National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Arpan Jhamb
Student ID: x20140771

School of Computing
National College of Ireland

Supervisor: Dr. Majid Latifi

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Arpan Jhamb
Student ID:	x20140771
Programme:	Data Analytics
Year:	2021
Module:	MSc Research Project
Supervisor:	Dr. Majid Latifi
Submission Due Date:	16/08/2021
Project Title:	Configuration Manual
Word Count:	XXX
Page Count:	6

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:	15th August 2021

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	Q
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	Q
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.	Q

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

Arpan Jhamb
x20140771

1 Introduction

The configuration manual is a step-by-step guide for the project 'Optimization of Supply Chain Workflow in Food Industry' from the report's creation, installation, implementation, and deployment. The purpose of this report is to support and guide through each stage of the process to get the required output and results given in a technical report.

1.1 Project Overview

The objective of the research is Optimization of Supply Chain Workflow in Food Industry. The quantity of cheese required for making pizza is predicted using the time series model. The comparison is made between ARIMA and TBATS for predicting the sales and an auto email system is created for delivering timely information to the owner of the restaurant. The model will solve the problem of optimizing the workflow in a restaurant.

2 Hardware/Software Requirements:

2.1 Hardware:

- **Processor:** Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz
- **Installed Memory:** 8:00 RAM
- **Storage:** 1 TB 5400 rpm SATA SSHD
- **Operating System:** Windows 10, 64-bit

2.2 Software

- **Jupyter-Lab:** Python programming language software with Jupyer-lab is used for data cleaning, pre-processing, transformations and implementation of all the models.
- **Microsoft Excel:** Used for saving data.
- **Draw.io:** For creating methodology diagram and implementation framework.
- **Email System:** For creation of auto email system.

3 Software Installation Guide:

3.1 Anaconda Navigator and Jupyter Notebook:

- Download Anaconda installer.
- Double click on installer to start.
- Check and address the Read Me and License agreement.
- Install it by clicking install button “Just Me’ unless if installing for other users.
- Select a destination directory or any of your preferred directory.

4 Implementation of Project

The Data cleaning and pre-processing is done using Jupyter-lab. Figure 1 below shows code for importing libraries.

```
# importing the necessary libraries
import pandas as pd
from datetime import datetime
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.tsa.seasonal import seasonal_decompose
from matplotlib import pyplot
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
import math
import numpy as np
import matplotlib
#matplotlib.use('GTKAgg')
import matplotlib.pyplot as plt
from sklearn import linear_model
import pandas as pd
import sys
import csv
import os
import os.path
import warnings
warnings.filterwarnings("ignore")

import statsmodels.api as sm

import pylab as py
```

Figure 1: Importing Python libraries for pre-processing

4.1 Time Series Plot

The time series plot shows the trend, seasonality and any other trends.

```
series = pd.read_csv(historical_pizza_sales_data, header=0, parse_dates=[0], index_col=0, squeeze=True, date_parser=parser)
series.plot()
pyplot.show() # TIme series Plot
```

Figure 2: Time Series Plot

4.2 Multipicative and Additive Series

Figure 3 shows the trend, seasonality and the residuals for Multiplicative series.

```
from statsmodels.tsa.seasonal import seasonal_decompose
from dateutil.parser import parse

# Import Data
df = pd.read_csv(historical_pizza_sales_data, parse_dates=['DATE'], index_col='DATE')

# Multiplicative Decomposition
result_mul = seasonal_decompose(df['#Sales'], model='multiplicative', extrapolate_trend='freq',  period=14)



# Plot
plt.rcParams.update({"figure.figsize": (10,10)})
result_mul.plot().suptitle("M.D",fontsize=22)
```

Figure 3: Multipicative time series Model

Figure 4 shows the trend, seasonality and the residuals for additive series.

```
# Additive Decomposition
result_add = seasonal_decompose(df['#Sales'], model='additive', extrapolate_trend='freq',  period=14)
plt.rcParams.update({"figure.figsize": (10,10)})
result_add.plot().suptitle('Additive Decompose', fontsize=22)
plt.show()
```

