PREDICTING THE NUMBER OF BREAKDOWNS USING TIME SERIES WITH MACHINE LEARNING & PYTHON

A Project in The Fulfilment of The Industrial Orientation Training in IT &ERP DEPARTMENT

VISAKHAPATNAM STEEL PLANT

Submitted by

VAIDEHI VIPPARTI

Trainee No.: **100030786**

Department of Information Technology Engineering

Anil Neerukonda Institute of Technology and Sciences

Under the esteemed guidance of

Mr. T. V. Kameswara Rao

RASHTRIYA ISPAT NIGAM LIMITED (RINL), VISAKHAPATNAM

(Duration: **27 MAY 2024 – 22 JUNE 2024**)

PREDICTING THE NUMBER OF BREAKDOWNS USING TIME SERIES WITH MACHINE LEARNING & PYTHON

A Project in The Fulfilment of The Industrial Orientation Training in IT &ERP DEPARTMENT

<u>VISAKHAPATNAM STEEL PLANT</u>

Submitted by

VIPIN KAULWAR

Trainee No.: **100032482**

Department of Biotechnology

NIT DURGAPUR

Under the esteemed guidance of

Mr. T. V. Kameswara Rao

RASHTRIYA ISPAT NIGAM LIMITED (RINL), VISAKHAPATNAM

(Duration: **27 MAY 2024 – 22 JUNE 2024**)

PREDICTING THE NUMBER OF BREAKDOWNS USING TIME SERIES WITH MACHINE LEARNING & PYTHON

A Project in The Fulfilment of The Industrial Orientation Training in IT &ERP DEPARTMENT

<u>VISAKHAPATNAM STEEL PLANT</u>

Submitted by

CHEEKOLU ANU

Trainee No.: **100032365**

Department of Biotechnology

NIT DURGAPUR

Under the esteemed guidance of

Mr. T. V. Kameswara Rao

RASHTRIYA ISPAT NIGAM LIMITED (RINL), VISAKHAPATNAM

(Duration: **27 MAY 2024 – 22 JUNE 2024**)

PREDICTING THE NUMBER OF BREAKDOWNS USING TIME SERIES WITH MACHINE LEARNING & PYTHON

A Project in The Fulfilment of The Industrial Orientation Training in IT &ERP DEPARTMENT

<u>VISAKHAPATNAM STEEL PLANT</u>

Submitted by

BANDARU DEEPTHI BHAVANI

Trainee No.: **100035420**

Department of Computer Science and Engineering

Andhra University College of Engineering

Under the esteemed guidance of

Mr. T. V. Kameswara Rao

RASHTRIYA ISPAT NIGAM LIMITED (RINL), VISAKHAPATNAM

(Duration: **27 MAY 2024 – 22 JUNE 2024**)

PREDICTING THE NUMBER OF BREAKDOWNS USING TIME SERIES WITH MACHINE LEARNING & PYTHON

A Project in The Fulfilment of The Industrial Orientation Training in IT &ERP DEPARTMENT

VISAKHAPATNAM STEEL PLANT

Submitted by

Y JAYA KAIVALYA

Trainee No.: **100032421**

Department of Biotechnology

NIT DURGAPUR

Under the esteemed guidance of

Mr. T. V. Kameswara Rao

RASHTRIYA ISPAT NIGAM LIMITED (RINL), VISAKHAPATNAM

(Duration: **27 MAY 2024 – 22 JUNE 2024**)

PREDICTING THE NUMBER OF BREAKDOWNS USING TIME SERIES WITH MACHINE LEARNING & PYTHON

A Project in The Fulfilment of The Industrial Orientation Training in IT &ERP DEPARTMENT

VISAKHAPATNAM STEEL PLANT

Submitted by

Tankala Kavyasri

Trainee No.: **100032076**

Department of Computer Science and Engineering

Andhra University college of Engineering for Women's

Under the esteemed guidance of

Mr. T. V. Kameswara Rao

RASHTRIYA ISPAT NIGAM LIMITED (RINL), VISAKHAPATNAM

(Duration: **27 MAY 2024 – 22 JUNE 2024**)

PREDICTING THE NUMBER OF BREAKDOWNS USING TIME SERIES WITH MACHINE LEARNING & PYTHON

A Project in The Fulfilment of The Industrial Orientation Training in IT &ERP DEPARTMENT

VISAKHAPATNAM STEEL PLANT

Submitted by

CH DEEPIKA

Trainee No.: **100033639**

Department of Computer Science and Engineering

Andhra University college of Engineering for Women's

Under the esteemed guidance of

Mr. T. V. Kameswara Rao

RASHTRIYA ISPAT NIGAM LIMITED (RINL), VISAKHAPATNAM

(Duration: **27 MAY 2024 – 22 JUNE 2024**)

CERTIFICATE

This to Certify that following students are engaged in the project titled

This is to certify that the project work entitled " **PREDICTING THE NUMBER OF BREAKDOWNS USING TIME SERIES**" is a record of work done by **Vaidehi Vipparti (Tr.No:100030786)** student of ANITS Bheemunipatnam, **Vipin Kaulwar (Tr. No:100032482),** student of NIT DURGAPUR, **Cheekolu Anu (Tr. No:100032365)** student of NIT DURGAPUR, **Bandaru Deepthi Bhavani (Tr. No: 100035420)** student of ANDHRA UNIVERSITY Visakhapatnam, **Y Jaya Kaivalya (Tr. No: 100032421)** student of NIT DURGAPUR, **Tankala Kavyasri (Tr. No.: 100032076), CH Deepika (Tr. No.: 100033639)** student of ANDHRA UNIVERSITY Visakhapatnam, in the partial fulfillment of the requirement for the completion of internship during the period 27 May 2023 to 22 June 2023 in IT & ERP Department RINL-VSP.

(Signature of Project guide)
Shri Mr. T. V. Kameswara Rao
Deputy General Manager
D.G.M (IT &ERP)
IT & ERP Department
RINL-VSP

ACKNOWLEDGEMENT

I would like to extend our sincere appreciation and gratitude to all the individuals and organization who have contributed to the successful completion of the data analytics project on material consumption in steel plant. Their invaluable support, expertise, and dedication have been instrumental in the accomplishment of our objectives and the enhancement of our operations.

I take this opportunity to sincerely thank Mr. T. V. Kameswara Rao, D.G.M., IT & ERP Department, Visakhapatnam Steel Plant, for guiding me with his immense knowledge and helping in complete this project successfully.

I sincerely thank Mr. S. K. Mishra, GM I/C, IT & ERP Department, and Visakhapatnam Steel Plant, for investing his time in giving us an overview and helping us in completing the project successfully.

I wish to express my sincere thanks to the IT and ERP Department employees of Visakhapatnam Steel Plant for their valuable guidance in completing this project.

INDEX

1. ABSTRACT

In the industrial sector, equipment breakdowns can lead to significant production losses, downtime and increased maintenance costs. Predicting equipment failures in advance plays a crucial role in preventing such disruptions and optimizing maintenance strategies. This abstract presents a machine learning project that primarily focuses on predicting equipment breakdowns in an industry using time series models. The project utilizes historical data collected from the industry's equipment, including various operational parameters and maintenance records. A time series analysis approach is adopted to model the temporal patterns and dependencies in the data, allowing for accurate prediction of future equipment failures Several time series models are explored, including autoregressive integrated moving averages (ARIMA), seasonal ARIMA (SARIMA), and recurrent neural networks (RNNs). The models are trained on the historical data, and their performance is evaluated using appropriate evaluation metrics such as mean squared error (MSE) or root mean squared error (RMSE).

To improve the predictive accuracy, hyperparameter tuning and model selection techniques are applied. Cross-validation is performed to assess the generalization performance of the models, ensuring their robustness when applied to unseen data. The project aims to provide a reliable and proactive maintenance strategy for the industry, allowing for timely interventions before equipment failures occur. By accurately predicting breakdowns, the industry can optimize maintenance schedules, minimize downtime, reduce repair costs, and enhance overall operational efficiency. The results of the project will be validated using real-time data from the industry, and the predictive models will be integrated into the existing maintenance system for practical deployment. The successful implementation of this machine learning project will significantly contribute to the industry's productivity, reliability, and cost-effectiveness.

2. INTRODUCTION

1.1 INTRODUCTION TO THE ORGANISATION

Visakhapatnam Steel Plant, the first coast-based Steel Plant of India is located, 16km South West of city of destiny ie, Visakhapatnam. Bestowed with modern technologies, VSP has an installed capacity of 3 million Tons per annum of Liquid Steel and 2.656 million Tons of saleable steel.

At VSP there is emphasis on total automation, seamless integration and efficient up gradation, which result in wide range of long and structural products to meet stringent demands of discerning customers within India and abroad. VSP products meet exalting International Quality Standards such as JIS, BIS, DIN, and BS etc.

VSP has become the first integrated Steel Plant in the country to be certified to all the three international standards for quality (ISO-9001), for Environment Management (ISO-14001) and for Occupational Health and Safety (OHSAS-18001). The certificate covers quality systems of all operational, maintenance, service units besides Purchase systems, Training and Marketing functions.

Spreading over 4 Regional Marketing Offices, 20 branch offices and 22 stock yards located all over the country.

VSP by successfully installing and operating efficiently Rs. 460 crores worth of Pollution Control and Environment Control Equipment's and covering the barren landscape by planting more than 3 million plants has made the Steel Plant, Steel Township and surrounding areas into a heaven of lush greenery. This has made Steel Township greenery. This has made Steel Township a greener, cleaner and cooler place, which can boast of 3 to 4 degrees C lesser temperature even in the peak summer compared to Visakhapatnam City.

VSP exports Quality Pig Iron &amp; products to Sri Lanka, Myanmar, Nepal, Middle East, USA and South East Asia (Pig iron). RINL-VSP was awarded "Star Trading House" status during 1997-2000. Having established a fairly dependable export market, VSP plans to make a continuous presence in the export market.

Having a total manpower of about 14,449 VSP has envisaged a labour productivity of 265 Tons per man year of Liquid Steel which is the best in the country and comparable with the international levels.

1.2 HALLMARK OF VIZAG STEEL AS AN ORGANISATION:

Today, VSP is moving forward with an aura of confidence and with pride amongst its employees who are determined to give best for the company to enable it to reach new heights in organizational excellence.

Futuristic enterprises, academic activity, planned and progressive residential localities are but few of the plentiful ripple effects of this transformation and each one of us take immense pride to uphold the philosophy of mutual (i.e., individual and societal) progress.

3. MODEL DEVELOPMENT

The approach used to predict the development model for the project involves the following steps:

1. Data Preprocessing: The input dataset is pre-processed to handle missing values. In this case, the missing values in the BD Report time and BD production delay are filled with the median values. Additionally, the dataset is split into input features (X) and target variables (y).

2. Training and Testing Split: The dataset is split into training and testing sets using the train_test_split function from the scikit-learn library. This allows for model training on a portion of the data and evaluation on unseen data.

3. Model Selection and Training: ARIMA model is chosen as the prediction model. The model is trained using the training set at different time intervals as target variables.

4. Model Evaluation: The trained model is evaluated using the testing set. Mean squared error (MSE) is calculated to assess the performance of the model.

5. Saving and Loading the Model: The trained model is saved to a file using the pickle module, allowing for future use without the need for retraining. The saved model can be loaded later for making predictions on new data.

4. METHODOLOGY

1. Data Collection:

- Identify and obtain the dataset containing historical breakdown data, including the relevant attributes such as year, week, department code, and breakdown count.

2. Data Preprocessing:

- Handle missing values: Check for missing values in the dataset and apply appropriate techniques such as imputation or removal of missing data points.
- Data type conversion: Convert the data types of relevant attributes to their appropriate formats, such as converting dates to datetime objects.
- Grouping and aggregation: Group the breakdown counts by day and department code to obtain the desired level of granularity for analysis.

3. Exploratory Data Analysis (EDA):

- Conduct descriptive analysis: Generate summary statistics, histograms, and other visualizations to gain insights into the dataset.
- Analyze trends and patterns: Explore the temporal trends, seasonality, and any other relevant patterns in the breakdown data.

4. Stationarity Analysis:

- Apply the Dickey-Fuller test: Check the stationarity of the time series data. If the series is found to be non-stationary, proceed to the next step.

5. Data Transformation:

- Differencing: Perform differencing or seasonal differencing to achieve stationarity. Apply appropriate transformations to make the time series data stationary.

6. Model Selection:

- Analyse ACF and PACF plots: Examine the autocorrelation and partial autocorrelation functions to identify the appropriate orders of the ARIMA model.
- Select the optimal model: Choose the appropriate values for the order of differencing (d), autoregressive (p), and moving average (q) components based on the analysis.

7. Model Training:

- Split the dataset: Divide the pre-processed data into training and validation sets using time series split.
- Train the ARIMA model: Fit the ARIMA model to the training dataset using the selected parameters.

8. Model Validation:

- Make predictions: Use the trained ARIMA model to predict the breakdown counts for the validation dataset.
- Evaluate the model: Assess the accuracy of the model predictions using evaluation metrics such as Mean Squared Error (MSE).

