set p and q as 2. Since we have a large amount of observations for ARIMA data, the autocorrelation lag (p) can be set to 4 conservatively.

3.1.6 Model Training and Building

The time series is split into training set and test set, the training set is fitted into the model, the prediction is each value on the test set.

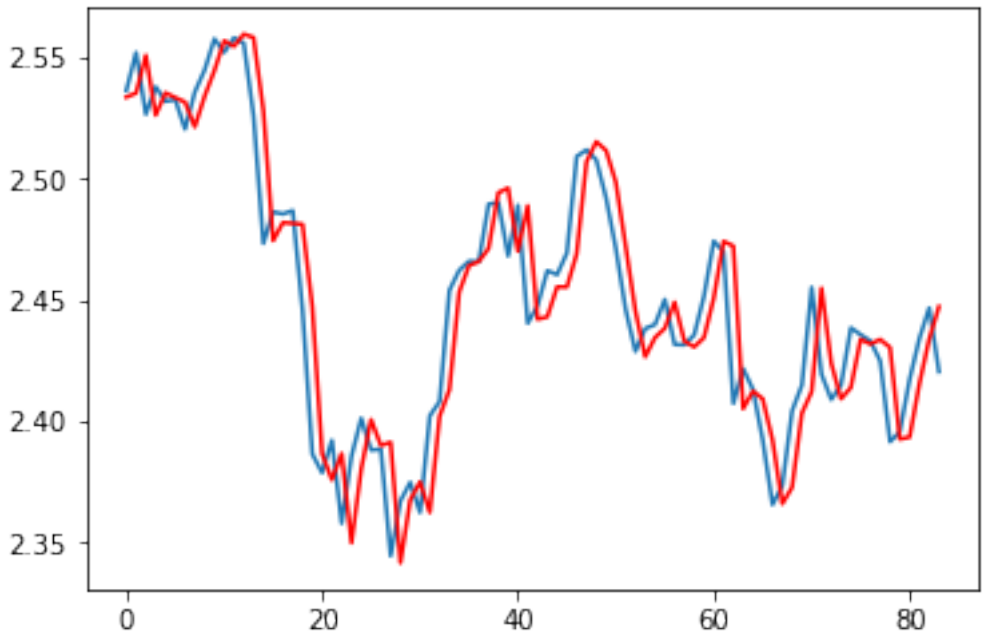
The model is defined and the parameters are passed by calling ARIMA(). The training data are fitted into the model by calling fit(). We used the forecast() function on the training set to predict the future time steps.

The indexes are from the start of the training set to predict the next value, which implies one-step prediction.

By calculating the Mean-Sqaure-Error, we can evaluate the performance of the model and complete comparative studies.

```
In [30]: X = data_log.values
         size = int(len(X) * 0.8)
         train_A, test_A = X[0:size], X[size:len(X)]
         history = [i for i in train_A]
         predictions = list()

         for sample in range(len(test_A)):
                 model = ARIMA(history, order=(4,1,2)) #p=2, d=1, q=2
                 model_fit = model.fit(disp=-1, dynamic=False)
                 output = model_fit.forecast()
                 pred = output[0]
                 predictions.append(pred)
                 obs = test_A[sample]
                 history.append(obs)
         error = mean_squared_error(test_A, predictions)
         print('Test MSE: %f' % error)
         plt.plot(test_A)
         plt.plot(predictions, color='red')
         plt.show()
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tsa/tsatools.py:668: RuntimeWarning: overflo
  newparams = ((1-np.exp(-params))/(1+np.exp(-params))).copy()
/usr/local/lib/python3.6/dist-packages/statsmodels/tsa/tsatools.py:668: RuntimeWarning: invalid
  newparams = ((1-np.exp(-params))/(1+np.exp(-params))).copy()
/usr/local/lib/python3.6/dist-packages/statsmodels/tsa/tsatools.py:669: RuntimeWarning: overflo
  tmp = ((1-np.exp(-params))/(1+np.exp(-params))).copy()
/usr/local/lib/python3.6/dist-packages/statsmodels/tsa/tsatools.py:669: RuntimeWarning: invalid
  tmp = ((1-np.exp(-params))/(1+np.exp(-params))).copy()
/usr/local/lib/python3.6/dist-packages/statsmodels/base/model.py:492: HessianInversionWarning:
  'available', HessianInversionWarning)
/usr/local/lib/python3.6/dist-packages/statsmodels/base/model.py:492: HessianInversionWarning:
  'available', HessianInversionWarning)


Test MSE: 0.000525
```



After setting dynamic=False, the in-sample lagged values are utilized for forecasting, which means the model gets trained up until the previous value to predict the next value. This can be the reason why the fitted forecast values and underground truth are very close.

