National College of Ireland

Configuration Manual

MSc Research Project
MSc in Data Analytics

Shree Hari Krishnamurthy
Student ID: x21165441

School of Computing
National College of Ireland

Supervisor: Dr. Anh Duong Trinh

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Shree Hari Krishnamurthy
Student ID:	x21165441
Programme:	MSc in Data Analytics
Year:	2023
Module:	MSc Research Project
Supervisor:	Dr. Anh Duong Trinh
Submission Due Date:	14/08/2023
Project Title:	Configuration Manual
Word Count:	2830
Page Count:	25

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:	*Shree Hari Krishnamurthy*
Date:	14th August 2023

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

Shree Hari Krishnamurthy
x21165441

1 Introduction

The present Configuration Manual provides an detailed instructions for the implementation of the project focused on the prediction of unemployment rate in the Ireland, employing an various range of statistical and machine learning models. This manual is based on the Cross-Industry Standard Process for Data Mining (CRISP-DM) methodology that offers an comprehensive guide through each stage of the process. It covers the steps, that includes, data collection, preprocessing, modelling, and inspection. The manual has been carefully written to ensure that it can be easily understood by individuals with both technical and non-technical backgrounds.

2 Hardware and Software Requirements

The following tables provide information on the hardware and software requirements.

Table 1: Hardware Specifications

Device Name/OS	MacBook Air/macOS Big Sur Version 11.0.1
RAM/CPU	8 GB/1.6 GHz Dual-Core Intel Core i5
Hard Disk	128 GB SSD
GPU	Intel UHD Graphics 617

Table 2: Software Specifications

Programming Language	Python
Python Version	3.8.5
IDE	Jupyter Notebook
Jupyter Notebook Version	6.1.4
Main Libraries Used	pandas, numpy, matplotlib, sklearn, statsmodels

3 Data Selection

3.1 Importing Python Libraries

The Figure 1 below, shows the main Python libraries that has been imported for the purpose of this project. Libraries like as NumPy, Pandas, and Matplotlib are essential components in the data analysis and model development phases of this work.

```
# Importing necessary libraries for data manipulation and mathematical operations
import numpy as np
import pandas as pd

# Importing visualization libraries
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px

# Importing libraries for time series analysis
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Importing machine learning libraries
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Ridge
from sklearn.preprocessing import StandardScaler
from sklearn import neighbors

# Importing the auto ARIMA function
from pmdarima import auto_arima

# Importing XGBoost library
import xgboost as xgb

# Filter warnings
import warnings
warnings.filterwarnings('ignore')

# Magic function for inline plotting
%matplotlib inline
```

Figure 1: Python Libraries

3.2 Retrieving the Unemployment Dataset from the CSO Ireland

To Download Irish Central Statistics Office (CSO) unemployment data for this research.

- Visit https://www.cso.ie/en/statistics/labourmarket/monthlyunemployment/ for CSO Ireland's monthly unemployment data
- Move dataset to a directory after downloading. This folder's file directory will be utilised later to load data to a dataframe for analysis.

3.3 Loading and Preparing the Data

The unemployment data that has been imported into a Python environment as a time series by utilising the pandas package. By selecting the 'Month' column as the index and using date parsing, the data undergoes the automated conversion into a time series structure. Furthermore, it can be essential to exclude the unnecessary variables from the dataset to simplify data and concentrate just on the data that is relevant. In this scenario, unnecessary columns are eliminated from dataset by employing the drop function, as shown in Figure 2 below:

```
# Read data in form of Time series
df_timeseries=pd.read_csv('/Users/shreehari/NCI/RIC/Model/Irish_unemployment_data.csv',index_col='Month', parse_dates=True)
df_timeseries.head()
```

✓ 0.0s

Month	Statistic Label	Age Group	Sex	UNIT	VALUE
1998-01-01	Seasonally Adjusted Monthly Unemployment	15 - 74 years	Both sexes	Thousand	151.3
1998-02-01	Seasonally Adjusted Monthly Unemployment	15 - 74 years	Both sexes	Thousand	153.3
1998-03-01	Seasonally Adjusted Monthly Unemployment	15 - 74 years	Both sexes	Thousand	149.5
1998-04-01	Seasonally Adjusted Monthly Unemployment	15 - 74 years	Both sexes	Thousand	140.2
1998-05-01	Seasonally Adjusted Monthly Unemployment	15 - 74 years	Both sexes	Thousand	133.8

```
df_timeseries = df_timeseries.drop(['Statistic Label','Age Group','Sex','UNIT'], axis=1)
df_timeseries.head()
```

✓ 0.0s

Month	VALUE
1998-01-01	151.3
1998-02-01	153.3
1998-03-01	149.5
1998-04-01	140.2
1998-05-01	133.8

Figure 2: Load Data to Data frame

4 Exploratory Data Analysis

4.1 Plotting the Trend Over Time

Matplotlib's plot function shows unemployment trends. The x-axis shows months, and the y-axis shows unemployment in thousands.

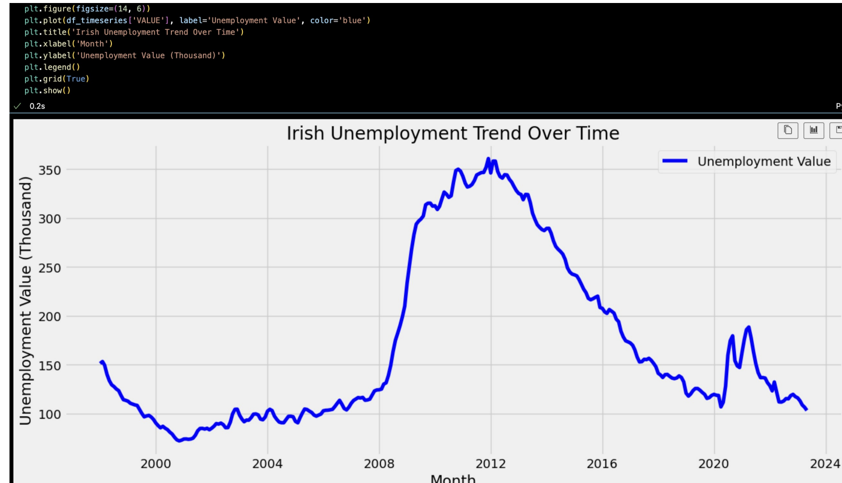

Figure 3: Irish Unemployment Trend Over Time

4.2 Histogram of Unemployment Values

To understand the distribution of unemployment values, a histogram is created using matplotlib's hist function. This visualization 4 helps to comprehend the frequency of different ranges of unemployment values

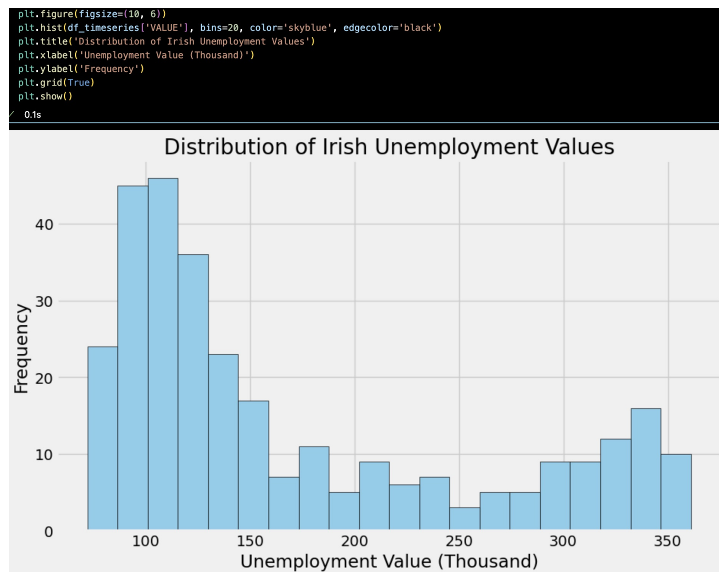

Figure 4: Distribution of Irish Unemployment Values

4.3 Interactive Slider Graph

The plotly.express package creates a slider-enabled line graph for data analysis. This let you analyse unemployment data over time as shown in Figure 5.

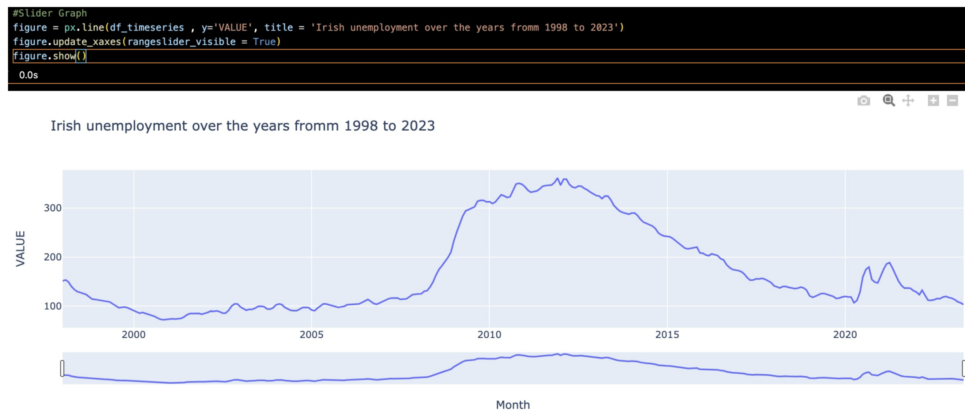

Figure 5: Irish unemployment over the years from 1998 to 2023

4.4 Seasonal Decomposition of Time Series Data

Seasonal_decompose from the statsmodels.tsa.seasonal module splits time series data into trend, seasonal, and residual components. To interpret time series data, additive and multiplicative models are decomposed as shown in Figure 6 and 7.