9. Model Refinement:

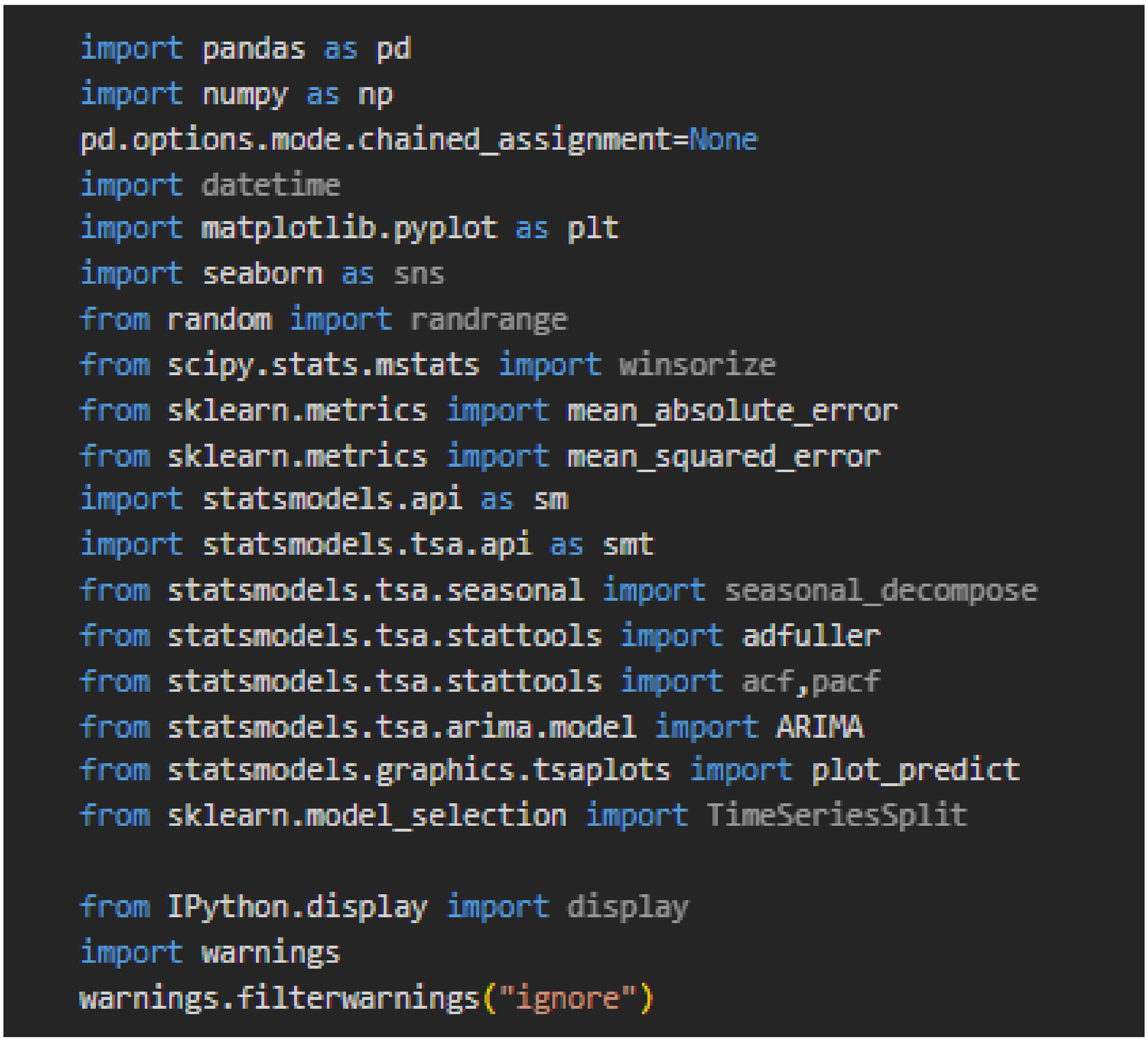
- Fine-tuning: If the model performance is not satisfactory, consider adjusting the model parameters or applying additional techniques such as hyperparameter tuning, seasonality adjustments or data transformations to improve the accuracy.

10. Model Deployment and Reporting:

- Apply the final model: Use the refined model to make predictions on new or unseen data.

- Document the findings: Summarize the methodology, data preprocessing steps, model selection process, evaluation metrics, and the overall results of the project.
- Report the conclusions: Provide insights and recommendations based on the analysis, highlighting the predictive capabilities of the ARIMA model for breakdown prediction.

5. DETAILED CODE EXPLANATION



```
import pandas as pd
import numpy as np
pd.options.mode.chained_assignment=None
import datetime
import matplotlib.pyplot as plt
import seaborn as sns
from random import randrange
from scipy.stats.mstats import winsorize
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
import statsmodels.api as sm
import statsmodels.tsa.api as smt
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.stattools import acf,pacf
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_predict
from sklearn.model_selection import TimeSeriesSplit

from IPython.display import display
import warnings
warnings.filterwarnings("ignore")
```

The first cell of the code imports the necessary modules for data handling, visualization , time series forecasting, and evaluation. Here's a breakdown of the modules imported:

Data Reading and Handling:

- **pandas (imported as pd):**

  Pandas is a library for data manipulation and analysis. It provides data structures like Data Frames and Series, which allow for easy handling and analysis of structured data. Pandas are a powerful open-source python library. It simplifies common data operations, enabling users to perform tasks like data cleaning, exploration and transformation. It also offers robust support for handling missing data and time series data. Overall, panadas is an essential tool for data manipulation and analysis in python.

- **numpy** (imported as **np**):

NumPy is a library for the Python programming language that provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently.

- **pd.options.mode.chained assignment:**

This is a pandas option that controls how warnings are displayed when performing chained assignment operations. It can be used to suppress or raise warnings related to chained assignment.

- **display from IPython.display:**

This is a module in the IPython library that provides utilities for displaying and rendering various types of output in Jupyter notebooks. It is commonly used to show data frames or visualizations in a notebook.

Data Visualization:

- **matplotlib.pyplot (imported as plt):** Matplotlib is a plotting library for Python. The pyplot module provides a convenient interface for creating various types of plots and visualizations.
- **seaborn (imported as sns):** Seaborn is a Python data visualization library based on Matplotlib. It provides a high-level interface for creating aesthetically pleasing and informative statistical graphics.

Time Series Forecasting:

- **statsmodels.tsa.stattools.adfuller:** This function is part of the statsmodels library and is used to perform the Augmented Dickey-Fuller test for testing the stationarity of a time series.
- **statsmodels.tsa.stattools.acf and statsmodels.tsa.stattools.pacf:** These functions are used to compute the autocorrelation function (ACF) and partial autocorrelation function (PACF) of a time series, respectively.
- **statsmodels.tsa.arima_model.ARIMA:** This class is used to create an ARIMA model for time series analysis. It is part of the statsmodels library and provides functionality for fitting and forecasting with ARIMA models.
- **statsmodels.graphics.tsaplots.plot predict:** This function is used to plot the predicted values from a time series model along with confidence intervals.

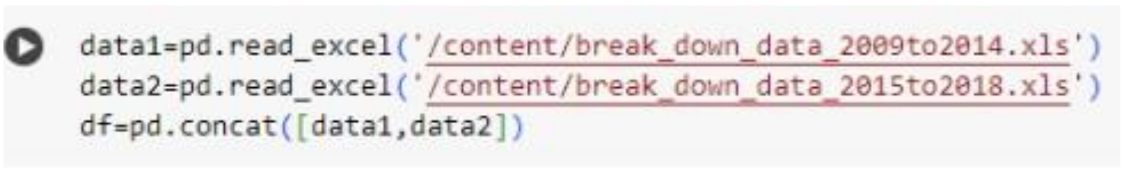
Evaluation:

- **sklearn.metrics.mean_absolute_error:** This function from the scikit-learn library is used to compute the mean absolute error (MAE) between the predicted values and the true values.
- **sklearn.metrics.mean_squared_error:** This function from scikit-learn is used to compute the mean squared error (MSE) between the predicted values and the true values. Visualizing AR Plots.
- **statsmodels.api:** This module in the stats models library provides an interface to access various statistical models and functions for data analysis.
- **statsmodels.tsa.api:** This module in the statsmodels library provides an API for time series analysis. It includes classes and functions for working with time series data, such as ARIMA models, seasonal decomposition, and forecasting.

Ignoring Warnings:

- **warnings:** The warnings module is a Python standard library module that provides a way to control warnings issued by the Python interpreter or other libraries. It allows you to filter, ignore, or raise warnings as needed.
- **Warnings.filterwarnings("ignore"):** This line of code is used to suppress warnings from being displayed. In this case, it ignores the warnings generated certain functions or operations.

These modules will be used throughout the project for various data operations, visualization, forecasting, and evaluation tasks.

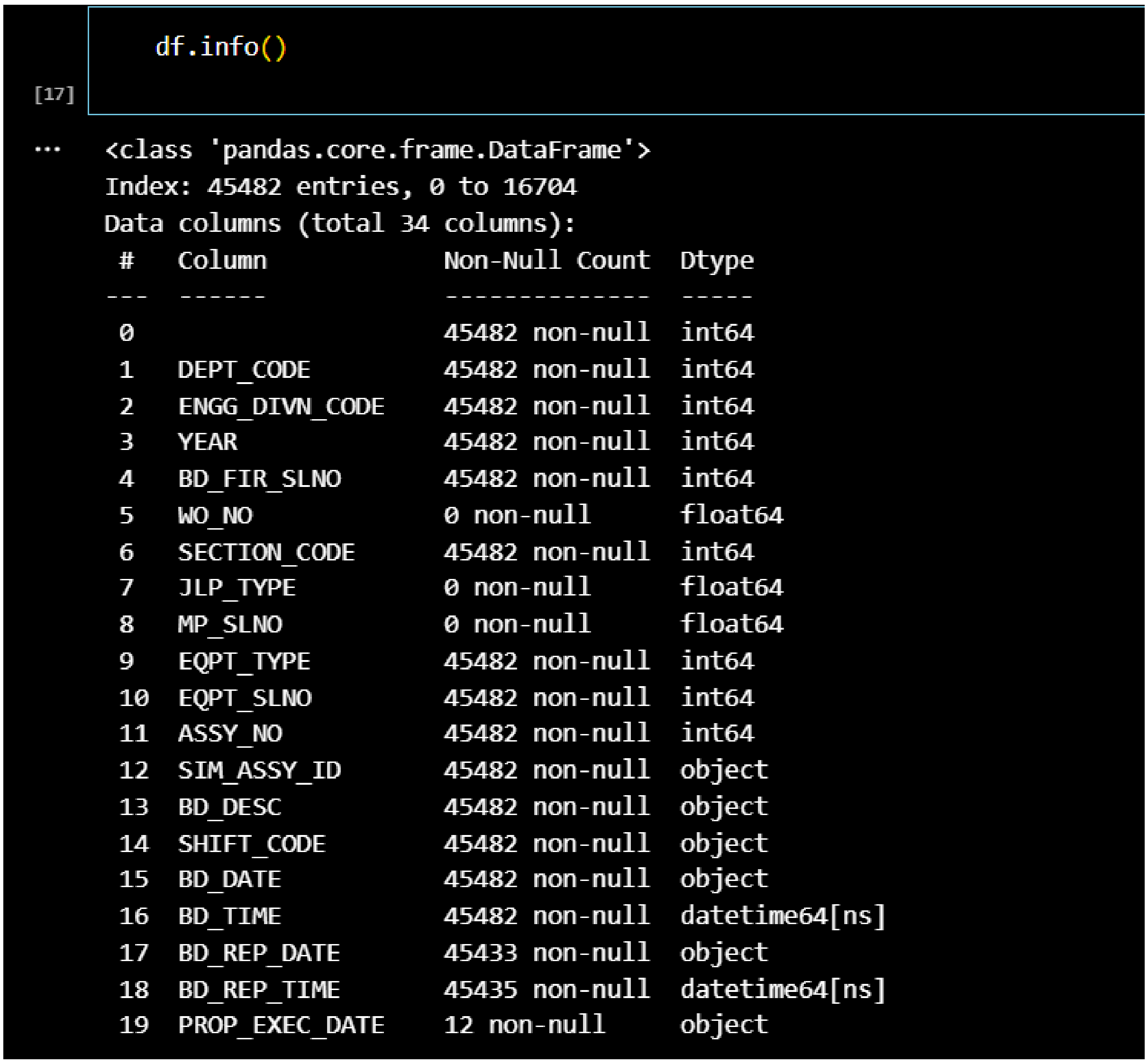


```
data1=pd.read_excel('/content/break_down_data_2009to2014.xls')
data2=pd.read_excel('/content/break_down_data_2015to2018.xls')
df=pd.concat([data1,data2])
```

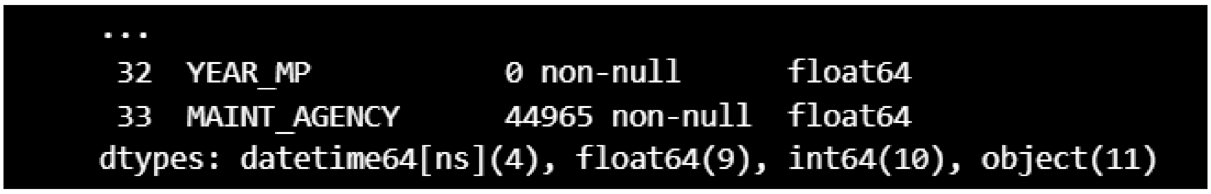
The code in this cell reads data from two Excel files and combines them into a single DataFrame. The process is explained as follows:

Firstly, the data is read using the **pd.read_excel()** function from the pandas library. The function takes the file paths of the Excel files as arguments. The data from the file 'break_down_data_2009to2014.xls' is read into a DataFrame named **data1**, while the data from the file 'break_down_data_2015to2018.xls' is read into a Data Frame named **data2.**

Next, the data from **data1 and data2** is combined into a single Data Frame using the **pd.concat()** function. This function concatenates the two Data Frames vertically, resulting in a merged Data Frame named **df.**

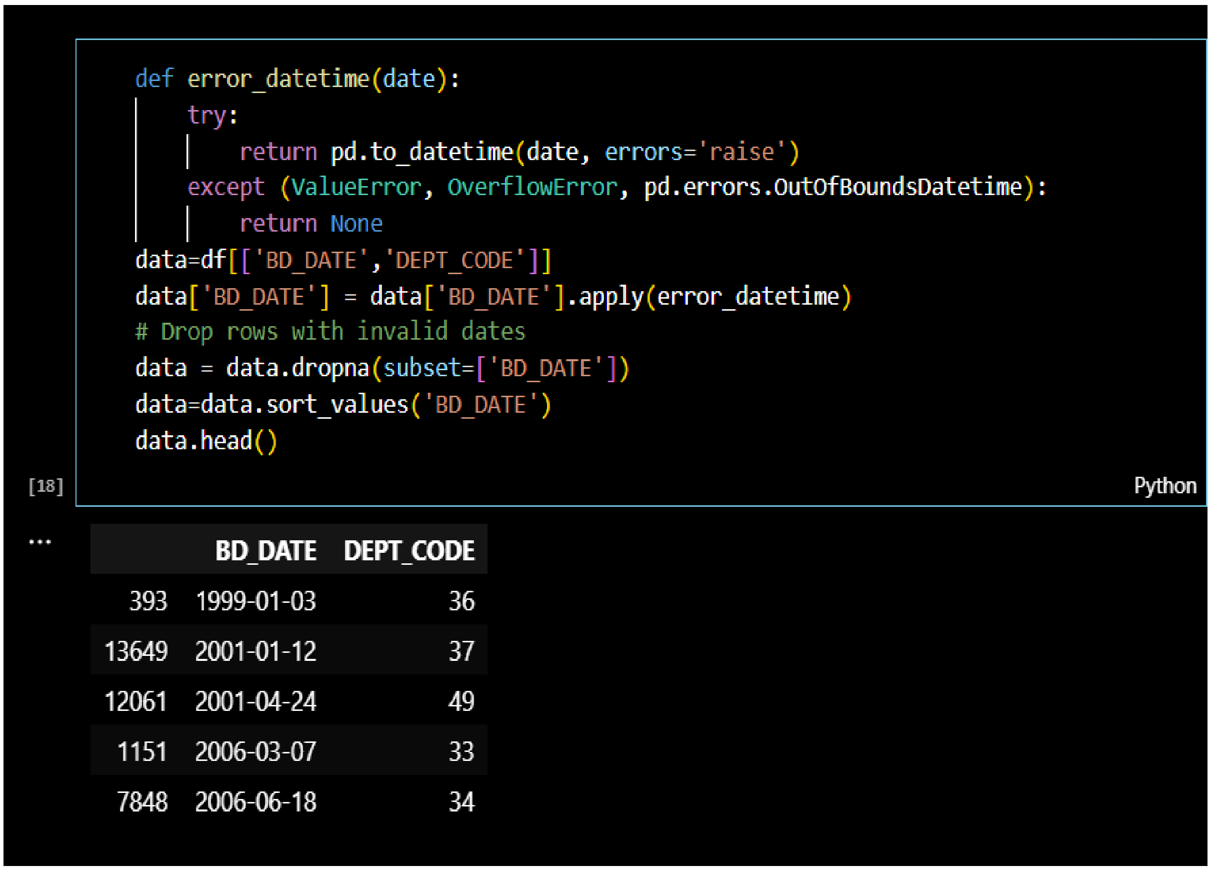


```
    df.info()
[17]

... <class 'pandas.core.frame.DataFrame'>
    Index: 45482 entries, 0 to 16704
    Data columns (total 34 columns):
     #   Column           Non-Null Count  Dtype
    ---  ------           --------------  -----
     0                    45482 non-null  int64
     1   DEPT_CODE        45482 non-null  int64
     2   ENGG_DIVN_CODE   45482 non-null  int64
     3   YEAR             45482 non-null  int64
     4   BD_FIR_SLNO      45482 non-null  int64
     5   WO_NO            0 non-null      float64
     6   SECTION_CODE     45482 non-null  int64
     7   JLP_TYPE         0 non-null      float64
     8   MP_SLNO          0 non-null      float64
     9   EQPT_TYPE        45482 non-null  int64
     10  EQPT_SLNO        45482 non-null  int64
     11  ASSY_NO          45482 non-null  int64
     12  SIM_ASSY_ID      45482 non-null  object
     13  BD_DESC          45482 non-null  object
     14  SHIFT_CODE       45482 non-null  object
     15  BD_DATE          45482 non-null  object
     16  BD_TIME          45482 non-null  datetime64[ns]
     17  BD_REP_DATE      45433 non-null  object
     18  BD_REP_TIME      45435 non-null  datetime64[ns]
     19  PROP_EXEC_DATE   12 non-null     object
```



```
 ...
 32  YEAR_MP           0 non-null      float64
 33  MAINT_AGENCY      44965 non-null  float64
dtypes: datetime64[ns](4), float64(9), int64(10), object(11)
```

