Configuration Manual

MSc Research Project
Data Analytics

Mrunali More
Student ID: x18189059

School of Computing
National College of Ireland

Supervisor: Dr. Muhammad Iqbal

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Mrunali More		
Student ID:	X18189059		
Programme:	Data Analytics	**Year:**	2020
Module:	MSc Research Project		
Lecturer:	Dr. Muhammad Iqbal		
Submission Due Date:	17/08/2020		
Project Title:	ANALYZING THE IMPACT OF MULTIPLE STOCK INDICES IN PREDICTION OF US DOLLAR INDEX		
Word Count:	1170 **Page Count:** 14		

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: ....

Date: 17/08/2020

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Mrunali More
x18189059
MSc Research Project in Data Analytics

1 Introduction

This configuration manual outlines the details about hardware, software specifications and programming steps required for the implementation of the research project "ANALYZING THE IMPACT OF MULTIPLE STOCK INDICES IN PREDICTION US DOLLAR INDEX."

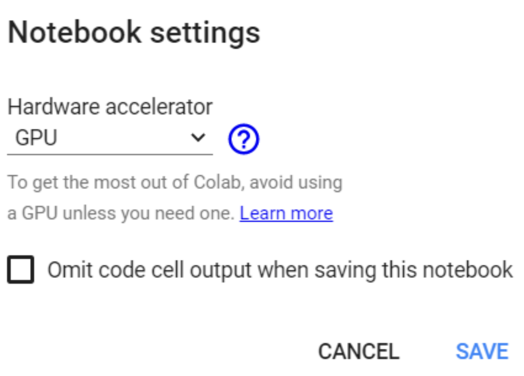
2 System Configurations

2.1 Hardware

- Processor: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz
- RAM: 8 GB
- System Type: Windows 10 64-bit Operating System, x64-based processor
- GPU: Intel(R) UHD Graphics 620, 4GB and NVIDIA GeForce MX250, 2GB
- Storage: 512GB SSD

2.2 Software

- **Google Colaboratory**
  Google Colaboratory, also called as *"Colab"* similar to Jupyter Notebook, enables the user to write and execute python code using free cloud services and GPU & TPU services, specifically build for Machine Learning and Data Analysis projects. In this project, colab is used for data download, data pre-processing and transformation, modelling, evaluation and visualization of predicted results. The GPU services can be enabled from Runtime -> Change runtime type -> Hardware accelerator -> GPU.



3 Project Development

3.1 Data Collection

The data of the US dollar index and all four stock indices (NASDAQ, NYSE, S&P and Dow Jones Industrial Average-DJIA) is gathered from Yahoo Finance. The data of Yahoo Finance can be directly downloaded using the library 'pandas_datareader' as shown in Figure 1. The data is retrieved as a pandas data frame containing six variables and date as an index of the data frame.

```
## Download data of US Dollar Index
import pandas_datareader as web
dfUSD = web.DataReader('DX-Y.NYB', data_source='yahoo',start='1990-01-01',end='2020-01-01')
dfUSD.head()
```

Date	High	Low	Open	Close	Volume	Adj Close
1990-01-01	93.309998	93.080002	93.190002	93.209999	0.0	93.209999
1990-01-02	94.309998	93.080002	93.129997	94.290001	0.0	94.290001
1990-01-03	94.519997	94.080002	94.150002	94.419998	0.0	94.419998
1990-01-04	93.879997	92.389999	93.720001	92.519997	0.0	92.519997
1990-01-05	93.419998	92.550003	93.339996	92.849998	0.0	92.849998

Figure 1. Data Retrival using pandas_datareader

3.2 Data Pre-processing

3.2.1 Merging of Data

Pre-processing of data involves removing missing values and merging of datasets, as shown in Figure 2. US dollar dataset contains a total of 7636 rows, and all stock indices data contains 7559 rows.

First, the data of the US dollar is stored in a new data frame, which contains values present for dates common for both US dollar index and stock indices. Then the data which is not present in the stock index but available in the US dollar index is stored in a new data frame, and all the absent values are removed from stock indices data.