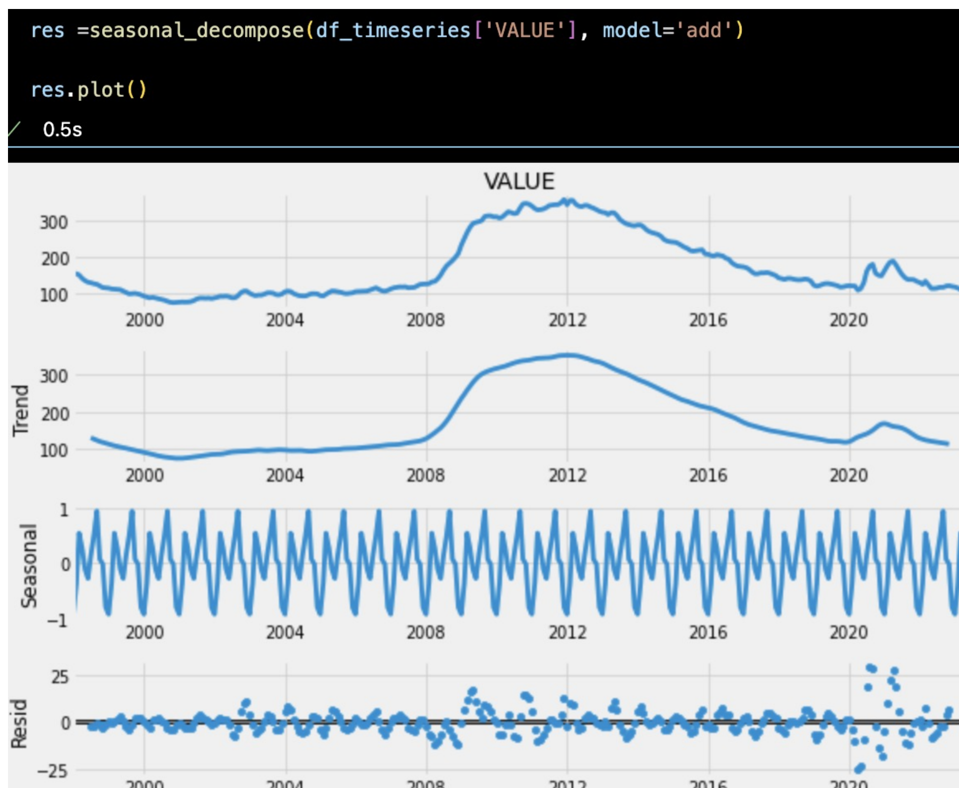

Figure 6: Additive decomposition

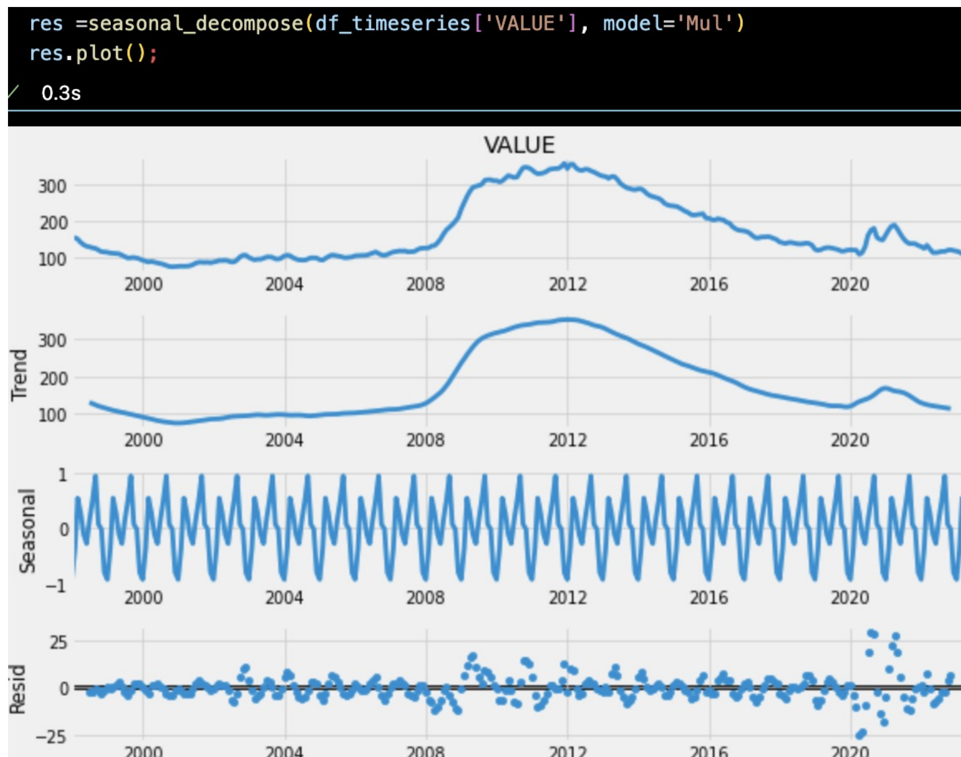

Figure 7: Multiplicative decomposition

4.5 Checking for Null Values

A dataset null value check ensures data integrity and completeness. The check is done by using isnull() function as shown in Figure 8

```
Check the Null Values

df_timeseries.isnull().sum()
✓ 0.0s

VALUE    0
dtype: int64
```

Figure 8: Null Value check

4.6 Augmented Dickey-Fuller Test

An Augmented Dickey-Fuller (ADF) test determines time series data stationarity. The test statistic and p-value reveal time series data characteristics. This test is part of the statsmodels.tsa.stattools module as shown below 9

Augmented Dickey Fuller Test (ADF Test)

```python
def ad_test(dataset):
    dftest = adfuller(dataset, autolag = 'AIC')
    print("1. Test Statistic : ",dftest[0])
    print("2. P-Value : ", dftest[1])
    print("3. Num Of Lags : ", dftest[2])
    print("4. Num Of Observations Used For ADF Regression:",      dftest[3])
    print("5. Critical Values :")
    for key, val in dftest[4].items():
        print("\t",key, ": ", val)
```

[4] ✓ 0.0s

If Test statistic < Critical Value and p-value < 0.05 - Stationary

```python
ad_test(df_timeseries['VALUE'])
```

[5] ✓ 0.0s

```
1. Test Statistic :  -1.8041306012586598
2. P-Value :  0.3784129017114926
3. Num Of Lags :  8
4. Num Of Observations Used For ADF Regression: 296
5. Critical Values :
	1% :  -3.452636878592149
	5% :  -2.8713543954331433
	10% :  -2.5719993576515705
```

Figure 9: ADF Test

5 Data Preprocessing and Transformation

5.1 Stationarity Check with Augmented Dickey-Fuller (ADF) Test

To enhance the stability of the variance, the data that has been converted logarithmically can be subjected to a square root transformation using the sqrt function from the numpy library. Additionally, In order to eliminate any of the underlying pattern in the data, it is recommended to apply a shift transformation by subtracting the preceding value from the current value. Following the transformation of the data, the Augmented Dickey-Fuller (ADF) test on the modified data in order to verify its stationarity as shown in Figure 10.

```
Transforms the time series data by taking the log and then the square root, and finally the shift difference

log_df = df_timeseries[['VALUE']]
log_df['log'] = np.log(log_df['VALUE'])

log_sqrt_df = log_df[['log']]
log_sqrt_df['sqrt_log'] = np.sqrt(log_sqrt_df['log'])

log_sqrt_df['shift_log_sqrt'] = log_sqrt_df['sqrt_log'] - log_sqrt_df['sqrt_log'].shift()
log_sqrt_df = log_sqrt_df.dropna()
log_sqrt_df

#Checking ADF test for Transformed Data
ad_test(log_sqrt_df['shift_log_sqrt'])
✓ 0.0s

1. Test Statistic :  -3.481329667646232
2. P-Value :  0.008479735799720136
3. Num Of Lags :  7
4. Num Of Observations Used For ADF Regression: 296
5. Critical Values :
        1% :  -3.452636878592149
        5% :  -2.8713543954331433
        10% :  -2.5719993576515705
```

Figure 10: Applying Transformation for original data

The ADF test is executed by employing the adfuller function from the statsmodels.tsa.stattools package. The Stationaty Plot of Transformed data can be seen in Figure 11