By executing this code, the user successfully reads and combines the data from the two Excel files, creating a Data Frame df that contains the merged data. The subsequent use of df.info() provides a concise summary of the Data Frame, including information such as the number of rows, columns, and data types of each column.



```
def error_datetime(date):
    try:
        return pd.to_datetime(date, errors='raise')
    except (ValueError, OverflowError, pd.errors.OutOfBoundsDatetime):
        return None
data=df[['BD_DATE','DEPT_CODE']]
data['BD_DATE'] = data['BD_DATE'].apply(error_datetime)
# Drop rows with invalid dates
data = data.dropna(subset=['BD_DATE'])
data=data.sort_values('BD_DATE')
data.head()
```

[18] Python

...

```
          BD_DATE  DEPT_CODE
393    1999-01-03         36
13649  2001-01-12         37
12061  2001-04-24         49
1151   2006-03-07         33
7848   2006-06-18         34
```

In the above cell, the code focuses on selecting the variable that we want to forecast and performing some data preprocessing steps. Here's an explanation of the code:

1.Variable Selection:

- 'data=df[['BD_DATE','DEPT CODE']]': Selects the columns 'BD DATE' and 'DEPT CODE' from the Data Frame df and assigns the resulting subset to the variable 'data'. This step helps us isolate the specific variables we are interested in forecasting.

2. Data Preprocessing:

- ‘def error_date_time(date)’: Converts the 'BD_DATE’ column to datetime format using pd.to_datetime(). The errors = 'raise’ argument ensures that if the conversion fails due to an invalid date format, out-of-bounds date or overflow error, it returns NaN value.

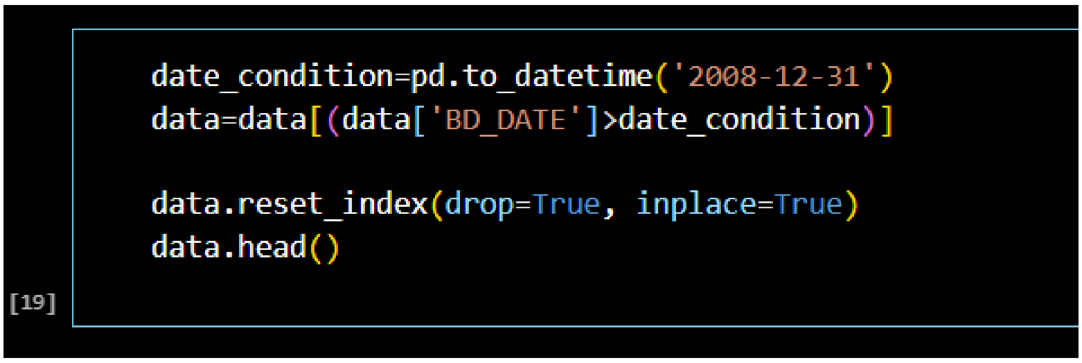
- ‘data=data.dropna(subset=['BD_DATE'])’: Drops any rows in the DataFrame data that contain missing values (NaN).

-‘data = data.sort_values('BD_DATE')’: Sorts the DataFrame data in ascending Order based on the 'BD_DATE' column. This step is important for time series analysis, as it ensures that the data is arranged chronologically.

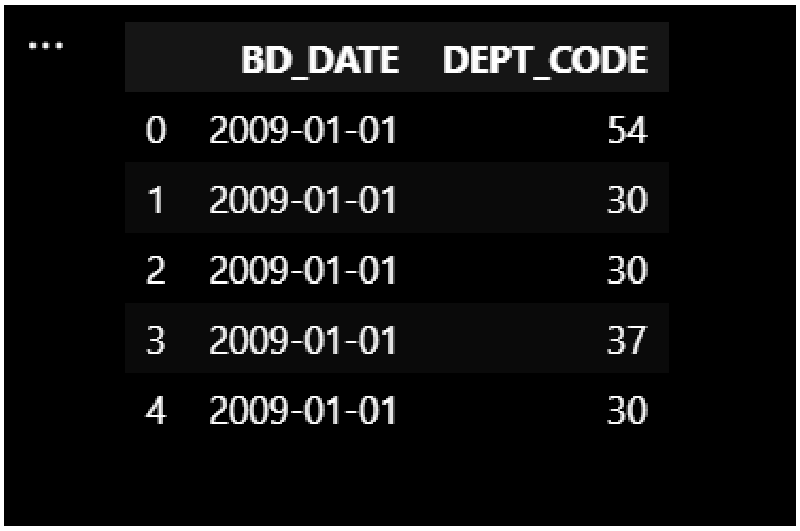
3. Displaying the Head of the DataFrame:

- ‘data.head()’: Displays the first few rows of the pre-processed Data Frame data using the ‘head()’ function.

By executing this code, the user selects the 'BD_DATE' and 'DEPT_CODE' columns from the original Data Frame df and assigns them to the variable 'data'. The 'BD DATE' column is then converted to datetime format, and the data is sorted chronologically. Any rows with missing values are dropped from the Data Frame. Finally, the data.head() statement is used to display the first few rows of the pre-processed Data Frame.



```
date_condition=pd.to_datetime('2008-12-31')
data=data[(data['BD_DATE']>date_condition)]

data.reset_index(drop=True, inplace=True)
data.head()
```
[19]



```
...
     BD_DATE   DEPT_CODE
0  2009-01-01         54
1  2009-01-01         30
2  2009-01-01         30
3  2009-01-01         37
4  2009-01-01         30
```

In the above cell, the code discards data before the year 2008, resets the index of the Data Frame, and performs data resampling. Here's an explanation of the code:

1.Discarding Data before 2008:

- ‘condition = pd.to_datetime("2008-12-31')’: Creates a condition by converting the specified date '2008-12-31' to a pandas datetime object using pd.to_datetime().

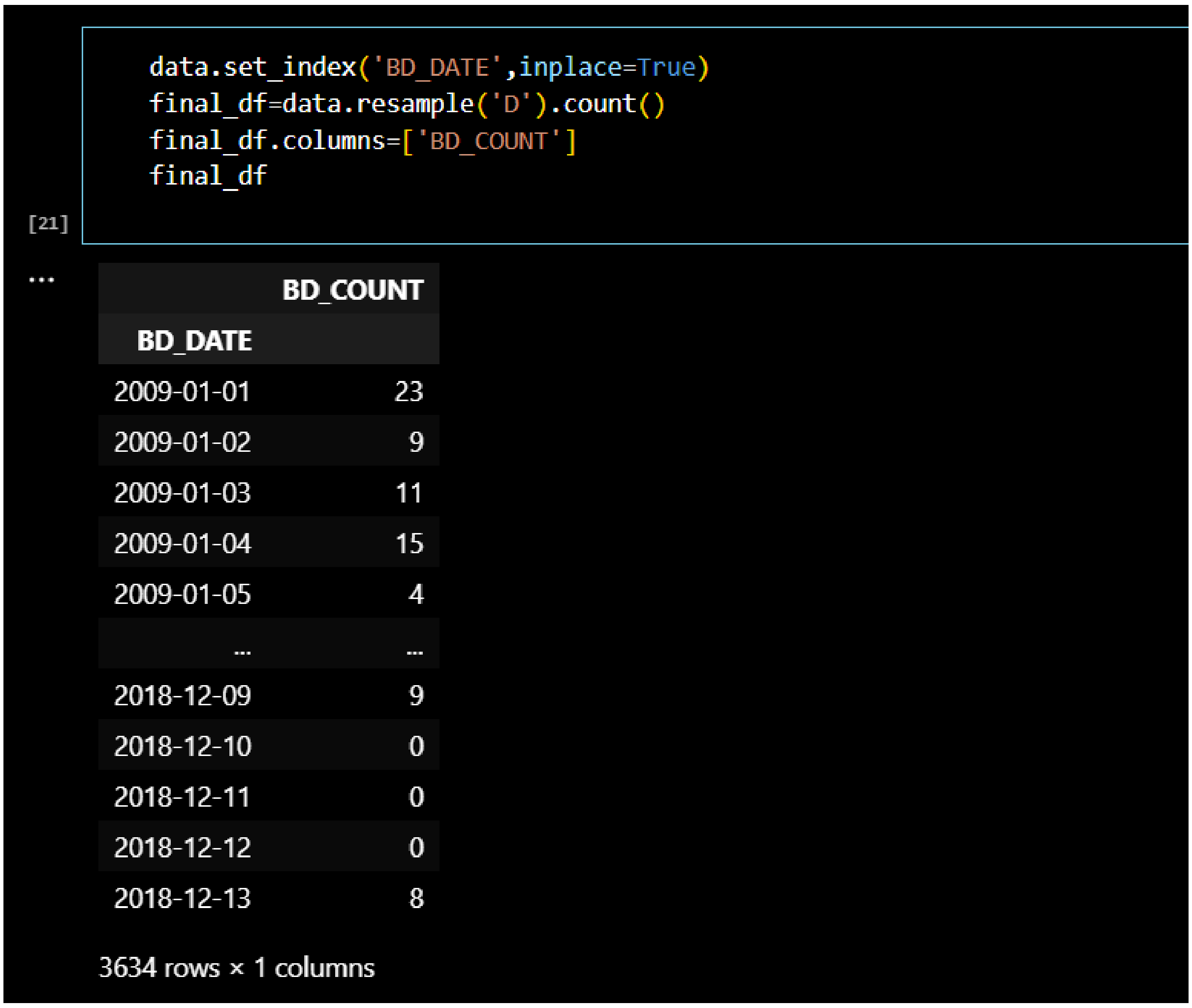
- ‘data=data[(data['BD_DATE] > condition)]’: Filters the DataFrame data to include only rows where the 'BD_DATE' is later than the specified condition. This step discards any data before the year 2008.

2.Resetting DataFrame Index:

- ‘data.reset_index(drop=True, inplace=True)’: Resets the index of the DataFrame 'data' after discarding data before 2008. The ‘drop=True’ argument ensures that the old index is dropped, and the ‘inplace=True’ argument modifies the DataFrame in place.

3. Displaying the Head of the DataFrame:

- ‘data.head()': Displays the first few rows of the modified DataFrame 'data' using the 'head() function.



```
[21]
    data.set_index('BD_DATE',inplace=True)
    final_df=data.resample('D').count()
    final_df.columns=['BD_COUNT']
    final_df
```

```
...
                BD_COUNT
    BD_DATE
    2009-01-01        23
    2009-01-02         9
    2009-01-03        11
    2009-01-04        15
    2009-01-05         4
           ...       ...
    2018-12-09         9
    2018-12-10         0
    2018-12-11         0
    2018-12-12         0
    2018-12-13         8

   3634 rows × 1 columns
```

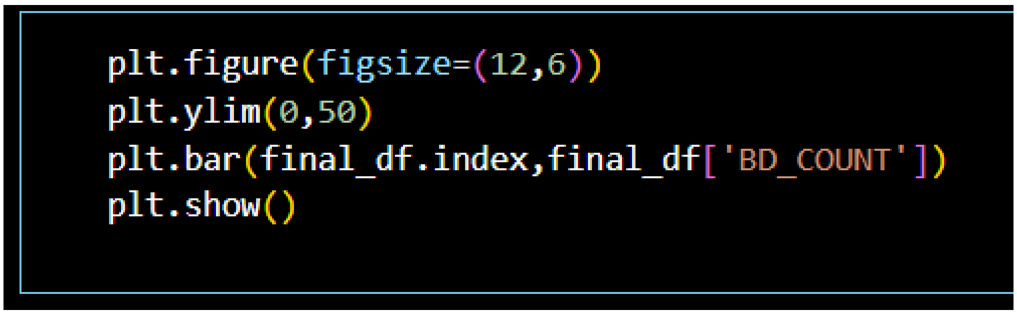
4.Resampling Data:

- ‘data.set_index("BD_DATE, inplace=True)’: Sets the 'BD_DATE' column as the index of the DataFrame data using the set index() function.