However, we can not conclude that the model has outstanding performance as at this point we haven't actually predicted the future values and compared the forecast with the actual values. Moreover, the model is possibly overfitted.

The next section will describe validation on ARIMA model, and introduce our solution to detecting overfitting.

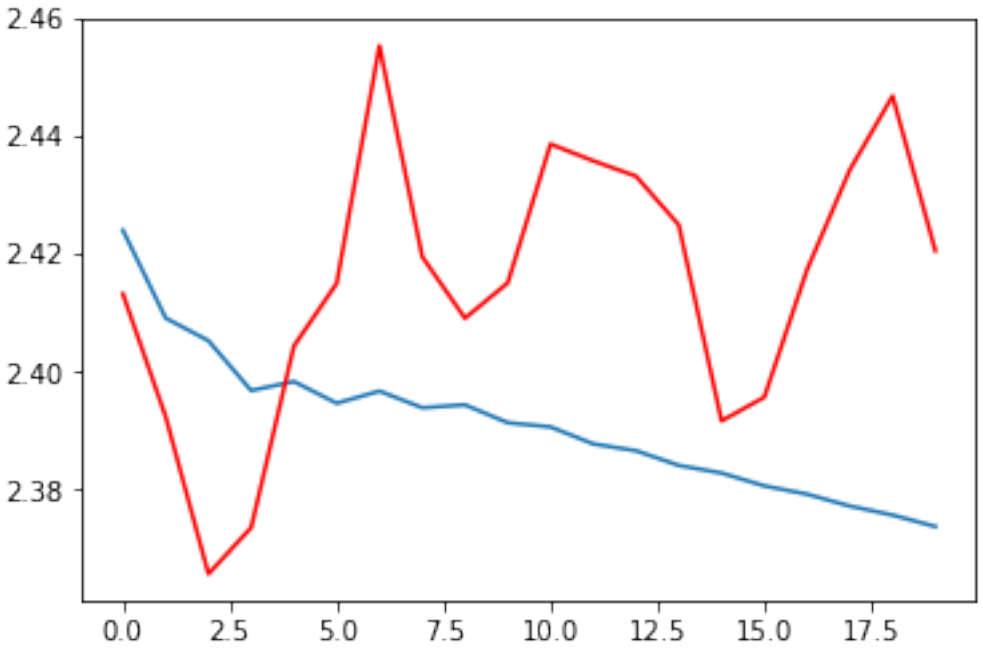
3.1.7 Validation

To validate ARIMA algorithm, instead of using all our training set to predict the next value, we use existing training data to predict the values of the next 20 days and compare them with true values.

```
In [0]: train_A = X[:399]
        Test_A = X[399:]
        model_val = ARIMA(train_A, order=(4,1,2))
        fit = model_val.fit(disp=-1)
        fc, se, conf = fitted.forecast(20, alpha=0.05)
        train_A = pd.DataFrame(train_A)
        fc = pd.DataFrame(fc)
        fc_ori = np.exp(fc)
        tr_ori = np.exp(train_A)
        fc_ori.index = ['400','401','402','403','404','405','406','407','408','409','410','411
        tr_y = tr_ori.append(fc_ori)
```

```
        error = mean_squared_error(fc, Test_A)
        print('Test MSE: %f' % error)
        plt.plot(fc)
        plt.plot(Test_A, color='red')
        plt.show()
```

```
Test MSE: 0.001424
```



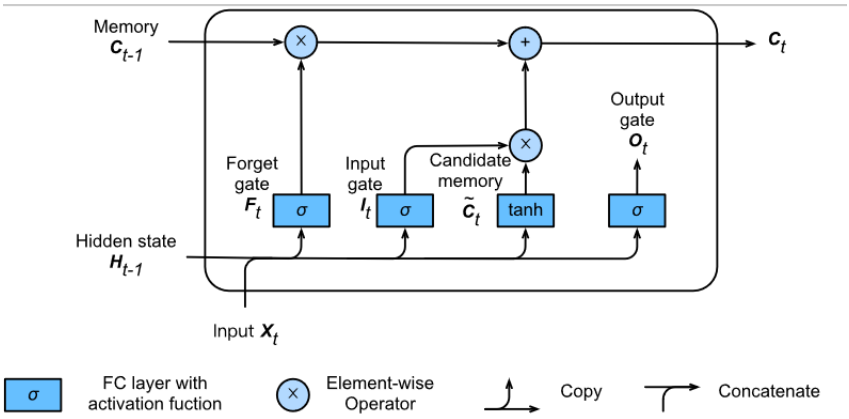
As the diagram shows above, in reality, ARIMA model only features the patterns and the trends on the previous observations. Due to the uncertainty of the time series, the long-term forecasting is highly innacurate.