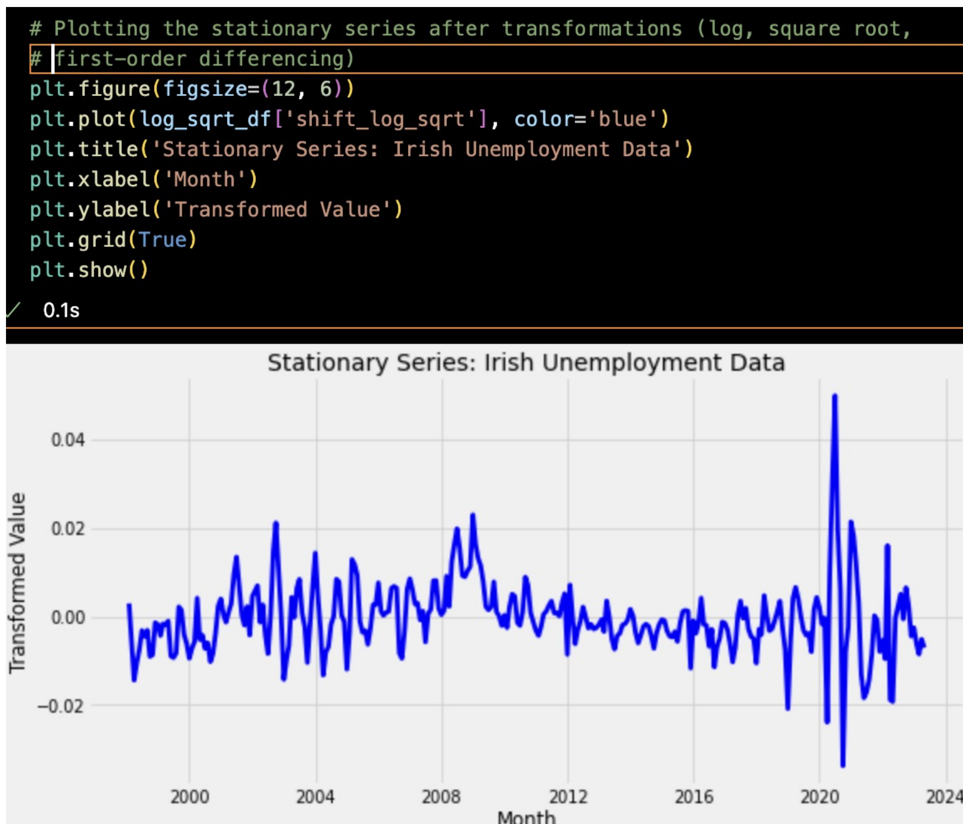

Figure 11: Stationary Plot for unemployment data

5.2 Visualizing Autocorrelation and Partial Autocorrelation

The autocorrelation function (ACF) may be visualised by employing the plot_acf function from the statsmodels.graphics.tsaplots module to analyse the autocorrelation of transformed data. The Autocorrelation Function (ACF) plot is a valuable tool for determining the order of the Moving Average (MA) component inside a time series model. The partial autocorrelation function (PACF) may be visualised by employing plot_pacf function to analyse altered data. Partial Autocorrelation Function (PACF) plot is employed for purpose of determining the order of the autoregressive (AR) component inside the time series model.

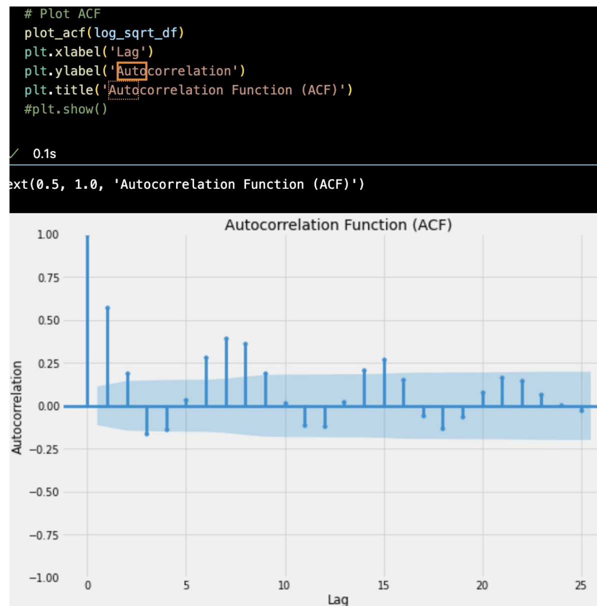

Figure 12: ACF plot

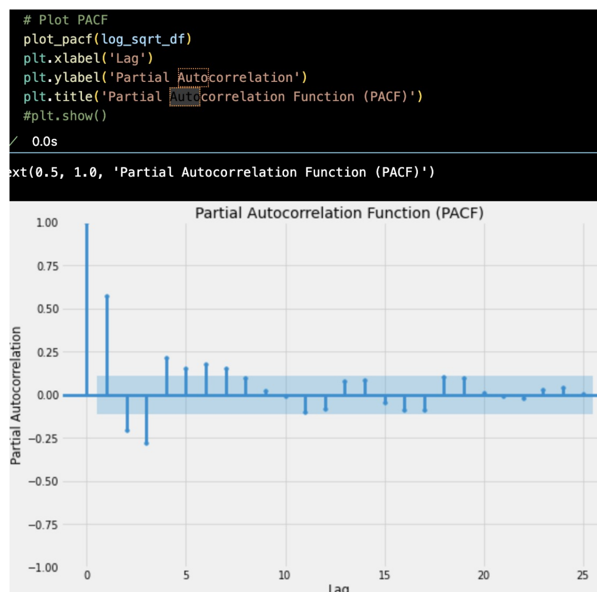

Figure 13: PACf plot

```
ARIMA

(variable) train_data: Series
ing and test sets
train_data = log_sqrt_df['shift_log_sqrt'].iloc[:-30]
test_data = log_sqrt_df['shift_log_sqrt'].iloc[-30:]

orders = [(2, 1, 0), (2,1,1)]

# Running the experiments with different ARIMA orders
results_df = pd.DataFrame(columns=["Order", "RMSE", "R-squared", "MAPE (%)"])
for order in orders:
    arima_model_exp = ARIMA(train_data, order=order)
    arima_fit_exp = arima_model_exp.fit()
    forecast_values_exp = arima_fit_exp.get_forecast(steps=len(test_data)).predicted_mean
    rmse_exp = np.sqrt(mean_squared_error(test_data, forecast_values_exp))
    r_square_exp = r2_score(test_data, forecast_values_exp)
    mape_exp = np.mean(np.abs((test_data - forecast_values_exp) / test_data)) * 100
    results_df = results_df.append({
        "Order": str(order),
        "RMSE": rmse_exp,
        "R-squared": r_square_exp,
        "MAPE (%)": mape_exp
    }, ignore_index=True)

results_df
```

Figure 14: Model performance comparison for different ARIMA orders

6 Modelling And Evaluation

6.1 ARIMA

Prior to getting started, it is important to verify the presence of all necessary libraries, that are pandas, numpy, statsmodels, and the scikit-learn, within the system. pre-processing on the data, resulting in creation of a DataFrame named log_sqrt_df. This DataFrame should have a column entitled shift_log_sqrt, which will hold the time-series data after transformation. The dataset is partitioned into the separate training and test sets. The training set is composed of all the data points, excluding the final 30, whereas the test set is comprised only of latest 30 data points. The configurations of ARIMA models are specified in the variable called "orders". This code in Figure 14 utilises two ARIMA setups, specifically (2, 1, 0) and (2, 1, 1).

For every ARIMA order, A model instance of ARIMA is instantiated using the current order and then trained on the training data. Subsequently, projected values for the duration of the test data are created. Evaluation measures, namely RMSE (Root Mean Squared Error), R-squared, and MAPE (Mean Absolute Percentage Error), are computed for every individual order. The measurements are saved into a DataFrame referred to as results_df. Upon executing the code, the user may access the results_df DataFrame in order to observe the evaluation metrics corresponding to the each ARIMA order.

```
# Applying auto_arima to find the best order for the ARIMA model
auto_arima_model = auto_arima(log_sqrt_df['shift_log_sqrt'], trace=True, suppress_warnings=True)
best_order_auto_arima = auto_arima_model.order
best_order_auto_arima
```