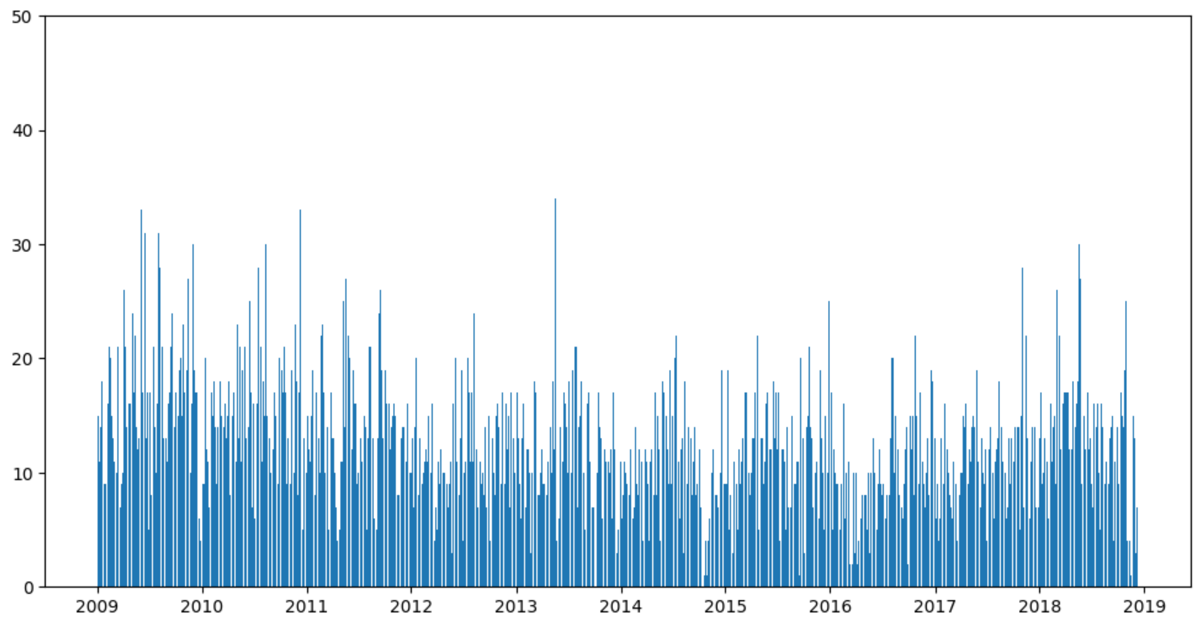
- ‘final_data = data.resample(D).count()’: Resamples the data to find the count of breakdowns per day ('D' represents daily frequency) using the resample() function. The count is calculated by applying the ‘count()’ function to each resampled period.

- 'final_data.columns = [BD_COUNT]': Renames the column of the resampled DataFrame as 'BD_COUNT' to indicate the number of breakdowns.

By executing this code, the user discards data before the year 2008, resets the index of the DataFrame, and resamples the data to obtain the count of breakdowns per day. The resulting DataFrame 'final data' contains the resampled data, and the final data info() statement provides an overview of its structure and column data types.



```
plt.figure(figsize=(12,6))
plt.ylim(0,50)
plt.bar(final_df.index,final_df['BD_COUNT'])
plt.show()
```



In the above cell, the code plots a bar plot to visually inspect any potential outliers in the data. Here's an explanation of the code:

1. Setting Figure Size:

- ‘plt.figure(figsize=(12,6))’: Sets the size of the plot figure to (12,6) using the figure() function from matplotlib.pyplot.

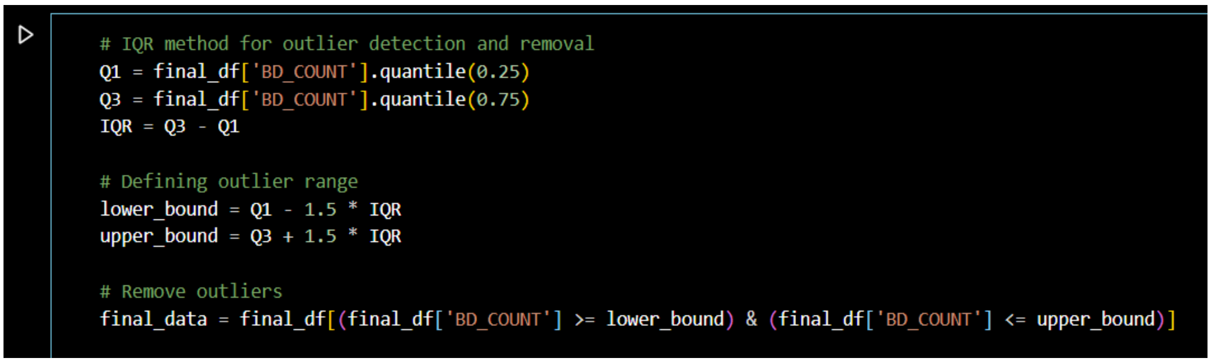
2. Setting y-Axis Limits:

- ‘plt.ylim(0, 50)’: Sets the limits of the y-axis from 0 to 50 using the ylim() function. This step ensures that the plot focuses on the range of values of interest.

3. Scatter Plot:

- ‘plt.scatter(final_data.index, final_data[‘BD_COUNT’)’: Plots a scatter plot using the ‘BD_COUNT’ column of the DataFrame final_data. The x- axis represents the index (date) of the DataFrame, while the y-axis represents the count of breakdowns.

By executing this code, the user generates a scatter plot to visually examine the distribution of breakdown counts over time. The figure size is set, and the y-axis limits are defined to ensure clear visualization. The scatter plot shows the relationship between the index (date) and the number of breakdowns (BD_COUNT") in the data.



```
# IQR method for outlier detection and removal
Q1 = final_df['BD_COUNT'].quantile(0.25)
Q3 = final_df['BD_COUNT'].quantile(0.75)
IQR = Q3 - Q1

# Defining outlier range
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Remove outliers
final_data = final_df[(final_df['BD_COUNT'] >= lower_bound) & (final_df['BD_COUNT'] <= upper_bound)]
```

In the above cell, the code detects and removesoutliers in the data by using interquartile range. Here's an explanation of the code:

1.Calculating Quartiles and IQR:

- Q1 is the first quartile (25th percentile) of the 'BD_COUNT' column.

- Q3 is the third quartile (75th percentile) of the 'BD_COUNT' column.

- IQR is the Interquartile Range, calculated as Q3 - Q1.

2. Defining Outlier Range:

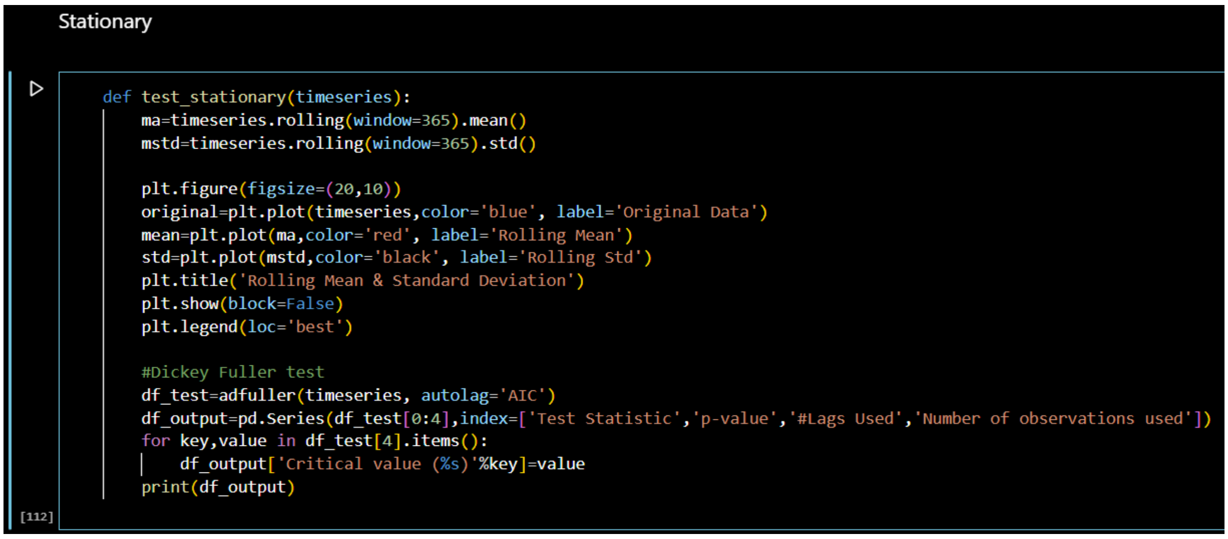
- Lower Bound: The lower bound for identifying outliers is calculated as Q1 - 1.5 * IQR.

- Upper Bound: The upper bound for identifying outliers is calculated as Q3 + 1.5 * IQR.

4. Removing Outliers:

- Filtering Data: This line filters final_df to include only the rows where 'BD_COUNT' is within the defined outlier range (i.e., between lower_bound and upper_bound).

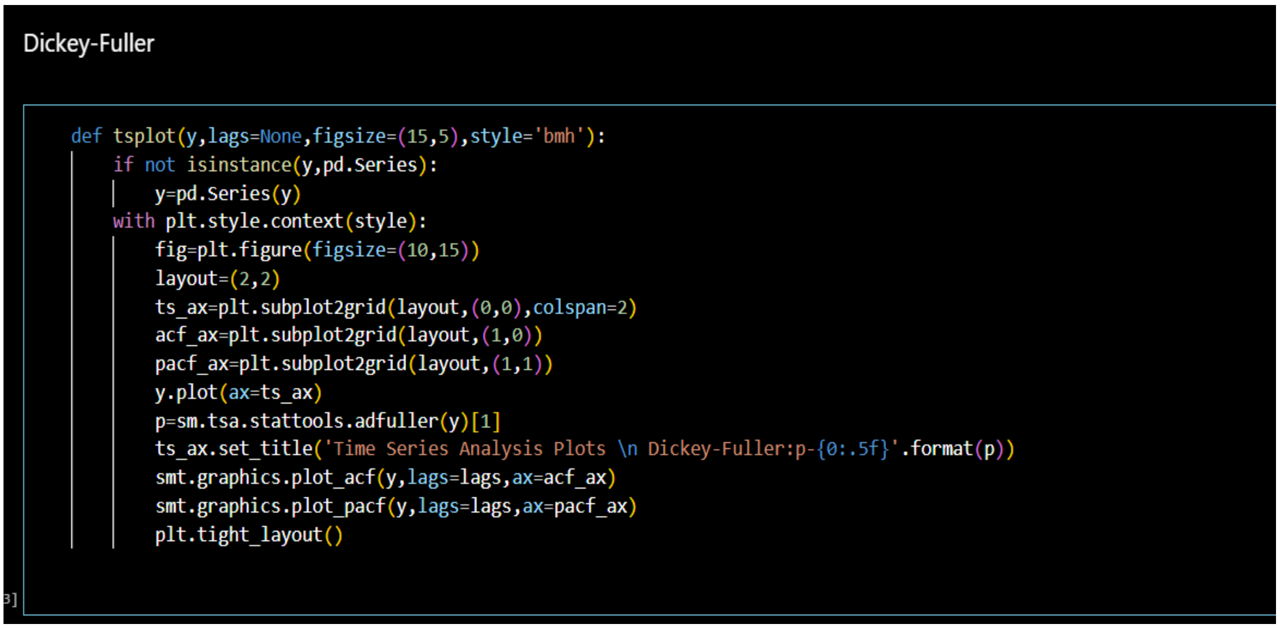
- The resulting DataFrame final_data contains no outliers in the 'BD_COUNT' column.



Stationary

```
def test_stationary(timeseries):
    ma=timeseries.rolling(window=365).mean()
    mstd=timeseries.rolling(window=365).std()

    plt.figure(figsize=(20,10))
    original=plt.plot(timeseries,color='blue', label='Original Data')
    mean=plt.plot(ma,color='red', label='Rolling Mean')
    std=plt.plot(mstd,color='black', label='Rolling Std')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False)
    plt.legend(loc='best')

    #Dickey Fuller test
    df_test=adfuller(timeseries, autolag='AIC')
    df_output=pd.Series(df_test[0:4],index=['Test Statistic','p-value','#Lags Used','Number of observations used'])
    for key,value in df_test[4].items():
        df_output['Critical value (%s)'%key]=value
    print(df_output)
```

[112]


Dickey-Fuller

```
def tsplot(y,lags=None,figsize=(15,5),style='bmh'):
    if not isinstance(y,pd.Series):
        y=pd.Series(y)
    with plt.style.context(style):
        fig=plt.figure(figsize=(10,15))
        layout=(2,2)
        ts_ax=plt.subplot2grid(layout,(0,0),colspan=2)
        acf_ax=plt.subplot2grid(layout,(1,0))
        pacf_ax=plt.subplot2grid(layout,(1,1))
        y.plot(ax=ts_ax)
        p=sm.tsa.stattools.adfuller(y)[1]
        ts_ax.set_title('Time Series Analysis Plots \n Dickey-Fuller:p-{0:.5f}'.format(p))
        smt.graphics.plot_acf(y,lags=lags,ax=acf_ax)
        smt.graphics.plot_pacf(y,lags=lags,ax=pacf_ax)
        plt.tight_layout()
```

In the above cells, two functions are defined: test stationarity() and tsplot().

1. test stationarity (timeseries):

This function is used to perform rolling statistics and the Dickey-Fuller test on a given time series.

Parameters:

timeseries: The input time series data.

Steps:

- Calculates the rolling mean and rolling standard deviation of the input time series using a window of 365 days.
- Plots the original time series, rolling mean, and rolling Standard deviation on a single graph.
- Displays the results of the Dickey-Fuller test, including the test statistic, p-value, number of lags used, and critical values.
- Example usage: test_stationarity(timeseries).

2. tsplot(y, lags None, figsize (15, 5), style='bmh'):

This function is used to plot the time series analysis plots, including the time series itself, the autocorrelation function (ACF), and the partial autocorrelation function (PACF).