Figure 4: Additive time series Model

4.3 Test for Checking Stationarity

Figure 5 shows the Dickey Fuller test. This test is used for checking the stationarity of the data. The null hypothesis is where the time series is non-stationary and has a unit root, the null hypothesis is rejected if the P-Value in the test is below the significance level (0.05).

```
#Dickey-Fuller's test
from statsmodels.tsa.stattools import adfuller
def adf_test(timeseries):
    print("Results of Dickey-Fuller Test:")
    dftest = adfuller(timeseries, 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)
```

Figure 5: Dickey Fuller Test

Figure 6 shows the KPSS test to check for stationarity. This is a stationarity test that evaluates if a series is stationary around the mean or not.

```
# KPSS test for stationarity
def kpss_test(timeseries):
    print("Results of KPSS Test:")
    kpsstest = kpss(timeseries, regression="c", nlags="auto")
    kpss_output = pd.Series(
        kpsstest[0:3], index=["Test Statistic", "p-value", "Lags Used"]
    )
    for key, value in kpsstest[3].items():
        kpss_output["Critical Value (%s)" % key] = value
    print(kpss_output)
```

Figure 6: KPSS Test

4.4 TBATS Impelementation

To model time series data, TBATS is a predictive approach. The main goal is to use exponential smoothing to anticipate time series with complicated seasonal trends. The TBATS implementation is seen in figure 7.

```
from tbats import TBATS, BATS
# Fit the TBATS model
estimator = TBATS()
model = estimator.fit(y_to_train)
y_forecast = model.forecast(steps=55)
```

Figure 7: TBATS Test

4.5 Evaluating Performance of TBATS and ARIMA

TBATS performance was evaluated based on the different performance criteria like ME,MAE, MPE, MSE, RMSE. Figure 8 shows the result obtained after evaluation.

```
mse = mean_squared_error(y_to_test, y_forecast)
rmse = math.sqrt(mse)
me = np.mean(y_forecast - y_to_test)             # ME
mae = np.mean(np.abs(y_forecast - y_to_test))    # MAE
mpe = np.mean((y_forecast - y_to_test)/y_to_test)   # MPE
print("MSE: {}".format(mse))
print("RMSE: {}".format(rmse))
print("ME: {}".format(me))
print("MAE: {}".format(mae))
print("MPE: {}".format(mpe))

forecast_errors = [abs(((y_to_test[i]-y_forecast[i])*100)/(y_to_test[i])) for i in range(len(y_to_test))]
error = sum(forecast_errors) * 1.0/len(y_to_test)
print('Error percentage (MAPE): %f' % error)
```

Figure 8: TBATS Performance Evaluation

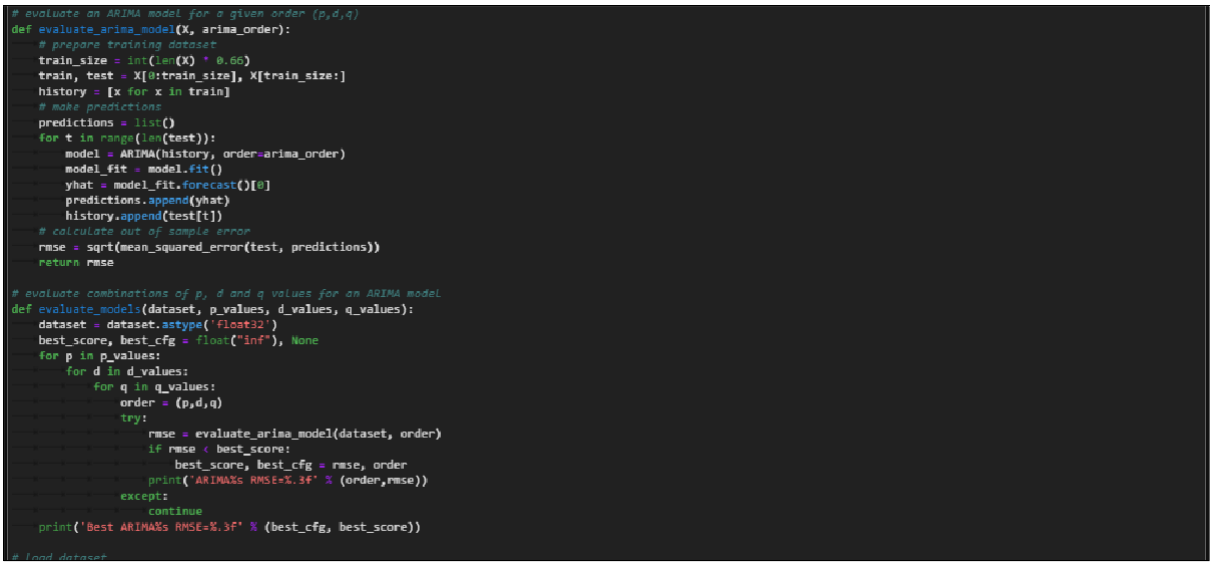
```
mse = mean_squared_error(test, predictions)
rmse = math.sqrt(mse)
me = np.mean(predictions - test)             # ME
mae = np.mean(np.abs(predictions - test))    # MAE
mpe = np.mean((predictions - test)/test)   # MPE
print("MSE: {}".format(mse))
print("RMSE: {}".format(rmse))
print("ME: {}".format(me))
print("MAE: {}".format(mae))
print("MPE: {}".format(mpe))

forecast_errors = [abs(((test[i]-predictions[i])*100)/(test[i])) for i in range(len(test))]
error = sum(forecast_errors) * 1.0/len(test)
print('Error percentage (MAPE): %f' % error)
```

