



AMAZON STOCK PRICE:

VISUALIZATION, FORECASTING, AND PREDICTION

USING

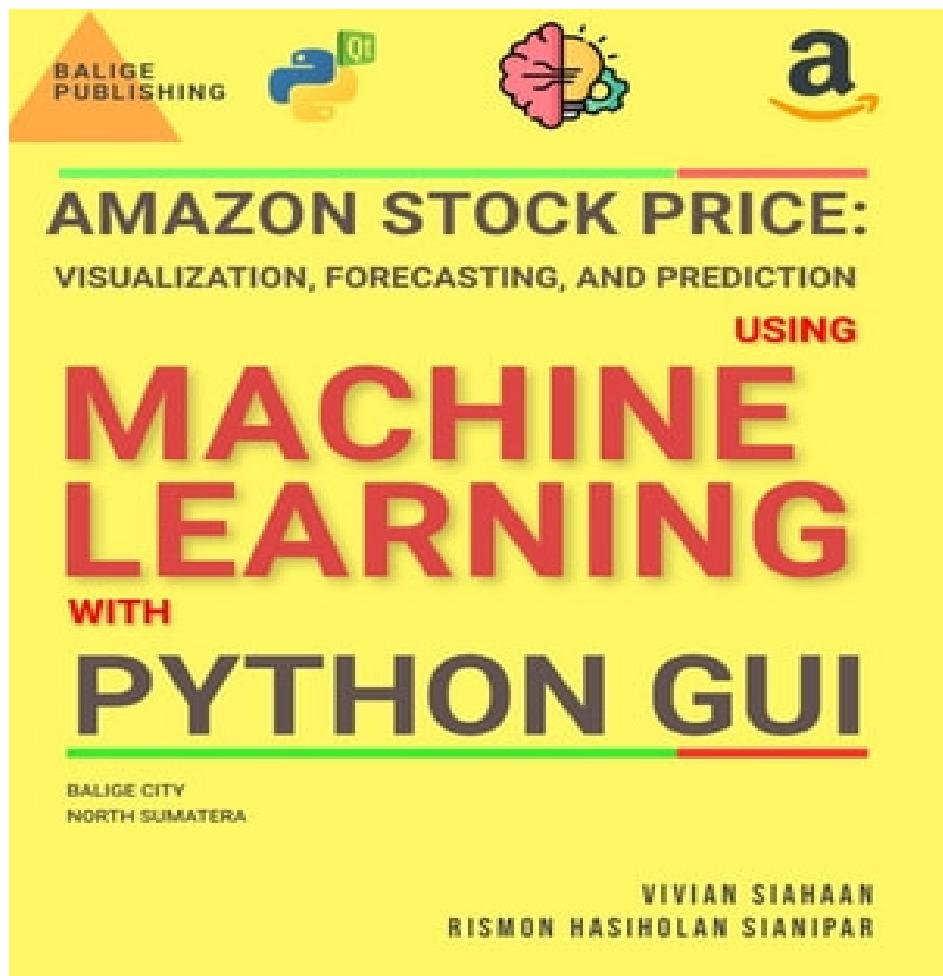
MACHINE LEARNING

WITH

PYTHON GUI

BALIGE CITY
NORTH SUMATERA

VIVIAN SIAHAAN
RISMON HASIHOLAH SIANIPAR



AMAZON STOCK PRICE: VISUALIZATION, FORECASTING, AND PREDICTION USING MACHINE LEARNING WITH PYTHON GUI

AMAZON STOCK PRICE: VISUALIZATION, FORECASTING, AND PREDICTION USING MACHINE LEARNING WITH PYTHON GUI

Second Edition

VIVIAN SIAHAAN
RISMON HASIHOLAN SIANIPAR

Copyright © 2023 BALIGE Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor BALIGE Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book. BALIGE Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BALIGE Publishing cannot guarantee the accuracy of this information.

Copyright © 2023 BALIGE Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor BALIGE Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book. BALIGE Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BALIGE Publishing cannot guarantee the accuracy of this information.

Published: JUNE 2023
Production reference: 06060223
Published by BALIGE Publishing Ltd.
BALIGE, North Sumatera

ABOUT THE AUTHOR

ABOUT THE AUTHOR



Vivian Siahaan is a fast-learner who likes to do new things. She was born, raised in Hinalang Bagasan, Balige, on the banks of Lake Toba, and completed high school education from SMAN 1 Balige. She started herself learning Java, Android, JavaScript, CSS, C ++, Python, R, Visual Basic, Visual C #, MATLAB, Mathematica, PHP, JSP, MySQL, SQL Server, Oracle, Access, and other programming languages. She studied programming from scratch, starting with the most basic syntax and logic, by building several simple and applicable GUI applications. Animation and games are fields of programming that are interests that she always wants to develop. Besides studying mathematical logic and programming, the author also has the pleasure of reading novels. Vivian Siahaan has written dozens of ebooks that have been published on Sparta Publisher: Data Structure with Java; Java Programming: Cookbook; C ++ Programming: Cookbook; C Programming For High Schools / Vocational Schools and Students; Java Programming for SMA / SMK; Java Tutorial: GUI, Graphics and Animation; Visual Basic Programming: From A to Z; Java Programming for Animation and Games; C # Programming for SMA / SMK and Students; MATLAB For Students and Researchers; Graphics in JavaScript: Quick Learning Series; JavaScript Image Processing Methods: From A to Z; Java GUI Case Study: AWT & Swing; Basic CSS and JavaScript; PHP / MySQL Programming: Cookbook; Visual Basic: Cookbook; C ++ Programming for High Schools / Vocational Schools and Students; Concepts and Practices of C ++; PHP / MySQL For Students; C # Programming: From A to Z; Visual Basic for SMA / SMK and Students; C # .NET and SQL Server for High School / Vocational School and Students. At the ANDI Yogyakarta publisher, Vivian Siahaan also wrote a number of books including: Python Programming Theory and Practice; Python GUI Programming; Python GUI and Database; Build From Zero School Database Management System In Python / MySQL; Database Management System in Python / MySQL; Python / MySQL For Management Systems of Criminal Track Record Database; Java / MySQL For Management Systems of Criminal Track Records Database; Database and Cryptography Using Java / MySQL; Build From Zero School Database Management System With Java / MySQL.



Rismon Hasiholan Sianipar was born in Pematang Siantar, in 1994. After graduating from SMAN 3 Pematang Siantar 3, the writer traveled to the city of Jogjakarta. In 1998 and 2001 the author completed his Bachelor of Engineering (S.T) and Master of

Engineering (M.T) education in the Electrical Engineering of Gadjah Mada University, under the guidance of Prof. Dr. Adhi Soesanto and Prof. Dr. Thomas Sri Widodo, focusing on research on non-stationary signals by analyzing their energy using time-frequency maps. Because of its non-stationary nature, the distribution of signal energy becomes very dynamic on a time-frequency map. By mapping the distribution of energy in the time-frequency field using discrete wavelet transformations, one can design non-linear filters so that they can analyze the pattern of the data contained in it. In 2003, the author received a Monbukagakusho scholarship from the Japanese Government. In 2005 and 2008, he completed his Master of Engineering (M.Eng) and Doctor of Engineering (Dr.Eng) education at Yamaguchi University, under the guidance of Prof. Dr. Hidetoshi Miike. Both the master's thesis and his doctoral thesis, R.H. Sianipar combines SR-FHN (Stochastic Resonance Fitzhugh-Nagumo) filter strength with cryptosystem ECC (elliptic curve cryptography) 4096-bit both to suppress noise in digital images and digital video and maintain its authenticity. The results of this study have been documented in international scientific journals and officially patented in Japan. One of the patents was published in Japan with a registration number 2008-009549. He is active in collaborating with several universities and research institutions in Japan, particularly in the fields of cryptography, cryptanalysis and audio / image / video digital forensics. R.H. Sianipar also has experience in conducting code-breaking methods (cryptanalysis) on a number of intelligence data that are the object of research studies in Japan. R.H. Sianipar has a number of Japanese patents, and has written a number of national / international scientific articles, and dozens of national books. R.H. Sianipar has also participated in a number of workshops related to cryptography, cryptanalysis, digital watermarking, and digital forensics. In a number of workshops, R.H. Sianipar helps Prof. Hidetoshi Miike to create applications related to digital image / video processing, steganography, cryptography, watermarking, non-linear screening, intelligent descriptor-based computer vision, and others, which are used as training materials. Field of interest in the study of R.H. Sianipar is multimedia security, signal processing / digital image / video, cryptography, digital communication, digital forensics, and data compression / coding. Until now, R.H. Sianipar continues to develop applications related to analysis of signal, image, and digital video, both for research purposes and for commercial purposes based on the Python programming language, MATLAB, C ++, C, VB.NET, C # .NET, R, and Java.

ABOUT THE BOOK

Amazon is an American multinational technology company that is known for its e-commerce, cloud computing, digital streaming, and artificial intelligence services. It was founded by Jeff Bezos in 1994 and is headquartered in Seattle, Washington. Amazon's primary business is its online marketplace, where it offers a wide range of products, including books, electronics, household items, and more. The company has expanded its operations to various countries and is one of the largest online retailers globally.

In addition to its e-commerce business, Amazon has ventured into other areas. It provides cloud computing services through Amazon Web Services (AWS), which offers on-demand computing power, storage, and other services to individuals, businesses, and governments. AWS has become a significant revenue source for Amazon. Amazon has also made a significant impact on the entertainment industry. It operates Amazon Prime Video, a streaming platform that offers a wide selection of movies, TV shows, and original content.

As for Amazon's stock price, it has experienced substantial growth since the company went public in 1997. The stock has been highly valued by investors due to Amazon's consistent revenue growth, market dominance, and innovation. The stock price has seen both ups and downs over the years, reflecting market trends and investor sentiment.

The dataset used in this project starts from 14-May-1997 and is updated till 27-Oct-2021. It contains 6155 rows and 7 columns. The columns in the dataset are Date, Open, High, Low, Close, Adj Close, and Volume. In this project, you will involve technical indicators such as daily returns, Moving Average Convergence-Divergence (MACD), Relative Strength Index (RSI), Simple Moving Average (SMA), lower and upper bands, and standard deviation.

To perform forecasting based on regression on Adj Close price of Amazon stock price, you will use: Linear Regression, Random Forest regression, Decision Tree regression, Support Vector Machine regression, Naïve Bayes regression, K-Nearest Neighbor regression, Adaboost regression, Gradient Boosting regression, Extreme Gradient

Boosting regression, Light Gradient Boosting regression, Catboost regression, MLP regression, Lasso regression, and Ridge regression.

The machine learning models used predict Amazon stock daily returns as target variable are K-Nearest Neighbor classifier, Random Forest classifier, Naive Bayes classifier, Logistic Regression classifier, Decision Tree classifier, Support Vector Machine classifier, LGBM classifier, Gradient Boosting classifier, XGB classifier, MLP classifier, and Extra Trees classifier. Finally, you will develop GUI to plot boundary decision, distribution of features, feature importance, predicted values versus true values, confusion matrix, learning curve, performance of the model, and scalability of the model.