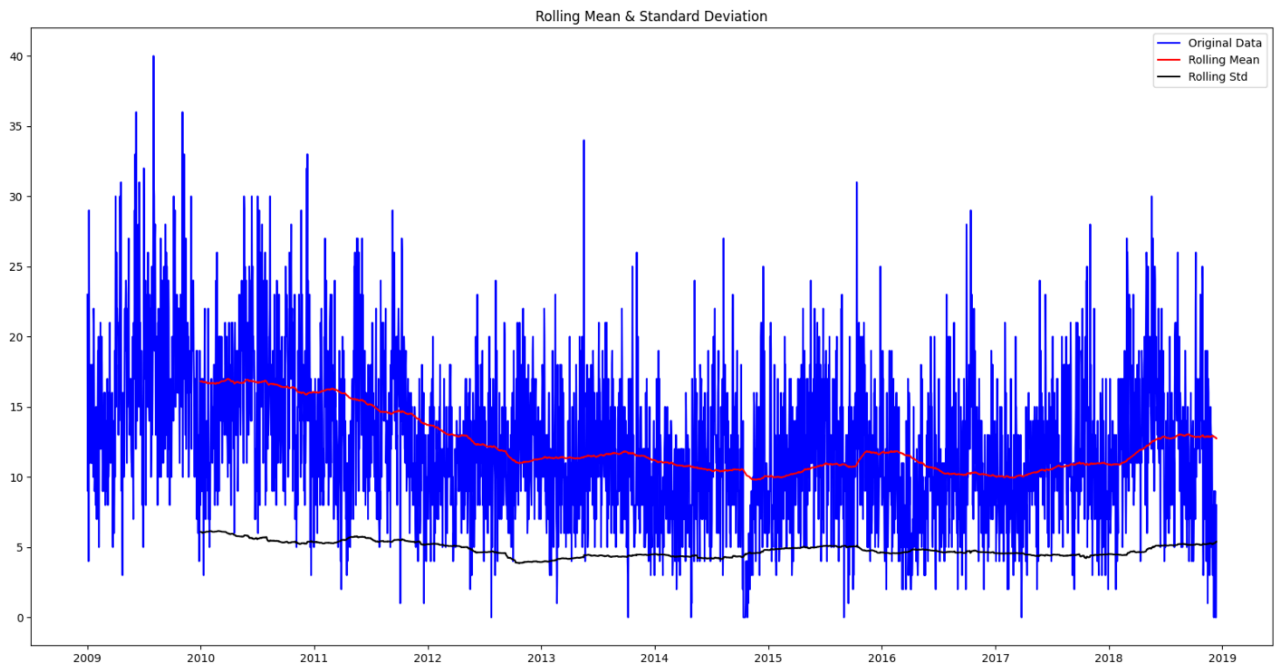
Parameters:

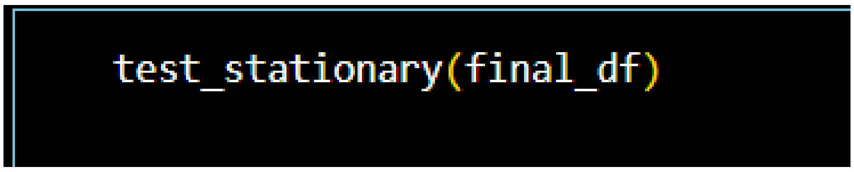
- **y:** The input time series data.
- **Lags:** The number of lags to include in the ACF and PACF plots.
- **Figsize:** The size of the plot figure.
- **style:** The style of the plot

Steps:

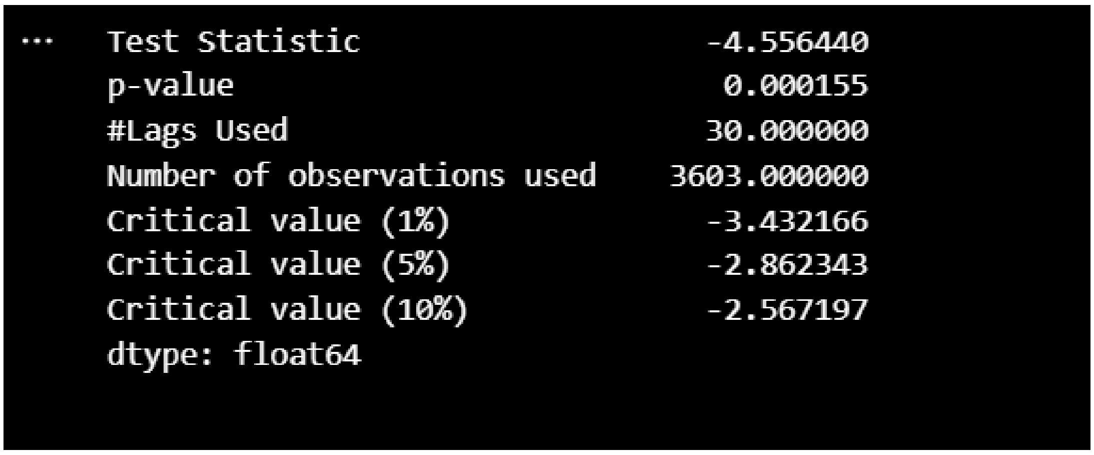
- Converts the input time series data to a pandas Series if it is not already.
- Creates a plot figure with subplots for the time series, ACF, and PACF.
- Plots the time series data and overlays the Dickey-Fuller test p-value the plot title.
- Plots the ACF on one subplot and the PACF on another subplot.
- Example usage: tsplot(lags=None, figsize (15,5), style='bmh ')

By executing this code, the user defines two functions: test_stationarity() for performing rolling statistics and the Dickey-Fuller test, and tsplot() for plotting time series analysis plots. These functions can be utilized for analyzing and visualizing time series data.





```
test_stationary(final_df)
```



```
...    Test Statistic                      -4.556440
       p-value                              0.000155
       #Lags Used                          30.000000
       Number of observations used       3603.000000
       Critical value (1%)                 -3.432166
       Critical value (5%)                 -2.862343
       Critical value (10%)                -2.567197
       dtype: float64
```

In the above cell, the test_stationary() function checks the stationarity of the time series by plotting the rolling mean and standard deviation on the data. Here's an explanation of the code and the output:

1.Calling the test stationarity() function:

- test_stationary(final_df): Calls the test_stationary() function and passes the 'final_df' as the input.

2. Calculate Rolling Mean and Standard Deviation:

- Rolling Mean (ma): The mean of the time series data calculated over a rolling window of 365 days. It computes the mean for each window, which helps in identifying trends in the time series data.

- Rolling Standard Deviation (mstd): The standard deviation of the time series data calculated over a rolling window of 365 days, which helps in identifying the variability in the time series data.

3. Performing Augumented Dickey-Fuller (ADF) Test:

- ADF Test: The adfuller function from the statsmodels.tsa.stattools module is performed to statistically check the stationarity of the time series.

4. Output:

- The function generates a graph that includes the original time series, rolling mean, and rolling standard deviation.

- This helps to visually inspect the stationarity of the time series by checking if the mean and standard deviation remain constant over time.

- Test Statistic: -4.087795 is used to compare against critical values to determine stationarity.

- p-value: 0.001015 indicates the probability that the time series is non-stationary. Since it is less than 0.05, we reject the null hypothesis and conclude the series is stationary.

- #Lags Used: 27 is the number of lagged terms used in the test regression.
- Number of observations used: 2477 is the number of observations included in the ADF regression.

- Critical Values: 1%: -3.432993, 5%: -2.862708, 10%: -2.567392. If the test statistic is less than these critical values, the null hypothesis of non-stationarity is rejected at the corresponding significance level.

Time Series Splitting:

Traditional cross-validation methods, such as k-fold cross-validation, are not suitable for time series data due to the inherent temporal order. In time series, the data points are time-dependent, and using future data to predict past data violates the chronological sequence, leading to information leakage and biased results.

Time Series Cross-Validation (TSCV) is specifically designed to address these challenges. It ensures that the training set always precedes the test set, preserving the temporal order and providing a more accurate evaluation of model performance on unseen data.

By preserving the order of data, TSCV provides a more realistic evaluation of how the model will perform on future data. This helps in selecting a model that generalizes well to unseen data.

It allows for robust hyperparameter tuning by evaluating the model across multiple train-test splits, ensuring that the chosen parameters perform well over different periods within the data.

Ensures that the model does not gain unfair advantage by being trained on future information, which would not be available in a real-world forecasting scenario.

ARIMA Model:

The ARIMA (AutoRegressive Integrated Moving Average) model is a powerful and flexible class of forecasting models that is used extensively in time series analysis. It combines three components - autoregression (AR), differencing (I for "Integrated"), and moving average (MA) - to model and predict future points in a time series.

Components in ARIMA:

1. Autoregressive (AR) Component:

- The AR component of the model specifies that the output variable depends linearly on its own previous values.

- Order (p): Represents the number of lagged observations included in the model.

2. Integrated (I) Component:

- The I component deals with making the time series stationary by differencing the data points.

- Order (d): Represents the number of times the data values have had past values subtracted.

3. Moving Average (MA) Component:

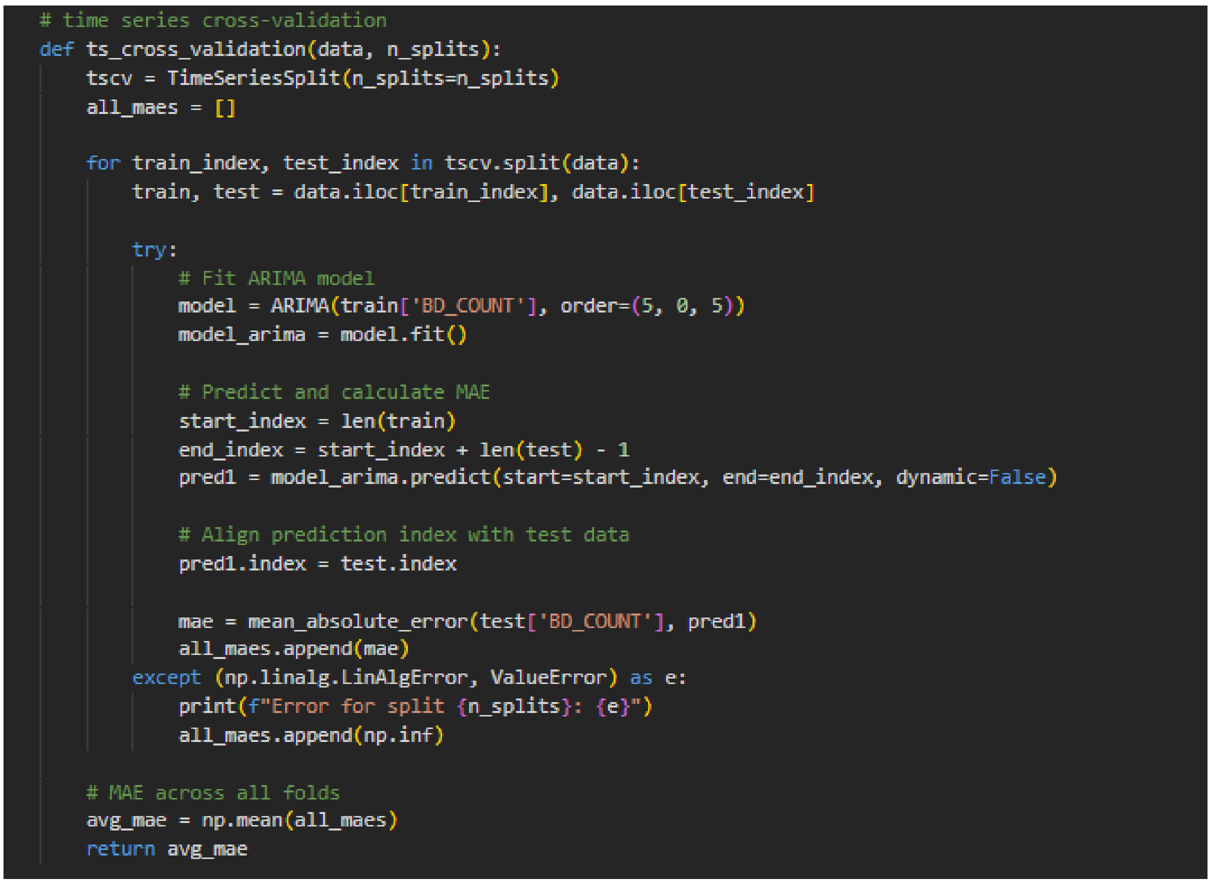
- The MA component specifies that the output variable depends linearly on the past forecast errors.

- Order (q): Represents the number of lagged forecast errors in the prediction equation.

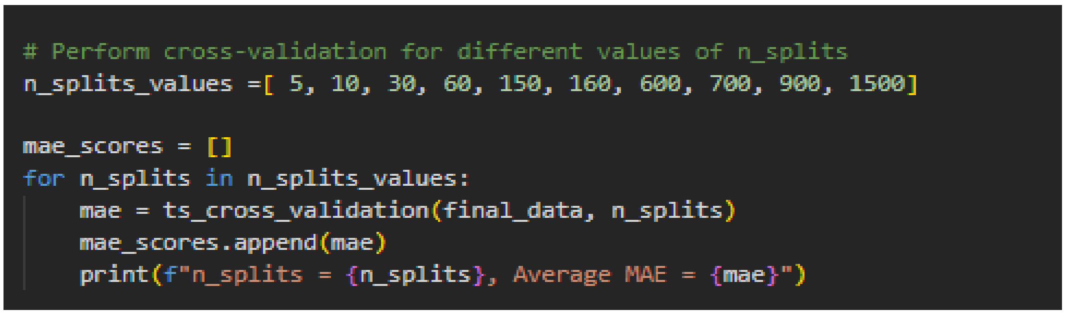
Finding Optimal Parameters for ARIMA:

- Hyperparameter search iterates over a range of values for p, d, and q to find the combination that minimizes the Mean Absolute Error (MAE) on the validation set.

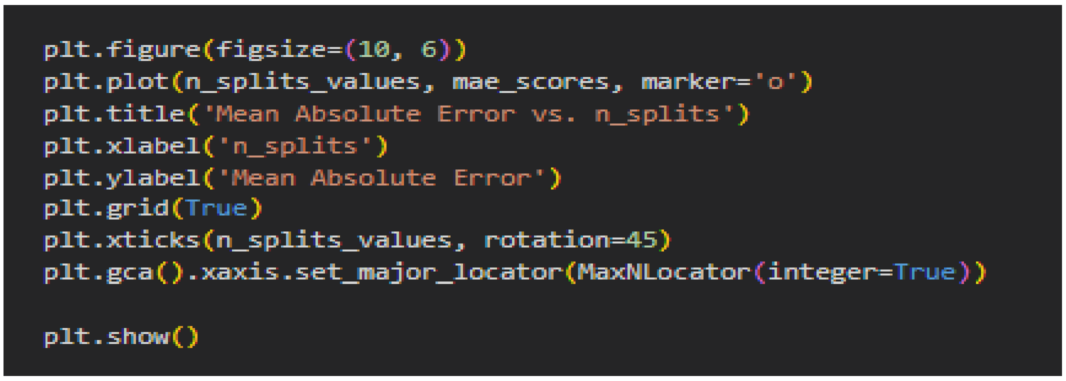
- The ARIMA model is fit on the training data for each combination of parameters.
- Predictions are made for the validation set, and the MAE is calculated. The combination of p, d, and q that results in the lowest MAE is selected as the best model.