✓ 5.9s

```
Performing stepwise search to minimize aic
 ARIMA(2,0,2)(0,0,0)[0] intercept   : AIC=-2214.207, Time=0.16 sec
 ARIMA(0,0,0)(0,0,0)[0] intercept   : AIC=-2061.562, Time=0.05 sec
 ARIMA(1,0,0)(0,0,0)[0] intercept   : AIC=-2179.974, Time=0.11 sec
 ARIMA(0,0,1)(0,0,0)[0] intercept   : AIC=-2155.183, Time=0.10 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=-2063.184, Time=0.04 sec
 ARIMA(1,0,2)(0,0,0)[0] intercept   : AIC=-2212.774, Time=0.11 sec
 ARIMA(2,0,1)(0,0,0)[0] intercept   : AIC=-2195.090, Time=0.24 sec
 ARIMA(3,0,2)(0,0,0)[0] intercept   : AIC=-2218.300, Time=0.35 sec
 ARIMA(3,0,1)(0,0,0)[0] intercept   : AIC=-2216.857, Time=0.23 sec
 ARIMA(4,0,2)(0,0,0)[0] intercept   : AIC=-2216.248, Time=0.36 sec
 ARIMA(3,0,3)(0,0,0)[0] intercept   : AIC=inf, Time=0.19 sec
 ARIMA(2,0,3)(0,0,0)[0] intercept   : AIC=-2163.912, Time=0.22 sec
 ARIMA(4,0,1)(0,0,0)[0] intercept   : AIC=-2218.139, Time=0.30 sec
 ARIMA(4,0,3)(0,0,0)[0] intercept   : AIC=-2215.312, Time=0.39 sec
 ARIMA(3,0,2)(0,0,0)[0]             : AIC=-2219.631, Time=0.08 sec
 ARIMA(2,0,2)(0,0,0)[0]             : AIC=-2216.543, Time=0.10 sec
 ARIMA(3,0,1)(0,0,0)[0]             : AIC=-2217.914, Time=0.06 sec
 ARIMA(4,0,2)(0,0,0)[0]             : AIC=-2218.032, Time=0.20 sec
 ARIMA(3,0,3)(0,0,0)[0]             : AIC=inf, Time=0.35 sec
 ARIMA(2,0,1)(0,0,0)[0]             : AIC=-2196.914, Time=0.13 sec
 ARIMA(2,0,3)(0,0,0)[0]             : AIC=-2206.022, Time=0.20 sec
 ARIMA(4,0,1)(0,0,0)[0]             : AIC=-2236.403, Time=0.20 sec
 ARIMA(4,0,0)(0,0,0)[0]             : AIC=-2225.774, Time=0.08 sec
 ARIMA(5,0,1)(0,0,0)[0]             : AIC=-2224.204, Time=0.11 sec
...
 ARIMA(5,0,2)(0,0,0)[0] intercept   : AIC=-2158.168, Time=0.51 sec

Best model:  ARIMA(5,0,2)(0,0,0)[0]
Total fit time: 5.943 seconds
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
(5, 0, 2)
```

Figure 15: Auto ARIMA model selection output

The code sets the order of the ARIMA model to (5, 0, 2) by the auto ARIMA, where 5 is the number of lag, with no difference (d), and 2 is the size of the moving average got with lowest AIC. Then, the train_data are used to set up and fit an ARIMA model. After the training phase, the model uses get_forecast() method to make predictions based on the length of test_data. The mean projected values are then used to evaluate model. Lastly, the evaluation measures and the order of model is added to the results_df dataframe so that they can be tracked and compared.

Appply Again ARIMA based on auto ARIMA order

```
# Adding the order provided by auto_arima
auto_arima_order = (5, 0, 2)

# Running the experiment with the order provided by auto_arima
arima_model_auto = ARIMA(train_data, order=auto_arima_order)
arima_fit_auto = arima_model_auto.fit()
forecast_values_auto = arima_fit_auto.get_forecast(steps=len(test_data)).predicted_mean
rmse_auto = np.sqrt(mean_squared_error(test_data, forecast_values_auto))
r_square_auto = r2_score(test_data, forecast_values_auto)
mape_auto = np.mean(np.abs((test_data - forecast_values_auto) / test_data)) * 100

# Adding the results to the results_df
results_df = results_df.append({
    "Order": str(auto_arima_order),
    "RMSE": rmse_auto,
    "R-squared": r_square_auto,
    "MAPE (%)": mape_auto
}, ignore_index=True)

results_df
```

✓ 0.4s

	Order	RMSE	R-squared	MAPE (%)
0	(2, 1, 0)	0.011782	-0.281562	420.855880
1	(2, 1, 1)	0.011627	-0.248074	405.357544
2	(5, 0, 2)	0.009647	0.140857	268.453309

Figure 16: Model performance comparison including the order suggested by auto ARIMA

```python
# Predicting the values for the test data
forecast_values_auto = arima_fit_auto.get_forecast(steps=len(test_data)).predicted_mean

# Plotting the predictions and actual test data
plt.figure(figsize=(12, 6))
plt.plot(forecast_values_auto, label='ARIMA Predictions', color='blue')
plt.plot(test_data.index, test_data.values, label='Actual Test Data', color='orange')
plt.legend()
plt.title('ARIMA(5, 0, 2) Predictions vs. Actual Test Data')
plt.xlabel('Date')
plt.ylabel('Value')
plt.show()
```

✓ 0.1s

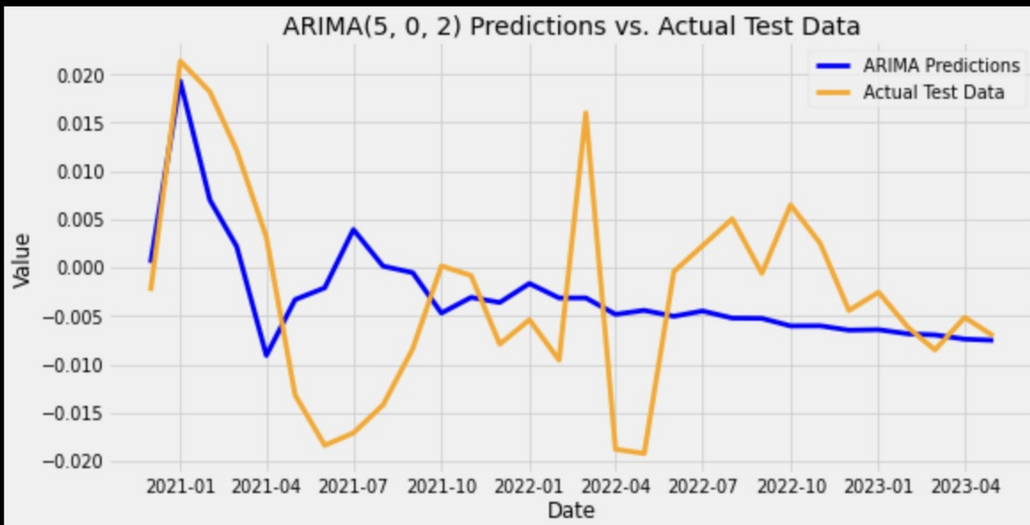

Figure 17: ARIMA model predictions vs. actual test data

6.2 SARIMA

The code is about setting up, the training, evaluating, and visualising SARIMAX model on time series data with different orders. At first, three different SARIMAX orders is set up so that they can be tested. Each of the order is made up of an main ARIMA order and a yearly component. The seasonal component stays the same in all configurations, which shows that there is no seasonal effects and only the 12-period yearly trend. subsequently each SARIMAX setup is checked against train_data in an loop. In this loop, a SARIMAX model is created with the given order, matched to training data, and then used to make the predictions for the test_data. After making a forecast, three evaluation metrics—RMSE, 2 R 2, and MAPE—are used to figure out how well model did. The sarimax_results_df dataframe stores these measure along with the SARIMAX orders so that they can be compared later .

SARIMA

```python
# Defining different SARIMAX orders for evaluation
sarimax_orders = [
    ((2, 1, 0), (0, 0, 0, 12)),
    ((2, 1, 1), (0, 0, 0, 12)),
    ((5, 0, 2), (0, 0, 0, 12)) # The order previously selected
]

# Running the experiments with different SARIMAX orders
sarimax_results_df = pd.DataFrame(columns=["Order", "Seasonal Order", "RMSE", "R-squared", "MAPE (%)"])

for order, seasonal_order in sarimax_orders:
    sarimax_model_exp = SARIMAX(train_data, order=order, seasonal_order=seasonal_order)

    sarimax_fit_exp = sarimax_model_exp.fit()
    forecast_values_exp = sarimax_fit_exp.get_forecast(steps=len(test_data)).predicted_mean
    rmse_exp = np.sqrt(mean_squared_error(test_data, forecast_values_exp))
    r_square_exp = r2_score(test_data, forecast_values_exp)
    mape_exp = np.mean(np.abs((test_data - forecast_values_exp) / test_data)) * 100
    sarimax_results_df = sarimax_results_df.append({
        "Order": str(order),
        "Seasonal Order": str(seasonal_order),
        "RMSE": rmse_exp,
        "R-squared": r_square_exp,
        "MAPE (%)": mape_exp
    }, ignore_index=True)

sarimax_results_df
```