CONTENT

CONTENT

FEATURES AND TECHNICAL INDICATORS ANALYSIS	1
DESCRIPTION	2
READING DATASET AND CHECKING NULL VALUES	3
CHECKING CORRELATION	7
EXTRACTING TIME-RELATED FEATURES	11
VISUALIZING TIME-RELATED FEATURES	14
VISUALIZING SCATTER DISTRIBUTION	18
VISUALIZING GROUPED DATAFRAME	22
STATISTICAL DESCRIPTION AND CATEGORIZED FEATURES HISTOGRAM	27
ANALYZING YEAR-WISE DATA	35
ANALYZING MONTH-WISE DATA	42
COMPUTING TECHNICAL INDICATORS	51
SOURCE CODE	64
REGRESSION USING MACHINE LEARNING	73
FORECASTING ON ADJ CLOSE VALUE USING MACHINE LEARNING	73
FORECASTING USING LINEAR REGRESSION	79
FORECASTING USING RANDOM FOREST REGRESSION	83
FORECASTING USING DECISION TREE REGRESSION	88
FORECASTING USING K-NEAREST NEIGHBOURS (KNN) REGRESSION	92
FORECASTING USING ADABOOST REGRESSION	96
FORECASTING USING GRADIENT BOOSTING REGRESSION	100
FORECASTING USING EXTREME GRADIENT BOOSTING REGRESSION	105
FORECASTING USING LIGHT GRADIENT BOOSTING MACHINE	109
REGRESSION	113
FORECASTING USING CATBOOST REGRESSION	118
FORECASTING USING SUPPORT VECTOR REGRESSION	122
FORECASTING USING MULTI LAYER PERCEPTRON REGRESSION	125
FORECASTING USING LASSO REGRESSION	129
FORECASTING USING RIDGE REGRESSION	130
PREDICTING STOCK DAILY RETURN USING MACHINE LEARNING	135
DESCRIPTION	135
SPLITTING AND NORMALIZING DATA	136
DEFINING HELPER FUNCTIONS	141
	149
	157
	165
	172
	178

PREDICTING DAILY RETURN USING SUPPORT VECTOR MACHINE	185
PREDICTING DAILY RETURN USING LOGISTIC REGRESSION	191
PREDICTING DAILY RETURN USING K-NEAREST NEIGHBOURS	198
PREDICTING DAILY RETURN USING DECISION TREE	205
PREDICTING DAILY RETURN USING RANDOM FOREST	212
PREDICTING DAILY RETURN USING GRADIENT BOOSTING	218
PREDICTING DAILY RETURN USING EXTREME GRADIENT BOOSTING	225
PREDICTING DAILY RETURN USING MULTI LAYER PERCEPTRON	
PREDICTING DAILY RETURN USING LIGHT GRADIENT BOOSTING MACHINE	
PREDICTING DAILY RETURN USING GAUSSIAN MIXTURE MODEL	
PREDICTING DAILY RETURN USING EXTRA TREES	
SOURCE CODE	243
<i>CREATING GRAPHICAL USER INTERFACE USING PYQT5</i>	
DESCRIPTION	243
DESIGNING GUI	244
DEFINING HELPER FUNCTIONS	250
POPULATING COMBOBOXES AND READING DATASET	257
PLOTTING CASE DISTRIBUTION	261
PLOTTING FEATURES DISTRIBUTION	266
PLOTTING GROUPED DISTRIBUTION	271
PLOTTING MONTH-WISE AND YEAR-WISE DISTRIBUTION	275
PLOTTING TECHNICAL INDICATORS	279
<i>CREATING GUI FOR REGRESSION- BASED FORECASTING</i>	282
SPLITTING DATA FOR REGRESSION	282
FORECASTING USING LINEAR REGRESSION	284
FORECASTING USING RANDOM REGRESSION	288
FORECASTING USING DECISION TREE REGRESSION	289
FORECASTING USING K-NEAREST NEIGHBORS REGRESSION	291
FORECASTING USING ADABoost REGRESSION	293
FORECASTING USING GRADIENT BOOSTING REGRESSION	294
FORECASTING USING EXTREME GRADIENT BOOSTING REGRESSION	296
FORECASTING USING LIGHT GRADIENT BOOSTING MACHINE REGRESSION	298
FORECASTING USING CATBOOST REGRESSION	299
FORECASTING USING SUPPORT VECTOR REGRESSION	301
FORECASTING USING MULTI LAYER PERCEPTRON REGRESSION	302
FORECASTING USING LASSO REGRESSION	304
FORECASTING USING RIDGE REGRESSION	306
<i>CREATING GUI FOR DAILY RETURN PREDICTION</i>	309
IMPLEMENTING FEATURE SCALING AND SPLITTING DATA	309
DEFINING HELPER FUNCTIONS	315
PREDICTING DAILY RETURN USING LOGISTIC REGRESSION	323
PREDICTING DAILY RETURN USING SUPPORT VECTOR MACHINE	326
PREDICTING DAILY RETURN USING DECISION TREE	329
PREDICTING DAILY RETURN USING K-NEAREST NEIGHBORS	333
PREDICTING DAILY RETURN USING RANDOM FOREST	336
PREDICTING DAILY RETURN USING GRADIENT BOOSTING	339
PREDICTING DAILY RETURN USING NAÏVE BAYES	343
	346
	349
	352
	355
	358
	361

PREDICTING DAILY RETURN USING ADABOOST
PREDICTING DAILY RETURN USING EXTREME GRADIENT BOOSTING
PREDICTING DAILY RETURN USING LIGHT GRADIENT BOOSTING
MACHINE
PREDICTING DAILY RETURN USING MULTI LAYER PERCEPTRON
PREDICTING DAILY RETURN USING EXTRA TREES
SOURCE CODE

FEATURES AND TEHCNICAL INDICATORS

ANALYSIS FEATURES AND TEHCNICAL INDICATORS ANALYSIS

DESCRIPTION DESCRIPTION

Amazon.com, Inc. engages in the provision of online retail shopping services. It operates through the following business segments: North America, International, and Amazon Web Services (AWS). The North America segment includes retail sales of consumer products and subscriptions through North America-focused websites such as www.amazon.com and www.amazon.ca. The International segment offers retail sales of consumer products and subscriptions through internationally-focused websites. The Amazon Web Services segment involves in the global sales of compute, storage, database,

and AWS service offerings for start-ups, enterprises, government agencies, and academic institutions. The company was founded by Jeffrey P. Bezos in July 1994 and is headquartered in Seattle, WA.

The data starts from 14-May-1997 and is updated till 27-Oct-2021. It contains 6155 rows and 7 columns. The columns in the dataset are Date, Open, High, Low, Close, Adj Close, and Volume.

READING DATASET AND CHECKING NULL VALUES

READING DATASET AND CHECKING NULL VALUES

Step 1	Download dataset from https://viviansiahaan.blogspot.com/2023/06/amazon-stock-price-visualization.html and save it to your working directory. Unzip the file, Amazon.csv , and put it into working directory.
Step 2	Open a new Python script and save it as amazon.py .
Step 3	Import various libraries and modules for data analysis and machine learning tasks. The code snippet you provided imports a wide range of libraries and modules, including numpy, pandas, matplotlib, seaborn, sklearn, statsmodels, plotly, joblib, itertools, and several machine learning algorithms such as logistic regression, decision trees, support vector machines, random forests, and more.

```
1 #Amazon.py
2 import numpy as np
3 import pandas as pd
4 import matplotlib
5 import matplotlib.pyplot as plt
6 import seaborn as sns
7 import datetime as dt
8 sns.set_style('darkgrid')
9 from sklearn.preprocessing import LabelEncoder
10 import warnings
11 warnings.filterwarnings('ignore')
12 import os
13 import plotly.graph_objs as go
14 import joblib
15 import itertools
16 from sklearn.metrics import roc_auc_score,roc_curve,
17 explained_variance_score
18 from sklearn.model_selection import cross_val_score
19 from statsmodels.tsa.seasonal import seasonal_decompose as
20 season
21 from sklearn.metrics import mean_squared_error,
22 mean_absolute_error
23 from sklearn.metrics import accuracy_score,
24 balanced_accuracy_score
25 from sklearn.model_selection import train_test_split,
26 RandomizedSearchCV, GridSearchCV,StratifiedKFold
27 from sklearn.preprocessing import StandardScaler,
28 MinMaxScaler, RobustScaler
29 from sklearn.linear_model import LogisticRegression
30 from sklearn.naive_bayes import GaussianNB
```

```
31 from sklearn.tree import DecisionTreeClassifier
32 from sklearn.svm import SVC
33 from sklearn.ensemble import RandomForestClassifier,
34 ExtraTreesClassifier
35 from sklearn.neighbors import KNeighborsClassifier
36 from sklearn.ensemble import AdaBoostClassifier,
37 GradientBoostingClassifier
38 from xgboost import XGBClassifier
39 from sklearn.neural_network import MLPClassifier
40 from sklearn.linear_model import SGDClassifier
41 from sklearn.preprocessing import StandardScaler, \
42     LabelEncoder, OneHotEncoder
43 from sklearn.metrics import confusion_matrix,
44 accuracy_score, recall_score, precision_score
45 from sklearn.metrics import classification_report, f1_score,
46 plot_confusion_matrix
47 from catboost import CatBoostClassifier
48 from lightgbm import LGBMClassifier
49 from imblearn.over_sampling import SMOTE
50 from sklearn.model_selection import learning_curve
51 from mlxtend.plotting import plot_decision_regions
52 from sklearn.decomposition import PCA
53 from sklearn.linear_model import LinearRegression
54 from sklearn.ensemble import RandomForestRegressor
55 from sklearn.tree import DecisionTreeRegressor
56 from sklearn.svm import SVR
57 from sklearn.pipeline import make_pipeline
58 from sklearn.naive_bayes import GaussianNB
59 from sklearn.neighbors import KNeighborsRegressor
60 from sklearn.ensemble import AdaBoostRegressor
61 from sklearn.ensemble import GradientBoostingRegressor
62 from xgboost import XGBRegressor
63 from lightgbm import LGBMRegressor
64 from catboost import CatBoostRegressor
65 from sklearn.neural_network import MLPRegressor
66 from statsmodels.tsa.holtwinters import ExponentialSmoothing
67 from statsmodels.tsa.stattools import adfuller, acf, pacf
from statsmodels.tsa.arima_model import ARIMA
from pmdarima.utils import decomposed_plot
from pmdarima.arima import decompose
from pmdarima import auto_arima
from sklearn.linear_model import LassoCV
from sklearn.linear_model import RidgeCV
from sklearn.mixture import GaussianMixture
```