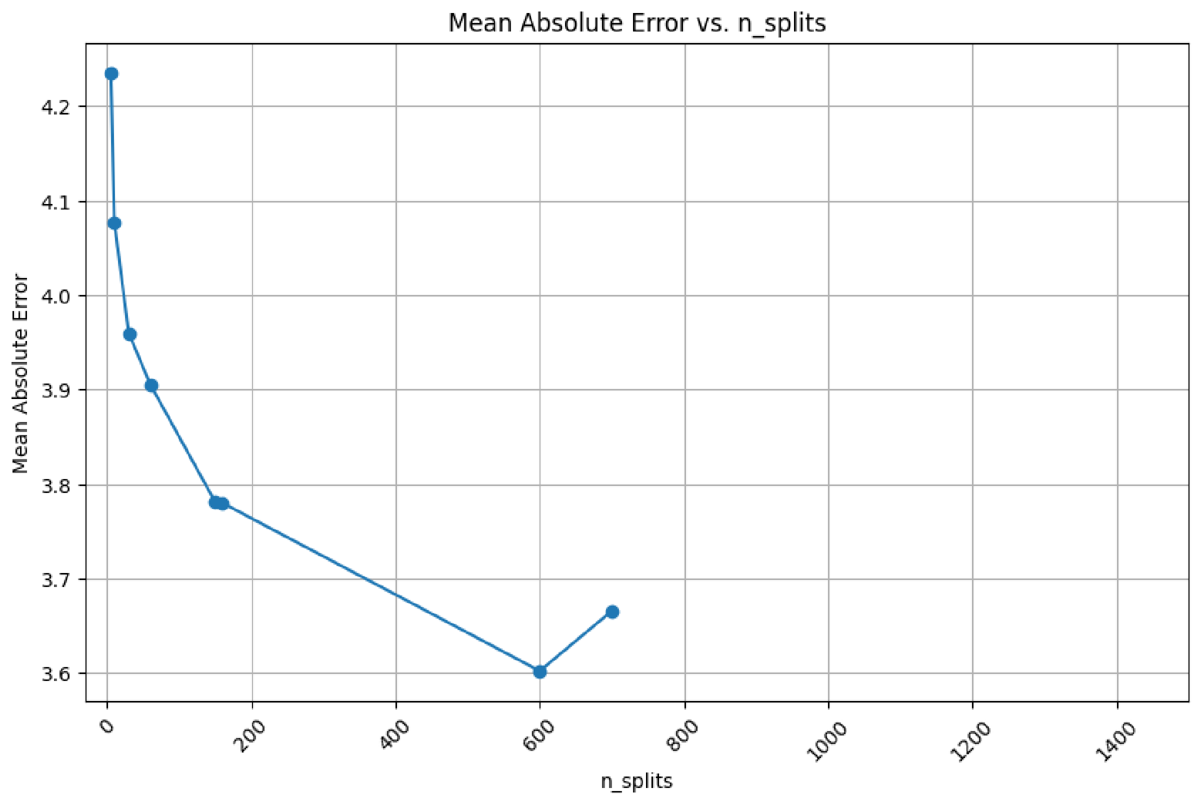
```
# time series cross-validation
def ts_cross_validation(data, n_splits):
    tscv = TimeSeriesSplit(n_splits=n_splits)
    all_maes = []

    for train_index, test_index in tscv.split(data):
        train, test = data.iloc[train_index], data.iloc[test_index]

        try:
            # Fit ARIMA model
            model = ARIMA(train['BD_COUNT'], order=(5, 0, 5))
            model_arima = model.fit()

            # Predict and calculate MAE
            start_index = len(train)
            end_index = start_index + len(test) - 1
            pred1 = model_arima.predict(start=start_index, end=end_index, dynamic=False)

            # Align prediction index with test data
            pred1.index = test.index

            mae = mean_absolute_error(test['BD_COUNT'], pred1)
            all_maes.append(mae)
        except (np.linalg.LinAlgError, ValueError) as e:
            print(f"Error for split {n_splits}: {e}")
            all_maes.append(np.inf)

    # MAE across all folds
    avg_mae = np.mean(all_maes)
    return avg_mae
```



```python
# Perform cross-validation for different values of n_splits
n_splits_values =[ 5, 10, 30, 60, 150, 160, 600, 700, 900, 1500]

mae_scores = []
for n_splits in n_splits_values:
    mae = ts_cross_validation(final_data, n_splits)
    mae_scores.append(mae)
    print(f"n_splits = {n_splits}, Average MAE = {mae}")
```



```python
plt.figure(figsize=(10, 6))
plt.plot(n_splits_values, mae_scores, marker='o')
plt.title('Mean Absolute Error vs. n_splits')
plt.xlabel('n_splits')
plt.ylabel('Mean Absolute Error')
plt.grid(True)
plt.xticks(n_splits_values, rotation=45)
plt.gca().xaxis.set_major_locator(MaxNLocator(integer=True))

plt.show()
```



1. Function Definition and Initialization: Defining the Function: def ts_cross_validation(data, n_splits): This function takes in a dataset and the number of splits (n_splits) for time series cross-validation.

- Initialize TimeSeriesSplit: tscv = TimeSeriesSplit(n_splits=n_splits): Initializes the TimeSeriesSplit object with the specified number of splits.

2. Splitting the Data:

- ‘for train_index, test_index in tscv.split(data)’: Splits the data into training and testing sets while maintaining the temporal order.

3. Fitting the ARIMA Model:

- ‘model = ARIMA(train['BD_COUNT'], order=(5, 0, 5))’: Creates an ARIMA model object with the specified order (5, 0, 5) based on the obtained values from the ACF and PACF graphs, p is set to 5. d is set to 0, and q is set to 5.
- ‘model_arima = model.fit()’: Fits the ARIMA model to the training data using the fit() method.

4. Making Predictions:

- ‘start_index = len(train)’: Defines the start index for predictions.
- ‘end_index = start_index + len(test) – 1’: Defines the end index for predictions.
- ‘pred1 = model_arima.predict(start = start_index, end = end_index, dynamic = False)’: Makes predictions on the training data using the fitted ARIMA model. The predict() method is called with the start and end indices of the test data without dynamic forecasting.
- ‘pred1.index = test.index’: Aligns the prediction index with the test data index.

5. Calculating Evaluation Metric Mean Absolute Error (MAE):

- ‘mae = mean_absolute_error(test['BD_COUNT'], pred1)’: Calculates the MAE between the actual and predicted values for the test set.

- ‘all_maes.append(mae)’: Appends the MAE to the list of all MAEs for each split.

6. Error Handling:

- ‘except (np.linalg.LinAlgError, ValueError) as e’: Catches any linear algebra errors or value errors during model fitting or prediction.

- ‘all_maes.append(np.inf)’: Appends infinity to the list of MAEs in case of an error.

7. Averaging MAE across All Folds:

- ‘avg_mae = np.mean(all_maes)’: Computes the average MAE across all folds. By taking the mean of all MAE scores (all_maes), avg_mae provides a single metric that represents the overall prediction accuracy of the model when evaluated using different training and validation sets.

- ‘return avg_mae’: Returns the average MAE.

8. Cross-Validation with different n_splits:

- ‘n_splits_values’ = [5, 10, 30, 60, 150, 160, 600, 700, 900, 1500]: Defines the values of n_splits to be tested.

- ‘mae_scores = []’: Initializes an empty list to store MAE scores.
- ‘for n_splits in n_splits_values:’: Iterates over each value of n_splits.
- ‘mae = ts_cross_validation(final_data, n_splits)’: Performs cross-validation and calculates MAE for each value of n_splits.
- ‘mae_scores.append(mae)’: Appends the MAE to the list of MAE scores.
- ‘print(f"n_splits = {n_splits}, Average MAE = {mae}")’: Prints the MAE for each value of n_splits.

9. Plotting the Test Data and Predictions:

- ‘plt.figure(figsize=(10, 6))’: Sets the figure size for the plot.

- ‘plt.plot(n_splits_values, mae_scores, marker='o')’: This line of code is to visually represent how the Mean Absolute Error (MAE) changes with different values of n_splits. Each point on the plot corresponds to a specific value of n_splits and its corresponding MAE score. marker='o' specifies that markers should be used at each data point and 'o' indicates circular markers.

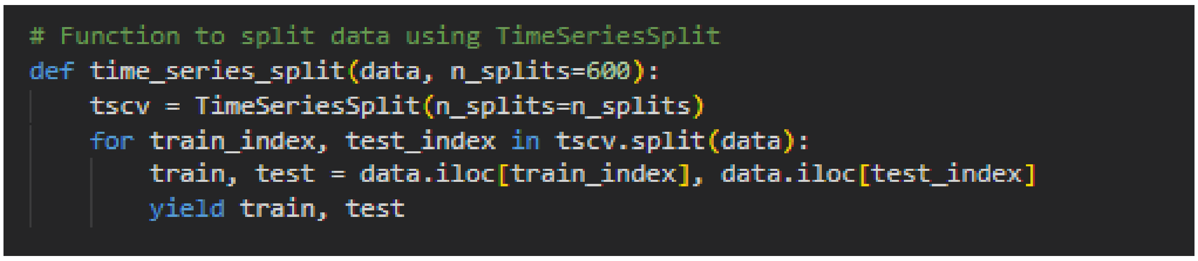
- ‘ plt.title('Mean Absolute Error vs. n_splits')’: Sets the plot title.
- ‘ plt.xlabel('n_splits')’: Labels the x-axis.
- ‘plt.ylabel('Mean Absolute Error')’: Labels the y-axis.
- ‘plt.grid(True)’: Enables the grid.
- ‘plt.xticks(n_splits_values)’: Sets the x-axis ticks.
- ‘plt.show()’: Displays the plot.

10. Determining Optimal n_splits value:

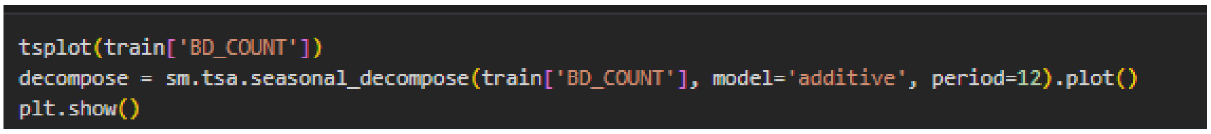
- ‘optimal_n_splits = n_splits_values [np.argmin(mae_scores)]’: Finds the value of n_splits that results in the lowest MAE by using the argmin() function of the numpy which returns the index of the minimum value in an array.

- ‘print(f"Optimal n_splits value: {optimal_n_splits}")’: Prints the optimal value of n_splits.

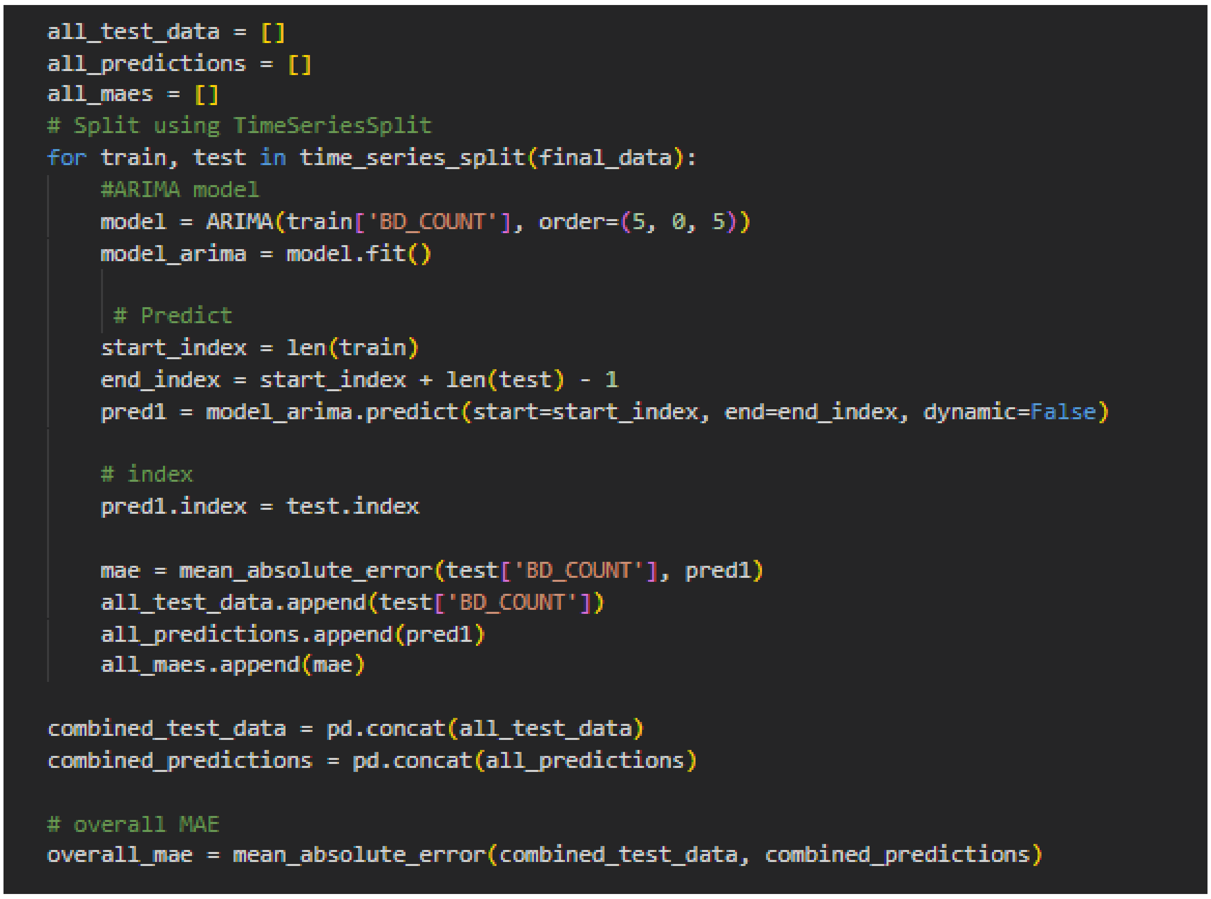
This section provides a comprehensive explanation of the time series cross-validation process for evaluating an ARIMA model. The function ts_cross_validation splits the data into training and testing sets, fits an ARIMA model, makes predictions, calculates the MAE, and averages the MAE across all splits. Different values of n_splits are tested to find the optimal split, ensuring robust model evaluation. This detailed approach ensures that the chosen ARIMA model is well-tuned and capable of making accurate forecasts based on historical data.



```
# Function to split data using TimeSeriesSplit
def time_series_split(data, n_splits=600):
    tscv = TimeSeriesSplit(n_splits=n_splits)
    for train_index, test_index in tscv.split(data):
        train, test = data.iloc[train_index], data.iloc[test_index]
        yield train, test
```



```
tsplot(train['BD_COUNT'])
decompose = sm.tsa.seasonal_decompose(train['BD_COUNT'], model='additive', period=12).plot()
plt.show()
```



```
all_test_data = []
all_predictions = []
all_maes = []
# Split using TimeSeriesSplit
for train, test in time_series_split(final_data):
    #ARIMA model
    model = ARIMA(train['BD_COUNT'], order=(5, 0, 5))
    model_arima = model.fit()

     # Predict
    start_index = len(train)
    end_index = start_index + len(test) - 1
    pred1 = model_arima.predict(start=start_index, end=end_index, dynamic=False)

    # index
    pred1.index = test.index

    mae = mean_absolute_error(test['BD_COUNT'], pred1)
    all_test_data.append(test['BD_COUNT'])
    all_predictions.append(pred1)
    all_maes.append(mae)

combined_test_data = pd.concat(all_test_data)
combined_predictions = pd.concat(all_predictions)

# overall MAE
overall_mae = mean_absolute_error(combined_test_data, combined_predictions)
```

1. Calling tsplot():

- The function ‘tsplot()’ provides a comprehensive set of plots, including the time series plot, ACF, and PACF plots, which are essential for diagnosing the characteristics of a time series and identifying appropriate models for forecasting.

2. Setting Plotting Style and Figure Layout:

- Applying a specified style to the plots for better visualization.

- plt.subplot2grid is used to create a grid layout for the figure.

- ts_ax is the axis for the time series plot, spanning two columns.