✓ 0.4s Python

Figure 18: SARIMA Training model with different order

After all the settings have been looked at, code hides any possible warnings to make result look better. Then, a specific SARIMAX model is built and put in the place, with an order of (5, 0, 2) and a yearly order of (0, 0, 12). A line plot is then used to compare this model forecasts to real test_data. The plot shows how the predictions made by SARIMAX model compare to real numbers from the test dataset.

```python
import warnings
warnings.filterwarnings('ignore')
# Building and fitting a SARIMA model (order and seasonal_order can be fine-tuned)
sarimax_model = SARIMAX(train_data, order=(5, 0, 2), seasonal_order=(0, 0, 0, 12))
sarimax_result = sarimax_model.fit()
# Predicting the values for the test data using the SARIMAX model
sarimax_forecast_values = sarimax_result.get_forecast(steps=len(test_data)).predicted_mean

# Plotting the SARIMAX predictions and actual test data
plt.figure(figsize=(12, 6))
plt.plot(sarimax_forecast_values, label='SARIMAX Predictions', color='blue')
plt.plot(test_data.index, test_data.values, label='Actual Test Data', color='orange')
plt.legend()
plt.title('SARIMAX(5, 0, 2) Predictions vs. Actual Test Data')
plt.xlabel('Date')
plt.ylabel('Value')
plt.show()
```

✓ 0.4s Python

Figure 19: SARIMA - Prediction of test Data

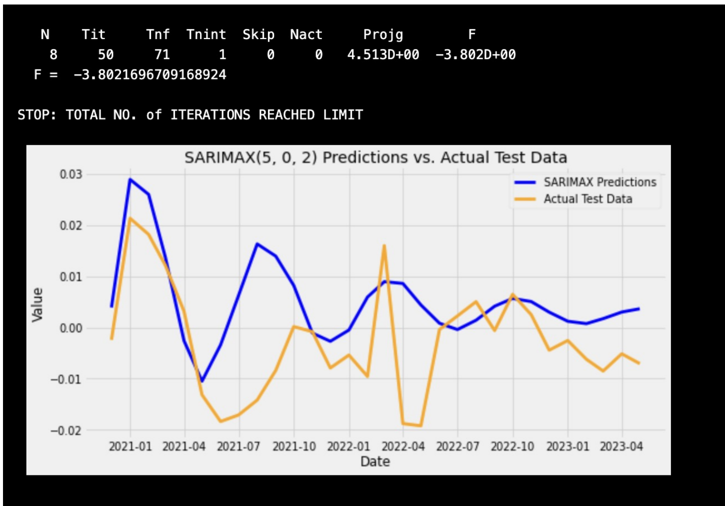

Figure 20: Visualisation of SARIMA Prediction

6.3 Random Forest

The provided code in Figure 21 enhances the df_timeseries dataframe by incorporating the 3 more columns. These columns correspond to lagged versions of VALUE column, with each column representing an different time period (1, 2 and 3 month). This modification aims to capture the historical dependencies existing in an time series data. Subsequently, rows that contain the NaN values, which were created as an result of the shifting process, that are eliminated in order to maintain the data integrity for further analysis.

Random Forest

```
df_timeseries['one_month_shift']= df_timeseries['VALUE'].shift(+1)
df_timeseries['two_month_shift']=df_timeseries['VALUE'].shift(+2)
df_timeseries['three_month_shift']=df_timeseries['VALUE'].shift(+3)
df_timeseries = df_timeseries.dropna()
```

✓ 0.0s Python

Figure 21: Introducing Lagged Features

In the code snippet 22, The numpy arrays are derived from the df_timeseries dataframe, consisting of the 3 lagged features and target variable. The arrays are undergo an process of reshaping to create two-dimensional structure, which are subsequently combined to produce an input matrix, final_x. The dataset is there after divided into the two subsets: a training set and a test set. The final 30 observations are specifically allocated for the purpose of the testing. In the meantime, features undergo standardisation using StandardScaler technique, which aims to achieve an mean of 0 and a standard deviation of 1. This process ensures that the features are an uniformly scaled, allowing training and the testing of the model.

```
import numpy as np
x1,x2,x3,y=df_timeseries['one_month_shift'],df_timeseries['two_month_shift'],df_timeseries['three_month_shift'],df_timeseries['VALUE']
x1,x2,x3,y=np.array(x1),np.array(x2),np.array(x3),np.array(y)
x1,x2,x3,y=x1.reshape(-1,1),x2.reshape(-1,1),x3.reshape(-1,1),y.reshape(-1,1)
final_x=np.concatenate((x1,x2,x3),axis=1)
print(final_x)
```

Python

```
X_train,X_test,y_train,y_test=final_x[:-30],final_x[-30:],y[:-30],y[-30:]
```

Python

```
# Scaling the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Python

Figure 22: Data Preprocessing and Feature Scaling

The code 23 given is set up to optimise and evaluate performance of an Random Forest Regression model by consistently trying out the different combination of the hyperparameter. First, training the dataset that is y_train is changed into a one-dimensional file to make sure it works with training process. Then, an fixed list of hyperparameter sets is given, which includes the different number of trees that is n_estimators, the number of features that are taken into the account for each decision split max_features, and a consistent random state to make results repeatable.

After that in code we can see an repeated process is used in which every combination of hyperparameter is used to create an Random Forest model and train it using training

data. After training, predictions is made on the test dataset. Three metric RMSE, R2, and the MAPE has been used to measure how accurate model's forecast are. Then, the metrics and hyperparameter combinations for each model are saved in an structured way in the 'evaluation_results_df' DataFrame so that the analysis can be done. After the evaluation loop finishes, the code finds and shows best set of hyperparameter.

```python
# Reshaping y_train to a one-dimensional array
y_train_reshaped = y_train.ravel()

# Defining different hyperparameters to try
hyperparameters_list = [
    {'n_estimators': 100, 'max_features': 2, 'random_state': 1},
    {'n_estimators': 200, 'max_features': 3, 'random_state': 1},
    {'n_estimators': 300, 'max_features': 2, 'random_state': 1},
    {'n_estimators': 150, 'max_features': 'auto', 'random_state': 1},
]

# DataFrame to store the evaluation results
evaluation_results_df = pd.DataFrame(columns=["Hyperparameters", "RMSE", "R-squared", "MAPE (%)"])

# Evaluating each hyperparameter combination
for hyperparameters in hyperparameters_list:
    # Creating and training the Random Forest model
    model = RandomForestRegressor(**hyperparameters)
    model.fit(X_train, y_train_reshaped)

    # Making predictions on the testing data
    Ran_pred = model.predict(X_test)

#Random Forest Evaluation
    Randomforest_rmse=sqrt(mean_squared_error(Ran_pred,y_test))
    Randomforest_r_square = r2_score(y_test, Ran_pred)
    Randomforest_MAPE= MAPE(y_test,Ran_pred)

    # Appending the results to the DataFrame
    evaluation_results_df = evaluation_results_df.append({
        "Hyperparameters": str(hyperparameters),
        "RMSE": Randomforest_rmse,
        "R-squared": Randomforest_r_square,
        "MAPE (%)": Randomforest_MAPE
    }, ignore_index=True)

# Finding the best hyperparameters based on the lowest RMSE
best_hyperparameters = evaluation_results_df.loc[evaluation_results_df['RMSE'].idxmin()]

evaluation_results_df, best_hyperparameters
```
✓ 0.7s Python

Figure 23: Hyperparameter Tuning for Random Forest Regressor