Figure 9: ARIMA Performance Evaluation

ARIMA performance was evaluated based on different performance criteria like ME, MAE, MPE, MSE, RMSE. We used different functions for ME, MAE, MSE, MPE, RMSE. Figure 9 shows the result obtained after evaluation. The result obtained highlighted the better performance of ARIMA. Now, this is used for forecasting.

4.6 Hyperparameter Tuning of ARIMA

After hyperparameter tuning, we found the best values for (p, d, q) for the implementation. Figure 10 is used to find the best values for (p,d,q).

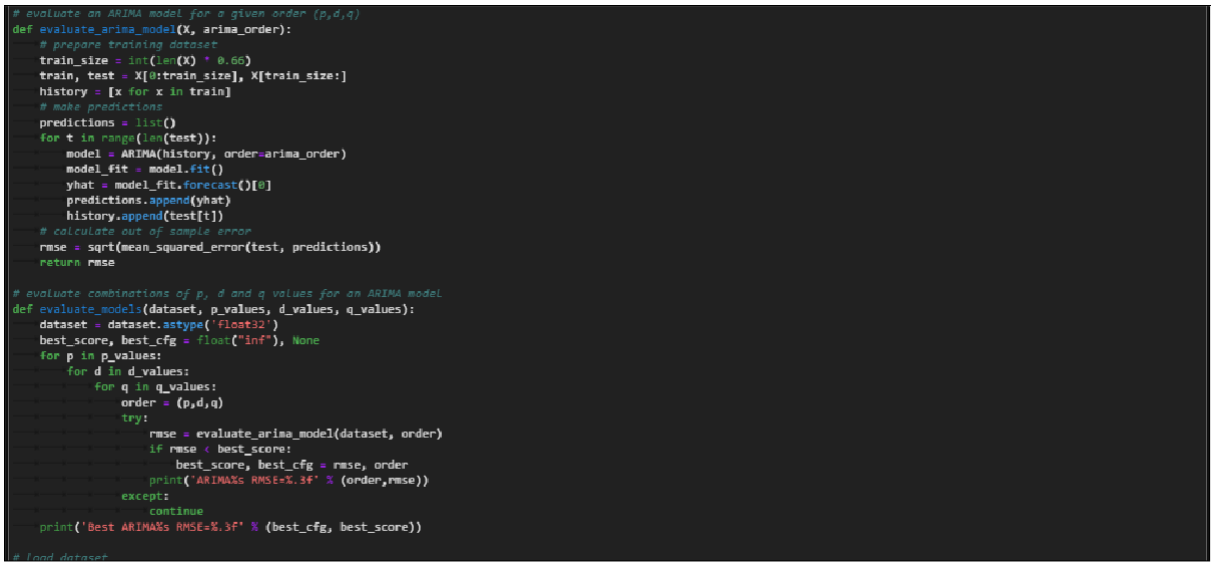


```
# evaluate an ARIMA model for a given order (p,d,q)
def evaluate_arima_model(X, arima_order):
    # prepare training dataset
    train_size = int(len(X) * 0.66)
    train, test = X[0:train_size], X[train_size:]
    history = [x for x in train]
    # make predictions
    predictions = list()
    for t in range(len(test)):
        model = ARIMA(history, order=arima_order)
        model_fit = model.fit()
        yhat = model_fit.forecast()[0]
        predictions.append(yhat)
        history.append(test[t])
    # calculate out of sample error
    rmse = sqrt(mean_squared_error(test, predictions))
    return rmse

# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(dataset, p_values, d_values, q_values):
    dataset = dataset.astype('float32')
    best_score, best_cfg = float("inf"), None
    for p in p_values:
        for d in d_values:
            for q in q_values:
                order = (p,d,q)
                try:
                    rmse = evaluate_arima_model(dataset, order)
                    if rmse < best_score:
                        best_score, best_cfg = rmse, order
                    print('ARIMA%s RMSE=%.3f' % (order,rmse))
                except:
                    continue
    print('Best ARIMA%s RMSE=%.3f' % (best_cfg, best_score))

# load dataset
```