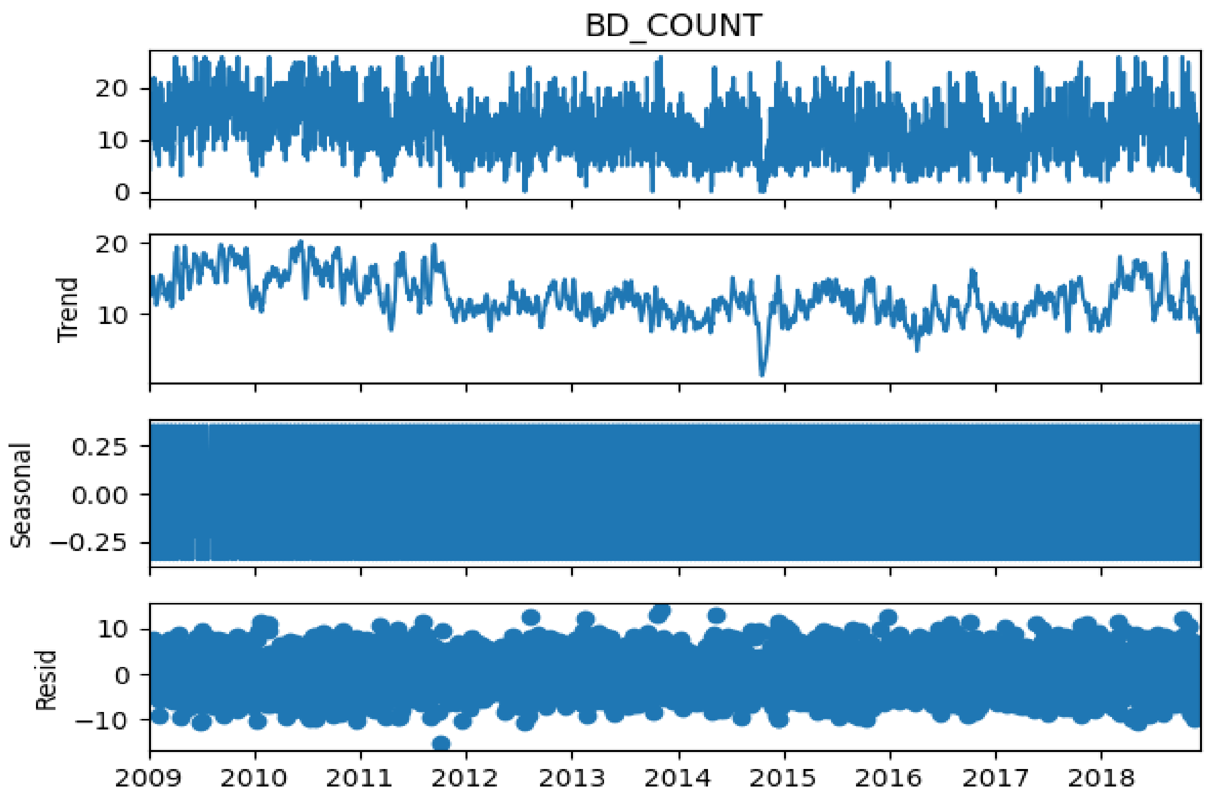
- acf_ax and pacf_ax are the axes for the ACF and PACF plots, respectively.

3. Plot Time Series, ACF and PACF:

- The plot method is used to visualize the time series data on the ts_ax axis.

- The plot_acf function from statsmodels.graphics.tsaplots plots the autocorrelation function, which shows the correlation of the time series with its own lagged values.

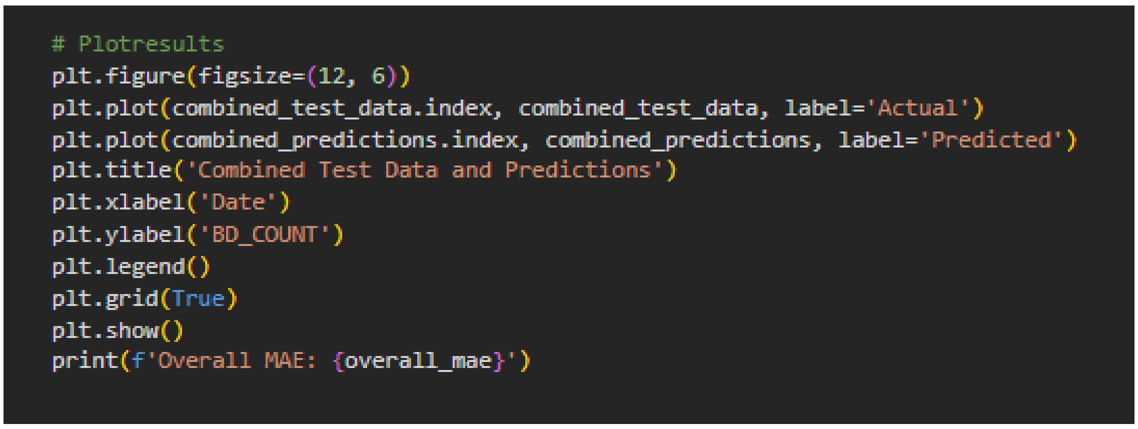
- The plot_pacf function from the same module plots the partial autocorrelation function, which shows the partial correlation of the time series with its lagged values, accounting for the correlations of the intermediate lags.



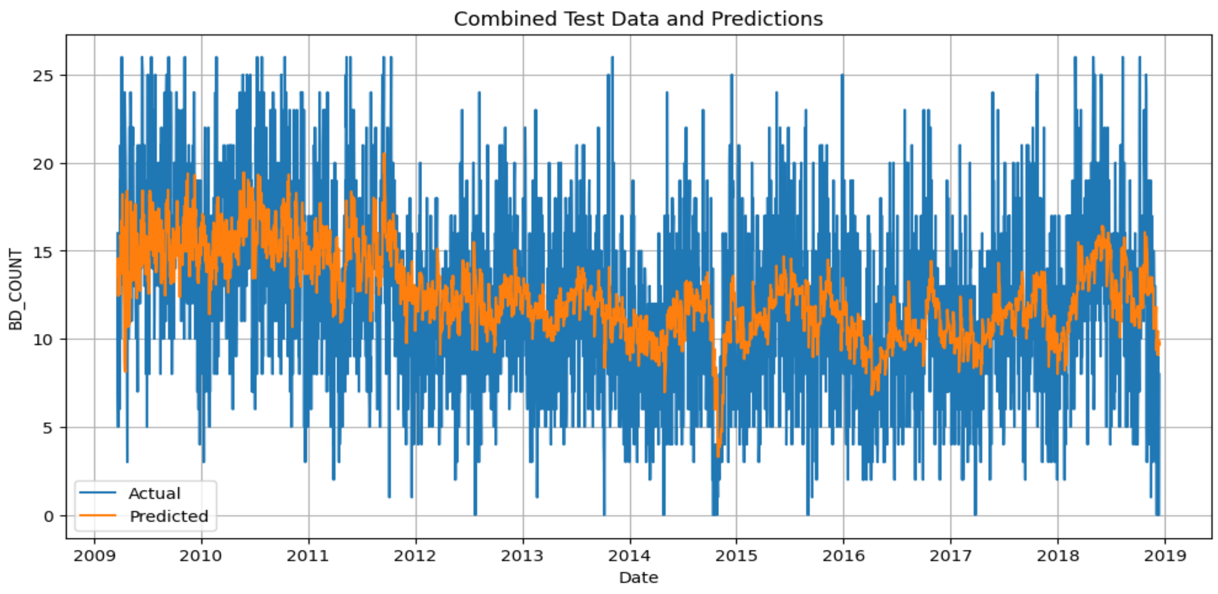
The above code performs seasonal decomposition on a time series to separate it into its trend, seasonal, and residual components. This helps in understanding the underlying patterns of the data.

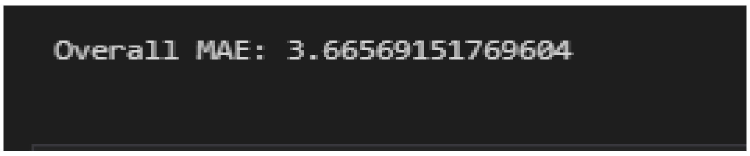
- ‘**model='additive'**’: Specifies the type of seasonal decomposition model to use. An additive model assumes that the components (trend, seasonal, residual) of the time series are added together. This is suitable when the seasonal fluctuations are roughly constant over time. Alternatively, a multiplicative model can be used if the seasonal variations change proportionally with the level of the time series.

- ‘**period=12**’: Indicates the length of the seasonal cycle. In this case, it is set to 12, implying a yearly cycle for monthly data (i.e., 12 months per year). Adjust this parameter according to the periodicity of your data.



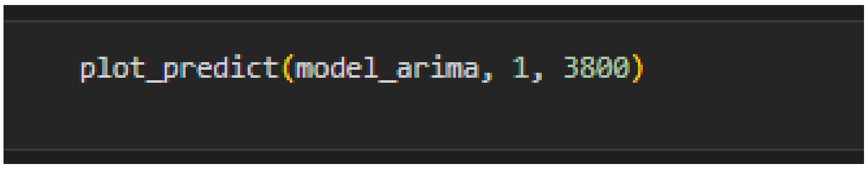
```
# Plotresults
plt.figure(figsize=(12, 6))
plt.plot(combined_test_data.index, combined_test_data, label='Actual')
plt.plot(combined_predictions.index, combined_predictions, label='Predicted')
plt.title('Combined Test Data and Predictions')
plt.xlabel('Date')
plt.ylabel('BD_COUNT')
plt.legend()
plt.grid(True)
plt.show()
print(f'Overall MAE: {overall_mae}')
```



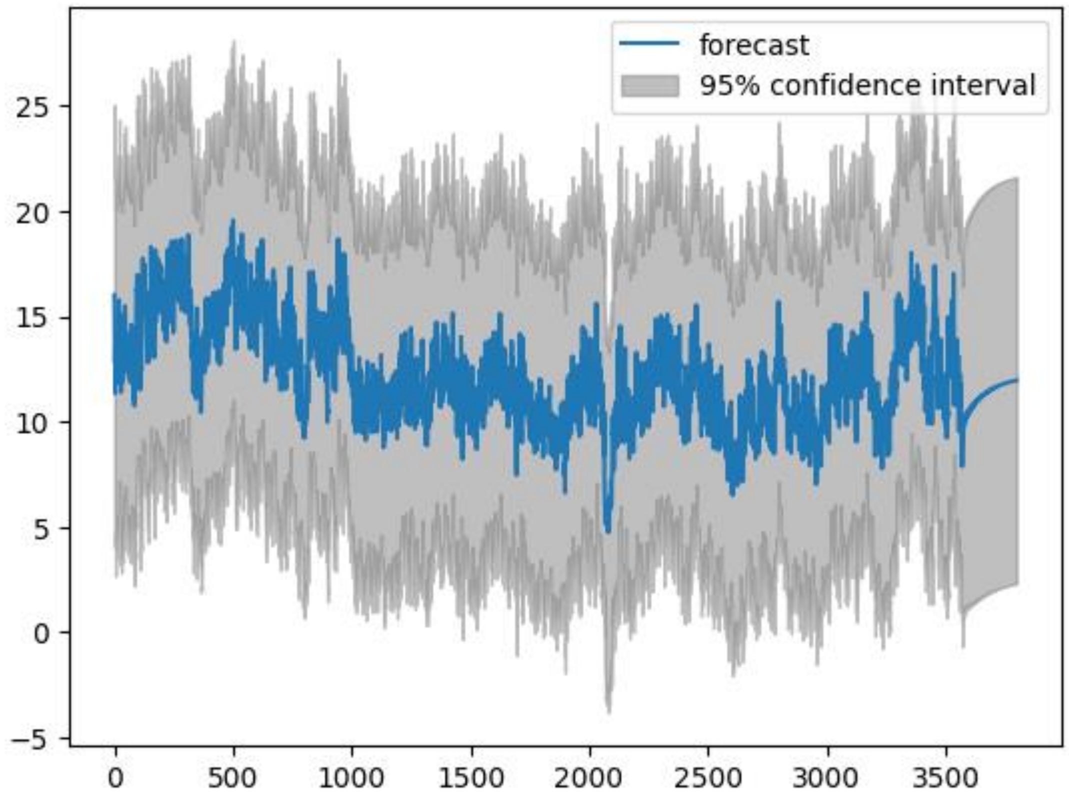


```
Overall MAE: 3.66569151769604
```

The above code demonstrates how to perform time series data splitting using TimeSeriesSplit and make predictions using the ARIMA model. The process includes training the model, making predictions, and evaluating the performance using Mean Absolute Error (MAE).



```
plot_predict(model_arima, 1, 3800)
```



In this cell, the ARIMA model is used to forecast future values. Here's an explanation of the code:

1.Forecasting Future Values:

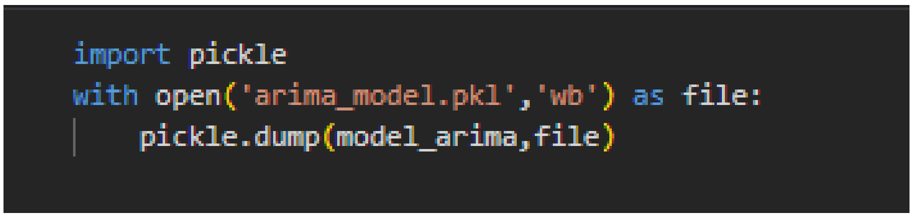
- ‘plot_predict(model_arima, 1, 3800)’: Generates a forecast of future values

using the fitted ARIMA model. The plot_predict() function is called with the model fit object and the start and end indices of the forecasted period.

2. Displaying the Forecast Plot:

- ‘plt.show()’: Displays the forecast plot.

By executing this code, the user generates a forecast of future values using the fitted ARIMA model. The forecasted values are plotted, showing the trend for the specified forecast period.



```
import pickle
with open('arima_model.pkl','wb') as file:
    pickle.dump(model_arima,file)
```

In the above cell, the fitted ARIMA model is saved using the pickle module. Here's an explanation of the code:

1. Importing the ‘pickle’ Module:

- ‘import pickle’: This statement imports the pickle module, which is used for serializing and deserializing Python objects.