Finally, the data with only close prices of all datasets are stored in a new data frame named *finaldata*.

```
[17] # finding values present in both US dollar index and stock index
     newdfUSD = dfUSD[(dfUSD.index).isin(dfnasdaq.index)]
```

```
[18] # finding values absent in both US dollar index and stock index
     absent = dfdaia[~(dfdaia.index).isin(newdfUSD.index)]
```

```
[19] #Removing all the absent values from Stock prices dataset
     newdfnasdaq = dfnasdaq[~(dfnasdaq.index).isin(absent.index)]
     newdfdaia = dfdaia[~(dfdaia.index).isin(absent.index)]
     newdfnyse = dfnyse[~(dfnyse.index).isin(absent.index)]
     newdfsp = dfsp[~(dfsp.index).isin(absent.index)]
```

```
#Merging of dataset
# Working on only close values

finaldata = pd.DataFrame(columns=['USD','NASDAQ','DJAI','NYSE','S&P'])

finaldata['USD'] = newdfUSD['Close']
finaldata['NASDAQ'] = newdfnasdaq['Close']
finaldata['DJAI'] = newdfdaia['Close']
finaldata['NYSE'] = newdfnyse['Close']
finaldata['S&P'] = newdfsp['Close']

finaldata.head()
```

Date	USD	NASDAQ	DJAI	NYSE	S&P
1990-01-02	94.290001	459.299988	2810.149902	2093.600098	359.690002
1990-01-03	94.419998	460.899994	2809.729980	2091.479980	358.760010

Figure 2. Removing missing values and Merging of data

3.3 Data Transformation

3.3.1 Data Decomposition

Decomposition of time series shows the seasonality, trend and white noise present in data. As shown in Figure 3., 'seasonal_decomposition' package from 'statsmodel' library is used for the decomposition of the US dollar index.

```
#decomposition
from statsmodels.tsa.seasonal import seasonal_decompose
data = finaldata['USD'] # US dollar prices
decomp = seasonal_decompose(x=data, model='additive', freq = 365)# decomposition with additive seasonality and freq=365 because data is daily

est_trend = decomp.trend #retrival of trend
est_seasonal = decomp.seasonal #retrival of seasonality
est_residual = decomp.resid #retrival of residuals

fig, axes = plt.subplots(4, 1, figsize=(10,10))
axes[0].plot(data, label='Original',color='indigo')
axes[0].legend()
axes[1].plot(est_trend, label='Trend',color="purple")
axes[1].legend()
axes[2].plot(est_seasonal, label='Seasonality',color='mediumvioletred')
axes[2].legend()
axes[3].plot(est_residual, label='Residuals',color='deeppink')
axes[3].legend()
```

Figure 3. Data Decomposition

3.3.2 Normalization of Data

To convert all the data into one range (0, 1), data is normalized using the 'MinMaxScaler' package is imported from 'sklearn' library can be seen in Figure 4. The output of the scaler is NumPy array which is then stored into a new pandas data frame named *normdata*.

```
[11] from sklearn.preprocessing import MinMaxScaler

     uscaler = MinMaxScaler(feature_range=(0, 1))
     unorm = uscaler.fit_transform(np.array(finaldata['USD']).reshape(-1,1))

     nqscaler = MinMaxScaler(feature_range=(0, 1))
     nqnorm = nqscaler.fit_transform(np.array(finaldata['NASDAQ']).reshape(-1,1))

     dscaler = MinMaxScaler(feature_range=(0, 1))
     dnorm = dscaler.fit_transform(np.array(finaldata['DJAI']).reshape(-1,1))

     nyscaler = MinMaxScaler(feature_range=(0, 1))
     nynorm = nyscaler.fit_transform(np.array(finaldata['NYSE']).reshape(-1,1))

     spscaler = MinMaxScaler(feature_range=(0, 1))
     spnorm = spscaler.fit_transform(np.array(finaldata['S&P']).reshape(-1,1))
```

```
[12] normdata = pd.DataFrame(columns=['USD','NASDAQ','DJAI','NYSE','S&P'])

     normdata['USD'] = unorm.flatten()
     normdata['NASDAQ'] = nqnorm.flatten()
     normdata['DJAI'] = dnorm.flatten()
     normdata['NYSE'] = nynorm.flatten()
     normdata['S&P'] = spnorm.flatten()

     normdata.index = finaldata.index
```