3.2 LSTM

3.2.1 Introduction to LSTM

LSTM(Long-Short-Term-Memory) is a type of Recurrent Neural Networks, which is based on gradient descent. Compared to RNNs, LSTM model can remember longer sequences of inputs by using three types of "gates". Also, it can the problem of vanishing and exploding gradient that may occur in RNNs. In this section, the model of LSTM is introduced for forecasting the Adj closing price.

One LSTM cell, shown in the above diagram, is composed of three gates that are forget gate, input gate and output gate. The diagram illustrates that memory state from the previous timestep Ct-1, hidden state from the previous timestep Ht-1 and the current timestep input Xt are inputs of the LSTM cell.



LSTM cell

Step 1

Firstly, Ht-1 and Xt will be passed through the forget gate that will squish values between 0 and 1 based on $F_t = sigmoid(W_f[H_{t-1}, X_t] + b_f)$. Then, Ft will be multiplied by Ct-1, if Ft = 0, LSTM will "forget the previous memory state", while if Ft = 1, it will "remember everything".

Step 2

Next, we will pass the Ht-1 and Xt through an input gate that has a sigmoid function and a tanh function in parallel. Next, the results of both operations will be combined to decide what new information will be retained in the new memory state.

Step 3

We will add the ouptput from step 1 and step 2 together to update the new meomory state Ct. The formula is $C_t = F_t * C_{t-1} + I_t * \hat{C}$.

Step 4

In this step, we will update the hidden state and output of this timestep. Ht-1 and Xt will be passed through the output gate that has a sigmoid function. Also, it will be multiplied by $tanh(C_{t-1})$. The output of this operation will be the new hidden state Ht and output of this timestep.

Finally, the new memory state Ct (output of step 3) and hidden state Ht (output of step 4) will be carried over to the next timestep. In conclusion, the LSTM can optionally remember the information from previous state, which makes it remember long-term information and avoid both vanishing and exploding gradient problems.

Data prepocessing, LSTM model building and traning and Tune hyperparameters of LSTM model are shown in this section.

3.2.2 Data Preprocessing

In this section, we load data to pandas dataframe, and rescale these data between 0 and 1 to make it easier to learn. In addition, to tune the hyperparameters of models and compare different algorithm, we divide the stock price dataset into three parts, which are training, validation and testing dataset. Training dataset is a dataset for model training, which includes 80% of stock price data. We use validation dataset to find the optimal setting of models. Also, the testing dataset is used for evaluating different algorithms.

Load Data

```
In [0]: # import libraries
        import torch
        import torch.nn as nn
        import torch.optim as optim
        import pandas_datareader as web
        import matplotlib.pyplot as plt
        from sklearn.preprocessing import MinMaxScaler
        import numpy as np
        import pandas as pd
        %matplotlib inline
```

```
In [0]: # Load data to pandas DataFrame
        data = web.data.get_data_yahoo('oil', start='2018-01-02', end='2019-09-02')
```

Min-Max Normalization Since the Adj close feature values have different ranges, we use Min-Max normalization to change values to a common scale(0-1).

```
In [0]: Data_Norm = data['Adj Close']
        Data_Norm = Data_Norm.values
        Data_Norm = Data_Norm.reshape(-1,1)
        Data_Norm.shape
```

```
Out[0]: (419, 1)
```

```
In [0]: minmaxscaler = MinMaxScaler()
        Data_scale = minmaxscaler.fit_transform(Data_Norm)
```