	<pre> 31 from sklearn.tree import DecisionTreeClassifier 32 from sklearn.svm import SVC 33 from sklearn.ensemble import RandomForestClassifier, 34 ExtraTreesClassifier 35 from sklearn.neighbors import KNeighborsClassifier 36 from sklearn.ensemble import AdaBoostClassifier, 37 GradientBoostingClassifier 38 from xgboost import XGBClassifier 39 from sklearn.neural_network import MLPClassifier 40 from sklearn.linear_model import SGDClassifier 41 from sklearn.preprocessing import StandardScaler, \ 42 LabelEncoder, OneHotEncoder 43 from sklearn.metrics import confusion_matrix, 44 accuracy_score, recall_score, precision_score 45 from sklearn.metrics import classification_report, f1_score, 46 plot_confusion_matrix 47 from catboost import CatBoostClassifier 48 from lightgbm import LGBMClassifier 49 from imblearn.over_sampling import SMOTE 50 from sklearn.model_selection import learning_curve 51 from mlxtend.plotting import plot_decision_regions 52 from sklearn.decomposition import PCA 53 from sklearn.linear_model import LinearRegression 54 from sklearn.ensemble import RandomForestRegressor 55 from sklearn.tree import DecisionTreeRegressor 56 from sklearn.svm import SVR 57 from sklearn.pipeline import make_pipeline 58 from sklearn.naive_bayes import GaussianNB 59 from sklearn.neighbors import KNeighborsRegressor 60 from sklearn.ensemble import AdaBoostRegressor 61 from sklearn.ensemble import GradientBoostingRegressor 62 from xgboost import XGBRegressor 63 from lightgbm import LGBMRegressor 64 from catboost import CatBoostRegressor 65 from sklearn.neural_network import MLPRegressor 66 from statsmodels.tsa.holtwinters import ExponentialSmoothing 67 from statsmodels.tsa.stattools import adfuller, acf, pacf from statsmodels.tsa.arima_model import ARIMA from pmdarima.utils import decomposed_plot from pmdarima.arima import decompose from pmdarima import auto_arima from sklearn.linear_model import LassoCV from sklearn.linear_model import RidgeCV from sklearn.mixture import GaussianMixture </pre>
Step 4	<p>Read and check the shape of dataset. Let's break down the code step by step:</p> <ol style="list-style-type: none"> Step 1: curr_path = os.getcwd() <p>This line of code assigns the current working directory (the directory where the Python script is located) to the variable curr_path using the os.getcwd() function from the os module. This will be used to construct the file path for reading the CSV file.</p> <ol style="list-style-type: none"> Step 2: df = pd.read_csv(curr_path+"/Amazon.csv") <p>This line of code reads the CSV file named "Amazon.csv" located in the current working directory. It uses the pd.read_csv() function from the pandas library (pd) to read the CSV file and assigns the resulting DataFrame to the variable df.</p> <ol style="list-style-type: none"> Step 3: print(df.shape) <p>This line of code prints the shape of the DataFrame df. The shape attribute of a DataFrame returns a tuple</p>

representing the dimensions of the **DataFrame**, with the number of rows followed by the number of columns. By printing **df.shape**, you can see the number of rows and columns in the DataFrame, providing information about the size of the dataset.

In summary, the code reads a CSV file named "Amazon.csv" located in the current working directory using pandas, assigns the data to a DataFrame df, and then prints the shape of the DataFrame to display the number of rows and columns in the dataset.

```
1 curr_path = os.getcwd()
2 df = pd.read_csv(curr_path+"/Amazon.csv")
3
4 #Checks shape
5 print(df.shape)
```

Output:

(4317, 7)

The dataset contains 4317 rows and 7 columns.

Step 5 Read every column in dataset. This line of code prints the column names of the DataFrame **df**. Let's break down the code:

- **print("Data Columns --> ", list(df.columns))**
- **print()** is a built-in Python function used to display output to the console.
- "Data Columns -->" is a string that serves as a label or prefix for the printed output.
- **list(df.columns)** converts the column names of the **DataFrame df** into a list. The **columns** attribute of a **DataFrame** returns an **Index** object containing the column names.
- The resulting list of column names is then printed along with the label.

Overall, this code provides a convenient way to quickly view the column names of the DataFrame. The output will be a list of column names prefixed with the label "Data Columns -->".

```
1 #Reads columns
2 print("Data Columns --> ",list(df.columns))
```

Output:

Data Columns --> ['Date', 'Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']

Step 6 Check the information of dataset. This code prints the information about the DataFrame df, including the data types

of each column, the number of non-null values, and the memory usage.

Let's break down the code:

1. Step 1: `print(df.info())`

- This line of code uses the `info()` method of the **DataFrame df** to retrieve and print information about the dataset. The `info()` method provides a concise summary of the **DataFrame**, including the column names, data types, and memory usage.
- The output of `df.info()` typically includes the following information:
 - The number of rows and columns in the **DataFrame**
 - The column names and their corresponding data types
 - The count of non-null values for each column
 - The memory usage of the **DataFrame**

Overall, this code is useful for quickly understanding the structure of the dataset, identifying the data types of each column, and checking for missing values.

```
1 #Checks dataset information
2 print(df.info())
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6155 entries, 1997-05-15 to 2021-10-27
Data columns (total 11 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   Open         6155 non-null   float64
 1   High          6155 non-null   float64
 2   Low           6155 non-null   float64
 3   Close          6155 non-null   float64
 4   Adj Close     6155 non-null   float64
 5   Volume         6155 non-null   int64  
 6   Day            6155 non-null   int64  
 7   Month          6155 non-null   int64  
 8   Year           6155 non-null   int64  
 9   Week           6155 non-null   int64  
 10  Quarter        6155 non-null   int64  
dtypes: float64(5), int64(6)
memory usage: 577.0 KB
None
```

Step 7 Checks null values and plot them. This code performs the following steps:

1. Step 1: `print(df.isnull().sum())`

This line of code calculates the sum of null values in each column of the **DataFrame df** using the `isnull()` function. The `isnull()` function returns a **DataFrame** of

the same shape as **df** with boolean values indicating whether each element is null or not. The **sum()** function is then applied to this **DataFrame**, which sums up the null values for each column. The resulting sums are printed to show the number of null values in each column.

2. Step 2: **print('Total number of null values: ', df.isnull().sum().sum())**

This line of code calculates the total number of null values in the entire **DataFrame** by applying the **sum()** function twice. The first **sum()** calculates the sum of null values for each column, and the second **sum()** calculates the sum of these column-wise sums. The total number of null values is then printed.

3. Step 3: **missing = df.isna().sum().reset_index()**

This line of code calculates the sum of null values in each column of the **DataFrame df** using the **isna()** function. The **isna()** function is similar to **isnull()** and also returns a **DataFrame** with boolean values indicating whether each element is null or not. The **sum()** function is then applied to this **DataFrame**, and the result is stored in the missing **DataFrame**. The **reset_index()** function is used to reset the index of the missing **DataFrame**.

4. Step 4: **missing.columns = ['features', 'total_missing']**

This line of code renames the columns of the missing **DataFrame** to 'features' and 'total_missing', which represent the column names and the corresponding total number of missing values, respectively.

5. Step 5: **missing['percent'] = (missing['total_missing'] / len(df)) * 100**

This line of code calculates the percentage of missing values for each column by dividing the total number of missing values in each column (**missing['total_missing']**) by the total number of rows in the **DataFrame** (**len(df)**), and then multiplying by 100. The result is stored in a new column called 'percent' in the missing **DataFrame**.

6. Step 6: **missing.index = missing['features']**

This line of code sets the index of the missing **DataFrame** to the 'features' column, which contains the column names.

7. Step 7: **del missing['features']**

This line of code deletes the 'features' column from the missing **DataFrame**, as it is now redundant with the index.

8. Step 8: **plt.figure(figsize=(15,8))**

This line of code creates a new figure for plotting with a specified figure size of 15 inches by 8 inches.

9. Step 9: **missing['total_missing'].plot(kind='bar')**

This line of code plots a bar chart using the 'total_missing' column from the missing **DataFrame**. This column contains the total number of missing values for each feature/column.

10. Step 10: **plt.title('Missing Values Count', fontsize=25)**

This line of code sets the title of the plot to "Missing Values Count" with a font size of 25.

Overall, this code provides a summary of the missing values in the **DataFrame df** by printing the number of null values in each column and plotting a bar chart showing the total count of missing values for each feature.

```
1 #Checks null values
2 print(df.isnull().sum())
3 print('Total number of null values: ', 
4 df.isnull().sum().sum())
5
6 #Plots null values
7 missing = df.isna().sum().reset_index()
8 missing.columns = ['features', 'total_missing']
9 missing['percent'] = (missing['total_missing'] / len(df)) *
10 100
11 missing.index = missing['features']
12 del missing['features']
13 plt.figure(figsize=(15,8))
missing['total_missing'].plot(kind = 'bar')
plt.title('Missing Values Count', fontsize = 25)
```

Output:

```
Date      0
High      0
Low       0
Open      0
Close     0
Volume    0
Adj Close 0
dtype: int64
Total number of null values:  0
```

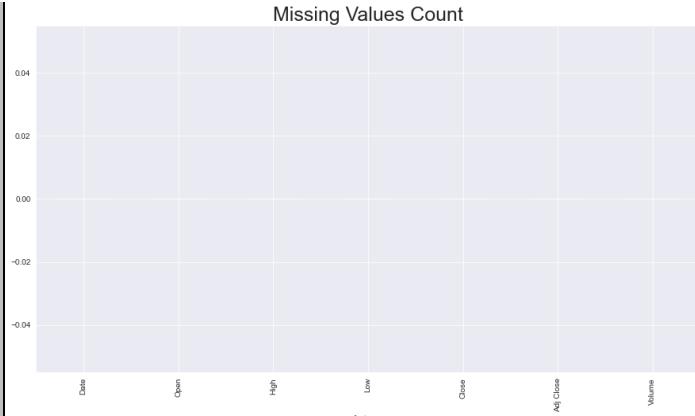


Figure 1.1 The null values in each column

CHECKING CORRELATION CHECKING CORRELATION

Step 1 Print coefficient correlation of every column with **Adj Close** column. This code calculates the coefficient correlation of every column in the **DataFrame df** with the 'Adj Close' column and prints the correlation values in descending order.

Let's break down the code:

1. Step 1: `all_corr = df.corr().abs()['Adj Close'].sort_values(ascending=False)`

This line of code calculates the correlation between each column in the **DataFrame df** and the 'Adj Close' column using the **corr()** function. The **corr()** function computes the pairwise correlation of columns, and **abs()** is used to take the absolute values of the correlation coefficients. The resulting correlation values are then accessed for the 'Adj Close' column using `['Adj Close']`. Finally, **sort_values(ascending=False)** sorts the correlation values in descending order and assigns the result to the **all_corr** variable.

2. Step 2: `print(all_corr.to_string())`

This line of code prints the **all_corr** variable, which contains the correlation values of each column with the 'Adj Close' column. The **to_string()** function is used to convert the **all_corr** variable to a string representation before printing.