Figure 4. Data Normalization

3.3.3 Unit Root test

The Augmented Dickey-Fuller test checks the stationarity of time series data. The properties of Stationary data like variance and mean do not vary with time. Package 'adfuller' from the library 'statsmodel' is used, as shown in Figure 5. The p_values less than 0.05 indicates stationarity in data.

```python
from statsmodels.tsa.stattools import adfuller

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

  ## Applying time shifting, we subtract every the point by the one that preceded it to make data stationary
df_stat = pd.DataFrame(columns=['NASDAQ','DJAI','NYSE','S&P'])
for i in normdata.columns:
  df = normdata[i] - normdata[i].shift()
  df.dropna(inplace=True)
  df_stat[i] = df

print("\n****************************After Difference*************************************\n")
for i in df_stat.columns:
  #Perform Dickey-Fuller test:
  print('Results of Dickey-Fuller Test:', i)
  dftest = adfuller(df_stat[i], autolag='AIC')
  dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','Number of Observations Used'])
  for key,value in dftest[4].items():
    dfoutput['Critical Value (%s)'%key] = value
  print(dfoutput)
  print("-----------------------------------------------")
```

Figure 5. Unit Root Test: ADF

3.3.4 Johansen Co-integration Test

Johansen co-integration test checks the long-term relationship between data. Package 'coint_johansen' is imported to perform the test. In this, coint_johansen().cvt gives critical values, coint_johansen().eig provides eigenvalues and coint_johansen().lr1 provides trace values to check the hypothesis.

```python
from statsmodels.tsa.vector_ar.vecm import coint_johansen

jcvtest = coint_johansen(normdata, det_order=0, k_ar_diff=1).cvt
jetest = coint_johansen(normdata, det_order=0, k_ar_diff=1).eig
jttest = coint_johansen(normdata, det_order=0, k_ar_diff=1).lr1


print("Eigen Values:\n", jetest)
print("\n\nCritical Values: \n", jcvtest)
print("\n\nTrace Values: \n", jttest)
```

```
Eigen Values:
 [0.00450676 0.00316367 0.00150947 0.0007697  0.00013474]


Critical Values:
 [[65.8202 69.8189 77.8202]
 [44.4929 47.8545 54.6815]
 [27.0669 29.7961 35.4628]
 [13.4294 15.4943 19.9349]
 [ 2.7055  3.8415  6.6349]]


Trace Values:
 [75.99982534 42.01434214 18.17313713  6.80731666  1.01385777]
```

Figure 6. Johansen Co-integration Test

3.3.5 Granger Causality Test

Granger causality test checks the null hypothesis that past values of stock indices do not cause the US dollar index. Package ‘grangercasualitytests’ is imported to perform the test. The p_value < 0.05 indicates the rejection of the null hypothesis.

```
[ ] from statsmodels.tsa.stattools  import   grangercausalitytests
    import numpy as np

    maxlag = 15
    test = 'ssr_chi2test'

    def gcm (data, variables, test='ssr_chi2test', verbose=False):
      df = pd.DataFrame(np.zeros((1, len(variables))), columns=variables)
      for c in df.columns:
        for r in df.index:
          result = grangercausalitytests(data[['USD' , c]], maxlag=maxlag, verbose=False)
          p_values = [round(result[i+1][0][test][1],4) for i in range(maxlag)]
          if verbose:
            print(f'y = {r}, x = {c}, p-value = {p_values}')
          minp_value = np.min(p_values)
          df.loc[r,c] = minp_value

      df.columns = [var+'_x' for var in variables]
      df.index = ['USD_y']

      return df

    gcm(df_stat, variables=df_stat.columns)
```

	NASDAQ_x	DJAI_x	NYSE_x	S&P_x	USD_x
USD_y	0.0001	0.0	0.0	0.0	1.0

Figure 7. Granger Causality Test

3.4 Implementation of models

To forecast the US dollar index, statistical model SARIMAX (Seasonal AutoRegressive Integrated Moving Average), new time series model Prophet developed by Facebook, machine learning model Extreme Gradient Boosting and Long Short Term Memory neural network are applied using stock index prices as external factors.