```
# Plotting both the predicted values and the actual values
plt.rcParams["figure.figsize"] = (12,8)
plt.plot(Ran_pred, label='Random Forest Predictions', marker='x')
plt.plot(y_test.ravel(), label='Actual Values', marker='o')
plt.legend(loc="upper left")
plt.xlabel('Observation')
plt.ylabel('Unemployment Value')
plt.title('Comparison of Random Forest Predictions with Actual Values')
plt.grid(True)
plt.show()
✓ 0.1s
```

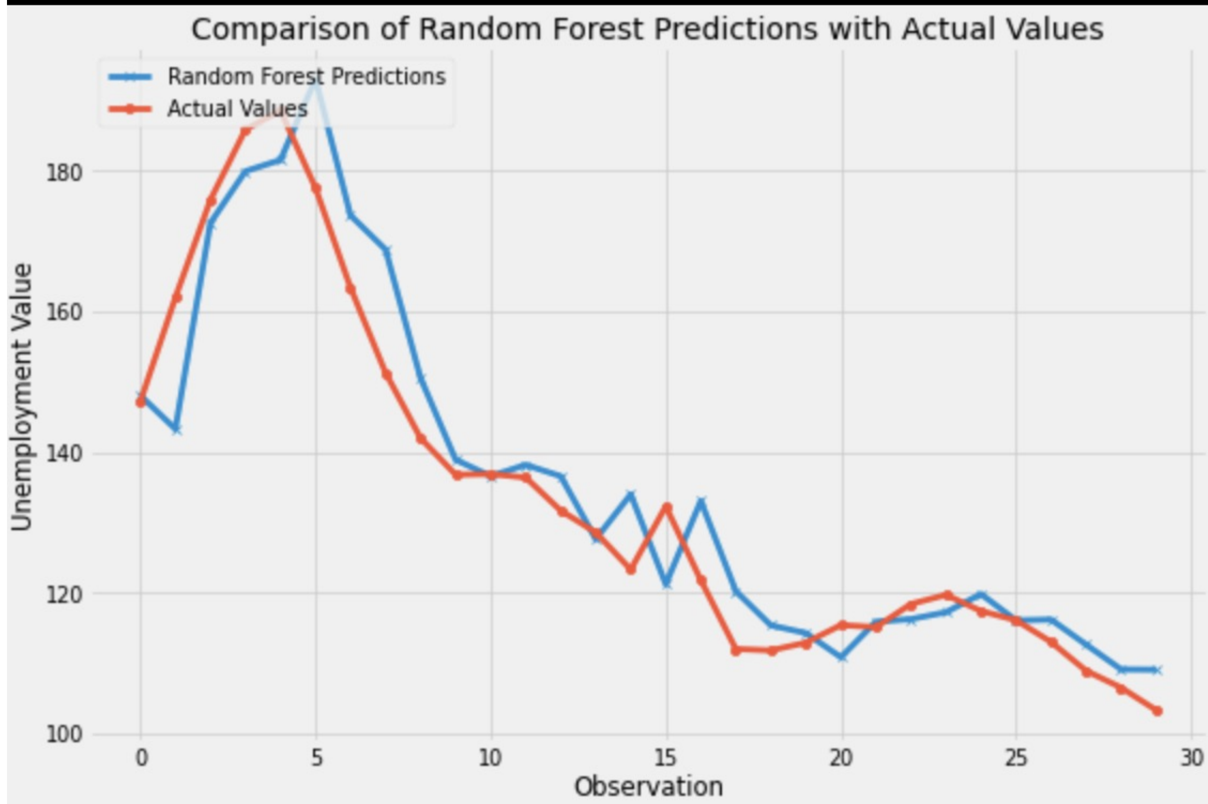

Figure 24: Prediction Plot of Random Forest

6.4 Ridge Regression

The code 25, is set up to test how well the Ridge Regression, a method for linear regression with L2 regularisation, works for an range of regularisation values that is set by the alphas list. By changing alpha, you can fine tune the mix between bias and the variation. The code is written such a way that , looping the each alpha number one by one. For each iteration, a Ridge Regression model are created with the current alpha, trained on the training dataset X_train and the y_train, and then used to make the predictions on test dataset (X_test). Then, for these prediction, the Root Mean Squared Error (RMSE), the $R\hat{2}$, and MAPE are calculated. These percentages show how big the model's errors are,

how well it fits data and how accurate the predictions. The ridge_results_df file keeps track of each model performance and the related alpha.

RIDGE REGRESSION

```python
# Defining different alpha values for Ridge Regression
alphas = [0.001, 0.01, 0.1, 1, 10]

# Scaling the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Evaluating Ridge Regression with different alpha values
ridge_results_df = pd.DataFrame(columns=["Alpha", "RMSE", "R-squared", "MAPE (%)"])
for alpha in alphas:
    ridge_model = Ridge(alpha=alpha)
    ridge_model.fit(X_train_scaled, y_train_reshaped)
    ridge_pred = ridge_model.predict(X_test_scaled)
    ridge_rmse = np.sqrt(mean_squared_error(ridge_pred, y_test))
    ridge_r_square = r2_score(y_test, ridge_pred)
    ridge_MAPE  = MAPE(y_test,ridge_pred) # Calculating MAPE
    ridge_results_df = ridge_results_df.append({
        "Alpha": alpha,
        "RMSE": ridge_rmse,
        "R-squared": ridge_r_square,
        "MAPE (%)": ridge_MAPE
    }, ignore_index=True)

best_alpha_ridge = ridge_results_df.loc[ridge_results_df['RMSE'].idxmin()]

ridge_results_df, best_alpha_ridge
```

✓ 0.0s Python Python

```
(    Alpha       RMSE  R-squared   MAPE (%)
 0   0.001   5.859365   0.942061  19.796930
 1   0.010   5.883889   0.941575  19.779870
 2   0.100   6.179207   0.935563  19.697289
 3   1.000   7.529578   0.904322  19.525697
 4  10.000  10.846215   0.801470  19.622250,
```

Figure 25: Performance Evaluation of Ridge Regression

The real test values are then plotted next to the predictions so that an clear comparison can be made. The graph shown in Figure 26 gives how well the Ridge Regression predictions match up with the real test data. Also, the models first five results are shown so that you can get an quick look at them.

```python
# Using the best alpha value for Ridge regression to forecast the test data
best_alpha = best_alpha_ridge["Alpha"]
ridge_best_model = Ridge(alpha=best_alpha)
ridge_best_model.fit(X_train_scaled, y_train_reshaped)
ridge_best_pred = ridge_best_model.predict(X_test_scaled)

# Plotting the actual vs predicted values
plt.plot(ridge_best_pred, label='Ridge Regression Predictions')
plt.plot(y_test, label='Actual Values')
plt.legend(loc="upper left")
plt.title("Ridge Regression Forecast vs Actual Values")
plt.xlabel("Time")
plt.ylabel("Value")
plt.show()

# Returning the predicted values
ridge_best_pred[:5]
```

✓ 0.1s

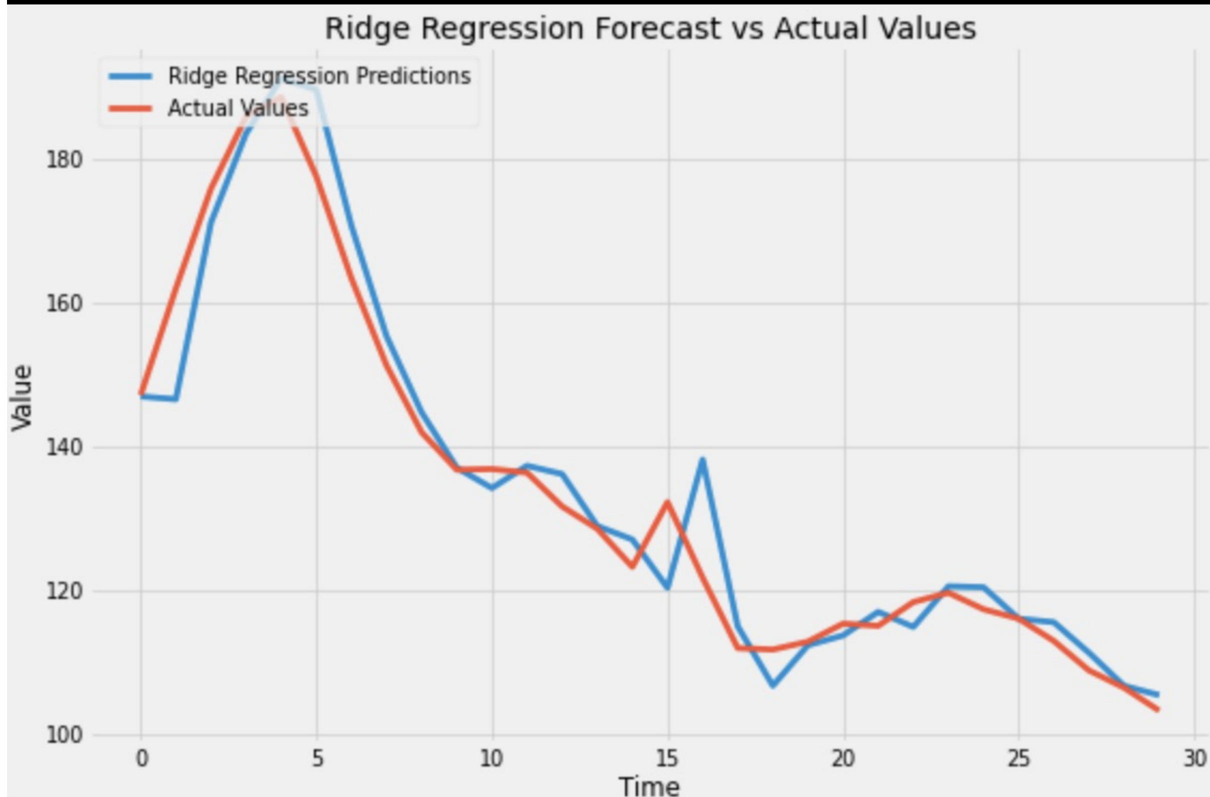

Figure 26: Visualization and Forecasting with Optimal Ridge Regression Model

6.5 KNN model

The code section gives the full evaluation of the k-Nearest regression method. It looks at how well the KNN works for the different neighbourhood sizes, specifically for the k values ranging from 1 to 5. The k-NN regression method predicts the goal value by taking an average of k nearest data points in the feature space.

The process works like this for each step in the range:

- A k-NN regressor is set up with an current k value. After the model is set up, it is trained with the X_train dataset, which gets it ready to make predictions on data. Once the model has trained, it can predict the values for the test sample called X_test.
- After making an prediction, 3 rating metrics are used to measure how well the model worked that are RMSE, MAPE and the R-square This evaluation shows how the choice of neighbours affects an capacity of the kNN model to predict future. It does this by iterating over an range of k numbers.

KNN Model