Train-Test spliting

```
In [0]: # 80% data for training
        train_Perc = 0.8
        total_rows = Data_scale.shape[0]
        spliting_index = int(train_Perc * total_rows)
        train = Data_scale[0:spliting_index,:]
        test = Data_scale[spliting_index:,:]
        train.shape, test.shape
```

```
Out[0]: ((335, 1), (84, 1))
```

```
In [0]: timesteps = 7
        x_train = []
        y_train = []
        for i in range(timesteps, len(train)):
          x_train.append(train[i-timesteps:i,0])
          y_train.append(train[i,0])

        x_train = torch.FloatTensor(x_train)
        y_train = torch.FloatTensor(y_train)
        x_train.shape, y_train.shape
```

```
Out[0]: (torch.Size([328, 7]), torch.Size([328]))

In [0]: # Reshape x_train to the format of RNN input:[sequence_length, batch_size, input_size]
        x_train = x_train.view(-1, 1, timesteps)
        y_train = y_train.unsqueeze(dim=1)
        # move data from cpu to gpu
        x_train, y_train = x_train.to('cuda'), y_train.to('cuda')
        x_train.shape,y_train.shape

Out[0]: (torch.Size([328, 1, 7]), torch.Size([328, 1]))

In [0]: # validation and test dataset
        x_valid = []
        y_valid = []
        for i in range(timesteps, len(test)):
          if i < len(test)*0.5 :
            x_valid.append(test[i-timesteps:i,0])
            y_valid.append(test[i,0])
          else:
            Test = test[i-timesteps:,0]
            break


        x_valid, y_valid = torch.FloatTensor(x_valid).to('cuda'), torch.FloatTensor(y_valid).to
        x_valid = x_valid.view(-1, 1, timesteps)
        y_valid = y_valid.unsqueeze(dim=1)
        x_valid.shape,y_valid.shape

Out[0]: (torch.Size([35, 1, 7]), torch.Size([35, 1]))
```

3.2.3 LSTM Model Building and Training

LSTM model building, training, testing and different hyperparameters comparison will be shown in the following sections.

we build a LSTM model(one LSTM cell connected to one linear regression layer) in pytorch and use GPU to train it.

```
In [0]: """
        sequech_length: length of time series
        input_size: the size of one input
        hidden_size: memory size for c,h
        input x:[sequence_length, batch_size, input_size]
        c/h : [num_layer, batch_size, hidden_size]
        lstm_output:[sequence_length, batch_size, hidden_size]

        """
        class NetWork(nn.Module):

          def __init__(self, input_size, hidden_size, num_layers, output_size=1):
```

```
        super(NetWork, self).__init__()
        self.hidden_size = hidden_size
        self.input_size = input_size
        self.output_size = output_size
        self.num_layers = num_layers

        self.LSTM = nn.LSTM(self.input_size, self.hidden_size, num_layers=self.num_layers)
        self.FC = nn.Linear(self.hidden_size, self.output_size)
        self.dropout = nn.Dropout(0.3)

    def forward(self, input):
        # input:[sequence_length,batch_size,input_size] -> lstm_output:[sequence_length, b
        lstm_output,(h,c) = self.LSTM(input)
        # run dropout layer, if model is overfitting
        # lstm_out = self.dropout(lstm_out)
        # reduce the dimension of lstm_output to 2D
        # [sequence_length * batch_size, hidden_size] -> output: [sequence_length * batch_
        output = self.FC(lstm_output.view(-1,self.hidden_size))
        return output
```

The LSTM model is trained on the training dataset using gradient descent method. Moreover, we passed seven consecutive days of stock prices data through the LSTM model to predict eighth day price, then calculate the mean square error between prediction and corresponding target to update weights by using back propagation algorithm. The data feeding progress is shown in the following part. > day 1,2,3,4,5,6,7 –> day 8

day 2,3,4,5,6,7,8 –> day 9

day 3,4,5,6,7,8,9 –> day 10,etc.

```
In [0]: # wrap up model training and testing to a function
        # in order to tune hyperparameters of lstm model
        def model_training(learning_rate, epochs, x_train, y_train, model, *test, Print=True,E
            """
            Train the model with x_train, y_train
            if Evaluation is True, test the trained model in testing dataset
            """

            # mean square error as criterion to update weights
            criterion = torch.nn.MSELoss().to('cuda')
            # use Adam optimization algorithm
            optimizer = torch.optim.Adam(params=model.parameters(), lr=learning_rate)
            model.train()
            train_loss = np.zeros(epochs)

            for i in range(epochs):
                out = model(x_train)
```

```
            # calculate loss
            loss = criterion(out, y_train)
            optimizer.zero_grad()
            # back propagation
            loss.backward()
            # update weights
            optimizer.step()
            train_loss[i] = loss
            if (i+1) % 100 == 0 and Print:
              print('Epoch: {}, Train Loss:{:.5f}'.format(i+1, loss.item()))

        if KeepModel:
          return model

        if Evaluation:
          model.eval()
          x_test = test[0]
          y_test = test[1]
          pred = model(x_valid)
          test_loss = criterion(pred, y_valid)
          print("Test Loss: {:.5f}".format(test_loss.item()))
          return train_loss, test_loss, pred

        else:
          return train_loss
```

```
In [0]: # train the model (input_size=7, hidden_size=16, lr=0.001, epochs= 500)
        model=NetWork(input_size=7, hidden_size=16, num_layers=1).to('cuda')
        train_loss = model_training(learning_rate=0.001, epochs=500, x_train=x_train, y_train=y
```

```
Epoch: 100, Train Loss:0.00939
Epoch: 200, Train Loss:0.00545
Epoch: 300, Train Loss:0.00431
Epoch: 400, Train Loss:0.00325
Epoch: 500, Train Loss:0.00259
```