3.4.1 Multivariate seasonal ARIMA (SARIMAX)

- **Data Spilt**

To validate the result of SARIMX model, data is divided into training (80% of total data) and testing (remaining 20% of data)

```
[13] train_data= normdata[0:int(len(normdata)*0.8)]
     test_data = normdata[int(len(normdata)*0.8):]

     plt.figure(figsize=(12,7))
     plt.xlabel('Dates')
     plt.ylabel('Prices')

     plt.plot(train_data['USD'], 'blue', label='Training Data')
     plt.plot(test_data['USD'], 'green', label='Testing Data')
     plt.legend()
```

Figure 8. Train and Test Split

- **Pre-processing for SARIMAX**

To find the least value of AIC (Akaike Information Criteria) different combinations of non-seasonal order (p, d, q) and seasonal order (P, D, Q) are executed as shown in Figure 9.

```
[ ] from statsmodels.tsa.statespace.sarimax import SARIMAX
    import warnings
    warnings.filterwarnings("ignore")
    import itertools
    p = q = range(0, 4)
    d = range(0,2)
    pdq = list(itertools.product(p, d, q))
    P = D = Q = range(0, 2)
    seasonal_pdq = [(x[0], x[1], x[2], 12) for x in list(itertools.product(P, D, Q))]


    for param in pdq:
        for param_seasonal in seasonal_pdq:
            #try:
                mod = SARIMAX(trainusd,order=param,seasonal_order=param_seasonal,exog = trainexog, initialization='approximate_diffuse')
                results = mod.fit()
                print('ARIMA{}x{}12 - AIC:{}'.format(param,param_seasonal,results.aic))
            #except:
                #continue
```

Figure 9. Getting values of AIC

To determine the values of non-seasonal order (p, d, q) and seasonal order (P, D, Q) are cross-checked by plotting graphs of the autocorrelation function and partial autocorrelation. Packages 'plot_acf' and 'plot_pacf' are imported to plot the graphs of ACF and PACF, as shown in Figure 10.

```
from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
from pandas.plotting import autocorrelation_plot
from matplotlib import pyplot
import statsmodels.api as sm
%matplotlib inline

fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(trainusd_stationary, lags = 30, ax= ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(trainusd_stationary, lags = 30, ax = ax2)
pyplot.show()
```

Figure 10. ACF and PACF plot

- **SARIMAX model**

Package 'SARIMAX' is imported from 'statsmodel' to develop a model. The normalized values are reversed back to the original scale using 'inverse_tranform()'. Refer to Figure 11 for SARIMAX model.

```
[19] from statsmodels.tsa.statespace.sarimax import SARIMAX

     model = SARIMAX(trainusd,order=(2,1,3),seasonal_order=(0,1,1,12),exog = trainexog, initialization='approximate_diffuse')

     model_fit = model.fit()
     print(model_fit.summary())

     # invert the differenced forecast to something usable
     forecast = model_fit.predict(start = len(trainusd_stationary)+1, end= len(trainusd_stationary)+len(testexog), exog=testexog)

     p=np.array(forecast).reshape(-1,1)
     pt = uscaler.inverse_transform(p)
     pt = np.array(pt).flatten()
     p = pd.Series(pt, index=testusd.index)


     t = (finaldata['USD'][-len(p):])
     plt.figure(figsize=(15,8))
     plt.plot(t)
     plt.plot(p)
     plt.xlabel('Year')
     plt.ylabel('Values')
     plt.legend(['Original','Predicted'])
```

Figure 11. SARIMAX model

3.4.2 Prophet model

- **Pre-processing for Prophet**

The data is divided into train and test (80:20). Prophet model requires a data frame containing column 'y' as the dependent variable and column 'ds' with dates. As shown in Figure 12.

```
[ ] train = normdata[:int(len(normdata)*0.8)]
    test = normdata[int(len(normdata)*0.8):]

    df = train
    df = df.rename(columns={'USD':'y','Date':'ds'})

[ ] test = test.rename(columns={'USD':'y','Date':'ds'})

[ ] df
```