```
rmse_val_knn = []
mape_val_knn = []
r_squre_knn = []

for K in range(5):
    K = K+1
    model = neighbors.KNeighborsRegressor(n_neighbors = K)
    model.fit(X_train, y_train.ravel())
    knn_pred = model.predict(X_test)
    error = sqrt(mean_squared_error(y_test, knn_pred))
    rmse_val_knn.append(error)
    Knn_r_square = r2_score(y_test, knn_pred)
    r_squre_knn.append(Knn_r_square)

    Knn_MAPE = MAPE(y_test,knn_pred)# Calculating MAPE
    mape_val_knn.append(Knn_MAPE)
    print('RMSE for k= ' , K , 'is:', error)
    print('MAPE for k= ' , K , 'is:', Knn_MAPE)
    print('R-squared for k= ' , K , 'is:', Knn_r_square)
    print()

# Return the lists with metric values
rmse_val_knn, mape_val_knn, r_squre_knn
```

✓ 0.0s

```
RMSE for k=  1 is: 9.52741657883535
MAPE for k=  1 is: 19.899724342478876
R-squared for k=  1 is: 0.8468132842862826

RMSE for k=  2 is: 10.246743385095584
MAPE for k=  2 is: 19.628199959274326
R-squared for k=  2 is: 0.8228086505730655

RMSE for k=  3 is: 8.94335259036317
MAPE for k=  3 is: 19.553552099735647
R-squared for k=  3 is: 0.8650193542038314
```

Figure 27: Evaluating k-Nearest Neighbors Regression for Different Values of k

Based on what worked best in previous code, This code uses the best KNN with k=4. The model is learned on the X_train dataset, and then it predicts values for X_test dataset. A plot is made so that these expectations and the real y_test values can be seen side by side. The k-NN prediction are shown in the blue on graph, while actual number are shown in the red. This makes it easy to compare the predicted and the real data points.

```
# Using k=4 for the K-Neighbors Regressor (as it produced the best results)
k_best = 4
model_best_knn = neighbors.KNeighborsRegressor(n_neighbors=k_best)
model_best_knn.fit(X_train, y_train.ravel())
knn_pred_best = model_best_knn.predict(X_test)

# Plotting the predictions vs actual values
plt.figure(figsize=(12, 8))
plt.plot(knn_pred_best, label='KNN Predictions (k=4)', color='blue')
plt.plot(y_test, label='Actual Values', color='red')
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('K-Nearest Neighbors Prediction vs Actual Values')
plt.legend(loc="upper left")
plt.show()
```

✓ 0.0s

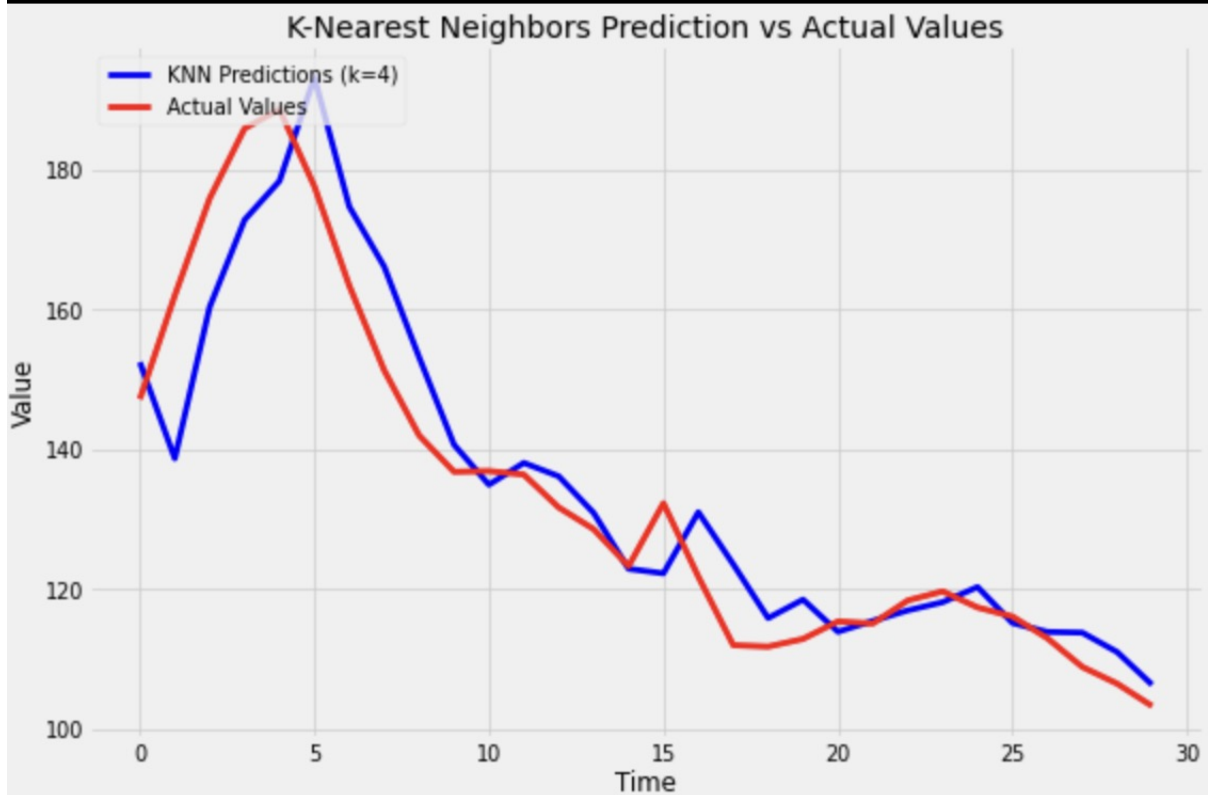

Figure 28: KNN Regression Predictions with Actual Data for k=4

6.6 XGBoost

The provided code 29 snippet divides xgboost_df dataset into the separate training and test set using an predetermined date limit of '01-01-2015'. The data preceding this specified date is utilised for purpose of the training, while the data later to this date is allocated for testing the model. A plot is constructed in order to graphically represent the division. The graphical representation offer an clear illustration of distribution of

the data and distinct boundary between training and test datasets.

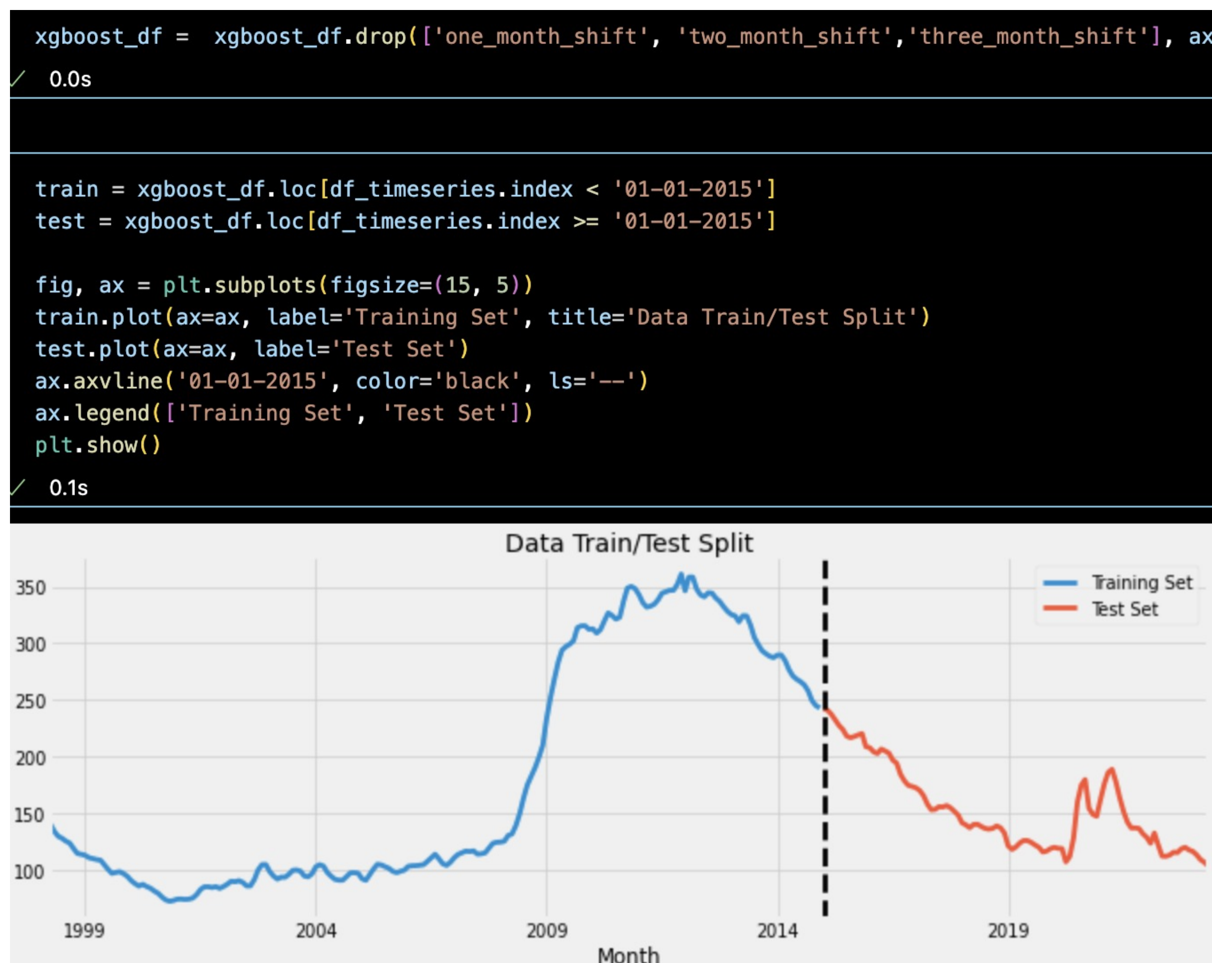

Figure 29: Visual Representation of Train/Test Data Split for XGBoost Model

```
def create_features(xgboost_df):
    """
    Create time series features based on time series index.