Overall, this code provides a summary of the correlation coefficients between each column in the **DataFrame df** and the 'Adj Close' column. The correlation values are printed in descending order, showing the columns that have the highest absolute correlation with the 'Adj Close' column at the top.

```
1 #Prints coefficient correlation of every column with Adj
2 Close column
3 all_corr = df.corr().abs()['Adj Close'].sort_values(\n4     ascending = False)
print(all_corr.to_string())
```

Output:

```
Adj Close      1.000000
Close          1.000000
Low            0.999928
High           0.999924
Open            0.999842
Volume         0.240122
```

Step 2 Plot pair correlation of all features. This code creates a pair plot of the correlation between all features in the **DataFrame df** using the seaborn library.

Let's break down the code:

1. Step 1: **sns.pairplot(df)**

This line of code uses the **pairplot()** function from the seaborn library (**sns**) to create a pair plot of all the features in the **DataFrame df**. The pair plot visualizes the pairwise relationships between all pairs of features, displaying scatter plots for numeric variables and bar plots for categorical variables.

2. Step 2: **plt.tight_layout()**

This line of code adjusts the layout of the plot to ensure that all elements fit within the figure area properly. It helps to prevent overlapping or cutoff labels and ensures a more visually appealing plot.

Overall, this code snippet generates a pair plot that allows you to visualize the correlation between all features in the **DataFrame df**. It provides a comprehensive overview of the relationships between variables, which can be helpful for identifying patterns, trends, or dependencies in the data.

```
1 #Plots pair correlation of all features
2 sns.pairplot(df)
3 plt.tight_layout()
```

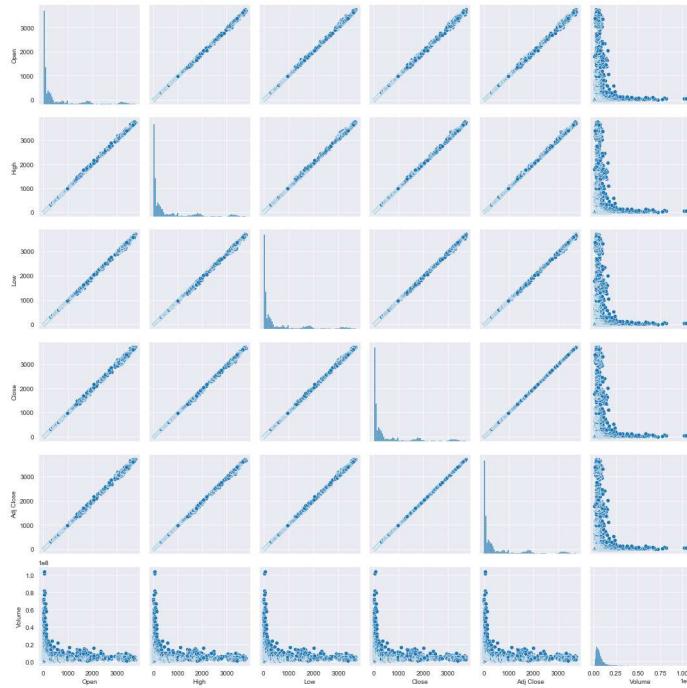


Figure 1.2 The pair correlation of all features

Step 3 Plot the correlation between each feature by using heatmap. This code creates a heatmap to visualize the correlation between each feature in the **DataFrame df**.

Let's break down the code:

1. Step 1: **fig, ax = plt.subplots(figsize=(20,10))**

This line of code creates a figure and an axes object for plotting the heatmap. The figsize parameter specifies the size of the figure, where (20, 10) represents the width and height of the figure in inches.

2. Step 2: **mask = np.triu(np.ones_like(df.corr()), dtype=np.bool)**

This line of code creates a mask array using **np.triu()** to ensure that only the lower triangular portion of the correlation matrix is shown in the heatmap. The **np.ones_like()** function creates an array of ones with the same shape as the correlation matrix, and **dtype=np.bool** sets the data type of the mask array to boolean.

3. Step 3: **sns.heatmap(df.corr(), annot=True, cmap="Reds", mask=mask, linewidth=0.5)**

This line of code generates the heatmap using the **heatmap()** function from the seaborn library (**sns**). The **df.corr()** calculates the correlation matrix of the **DataFrame df**, and the resulting matrix is passed to the

heatmap() function. Additional parameters are specified:

- **annot=True** adds numerical annotations to the heatmap cells, displaying the correlation values.
- **cmap="Reds"** sets the color map for the heatmap. In this case, the "Reds" colormap is used.
- **mask=mask** applies the mask to the heatmap, hiding the upper triangular portion of the correlation matrix.
- **linewidth=0.5** sets the width of the lines that separate each cell in the heatmap.

Overall, this code snippet generates a heatmap that visualizes the correlation between each feature in the **DataFrame df**. The color intensity and numerical annotations provide insights into the strength and direction of the correlations. The lower triangular portion of the heatmap is shown, while the upper triangular portion is masked. The resulting plot can help identify patterns, relationships, and potential multicollinearity among the features.

```

1 #Plots the correlation between each feature by using heatmap
2 fig, ax = plt.subplots(figsize=(20,10))
3 mask = np.triu(np.ones_like(df.corr(), dtype=np.bool))
4 sns.heatmap(df.corr(), annot=True, cmap="Reds", \
5             mask=mask, linewidth=0.5)

```

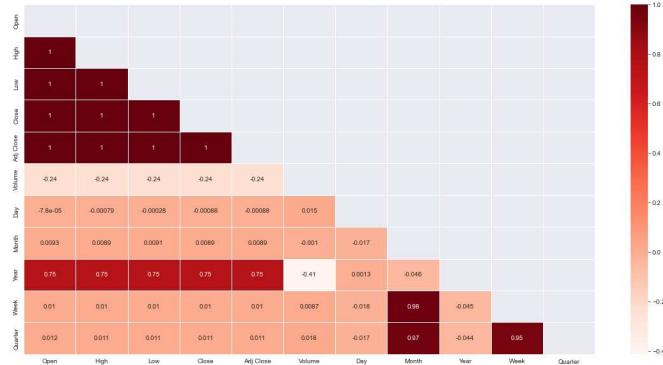


Figure 1.3 The correlation matrix

EXTRACTING TIME-RELATED FEATURES

EXTRACTING TIME-RELATED FEATURES

Step 1 Extract day, month, week, quarter, and year. Then, set **Date** column as index. This code extracts additional time-related features from the 'Date' column in the **DataFrame df**. It adds columns for day of the week, month, year, week number, and quarter. Then, it sets the 'Date' column as the index of the DataFrame.

Let's break down the code:

1. Step 1: **df['Date'] = pd.to_datetime(df['Date'])**

This line of code converts the 'Date' column in the **DataFrame df** to a datetime data type using the **pd.to_datetime()** function from the pandas library. This step is necessary to perform various time-related operations.

2. Step 2: **df['Day'] = df['Date'].dt.weekday**

This line of code extracts the day of the week from the 'Date' column using the **dt.weekday** attribute of the datetime object. It assigns the resulting day values (0 for Monday, 1 for Tuesday, and so on) to a new column called 'Day' in the **DataFrame df**.

3. Step 3: **df['Month'] = df['Date'].dt.month**

This line of code extracts the month from the 'Date' column using the **dt.month** attribute and assigns the month values (1 to 12) to a new column called 'Month' in the **DataFrame df**.

4. Step 4: **df['Year'] = df['Date'].dt.year**

This line of code extracts the year from the 'Date' column using the **dt.year** attribute and assigns the year values to a new column called 'Year' in the **DataFrame df**.

5. Step 5: **df['Week'] = df['Date'].dt.week**

This line of code extracts the week number from the 'Date' column using the **dt.week** attribute and assigns the week number values to a new column called 'Week' in the **DataFrame df**.

6. Step 6: **df['Quarter'] = df['Date'].dt.quarter**

This line of code extracts the quarter of the year from the 'Date' column using the **dt.quarter** attribute and assigns the quarter values (1 to 4) to a new column called 'Quarter' in the **DataFrame df**.

7. Step 7: **df = df.set_index("Date")**

This line of code sets the 'Date' column as the index of the **DataFrame df** using the **set_index()** method. This step is useful when working with time-series data, as it allows easy indexing and slicing based on dates.

8. Step 8: **print(df.head(10).to_string())**

This line of code prints the first 10 rows of the modified **DataFrame df**, including the newly added time-related columns ('Day', 'Month', 'Year', 'Week', 'Quarter') and their respective values. The **to_string()** method is used to ensure that the entire **DataFrame** is printed without truncation.

Overall, this code snippet adds several time-related columns to the **DataFrame df** and sets the 'Date' column as the index. It provides additional granularity and allows for time-based analysis and filtering in the dataset.

```
1 #Extracts day, month, week, quarter, and year
2 df['Date'] = pd.to_datetime(df['Date'])
3 df['Day'] = df['Date'].dt.weekday
4 df['Month'] = df['Date'].dt.month
5 df['Year'] = df['Date'].dt.year
6 df['Week']= df['Date'].dt.week
7 df['Quarter']= df['Date'].dt.quarter
8
9 #Sets Date column as index
10 df = df.set_index("Date")
11
12 print(df.head(10).to_string())
```

Output:

Date	Open	High	Low	Close	Adj Close	Volume	Day	Month	Year	Week	Quarter
1997-05-15	2.437500	2.500000	1.927083	1.958333	1.958333	72156000	3	5	1997	20	2
1997-05-16	1.968750	1.979167	1.708333	1.729167	1.729167	14700000	4	5	1997	20	2
1997-05-19	1.760417	1.770833	1.625000	1.708333	1.708333	6106800	0	5	1997	21	2
1997-05-20	1.729167	1.750000	1.635417	1.635417	1.635417	5467200	1	5	1997	21	2
1997-05-21	1.635417	1.645833	1.375000	1.427083	1.427083	18853200	2	5	1997	21	2
1997-05-22	1.437500	1.447917	1.312500	1.395833	1.395833	11776800	3	5	1997	21	2
1997-05-23	1.406250	1.520833	1.333333	1.500000	1.500000	15937200	4	5	1997	21	2
1997-05-27	1.510417	1.645833	1.458333	1.583333	1.583333	8697600	1	5	1997	22	2
1997-05-28	1.625000	1.635417	1.531250	1.531250	1.531250	4574400	2	5	1997	22	2
1997-05-29	1.541667	1.541667	1.479167	1.505208	1.505208	3472800	3	5	1997	22	2

Step 2 Create a dummy dataframe for visualization and convert days and months from numerics to meaningful string. This code creates a dummy DataFrame **df_dummy** based on the DataFrame **df** for visualization purposes. It also converts the numeric values in the 'Day', 'Month', and 'Quarter' columns to meaningful string representations.

Let's break down the code:

1. Step 1: **df_dummy = df.copy()**

This line of code creates a copy of the DataFrame **df** and assigns it to the new DataFrame **df_dummy**. Creating a copy ensures that the original DataFrame **df** is not modified during the visualization process.

2. Step 2: **days = {0:'Sunday',1:'Monday',2:'Tuesday',3:'Wednesday',4:'Thursday',5: 'Friday',6:'Saturday'}**

This line of code defines a dictionary **days** that maps numeric values (0 to 6) representing days of the week to their corresponding string names.