Date	y	NASDAQ	DJAI	NYSE	S&P	ds
1990-01-02	0.463183	0.015396	0.016935	0.030954	0.021813	1990-01-02
1990-01-03	0.465806	0.015580	0.016919	0.030781	0.021497	1990-01-03
1990-01-04	0.427476	0.015408	0.016399	0.029476	0.020448	1990-01-04
1990-01-05	0.434133	0.015270	0.015531	0.028049	0.019269	1990-01-05
1990-01-08	0.417995	0.015327	0.016334	0.028646	0.019809	1990-01-08

Figure 12. Developing data frame for Prophet

- **Prophet default model**

Refer Figure 13 for the Prophet model with default parameters.

```
[ ] from fbprophet import Prophet

    model = Prophet()
    model.add_regressor('DJAI')
    model.add_regressor('NASDAQ')
    model.add_regressor('NYSE')
    model.add_regressor('S&P')
    model.fit(df)

    pred=model.predict(test.drop('y', axis=1))
    pred.index = pred['ds']
    p = pred['yhat']
    y=np.array(p).reshape(-1,1)
    pt1 = uscaler.inverse_transform(y)
    pt1 = np.array(pt1).flatten()
    pt1 = pd.Series(pt1, index=test.index)

    plt.figure(figsize=(15,8))
    plt.plot(yt)
    plt.plot(pt1)
    plt.xlabel('Year')
    plt.ylabel('Values')
    plt.legend(['Original','Predicted'])
```

Figure 13. Prophet default model

- **Prophet with yearly seasonality parameter**

Refer to Figure 14 for the Prophet model with yearly seasonality parameter.

```
[ ] from fbprophet import Prophet

    model = Prophet(yearly_seasonality=False,weekly_seasonality=False, daily_seasonality=False)
    model.add_seasonality(name='yearly', period=365, fourier_order=20)
    model.add_regressor('DJAI')
    model.add_regressor('NASDAQ')
    model.add_regressor('NYSE')
    model.add_regressor('S&P')
    model.fit(df)

    INFO:numexpr.utils:NumExpr defaulting to 2 threads.
    <fbprophet.forecaster.Prophet at 0x7f94416f0550>

[ ] pred=model.predict(test.drop('y', axis=1))
```

Figure 14. Prophet model with seasonality parameter

3.4.3 Long Short-Term Memory (LSTM) model

- **Pre-processing for LSTM**

Date and its features are added to the data, as shown in Figure 15.

```
[ ] def create_features(df, label=None):
        """
        Creates time series features from datetime index
        """
        df['Date'] = df.index
        df['dayofweek'] = df['Date'].dt.dayofweek
        df['quarter'] = df['Date'].dt.quarter
        df['month'] = df['Date'].dt.month
        df['year'] = df['Date'].dt.year
        df['dayofyear'] = df['Date'].dt.dayofyear
        df['dayofmonth'] = df['Date'].dt.day
        df['weekofyear'] = df['Date'].dt.weekofyear

        return df
```

Figure 15. Date and its features

For LSTM, data is split into training, validation and testing (68:12:20) as shown in Figure 16.

```
[ ] train_data= stat_df[0:int(len(stat_df)*0.68)]
    Val_data = stat_df[int(len(stat_df)*0.68):int(len(stat_df)*0.8)]
    test_data = stat_df[int(len(stat_df)*0.8):]
```

```
[ ] a = train_data.drop(['USD','Date'], axis=1).values
    x_train = a.reshape(a.shape[0], a.shape[1], 1)
    print(x_train.shape)

    y_train = train_data['USD'].values
    print(y_train.shape)

    (5117, 11, 1)
    (5117,)
```

```
[ ] b = test_data.drop(['USD','Date'], axis=1).values
    x_test = b.reshape(b.shape[0], b.shape[1], 1)
    print(x_test.shape)

    y_test = test_data['USD'].values
    print(y_test.shape)

    (1505, 11, 1)
    (1505,)
```