    """
    xgboost_df['one_month_shift']=xgboost_df['VALUE'].shift(+1)
    xgboost_df['two_month_shift']=xgboost_df['VALUE'].shift(+2)
    xgboost_df['three_month_shift']=xgboost_df['VALUE'].shift(+3)
    xgboost_df = xgboost_df.dropna()

    return xgboost_df

df = create_features(xgboost_df)
✓ 0.0s                                                                Python

xgboost_df = xgboost_df.dropna()
xgboost_df
✓ 0.0s                                                                Python
```

Figure 30: Feature Engineering for Time Series Data in XGBoost Model.

The code 31 is set up to test how well the XGBoost regression model work with the different amounts of boosting round, also called estimators. The list n_estimators_list shows the different amount of the boosting rounds to be evaluated 10, 100, 200, and 1000. A XGBoost regressor is set up with certain setting, such as current number of estimator, a fixed learning rate of 0.01, and an maximum tree depth of 10.

The X_train dataset is used to train model, and both training and test data are used to test the model. After training process, model is used to predict value on the X_test dataset. The accuracy of these predictions is then measured using three metrics Root Mean Squared Error, coefficient of determination, and MAPE . The 'evaluation_results_df' dataframe stores these measures along with the amount of estimator that is currently running. This gives an consolidated view of how well different configurations are working. After the evaluations is done, the dataframe with the results for all the tried numbers of estimators that are printed. This gives an clear picture of how the performance of the model changes with the number of boosting round.

```
# List of n_estimators to try
n_estimators_list = [10, 100, 200  , 1000]

# DataFrame to store the evaluation results
evaluation_results_df = pd.DataFrame(columns=["n_estimators", "RMSE", "R-squared", "MAPE (%)"])

# Looping through different n_estimators
for n_estimators in n_estimators_list:
    # Defining and training the XGBoost model
    reg = xgb.XGBRegressor(base_score=0.5, booster='gbtree',
                           n_estimators=n_estimators,
                           early_stopping_rounds=100,
                           objective='reg:linear',
                           max_depth=10,
                           learning_rate=0.01)

    reg.fit(X_train, y_train,
        eval_set=[(X_train, y_train), (X_test, y_test)],
        verbose=100)

    # Making predictions on the test set
    xgb_pred = reg.predict(X_test)

    # Calculating evaluation metrics
    rmse = mean_squared_error(y_test, xgb_pred, squared=False)
    r_square = r2_score(y_test, xgb_pred)
    mape = MAPE(y_test, xgb_pred)  # Define MAPE function as per your requirement

    # Appending the results to the DataFrame
    evaluation_results_df = evaluation_results_df.append({
        "n_estimators": n_estimators,
        "RMSE": rmse,
        "R-squared": r_square,
        "MAPE (%)": mape
    }, ignore_index=True)

# Print the evaluation results
print(evaluation_results_df)
```

Figure 31: Evaluation of XGBoost Regression Model Across Different Numbers of Estimators

The code 32 first figures out best number of boosting rounds that is n_estimators. The XGBoost model is set up and trained on the X_train dataset using this best number. The model makes prediction based on the X_test data after it has trained. This makes sure that the prediction are made using best way to set up the model.

```python
# Identifying the best n_estimators value
best_n_estimators = evaluation_results_df.loc[evaluation_results_df['RMSE'].idxmin(), 'n_estimators']

# Training the XGBoost model with the best n_estimators
best_model = xgb.XGBRegressor(base_score=0.5, booster='gbtree',
                              n_estimators=int(best_n_estimators),
                              early_stopping_rounds=100,
                              objective='reg:linear',
                              max_depth=10,
                              learning_rate=0.01)
best_model.fit(X_train, y_train,
               eval_set=[(X_train, y_train), (X_test, y_test)],
               verbose=100)

# Making predictions using the best model
best_pred = best_model.predict(X_test)
```

```
[64] ✓ 0.4s                                                            Python

[23:23:25] WARNING: /Users/runner/work/xgboost/xgboost/python-package/build/temp.macosx-10.9-x86_64-cpython-38/xgboost/src/objective/regression_obj.cu:213: reg:linear is now deprecate
[0]	validation_0-rmse:205.53830	validation_1-rmse:153.23147
[100]	validation_0-rmse:77.74616	validation_1-rmse:54.59836
[200]	validation_0-rmse:29.74112	validation_1-rmse:17.06016
[300]	validation_0-rmse:11.80804	validation_1-rmse:9.39711
[388]	validation_0-rmse:5.73543	validation_1-rmse:11.43891
```

Figure 32: Optimal XGBoost Model Training and Prediction

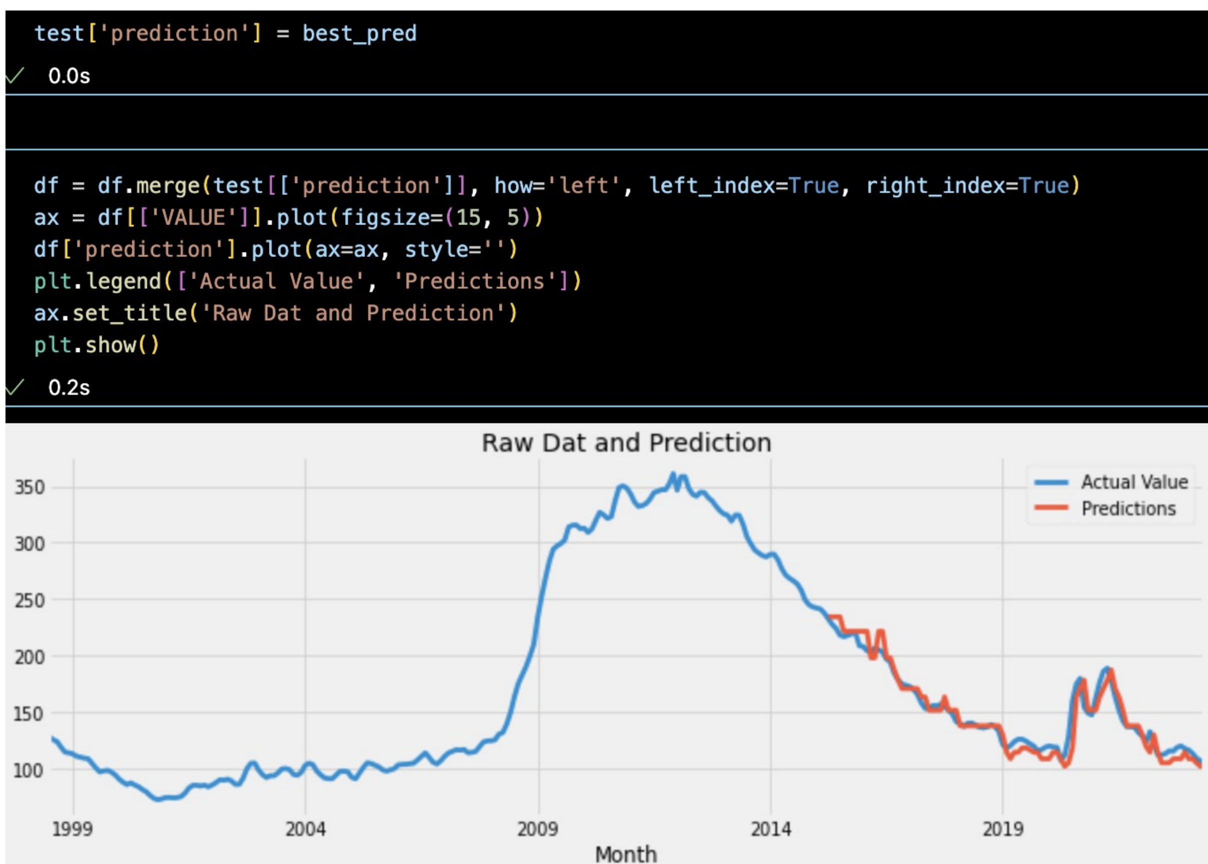

Figure 33: Visual Comparison of Actual Values and XGBoost Predictions