3. Step 3: **df_dummy['Day'] = df_dummy['Day'].map(days)**

This line of code maps the values in the 'Day' column of the DataFrame **df_dummy** to their corresponding string names using the **map()** function. It replaces the numeric values with the corresponding day names from the **days** dictionary.

4. Step 4: **months = {1:'January', 2:'February', 3:'March', 4:'April', 5:'May', 6:'June',7:'July',8:'August',9:'September',10:'October',11:'November',12:'December'}**

This line of code defines a dictionary **months** that maps numeric values (1 to 12) representing months to their corresponding string names.

5. Step 5: **df_dummy['Month'] = df_dummy['Month'].map(months)**

This line of code maps the values in the 'Month' column of the DataFrame **df_dummy** to their corresponding string names using the **map()** function. It replaces the numeric values with the corresponding month names from the **months** dictionary.

6. Step 6: **quarters = {1:'Jan-March', 2:'April-June',3:'July-Sept',4:'Oct-Dec'}**

This line of code defines a dictionary **quarters** that maps numeric values (1 to 4) representing quarters to their corresponding string names.

7. Step 7: **df_dummy['Quarter'] = df_dummy['Quarter'].map(quarters)**

This line of code maps the values in the 'Quarter' column of the DataFrame **df_dummy** to their corresponding string names using the **map()** function. It replaces the numeric values with the

corresponding quarter names from the quarters dictionary.

Overall, this code snippet creates a dummy DataFrame **df_dummy** based on **df** and converts the numeric values in the 'Day', 'Month', and 'Quarter' columns to meaningful string representations. This conversion improves the interpretability of the data and allows for more intuitive visualization of the time-related features.

```
1 #Creates a dummy dataframe for visualization
2 df_dummy=df.copy()
3
4 #Converts days and months from numerics to meaningful string
5 days =
6 {0: 'Sunday', 1: 'Monday', 2: 'Tuesday', 3: 'Wednesday', 4: 'Thursday', 5:
7 'Friday', 6: 'Saturday'}
8 df_dummy['Day'] = df_dummy['Day'].map(days)
9
10
11 months={1: 'January', 2: 'February', 3: 'March', 4: 'April',
12 5: 'May', 6: 'June', 7: 'July', 8: 'August', 9: 'September',
13 10: 'October', 11: 'November', 12: 'December'}
14 df_dummy['Month']= df_dummy['Month'].map(months)
15
16 quarters = {1: 'Jan-March', 2: 'April-June', 3: 'July-Sept', 4: 'Oct-
17 Dec'}
17 df_dummy['Quarter'] = df_dummy['Quarter'].map(quarters)
18
19 print(df_dummy.head(10).to_string())
```

Output:

VISUALIZING TIME-RELATED FEATURES

VISUALIZING TIME-RELATED FEATURES

Step 1 Define **plot_pie_bar_chart()** method to plot case distribution of a variable in pie chart and bar plot as subplots. It creates a pie chart and a horizontal bar plot as subplots. The function takes the following parameters:

- **df**: The **DataFrame** containing the data.
- **var**: The variable/column in the DataFrame for which the pie chart and bar plot will be created.
- **title** (optional): A title for the plots.

Let's break down the code:

1. Step 1: **plt.figure(figsize=(20,10))**

This line of code creates a new figure with a specified size using the **plt.figure()** function from the matplotlib library. The **figsize** parameter sets the width and height of the figure in inches.

2. Step 2: **plt.subplot(121)**

This line of code creates a subplot grid with 1 row and 2 columns and selects the first subplot as the current subplot for plotting. The (121) argument indicates that

there are 1 row, 2 columns, and the current subplot is the first one.

3. Step 3: **label_list** =
list(df[var].value_counts().index)

This line of code creates a list of unique values in the specified column **var** of the DataFrame **df**. The **value_counts()** function counts the occurrences of each unique value, and the **index** attribute retrieves the unique values as an index object, which is converted to a list.

4. Step 4: **df[var].value_counts().plot.pie(...)**

This line of code plots a pie chart using the **plot.pie()** function of the pandas Series. The **value_counts()** function counts the occurrences of each unique value in the specified column **var**. Additional parameters are specified to customize the pie chart appearance, such as colors, start angle, labels, wedge properties, shadow, and text properties.

5. Step 5: **plt.title("Case distribution of "+ var +"
variable "+title, fontsize=25)**

This line of code sets the title of the pie chart using the **title()** function from matplotlib. The title includes the variable name (**var**) and an optional custom title (**title** parameter).

6. Step 6: **plt.subplot(122)**

This line of code creates the second subplot and selects it as the current subplot for plotting. The (122) argument indicates that there are 1 row, 2 columns, and the current subplot is the second one.

7. Step 7: **ax** =
df[var].value_counts().plot(kind="barh")

This line of code plots a horizontal bar plot using the **plot()** function of the pandas Series with kind="barh". The **value_counts()** function counts the occurrences of each unique value in the specified column **var**. The resulting horizontal bar plot is assigned to the **ax** variable.

8. Step 8: **ax.text(.7,i,j,weight = "bold",fontsize=30)**

This line of code adds text annotations to the horizontal bar plot. It displays the count values (j) on the bars, with a specific position (0.7 x-coordinate), font weight, and font size.

9. Step 9: **plt.title("Count of "+ var +" cases "
+title, fontsize=30)**

This line of code sets the title of the horizontal bar plot, similar to Step 5.

10. Step 10: plt.show()

This line of code displays the overall figure with both subplots.

Overall, this function provides a convenient way to create a pie chart and a horizontal bar plot side by side, showcasing the distribution and count of cases for a specific variable in a **DataFrame**. The function allows for customization of the plot appearance and provides options to include a custom title.

```
1 #Defines function to create pie chart and bar plot as
2 subplots
3 def plot_pie_bar_chart(df,var, title=""):
4     plt.figure(figsize=(20,10))
5     plt.subplot(121)
6     label_list = list(df[var].value_counts().index)
7
8     df[var].value_counts().plot.pie(autopct = "%1.1f%%", \
9         colors = sns.color_palette("prism",7),\
10        startangle = 60,labels=label_list,\
11        wedgeprops={"linewidth":2,"edgecolor":"k"},\
12        shadow =True, textprops={'fontsize': 25}
13    )
14    plt.title("Case distribution of "+ var +" variable"
15 "+title, \
16         fontsize=25)
17
18
19     plt.subplot(122)
20     ax = df[var].value_counts().plot(kind="barh")
21
22     for i,j in enumerate(df[var].value_counts().values):
23         ax.text(.7,i,j,weight = "bold",fontsize=30)
24
25
26     plt.title("Count of "+ var +" cases " +title,
27 fontsize=30)
28     plt.show()
```

Step 2	Plot case distribution of Year . The code is used to create a pie chart and a horizontal bar plot to visualize the case distribution of the " Year " variable in the df_dummy DataFrame. The result is shown in Figure 1.4.
--------	--

```
1 #Plots case distribution of Year  
2 plot_pie_bar_chart(df_dummy, "Year")
```

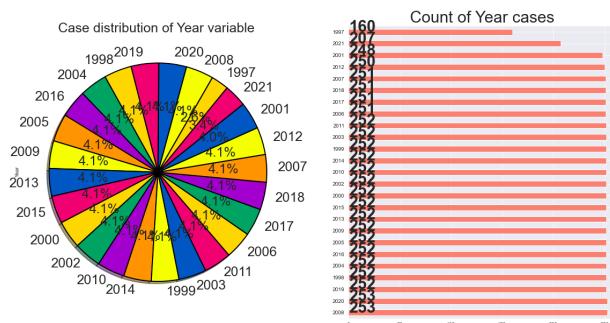


Figure 1.4 The case distribution of Year column

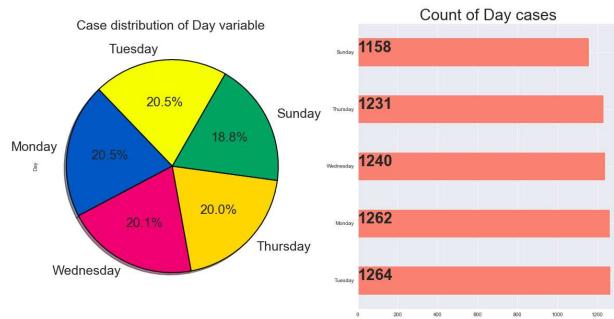


Figure 1.5 The case distribution of Day column

- Step 2 Plot case distribution of **Day**. This code is used to create a pie chart and a horizontal bar plot to visualize the case distribution of the "Day" variable in the **df_dummy DataFrame**.

```
1 #Plots case distribution of Day
2 plot_pie_bar_chart(df_dummy, "Day")
```

The result is shown in Figure 1.5.

- Step 3 Plot case distribution of **Month**. This code will generate a pie chart and a horizontal bar plot showing the distribution and count of cases for each month in the **df_dummy DataFrame**. The pie chart represents the proportion of cases for each month, while the horizontal bar plot displays the count of cases for each month.

```
1 #Plots case distribution of Month
2 plot_pie_bar_chart(df_dummy, "Month")
```

The result is shown in Figure 1.6.

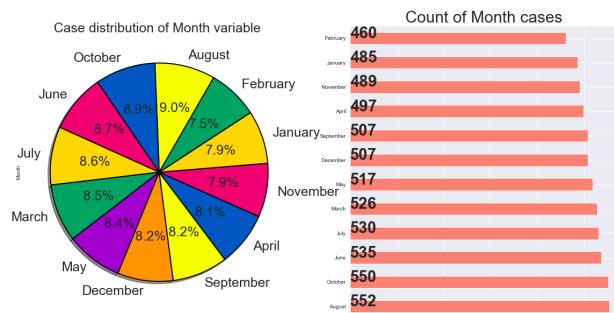


Figure 1.6 The case distribution of Month column

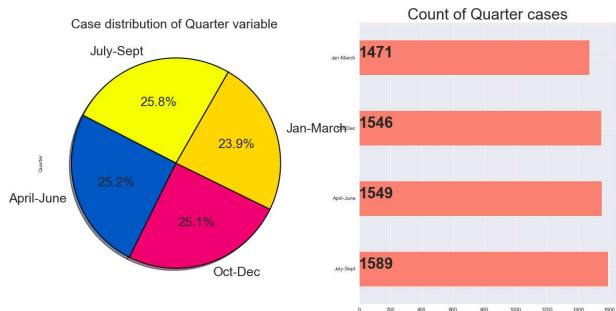


Figure 1.7 The case distribution of **Quarter** column

Step 4 Plot case distribution of **Quarter**. This code will generate a pie chart and a horizontal bar plot showing the distribution and count of cases for each quarter in the **df_dummy DataFrame**. The pie chart represents the proportion of cases for each quarter, while the horizontal bar plot displays the count of cases for each quarter.

```
1 #Plots case distribution of Quarter
2 plot_pie_bar_chart(df_dummy, "Quarter")
```

The result is shown in Figure 1.7.