Figure 16. Training, validation and Test split

- **LSTM model**
  Five layered (1 input layer - 3 hidden layers - 1output layer) is built as shown in Figure 17 and 18.

```
##run multiple times
import numpy as np
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout
from tensorflow.python.keras.layers import Dense,LSTM,Dropout
from tensorflow.python.keras import Sequential

np.random.seed(1234)
tf.random.set_seed(1234)

#Build the LSTM model
model = Sequential()
model.add(LSTM(units = 50 , activation='relu', return_sequences=True, input_shape= (x_train.shape[1], 1))) # Add input layer to our model, 50 Neurons, True as we are
                                                                                                             # going to use another LSTM layer

model.add(LSTM(units = 50, activation='relu', return_sequences=True)) # Add hidden layer to our model, 50 Neurons, True as we are going to use another LSTM layer

model.add(LSTM(units = 70, activation='relu', return_sequences=True)) # Add hidden layer to our model, 70 Neurons, True as we are going to use another LSTM layer

model.add(LSTM(units = 35, activation='relu')) # Add hidden layer to our model, 70 Neurons
model.add(Dense(25)) # Densely connected neural network with 25 Neurons
model.add(Dense(1))
model.add(Dropout(0.2))

model.compile(loss='mean_squared_error', optimizer='adam', metrics = ['accuracy']) # compiling the LSTM model
model.summary()

model.fit(x_train, y_train,validation_data=(x_val,y_val),epochs=50, batch_size=80, verbose=1) #fitting of LSTM
```

Figure 17. LSTM model

```
predictions = model.predict(x_test) # Test Data Prediction

y = pd.Series(y_test.flatten())
ycs = y.cumsum() # revert back the stationarized values

p = pd.Series(predictions.flatten())
pcs = p.cumsum() # revert back the stationarized values

y=np.array(ycs).reshape(-1,1)
yt = uscaler.inverse_transform(y) # revert back the normalized values
yt = np.array(yt).flatten()
yt = yt+ 8.729996
yt = pd.Series(yt, index=test_data.index)

p=np.array(pcs).reshape(-1,1)
pt = uscaler.inverse_transform(p) # revert back the normalized values
pt = np.array(pt).flatten()
pt = pt+8.729996
pt = pd.Series(pt, index=test_data.index)
```

Figure 18. LSTM prediction

3.4.4 Extreme Gradient Boosting (XGBoost)

- **Pre-processing in XGBoost**

  Date and its features are added to the data, as shown in Figure 15.

  Data is divided into train, validation and test as shown in Figures 19 and 20.

```
[ ] tdata = create_features(test_data)

    X_test = tdata.drop(['USD','Date'], axis=1)
    y_test = tdata['USD']
```

Figure 19. Testing data for XGBoost

```
[ ] data = create_features(train_data)
    X = data.drop(["USD","Date"],axis=1)
    Y = data['USD']

    X_train= X[0:int(len(X)*0.85)]
    X_val = X[int(len(X)*0.85):]

    y_train= Y[0:int(len(Y)*0.85)]
    y_val = Y[int(len(Y)*0.85):]
```

Figure 20. Training and Validation for XGBoost

- **XGBoost Model**

  Code for default XGBoost model is shown in Figure 21.

```
from xgboost import XGBRegressor
model = XGBRegressor() #model
model.fit(
    X_train,
    y_train,
    eval_metric="rmse",
    eval_set=[(X_train, y_train),(X_val,y_val)], #model fit
    verbose=True)

pred = model.predict(X_test) #prediction

pred = pd.Series(pred, index=test_data.index)
#pred[0] = pred[0]+0.17
pcs = (pred.cumsum()) # revert back the stationarized values
pred_cs = pd.Series(test_data.iloc[0], index=test_data.index)
pred_cs = pred_cs.add(pcs, fill_value=0)
pred_cs = pred_cs + 0.17

p=np.array(pred_cs).reshape(-1,1)
pt = uscaler.inverse_transform(p) # revert back the normalized values
pt = np.array(pt).flatten()
pecs = pd.Series(pt, index=y_test.index)
```