3.2.4 Tune hyperparameters of LSTM model

In this part, we compare the performance of the LSTM model trained by different hyperparameters to get an optimal model setting. We apply MSE(mean square error) between model predictions on validation dataset and corresponding labels as the criterion to evaluate the performance of LSTM models. In addition, we use pyplot function to visualize training and validation losses of different models, which can illustrate which model has the best performance and if any model is overfitting or underfitting.

```
In [0]: # compare the performance of model trained by different hyperparameters
        def parameters_comparsion(num_layers, hidden_size, learning_rate, input_size, epochs, x
          # learing_rate and hidden_size should be stored in list
```

```
    list = [[a,b] for a in num_layers for b in hidden_size]
    train_loss = []
    valid_loss = []
    pred = []
    for n, h in list:
      model=NetWork(input_size=input_size, hidden_size=h, num_layers=n).to('cuda')
      # train loss, validation loss, predication
      a,b,c = model_training(learning_rate, epochs, x_train, y_train, model, Print=p, Eva
      b = b.to('cpu').data.numpy()
      c = c.to('cpu').data.numpy()
      train_loss.append(a)
      valid_loss.append(b)
      pred.append(c)

    return train_loss, valid_loss, pred
```

```
In [0]: # different learning rate and hidden size
        num_layers= [1, 2]
        hidden_size = [4, 8, 16]
```

```
In [0]: # train and test models with different hyperparameters
        train_loss, valid_loss, pred = parameters_comparsion(learning_rate=0.001, epochs=500, 
                                                             input_size=7, hidden_size=hidden_s
```

```
Test Loss: 0.02630
Test Loss: 0.00995
Test Loss: 0.01248
Test Loss: 0.01449
Test Loss: 0.00483
Test Loss: 0.00631
```

```
In [0]: train_loss = np.array(train_loss)
        valid_loss = np.array(valid_loss)
        pred = np.array(pred)
        pred = pred.reshape(pred.shape[0],pred.shape[1])
        train_loss.shape, valid_loss.shape, pred.shape
```

```
Out[0]: ((6, 500), (6,), (6, 35))
```

```
In [0]: """
        M00: num_layer=1, hidden_size=4, M01: num_layer=1, hidden_size=8,
        M02: num_layer=1, hidden_size=16, M10: num_layer=2, hidden_size=4,
        M11: num_layer=2, hidden_size=8, M12: num_layer=2, hidden_size=16

        """

        Train_loss = pd.DataFrame({'M00': train_loss[0, :], 'M01': train_loss[1, :],'M02': tra
```

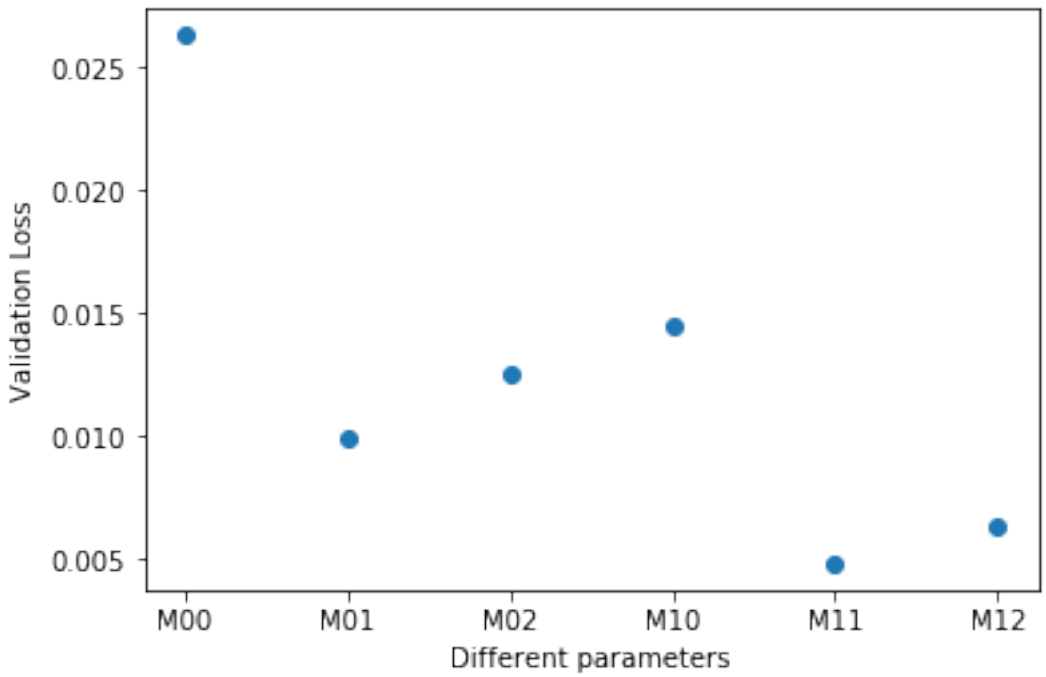
```
                'M10': train_loss[3, :], 'M11': train_loss[4, :], 'M12': train_loss[5, :]}
Train_loss.describe()
```