VISUALIZING SCATTER DISTRIBUTION

VISUALIZING SCATTER DISTRIBUTION

Step 1 Plot the scatter distribution of the "Open" versus "Volume" variables, with the points colored according to the "Year" variable in the **df_dummy DataFrame**. This code creates a figure with a specified size using the **subplots()** function from **matplotlib**. It then plots a scatter plot using the **scatterplot()** function from **seaborn**. The **x** parameter is set to "Open" and the **y** parameter is set to "Volume" to define the variables for the x-axis and y-axis, respectively. The **hue** parameter is set to "Year" to color the points based on the year. The **palette** parameter is set to "deep" to use a deep color palette for the different years. The resulting scatter plot is displayed on the figure.

```
1 #Plots the scatter distribution of Open versus Volume versus
2 Year
3 fig, ax1 = plt.subplots(figsize=(15,10))
4 sns.scatterplot(data=df_dummy, x="Open", y="Volume",
hue="Year", palette="deep", ax=ax1)
```

Let's break down the code step by step:

1. Step 1: **fig, ax1 = plt.subplots(figsize=(15,10))**

This line of code creates a new figure and axes object using the **subplots()** function from **matplotlib**. The

figsize parameter sets the size of the figure in inches, with a width of 15 and height of 10. The **subplots()** function returns both the figure (**fig**) and the axes object (**ax1**).

2. Step 2: `sns.scatterplot(data=df_dummy, x="Open", y="Volume", hue="Year", palette="deep", ax=ax1)`

This line of code plots a scatter plot using the **scatterplot()** function from seaborn (**sns**). The data parameter specifies the DataFrame (**df_dummy**) from which the data will be plotted. The **x** parameter is set to "Open", which represents the values for the x-axis. The **y** parameter is set to "Volume", representing the values for the y-axis. The **hue** parameter is set to "Year", indicating that the points will be colored based on the values of the "Year" column. The **palette** parameter is set to "deep" to use a deep color palette for the different years. Finally, the **ax** parameter is set to **ax1**, which represents the axes object where the scatter plot will be plotted.

3. Step 3: The resulting scatter plot is displayed on the figure.

In summary, the code generates a scatter plot that visualizes the relationship between the "Open" and "Volume" variables, with the points colored based on the "Year" variable. This allows for the exploration of any potential patterns or trends between these variables. The result is shown in Figure 1.8.

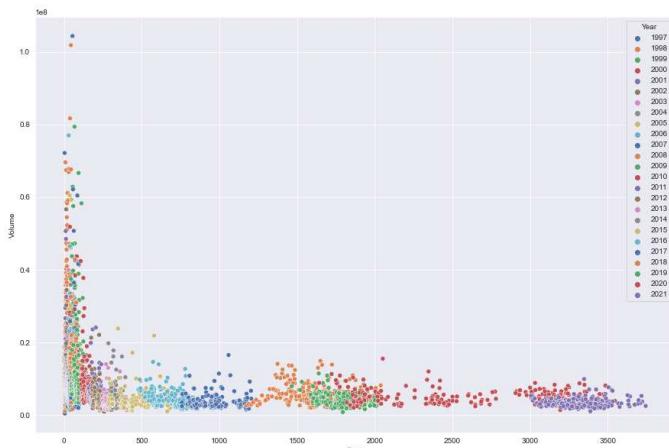


Figure 1.8 The scatter distribution of Open versus Volume versus Year

Step 2 Plot the scatter distribution of **Low** versus **Volume** versus **Month**.

```
1 #Plots the scatter distribution of Low versus Volume versus
```

```
2     Month
3     fig, ax1 = plt.subplots(figsize=(15,10))
4     sns.scatterplot(data=df_dummy, x="Low", y="Month",
hue="Month", palette="deep", ax=ax1)
```

Here's a step-by-step explanation of the code you provided:

1. **fig, ax1 = plt.subplots(figsize=(15,10))**: The first line of code creates a figure and axes object for plotting using the subplots function from the matplotlib.pyplot module. It sets the figure size to 15 units in width and 10 units in height. The returned objects are assigned to **fig** and **ax1** variables.
2. **sns.scatterplot(data=df_dummy, x="Low", y="Month", hue="Month", palette="deep", ax=ax1)**: The second line of code uses the scatterplot() function from the seaborn library to create a scatter plot. The data parameter specifies the **data frame (df_dummy)** that contains the data to be plotted. The **x** parameter is set to "Low" and represents the variable that will be plotted on the x-axis. The **y** parameter is set to "Month" and represents the variable that will be plotted on the y-axis. The **hue** parameter is set to "Month", which means that the color of each data point in the scatter plot will be determined by the corresponding value in the "Month" column of the data frame. This will allow you to distinguish different months visually. The **palette** parameter is set to "deep" to specify the color palette that will be used for the different months. The "deep" palette provides a set of distinct colors suitable for categorical data. The **ax** parameter is set to **ax1** to specify the axes object on which the scatter plot will be drawn. This is necessary because we have created the axes object **ax1** earlier using subplots.

Overall, this code will create a scatter plot with "Low" values on the x-axis, "Month" values on the y-axis, and different colors representing different months. The size of the figure will be 15 units in width and 10 units in height. The result is shown in Figure 1.9.

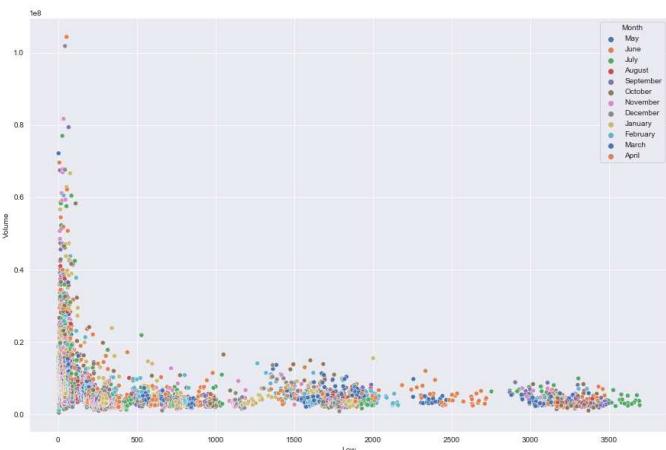


Figure 1.9 The scatter distribution of Low versus Volume versus Month

- Step 3 Plot the scatter distribution of **Adj Close** versus **Volume** versus **Quarter**. Here's a step-by-step explanation of the code:
1. The first line of code creates a figure and **axes** object for plotting using the subplots function from the matplotlib.pyplot module. It sets the figure size to 15 units in width and 10 units in height. The returned objects are assigned to **fig** and **ax1** variables.
 2. The second line of code uses the **scatterplot()** function from the seaborn library to create a scatter plot. The **data** parameter specifies the data frame (**df_dummy**) that contains the data to be plotted. The **x** parameter is set to "Adj Close" and represents the variable that will be plotted on the x-axis. The **y** parameter is set to "Volume" and represents the variable that will be plotted on the y-axis.
 3. The **hue** parameter is set to "Quarter", which means that the color of each data point in the scatter plot will be determined by the corresponding value in the "Quarter" column of the data frame. This will allow you to distinguish different quarters visually.
 4. The **palette** parameter is set to "deep" to specify the color palette that will be used for the different quarters. The "deep" palette provides a set of distinct colors suitable for categorical data.
 5. The **ax** parameter is set to **ax1** to specify the axes object on which the scatter plot will be drawn. This is necessary because we have created the axes object **ax1** earlier using subplots.

Overall, this code will create a scatter plot with "Adj Close" values on the x-axis, "Volume" values on the y-axis, and

different colors representing different quarters. The size of the figure will be 15 units in width and 10 units in height.

```
1 #Plots the scatter distribution of Adj Close versus Volume  
2 versus Quarter  
3 fig, ax1 = plt.subplots(figsize=(15,10))  
4 sns.scatterplot(data=df_dummy, x="Adj Close", y="Volume",  
hue="Quarter", palette="deep", ax=ax1)
```

The result is shown in Figure 1.10.

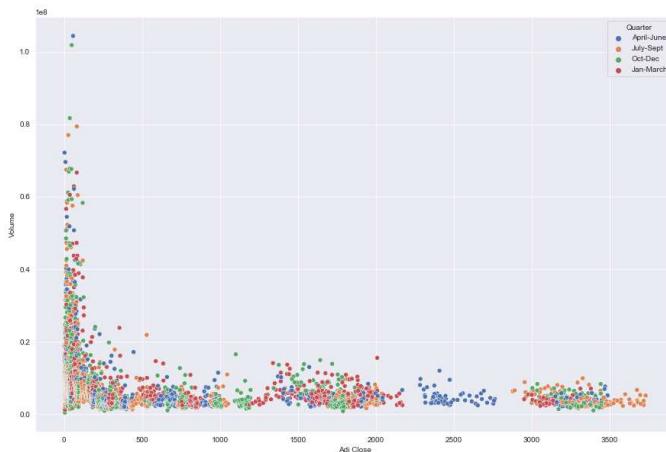


Figure 1.10 The scatter distribution of **Adj Close** versus **Volume** versus **Month**

Step 4 Plot the scatter distribution of **High** versus **Volume** versus **Day**:

```
1 #Plots the scatter distribution of high versus volume versus  
2 Day  
3 fig, ax1 = plt.subplots(figsize=(15,10))  
4 sns.scatterplot(data=df_dummy, x="High", y="Volume",  
hue="Day", palette="deep", ax=ax1)
```

The result is shown in Figure 1.11.