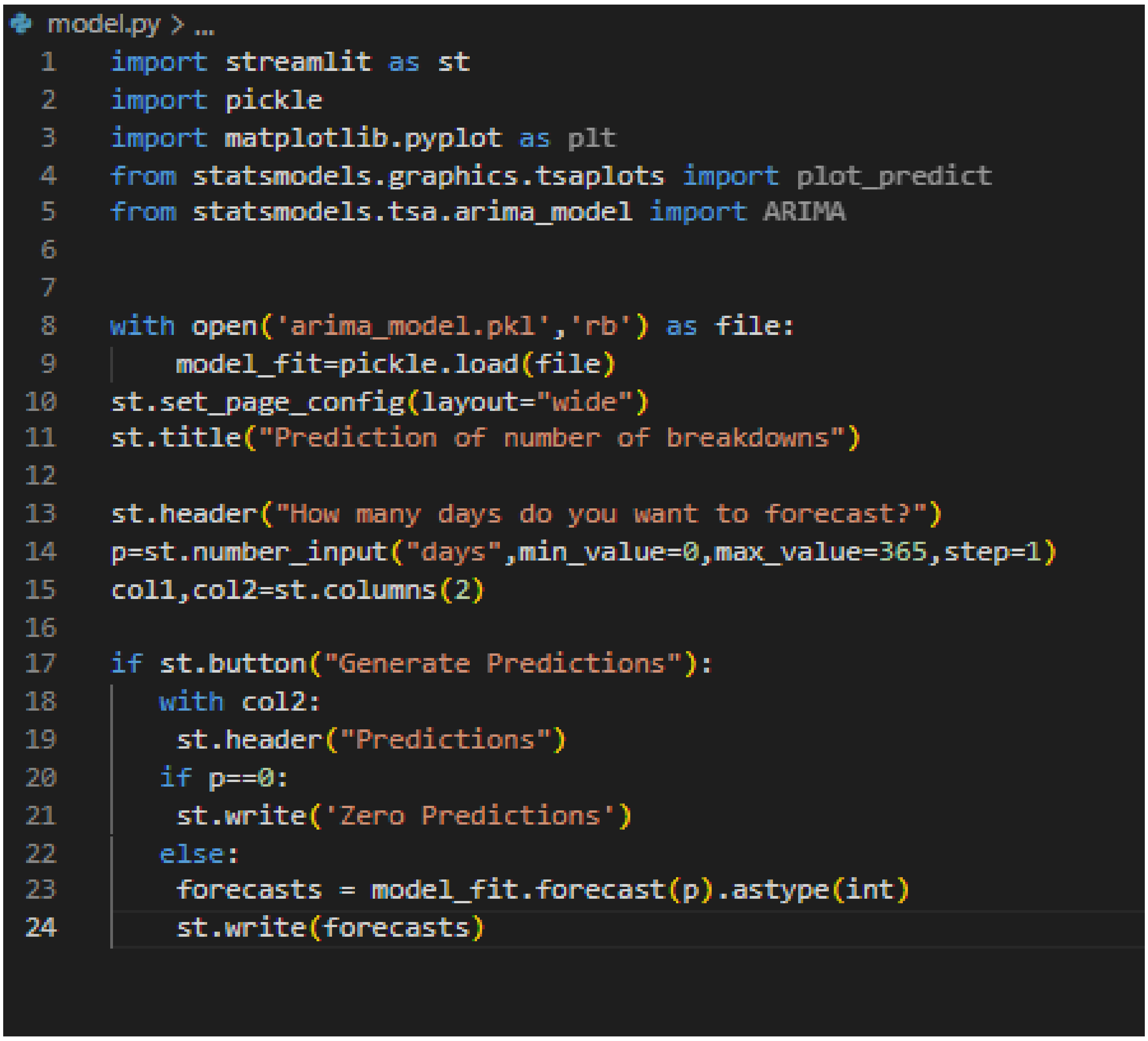
2. Saving the ARIMA Model:

- ‘with open('arima_model.pkl', 'wb') as file’: Opens a file named 'arima_model.pkl' in binary write mode. It assigns the opened file object to the variable file. The file will be automatically closed at the end of the with block.

- ‘pickle.dump(model_arima, file)’: Uses the pickle.dump() function to serialize and save the model fit object to the file.

By executing this code, the user saves the fitted ARIMA model as a serialized object in a file named 'arima_model.pkl'. This allows the model to be loaded and reused later without having to retrain it.

6. PROJECT DEPLOYMENT


model.py > ...

```python
import streamlit as st
import pickle
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_predict
from statsmodels.tsa.arima_model import ARIMA


with open('arima_model.pkl','rb') as file:
    model_fit=pickle.load(file)
st.set_page_config(layout="wide")
st.title("Prediction of number of breakdowns")

st.header("How many days do you want to forecast?")
p=st.number_input("days",min_value=0,max_value=365,step=1)
col1,col2=st.columns(2)

if st.button("Generate Predictions"):
   with col2:
    st.header("Predictions")
   if p==0:
    st.write('Zero Predictions')
   else:
    forecasts = model_fit.forecast(p).astype(int)
    st.write(forecasts)
```

Streamlit

- Streamlit is an open-source Python library used for building interactive web appl ications for data science and machine learning. It simplifies the process of creating and deploying web interfaces for data analysis and model visualization. Here's an explanation of the code:

1. Importing Required Libraries:

- **import streamlit as st:** This line imports the Streamlit library and allows you to use its functions and features in your code to build web applications."st" is a commonly used alias for Streamlit.
- **import pickle:** The pickle module in Python is used for serializing and de serializing Python objects. In this context, it is used to load a pre- trained ARIMA model that was previously saved using pickle.
- **import matplotlib.pyplot as plt:** This line imports the matplotlib library's pyplot module, which provides a simple interface for creating various type s of plots and visualizations.
- **from statsmodels.graphics.tsaplots import plot_predict:** This line imports the plot predict function from the statsmodels.graphics.tsaplots module. This function is used to plot the predicted values from a time series model

2. Loading the ARIMA Model:

- **with open('arima_model.pkl', 'rb') as file:** This line opens a file named arima model.pkl' in binary read mode. It is assumed that this file contains a pre-trained ARIMA model that was saved using pickle.
- **model_fit pickle.load(file):** This line loads the pre-trained ARIMA model from the opened file using pickle and assigns it to the variable "model_fit" for further use.

3. Setting Up the Streamlit Application:

- **st.set_page_config(layout="wide"):** This line sets the layout configuration of the Streamlit page to "wide", indicating a wider layout with more horizontal space.
- **st.title("Prediction of number of Breakdowns with ARIMA Model"):** Displays the title of the application.

4. Input Form for User to Enter Values:

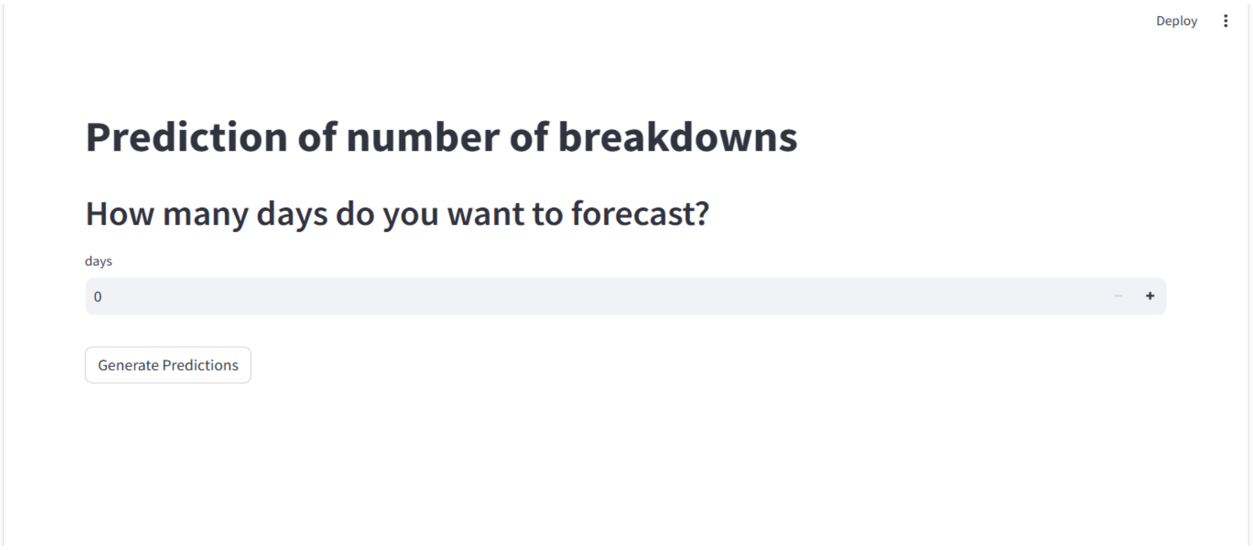
- **‘st.header("How many days you want to forecast?")’:** This line displays a header on the web page, indicating a question to the user about the number of days they want to forecast.
- **‘pst.number_input("days", min_value=0, max_value 365, step=1)’:** This line creates a number input field on the web page, allowing the user to enter the number of days they want to forecast. The input is assigned to the variable "p".

5. Generating Predictions:

- **‘if st.button("Generate Predictions")’:** This line creates a button on the web page with the label "Generate Predictions". It is used to trigger the generation of predictions when clicked.
- **‘forecasts = model_fit.forecast(p)’:** This line generates predictions for the next "p" days using the pre-trained ARIMA model stored in "model_fit". The predicted values are assigned to the variable "forecasts",
- **‘with col2’:** Sets up the second column for displaying the predictions.
- **‘st.header("Predictions")’:** This line displays a header on the web page, in dicating the section for displaying the predictions.
- **‘if p == 0’:** Ensures no predictions are attempted if p is 0.
- **‘else’:** Computes and displays predictions using model_fit.forecast(p) for the specified number of days (p), then outputs the forecasts using st.write().
- **‘st.write(forecasts)’:** This line displays the predicted values on the web pag e using the "st.write" function. The "forecasts" variable contains the predic ted values generated by the ARIMA model.

By executing this code, the user is presented with a Streamlit web application. The user can enter the number of days to forecast and click the "Generate

Predictions" button. The application then generates a forecast plot and displays the predictions.

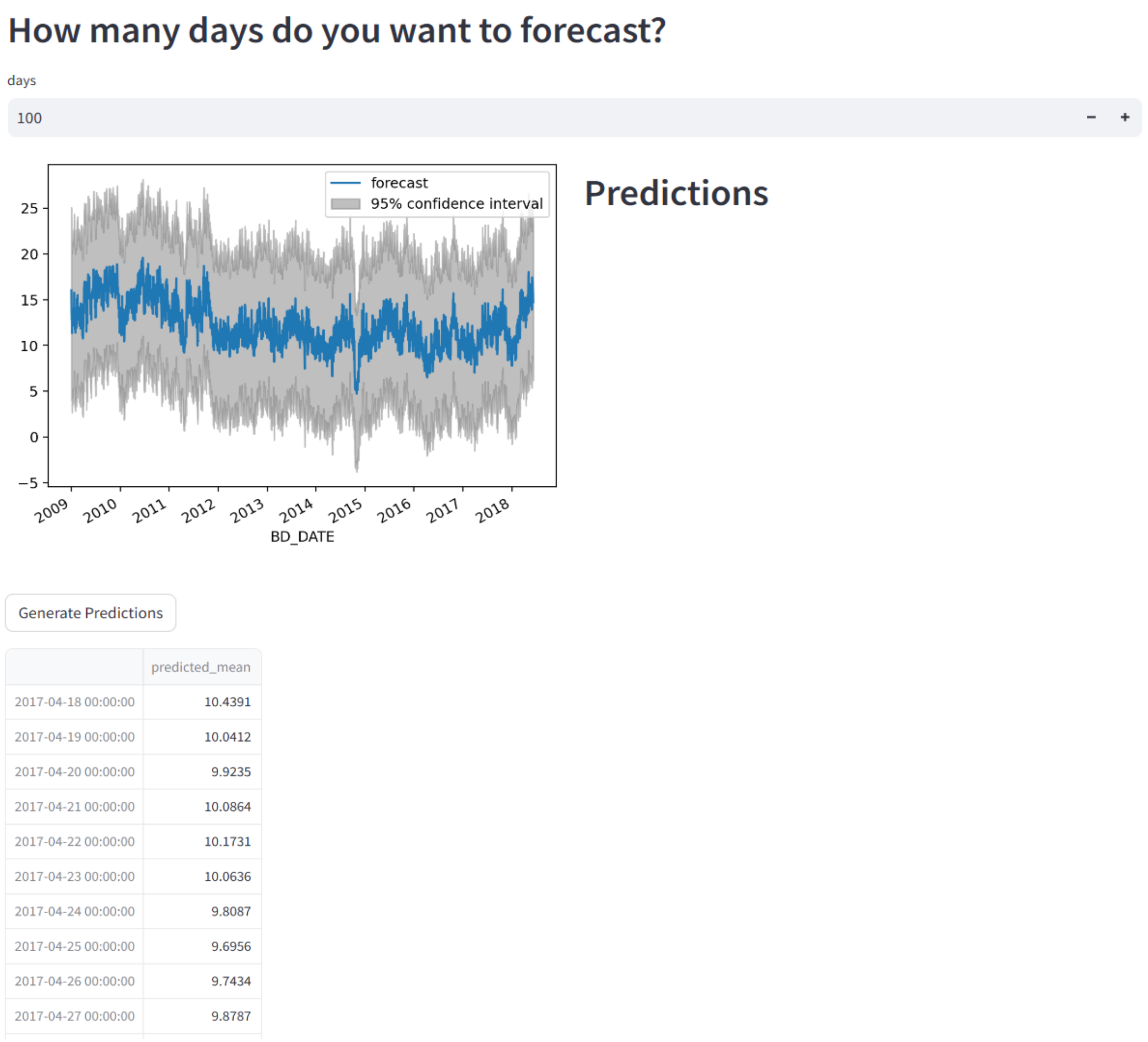


Prediction of number of breakdowns

How many days do you want to forecast?

days

0



How many days do you want to forecast?

days

100

	predicted_mean
2017-04-18 00:00:00	10.4391
2017-04-19 00:00:00	10.0412
2017-04-20 00:00:00	9.9235
2017-04-21 00:00:00	10.0864
2017-04-22 00:00:00	10.1731
2017-04-23 00:00:00	10.0636
2017-04-24 00:00:00	9.8087
2017-04-25 00:00:00	9.6956
2017-04-26 00:00:00	9.7434
2017-04-27 00:00:00	9.8787

7. CONCLUSION

In this project, we successfully carried out time series analysis and forecasting using the ARIMA model. We followed several steps to make sure our forecasts were accurate and reliable, giving us valuable insights into the data's behavior and paving the way for future improvements and real-time applications.

We started by preparing and organizing the data to ensure it was ready for time series analysis. This involved data cleaning, converting date columns to a Datetime Index and setting the correct frequencies, which are essential for accurate analysis and modelling.

To evaluate the model's performance, we used Time SeriesSplit to create multiple train-test splits. This approach allowed us to test the model's ability to predict across different time periods, ensuring it was both robust and reliable. By training the ARIMA model on each split and calculating the Mean Absolute Error (MAE) for each prediction, we were able to measure the model's accuracy effectively.

We then combined the predictions from all splits and calculated the overall MAE, providing a complete picture of the model's performance across the entire dataset. We also performed seasonal decomposition to break down the data into its trend, seasonal, and residual components. This analysis gave us deeper insights into the patterns within the data, helping us understand long-term movements, repeating seasonal cycles, and irregular fluctuations. We visualized this decomposition through plots, making it easier to interpret the results and identify key patterns.

Throughout the project, we created various visualizations, including plots of actual vs. predicted values and the decomposed components of the time series. These visualizations were essential for understanding the model's performance and the data's behaviour.

Our findings showed that the ARIMA model was effective in capturing both the seasonal patterns and long-term trends in the data. The model demonstrated predictive accuracy with an overall MAE of 3.66569151769.

In conclusion, this project successfully demonstrated how to use the ARIMA model for time series analysis and forecasting. The methods and results provide a strong foundation for accurate forecasting and valuable insights into the data's behavior. This project also sets the stage for future work, including optimizing the model, incorporating additional external data sources, and implementing real-time forecasting systems that update predictions as new data becomes available.