Out[0]:

	M00	M01	M02	M10	M11	M12
count	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000
mean	0.239991	0.020990	0.020518	0.031472	0.010737	0.011805
std	0.407858	0.052958	0.052943	0.055628	0.015122	0.027166
min	0.008151	0.003781	0.003925	0.005425	0.001818	0.002281
25%	0.009263	0.004486	0.004934	0.006958	0.002608	0.003098
50%	0.011026	0.005193	0.006478	0.010081	0.004007	0.004170
75%	0.288273	0.006035	0.009781	0.019656	0.009554	0.007513
max	1.437464	0.323139	0.395853	0.279811	0.079455	0.218230

In [0]:
```
# plot the validation losses of models with different hyperparameters
plt.plot(['M00','M01','M02','M10','M11','M12'],valid_loss,'o')
plt.ylabel('Validation Loss')
plt.xlabel('Different parameters')
```

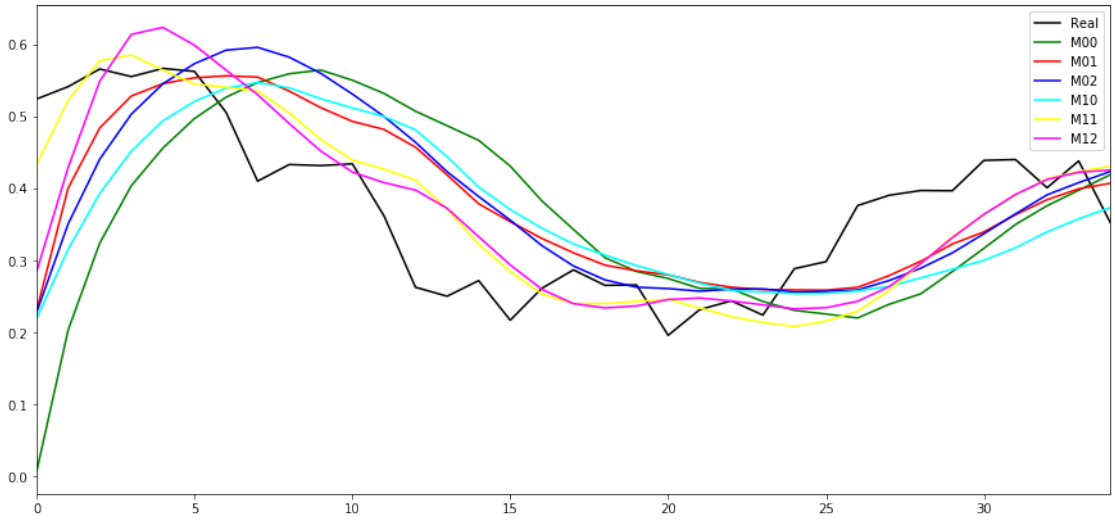
Out[0]: Text(0.5, 0, 'Different parameters')



In [0]:
```
# plot the predictions of models with different hyperparameters on validation dataset
y_validation = y_valid.to('cpu').data.numpy()
y_validation = y_validation.flatten()

Prediction = pd.DataFrame({'Real':y_validation, 'M00': pred[0, :], 'M01': pred[1, :],'M
```

```
                    'M10': pred[3, :], 'M11': pred[4, :], 'M12': pred[5, :]})
Prediction.plot(figsize=(15,7),color=['black','green','red','blue','cyan','yellow','mag
```

```
Out[0]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa92992f4e0>
```



In conclusion, according to the above plot figures about train and validation losses of different hyperparameters, it is clear that MO2 (num_layer=2, hidden_size=8) has the lowest loss among the model family. Therefore, we choose one as the number of layer of the lstm cell and thirty two as the size of the hidden state.