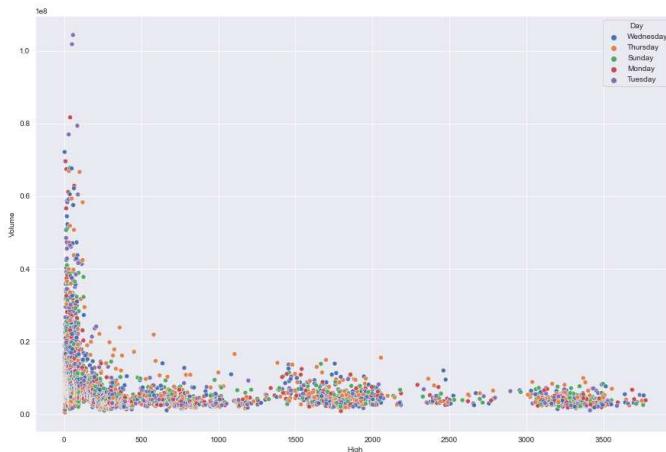


Figure 1.11 The scatter distribution of High versus Volume versus Day

VISUALIZING GROUPED DATAFRAME

VISUALIZING GROUPED DATAFRAME

Step 1	<p>Define plot_group() to plot distribution of a grouped dataframe. It is used to plot the distribution of a grouped dataframe. Here's a step-by-step explanation of the code:</p> <ol style="list-style-type: none">1. The function is defined with the syntax def plot_group(df, title=""): It takes two parameters: df, which represents the grouped dataframe to be plotted, and title, which is an optional parameter to specify the title of the plot. The default value for title is an empty string.2. The plt.figure(figsize=(20,10)) code creates a new figure with a specified size of 20 units in width and 10 units in height. This sets the overall size of the plot.3. The plt.subplot(121) code creates a subplot grid with 1 row and 2 columns, and sets the current subplot to the first position (i.e., the left subplot).4. The label_list = list(df.index) code extracts the index values of the dataframe and assigns them to the label_list variable. These values will be used as labels for the pie chart.5. The df.plot.pie(...) code creates a pie chart from the grouped dataframe (df). The autopct parameter formats the percentages displayed on the pie chart, colors specifies the color palette to use, startangle sets the angle at which the first wedge starts, labels assigns the label_list as the labels for each wedge, wedgeprops customizes the appearance of the wedges, shadow adds a shadow effect to the pie chart, and textprops specifies the font size for the labels on the pie chart.6. The plt.title(title, fontsize=25) code adds a title to the left subplot using the specified title parameter and sets the font size to 25.7. The plt.subplot(122) code creates the second subplot on the right side of the plot.8. The ax = df.plot(kind="barh", color={"salmon"}) code creates a horizontal bar chart from the grouped dataframe (df). The kind parameter is set to "barh" to create a horizontal bar chart, and the color parameter sets the color of the bars to "salmon".
--------	---

9. The **for i, j in enumerate(df.values)**: loop iterates over the values of the grouped dataframe. The **i** variable represents the index of the value, and **j** represents the value itself.
10. The **ax.text(.7, i, j, weight="bold", fontsize=15)** code adds the text label for each bar at a specified position (.7 on the x-axis, i on the y-axis). The **weight** parameter sets the font weight to "bold", and the **fontsize** parameter sets the font size to 15.
11. The **plt.title(title, fontsize=25)** code adds a title to the right subplot using the specified title parameter and sets the font size to 25.
12. The **plt.show()** code displays the plot.

Overall, this function creates a figure with two subplots: a pie chart on the left representing the distribution of the grouped dataframe, and a horizontal bar chart on the right showing the values of the grouped dataframe. The title parameter can be used to specify a title for the plot.

```

1 #Defines function to plot distribution of a grouped
2 #dataframe
3 def plot_group(df, title=""):
4     plt.figure(figsize=(20,10))
5     plt.subplot(121)
6     label_list = list(df.index)
7
8     df.plot.pie(autopct = "%1.1f%%", \
9                 colors = sns.color_palette("prism",7), \
10                startangle = 60,labels=label_list,\ 
11                wedgeprops={"linewidth":2,"edgecolor": "k"},\ 
12                shadow =True, textprops={'fontsize': 16})
13     plt.title(title, fontsize=25)
14
15     plt.subplot(122)
16     ax = df.plot(kind="barh",color={"salmon"})
17     for i,j in enumerate(df.values):
18         ax.text(.7,i,j, weight = "bold", fontsize=15)
19     plt.title(title, fontsize=25)
20
21     plt.show()

```

Step 2 Plot the distribution of the total volume by year. Here's how it works:

1. **df_dummy.groupby('Year')['Volume'].sum()** groups the dataframe **df_dummy** by the 'Year' column and calculates the sum of the 'Volume' column within each year. This creates a new dataframe that shows the total volume for each year.
2. The resulting grouped dataframe is passed as the first argument to the **plot_group()** function: **plot_group(df_dummy.groupby('Year') ['Volume'].sum(), "The distribution of Volume by Year")**. This will be the **df** parameter of the function.

3. The second argument passed to **plot_group()** is the title for the plot: "**The distribution of Volume by Year**". This will be the title parameter of the function.
4. Inside the **plot_group()** function, a figure is created using `plt.figure(figsize=(20,10))` with a size of 20 units in width and 10 units in height.
5. The function then creates a subplot grid with 1 row and 2 columns using **plt.subplot(121)**, indicating that the left subplot will be used.
6. A pie chart is created using **df.plot.pie(...)** with the grouped dataframe as the data. The pie chart displays the distribution of the total volume by year. The **autopct** parameter formats the percentages displayed on the pie chart, **colors** specifies the color palette to use, **startangle** sets the angle at which the first wedge starts, **labels** assigns the index values of the grouped dataframe as the labels for each wedge, **wedgeprops** customizes the appearance of the wedges, **shadow** adds a shadow effect to the pie chart, and **textprops** specifies the font size for the labels on the pie chart.
7. The **plt.title(title, fontsize=25)** code adds the specified title to the left subplot.
8. The function then creates the right subplot using `plt.subplot(122)`.
9. A horizontal bar chart is created using **df.plot(kind="barh", color={"salmon"})** with the grouped dataframe as the data. The kind parameter is set to "barh" to create a horizontal bar chart, and the color parameter sets the color of the bars to "salmon".
10. Text labels are added to each bar using `ax.text(.7, i, j, weight="bold", fontsize=15)`, where *i* represents the index of the bar and *j* represents the value. The labels are displayed on the right side of each bar, and the font weight and size are customized.
11. The **plt.title(title, fontsize=25)** code adds the specified title to the right subplot.
12. Finally, **plt.show()** is called to display the plot.

In summary, the code uses the **plot_group()** function to plot the distribution of the total volume by year. The function creates a figure with two subplots: a pie chart on the left showing the distribution of volume by year, and a horizontal bar chart on the right displaying the total volume for each year. The title of the plot is "The distribution of Volume by Year".

```

1 #Plots which year have most volumes
2 plot_group(df_dummy.groupby('Year')[['Volume']].sum(), "The
3 distribution of Volume by Year")

```

The result is shown in Figure 1.12.

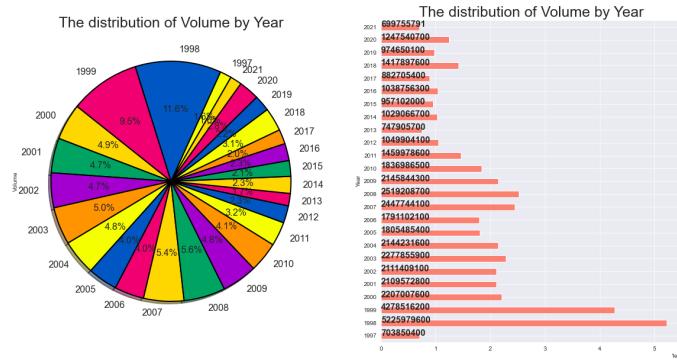


Figure 1.12 Showing which year have most volumes

Step 3 In the same way, plot the distribution of the total volume by day.

```
1 #Plots which days of the week have most volumes
2 plot_group(df_dummy.groupby('Day')['Volume'].sum(), "The
3 distribution of Volume by Day")
```

The result is shown in Figure 1.13.

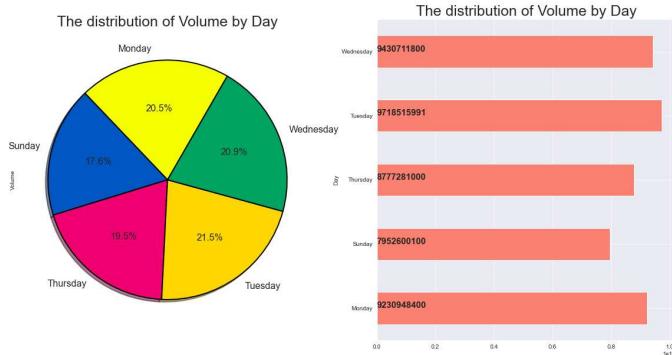


Figure 1.13 Showing which days of the week have most volumes

Step 4 In the same way, plot the distribution of the total volume by month.

```
1 #Plots which month have most volumes
2 plot_group(df_dummy.groupby('Month')['Volume'].sum(), "The
3 distribution of volume by Month")
```

The result is shown in Figure 1.14.

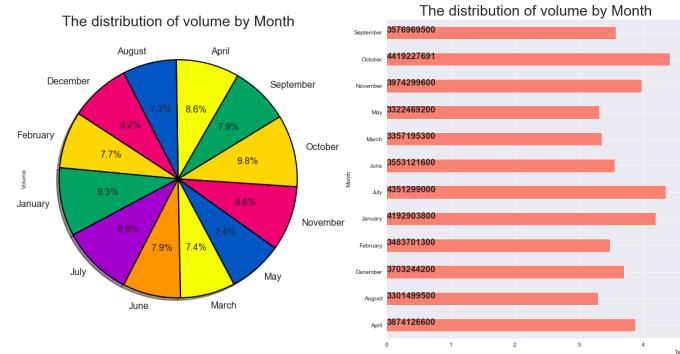


Figure 1.14 Showing which month have most volumes

Step 4 In the same way, plot the distribution of the total volume by quarter.

```
1 #Plots which quarter have most volumes
2 plot_group(df_dummy.groupby('Quarter')['Volume'].sum(), "The
3 distribution of volume by Quarter")
```

The result is shown in Figure 1.15.

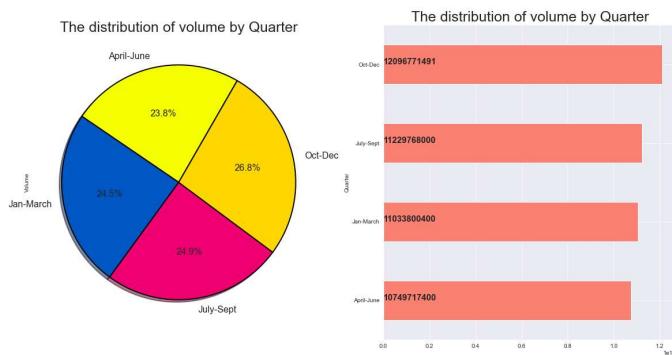


Figure 1.15 Showing which quarter have most volumes

STATISTICAL DESCRIPTION AND CATEGORIZED FEATURES HISTOGRAM

STATISTICAL DESCRIPTION AND CATEGORIZED FEATURES HISTOGRAM

Step 1 Print statistical description of dataset. The code prints the statistical description of the dataframe **df** using the **describe()** function. Here's how it works:

1. **df.describe()** computes the summary statistics of the dataframe **df**. It calculates various statistics such as count, mean, standard deviation, minimum, quartiles, and maximum for each column of the dataframe.
2. **.T** is used to transpose the resulting summary statistics dataframe, swapping the rows and

columns. This is done to make the output easier to read.

3. `.to_string()` converts the transposed summary statistics dataframe into a string representation.
4. Finally, `print()` is used to display the string representation of the transposed summary statistics dataframe.

The `describe()` function provides a quick overview of the central tendency, dispersion, and shape of the distribution of a dataset. The resulting output will show the statistical summary for each column of the dataframe, including the count (number of non-null values), mean, standard deviation, minimum value, quartiles (25%, 50%, and 75%), and maximum value.

By using `.T` and `.to_string()`, the output is formatted in a tabular structure where the columns represent the statistical measures, and each row corresponds to a specific column in the original dataframe.