Figure 21. XGBoost Default

Code for tuned hyperparameters is shown in Figure 22.

```
#hyperparameter tuning

from xgboost import XGBRegressor
import warnings
warnings.filterwarnings("ignore")

gbc = XGBRegressor() #XGBoost model

parameters = {
    "n_estimators":[1000, 800, 1200],
    "max_depth":[50, 60, 40],
    "min_child_weight":[500, 350, 600],        ##hyperparameters
    "learning_rate":[0.1,0.3,0.5]
}
from sklearn.model_selection import GridSearchCV
cv = GridSearchCV(gbc, parameters)        # gridsearchcv
cv.fit(X_train,y_train, eval_metric="rmse",       #model fit
    eval_set=[(X_train, y_train),(X_val,y_val)],
    verbose=False)

p = cv.predict(X_test) #prediction
pred = pd.Series(p, index=test_data.index)
pcs = (pred.cumsum()) # revert back the stationarized values
pred_cs = pd.Series(test_data.iloc[0], index=test_data.index)
pred_cs = pred_cs.add(pcs, fill_value=0)
pred_cs = pred_cs + 0.17
p=np.array(pred_cs).reshape(-1,1)
pt = uscaler.inverse_transform(p) # revert back the normalized values
pt = np.array(pt).flatten()
p1 = pd.Series(pt, index=y_test.index)
```

Figure 22. XGBoost tuned hyperparameters

Code for tuned hyperparameters with three-fold cross-validation is shown in Figure 23.

```
from xgboost import XGBRegressor
import warnings
warnings.filterwarnings("ignore")

gbc = XGBRegressor() #model

parameters = {
    "n_estimators":[1000],
    "max_depth":[50],
    "min_child_weight":[500],     #hyperparameters
    "learning_rate":[0.1]
}

from sklearn.model_selection import GridSearchCV
cv = GridSearchCV(gbc, parameters ,cv=3) # model with 3-fold cross validation
cv.fit(X_train,y_train, eval_metric="rmse",
    eval_set=[(X_train, y_train),(X_val,y_val)],
    verbose=False)

p = cv.predict(X_test) #prediction
pred = pd.Series(p, index=test_data.index)
pcs = (pred.cumsum()) # revert back the stationarized values
pred_cs = pd.Series(test_data.iloc[0], index=test_data.index)
pred_cs = pred_cs.add(pcs, fill_value=0)
pred_cs = pred_cs + 0.17
p=np.array(pred_cs).reshape(-1,1)
pt = uscaler.inverse_transform(p) # revert back the normalized values
pt = np.array(pt).flatten()
p1 = pd.Series(pt, index=y_test.index)
```

Figure 23. XGBoost with 3-fold cross-validation

3.5 Evaluation of predicted results

The predicted results are evaluated using sklearm metrics ‘mean_squared_error’ and ‘mean_absoulte_error’ as shown in Figure 19.

```
from math import sqrt
from sklearn.metrics import mean_squared_error, mean_absolute_error

rmse = math.sqrt(mean_squared_error(yt, pt))
print('Root Mean Squared Error: %.3f'% rmse)
mae = mean_absolute_error(yt, pt)
print('Mean Absolute Error: %.3f'% mae)
mape = np.mean(np.abs(pt - yt)/np.abs(yt))
print('Mean Absolute Percentage Error: %.3f'% (mape))
```

Figure 24. Evaluation Metrics

3.6 Visualization of predicted results

To plot the graphs of the predicted result, ‘matplotlib’ library is used, as shown in Figure 25.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(15,8))
plt.plot(yt)
plt.plot(pt)
plt.xlabel('Year')
plt.ylabel('Values')
plt.legend(['Original','Predicted'])
```

Figure 25. Visualization using matplotlib

Figure 26 shows the code to the plot of SARIMAX residual diagnosis.

```
[ ]  model_fit.plot_diagnostics(figsize=(15, 10))
     plt.show()
```

Figure 26. SARIMAX residual plot