4 Evaluation

In this section, we evaluate the performance of LSTM model and ARIMA model from two aspects that are **efficiency** and **MSE**(mean square error) with underground truth. For the efficiency, we evaluate the two algorithms from a computational point of view by counting the time of each algorithm execution including model training and testing. For the model accuracy evaluation, we pass the testing dataset though both models. Moreover, we use the trained model to predict 7-day future stock prices, then compute the MSE between future predictions and underground truth, which is not passed through the model.

The process of the future prediction is shown as follows. First, We feed seven-day real data to the model. Next, the model will predict the value of day 8, then discard day 1 data from the data list and insert the day 8 prediction to it as the next input. We will get seven-day predictions by repeating the above process seven times. Therefore, it is clear that the more days the forecast is, the less the accuracy. However, there is a difference between ARIMA model and LSTM model. ARIMA will remain the previous data instead of discarding it, critically beacuse the ARIMA model is trying to find a trend among the dataset, not doing a 7 to 1 prediction.

t1, t2, t3, t4, t5, t6, t7 –> p8

t2, t3, t4, t5, t6, t7, p8 –> p9

t3, t4, t5, t6, t7, p8, p9 –> p10,etc.

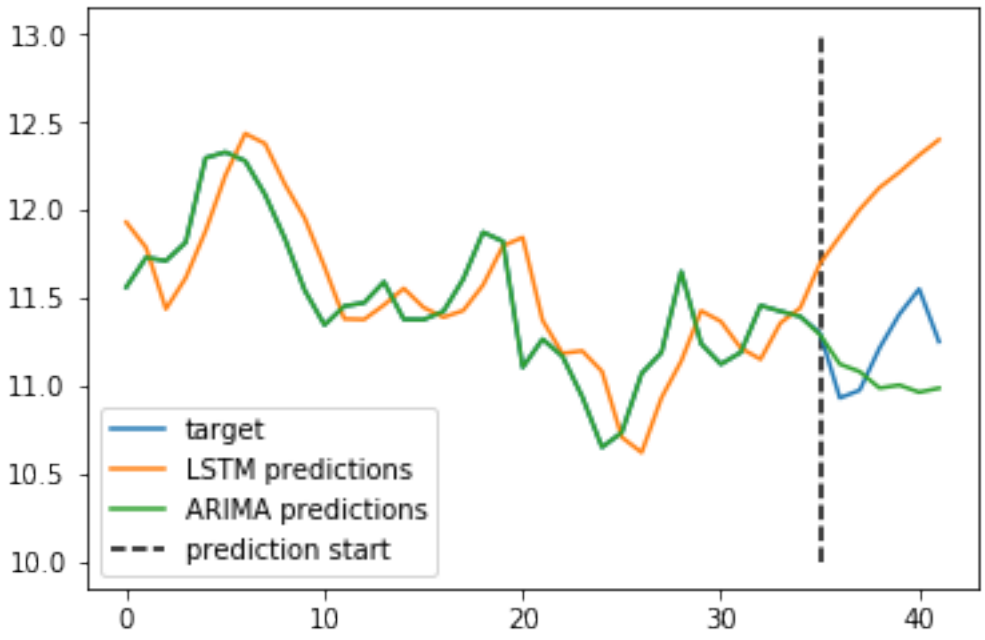
```
In [0]: # LSTM seven-day stock price predictions
        %%time
        model=NetWork(input_size=7, hidden_size=8, num_layers=2).to('cuda')
        Model = model_training(learning_rate=0.01, epochs=500, x_train=x_train, y_train=y_trai
        test_input = Test[0:42].reshape(-1,1)
        target = Test[7:]
        input1 = []
        # one-step ahead predictions
        for i in range(7, len(test_input)):
          input1.append(test_input[i-7:i,0])
        input1 = torch.FloatTensor(input1).to('cuda')
        input1 = input1.view(-1,1,7)
        # convert to evaluation pattern(no gradient)
        model.eval()
        preds1 = Model(input1)
        a = torch.FloatTensor(test_input[35:-1]).reshape(1,1,6).to('cuda')
        b = preds1[-1,:].reshape(1,1,1)
        input2 = torch.cat((a,b),dim=2)

        # seven-day future predictions
        predictions = []
        for i in range(7):
         preds2 = Model(input2[:,:,i:])
         predictions.append(preds2.item())
         input2 = torch.cat((input2,preds2.view(1,1,1)),dim=2)
        preds1 = preds1.view(-1).to('cpu').data.numpy()
        predictions = np.array(predictions)
        prediction_LSTM = np.concatenate((preds1,predictions))
        prediction_LSTM = minmaxscaler.inverse_transform(prediction_LSTM.reshape(-1,1))
        targets = minmaxscaler.inverse_transform(target.reshape(-1,1))

CPU times: user 20.2 s, sys: 8.73 s, total: 28.9 s
Wall time: 29 s


In [0]: #ARIMA seven-day stock price prediction
        %%time
        tr = X[370:412]
        model = ARIMA(tr, order=(4,1,2))
        fitted = model.fit(disp=-1)
        fc, se, conf = fitted.forecast(7, alpha=0.05)
        tr = pd.DataFrame(tr)
        fc = pd.DataFrame(fc)
        fc_ori = np.exp(fc)
        tr_ori = np.exp(tr)
        fc_ori.index = ['42','43','44','45','46','47','48']
        tr_y = tr_ori.append(fc_ori)
```

```
CPU times: user 750 ms, sys: 11.8 ms, total: 762 ms
Wall time: 761 ms
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/base/model.py:492: HessianInversionWarning:
  'available', HessianInversionWarning)
```