Figure 10: Calculating Hyperparameters for ARIMA

4.7 Implementation of Model

Figure 11 shows the implementation of ARIMA where we have forecasted the sales and calculated the quantity of cheese required.



```
# evaluate an ARIMA model for a given order (p,d,q)
def evaluate_arima_model(X, arima_order):
    # prepare training dataset
    train_size = int(len(X) * 0.66)
    train, test = X[0:train_size], X[train_size:]
    history = [x for x in train]
    # make predictions
    predictions = list()
    for t in range(len(test)):
        model = ARIMA(history, order=arima_order)
        model_fit = model.fit()
        yhat = model_fit.forecast()[0]
        predictions.append(yhat)
        history.append(test[t])
    # calculate out of sample error
    rmse = sqrt(mean_squared_error(test, predictions))
    return rmse

# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(dataset, p_values, d_values, q_values):
    dataset = dataset.astype('float32')
    best_score, best_cfg = float("inf"), None
    for p in p_values:
        for d in d_values:
            for q in q_values:
                order = (p,d,q)
                try:
                    rmse = evaluate_arima_model(dataset, order)
                    if rmse < best_score:
                        best_score, best_cfg = rmse, order
                    print('ARIMA%s RMSE=%.3f' % (order,rmse))
                except:
                    continue
    print('Best ARIMA%s RMSE=%.3f' % (best_cfg, best_score))

# load dataset
```

Figure 11: Hyperparameter tuning of ARIMA model

4.8 Prediction based on ARIMA

Based on the hyperparameter tuning, the best values chosen were (6,1,0) and the prediction was made on these values.

```
model = ARIMA(history, order=(6,1,0))
model_fit = model.fit(disp=0)
output = model_fit.forecast()
yhat = output[0]
print('Predicted=%f' % (int(yhat)))
#Multiply the predicted pizza sales figure with 50, indicating 50gms of cheese required for each pizza
n=yhat*50
fi=n/1000
print("Recommended Order Quantity of Cheddar Cheese for pizza is %f Kgs" % fi)
recommended_order = round(fi[0], 2)
```

Figure 12: Prediction through ARIMA model

4.9 Auto Email System

Figure 13 shows an auto email system used for sending the email to the owner of the restaurant. Port 465 is used for a secured connection (SSL). The smtplib module is used to establish a connection with the client-server, validating login credentials, and sending emails.

```
import smtplib, ssl

port = 465  # For SSL
smtp_server = "smtp.gmail.com"
sender_email = "prateekbabbar1242@gmail.com"  # Enter your address
receiver_email = "arpanjhamb95@gmail.com"  # Enter receiver address
with open("C://Users//user//Desktop//Pass.txt") as f:
    contents = f.readlines()
password =contents[0]

message = """\
Subject: Hi,Updated Quantity to be ordered:

Hi Arpan,

You need to order {} Kg of Cheese.

Have a nice day!

Your Machine learning model
""".format(recommended_order)


context = ssl.create_default_context()
with smtplib.SMTP_SSL(smtp_server, port, context=context) as server:
    server.login(sender_email, password)
    server.sendmail(sender_email, receiver_email, message)
```

Figure 13: Email Delivery System

This is the summary of the important extracts from the source code.