Although LSTM is much more computationally expensive than ARIMA algorithm, the following diagram illustrates that the ARIMA model has a better performance for a seven-day short term stock price prediction than the LSTM model. This is mainly because our project tries to predict seven-day stock prices by analysing a small size of dataset. Therefore, we have chosen the ARIMA as the algorithm for the stock price prediction. However, both algorithms could not predict the exact changes in stock price(when stock price rise and fall) because of the natural complexity of stock price prediction.

```
In [0]: y = tr_y[7:].to_numpy()
        plt.plot(targets)
        plt.plot(prediction_LSTM)
        plt.plot(y)
        plt.vlines(35,10,13,linestyles='dashed')
        plt.legend(['target','LSTM predictions', 'ARIMA predictions', 'prediction start'])
        plt.show()
        error = mean_squared_error(targets[35:], prediction_LSTM[35:])
        error_arima = mean_squared_error(targets[35:], y[35:])
        print("LSTM MSE:",error)
        print("ARIMA MSE:",error_arima)
```



```
LSTM MSE: 0.7762323831610406
ARIMA MSE: 0.09653109125534878
```

5 Conclusion

This report shows a procedure of applying machine learning techniques to perform time series forecasting. We extracted the dataset of the oil price index of the last two years from Yahoo API to perform a short term prediction of the next seven days. We focused on Adjcent Close price as it best describes the closing price for each day.

To find the optimal algorithm for this topic, we have compared two time series algorithms (LSTM and ARIMA) using the Mean Square Error as the evaluation criterion. Due to the non-linearity of the activation functions, LSTM model takes much longer training time than ARIMA model, although we have embedded GPU to implement LSTM model in Pytorch to make it more time-efficient. Also, due to the training dataset we used that is a small amount of data, LSTMs could not learn the nonlinearity of it. However, it is easier for ARMA to find the coefficients with fewer computations. Hence, we chose to use ARIMA model to predict the stock price. However, the ARIMA we trained could not predict the exact changes in stock price(when stock price rise and fall). Therefore, it can be argued that the current ARIMA solution could not help an individual or company to make profits. In conclusion, for the time series prediction, the models can barely predict the exact truth, there are just merely approximations towards reality.

For future improvement, we could increase the number of timestep from 7 to 30. Also, replacing LSTM with GRU could be enhanced by future studies.

6 Ethical Aspects

When it comes to data-based decision making, there is no doubt that machine learning plays an important role in financial analysis. While performing machine learning algorithms on financial data, it is essential that one to straddle the confluence of math/statistics, programming skills and also financial domain expertise. Lack of any required knowledge or incautious will lead to failure in time series prediction or capital losses.

According to Utilitarian Ethical Approach, an action should be assessed on the consequences and outcome. Although our initial motivation is to apply machine learning algorithms to achieve capital gain, data scientists in many companies and financial institutions have utilized powerful machine learning techniques to make a smarter decision. While a successful discovery can generate benefits or capital gains for stakeholders, a false discovery can also cause harm on a large scale.

With unnatural behaviors and physiological factors in time series data, machine learning techniques have not been fully accepted in its current state. While it is one of the biggest concerns that human labor would be replaced by automation technologies due to their rapid development, human performance is still the most important factor that is driving the changes in the stock market. Financial analysts still play a major role in decision making and they should not attain less credibility than machine learning algorithms do. On the other hand, Kantian's principle of universalizability requires that, for an action to be permissible, it must be possible to apply it to all people without a contradiction occurring. A successful machine learning implementation is likely to be applied in large companies/organizations, while the purpose of doing so is to generate capital gains, in terms of economics, the inflation rate will accelerate and eventually, the gap between

the poor and the rich will increase. Therefore, machine learning applied in financial analysis is controversial and individuals/stakeholders should take careful consideration before applying it in the real world.