Overall, this code prints a statistical description of the dataframe `df` in a readable tabular format.

```
1 #Prints statistical description
2 print(df.describe().T.to_string())
```

Output:

Step 2 Define `hist_dist()` method to plot histogram distribution of a feature. It plots the histogram distribution of a feature in a dataframe. Here's a step-by-step explanation of the code:

1. The function `hist_dist()` takes several parameters: `df` represents the dataframe, `feat` represents the feature to be plotted, `num_bin` specifies the number of bins for the histogram, lower and upper represent the lower and upper bounds for highlighting specific bars, and `ax1` is the axes object on which the histogram will be plotted.
2. The `plt.subplots_adjust(wspace=0.25, hspace=0.25)` code adjusts the spacing between subplots horizontally (`wspace`) and vertically (`hspace`). This is done to create some space between the subplots if multiple plots are created.
3. The `sns.histplot(...)` code creates a histogram using the `histplot()` function from the `seaborn` library. The `data` parameter specifies the dataframe (`df`), and the `x` parameter specifies the feature (`feat`) to be plotted. The `ax` parameter is set to `ax1` to specify the axes object on which the histogram will be drawn.

4. The **kde=True** parameter adds a kernel density estimate curve to the histogram, providing a smooth representation of the underlying distribution.
5. The **bins=num_bin** parameter sets the number of bins for the histogram, determining the granularity of the distribution representation.
6. The **line_kws={'lw': 5}** parameter sets the line width of the histogram bars to 5, making them visually prominent.
7. The **ax1.set_title(...)** code adds a title to the histogram plot, combining the string "**Histogram Distribution of**" with the specified feature (**feat**).
8. The **ax1.set_xlabel(...)** and **ax1.set_ylabel(...)** codes set the x-axis and y-axis labels of the histogram plot, respectively.
9. The **for p in ax1.patches:** loop iterates over each rectangle patch representing a histogram bar.
10. The **color = '#75bbfd' if (lower<=p.get_height() <=upper) else '#ff796c'** line sets the color of the histogram bar based on whether its height falls within the specified lower and upper bounds. If it does, the color is set to '#75bbfd', otherwise, it is set to '#ff796c'.
11. The **p.set_facecolor(color)** code sets the face color of the rectangle patch to the specified color.
12. The **ax1.annotate(...)** code adds a text annotation to each histogram bar. It displays the height of the bar using **format(p.get_height(), '.0f')**, positions the text at the center of the bar, and adds an offset of 10 points above the bar. The font weight, size, and color are customized.

Overall, this function creates a histogram plot with a kernel density estimate curve for a specified feature in a dataframe. The histogram bars are colored differently based on whether their heights fall within a specified range. The function also adds a title, x-axis and y-axis labels, and text annotations to the histogram bars.

```

1 #Plots histogram distribution of a feature
2 def hist_dist(df, feat, num_bin,lower,upper,ax1):
3     plt.subplots_adjust(wspace=0.25,hspace=0.25)
4     sns.histplot(data = df, x = feat, \
5                 ax=ax1, kde = True, bins=num_bin,line_kws={'lw': 5})
6     ax1.set_title('Histogram Distribution of ' + feat, \
7                   fontsize=25)
8     ax1.set_xlabel(feat, fontsize=20)
9     ax1.set_ylabel('Count', fontsize=20)
10    for p in ax1.patches:
11        color = '#75bbfd' if (lower<=p.get_height()<=upper)
12        else '#ff796c'
13        p.set_facecolor(color)
14        ax1.annotate(format(p.get_height(), '.0f'), \
15                     (p.get_x() + p.get_width() / 2.,
16                      p.get_height()), \

```

```
17 \ha = 'center', va = 'center', xytext = (0, 10),  
weight = "bold", fontsize=20, \  
textcoords = 'offset points'
```

Step 3 Define two new functions: **put_label_stacked_bar()** and **dist_one_vs_another_plot()**. Here's an explanation of each function:

1. **put_label_stacked_bar(ax, fontsize):** This function adds labels inside each stacked bar of a bar plot. Here's how it works:
 - The function takes two parameters: **ax**, which is the axes object representing the bar plot, and **fontsize**, which specifies the font size of the labels.
 - The loop **for rect in ax.patches:** iterates over each rectangle patch in the bar plot.
 - Inside the loop, the function retrieves information about each bar, such as its height, width, x-coordinate, and y-coordinate.
 - The height of the bar is used as the label for the bar, and `f'{height:.0f}'` formats the height as a string with no decimal places.
 - The **label_x** and **label_y** variables calculate the coordinates where the label will be placed, which is at the center of each bar.
 - The **if height > 0:** condition ensures that labels are only added to bars with a positive height.
 - Finally, the **ax.text(...)** code adds the label text at the specified coordinates (**label_x** and **label_y**) using the specified parameters such as horizontal alignment (**ha='center'**), vertical alignment (**va='center'**), font weight (**weight='bold'**), and font size (**fontsize**).

2. **dist_one_vs_another_plot(df, cat1, cat2, ax1, title=""):** This function plots the distribution of one variable (**cat1**) against another variable (**cat2**) using a stacked bar plot. Here's how it works:

- The function takes several parameters: **df** represents the dataframe, **cat1** represents the variable to be plotted on the x-axis, **cat2** represents the variable to be stacked on the y-axis, **ax1** is the axes object on which the stacked bar plot will be plotted, and **title** is an optional parameter for the plot title.
- The **group_by_stat = df.groupby([cat1, cat2]).size()** line groups the dataframe **df** by

both **cat1** and **cat2** and calculates the number of cases within each group.

- The **group_by_stat.unstack().plot(kind='bar', stacked=True, ax=ax1, grid=True)** code creates a stacked bar plot using the **plot** function. The **unstack()** method reshapes the grouped data to have **cat1** as the x-axis and each unique value of **cat2** as a stacked bar. The bars are stacked (**stacked=True**), and the plot is drawn on the specified **ax1** axes object. The **grid=True** parameter adds gridlines to the plot.
- The **ax1.set_title(...), ax1.set_ylabel(...), and ax1.set_xlabel(...)** codes set the plot title, y-axis label, and x-axis label, respectively.
- The **put_label_stacked_bar(ax1, 17)** code calls the **put_label_stacked_bar()** function to add labels inside each stacked bar with a font size of 17.
- Finally, **plt.show()** is called to display the stacked bar plot.

Overall, the **put_label_stacked_bar()** function adds labels inside stacked bars of a bar plot, while the **dist_one_vs_another_plot()** function creates a stacked bar plot showing the distribution of one variable against another variable.

```
1 #Puts label inside stacked bar
2 def put_label_stacked_bar(ax,fontsize):
3     #patches is everything inside of the chart
4     for rect in ax.patches:
5         # Find where everything is located
6         height = rect.get_height()
7         width = rect.get_width()
8         x = rect.get_x()
9         y = rect.get_y()
10
11        # The height of the bar is the data value and can be
12        used as the label
13        label_text = f'{height:.0f}'
14
15        # ax.text(x, y, text)
16        label_x = x + width / 2
17        label_y = y + height / 2
18
19        # plots only when height is greater than specified
20        value
21        if height > 0:
22            ax.text(label_x, label_y, label_text, \
23                    ha='center', va='center', \
24                    weight = "bold", fontsize=fontsize)
25
26
27 #Plots the distribution of one variable against another
28 #variable
29 def dist_one_vs_another_plot(df,cat1, cat2, ax1,title=""):
30     cmap=plt.cm.Blues
31     cmap1=plt.cm.coolwarm_r
```

```

33 group_by_stat = df.groupby([cat1, cat2]).size()
34 group_by_stat.unstack().plot(kind='bar', \
35     stacked=True,ax=ax1,grid=True)
36 ax1.set_title('Stacked Bar Plot of '+ \
37     cat1 + ' (number of cases) '+title, fontsize=20)
38 ax1.set_ylabel('Number of Cases', fontsize=20)
39 ax1.set_xlabel(cat1, fontsize=20)
40 put_label_stacked_bar(ax1,17)
41 plt.show()

```

Step 4	<p>Categorize High, Low, Open, Close, Adj Close, and Volume features. The code categorizes multiple features in the dataframe df_dummy by dividing them into predefined ranges. Here's an explanation of each step:</p> <ol style="list-style-type: none"> 1. Categorizing the "High" feature: <ul style="list-style-type: none"> The pd.cut() function is used to create categorical bins for the "High" feature of the dataframe df_dummy. The feature values are divided into specific ranges using the specified bins: [0, 200, 500, 750, 1000, 3000]. The labels parameter assigns labels to each bin range: ['0-200', '200-500', '500-750', '750-1000', '1000-3000']. The resulting categorical values are stored in a new column called "Cat_High" in df_dummy. 2. Categorizing the "Low" feature: <ul style="list-style-type: none"> Similar to the previous step, the "Low" feature is categorized using the same bins and labels. The resulting categorical values are stored in a new column called "Cat_Low" in df_dummy. 3. Categorizing the "Open" feature: <ul style="list-style-type: none"> The "Open" feature is categorized using the same bins and labels as the previous steps. The resulting categorical values are stored in a new column called "Cat_Open" in df_dummy. 4. Categorizing the "Close" feature: <ul style="list-style-type: none"> The "Close" feature is categorized using the same bins and labels as the previous steps. The resulting categorical values are stored in a new column called "Cat_Close" in df_dummy. 5. Categorizing the "Adj Close" feature:
--------	---

- The "Adj Close" feature is categorized using the same bins and labels as the previous steps.
- The resulting categorical values are stored in a new column called "Cat_Adj_Close" in **df_dummy**.

6. Categorizing the "Volume" feature:

- The "Volume" feature is categorized using custom bins: [400000, 2000000, 10000000, 50000000, 100000000, 200000000].
- The labels parameter assigns labels to each bin range: ['400k-2M', '2M-10M', '10M-50M', '50M-100M', '100M-200M'].
- The resulting categorical values are stored in a new column called "Cat_Volume" in **df_dummy**.

Overall, this code applies binning or categorization to several numerical features in the **df_dummy** dataframe and creates new columns to store the corresponding categorical values. The specific bin ranges and labels are customized for each feature.

```

1 #Categorizes High feature
2 labels = ['0-200', '200-500', '500-750', '750-1000', '1000-
3 3000']
4 df_dummy["Cat_High"] = pd.cut(df_dummy["High"], \
5     [0, 200, 500, 750, 1000, 3000], labels=labels)
6
7 #Categorizes Low feature
8 labels = ['0-200', '200-500', '500-750', '750-1000', '1000-
9 3000']
10 df_dummy["Cat_Low"] = pd.cut(df_dummy["Low"], \
11     [0, 200, 500, 750, 1000, 3000], labels=labels)
12
13 #Categorizes Open feature
14 labels = ['0-200', '200-500', '500-750', '750-1000', '1000-
15 3000']
16 df_dummy["Cat_Open"] = pd.cut(df_dummy["Open"], \
17     [0, 200, 500, 750, 1000, 3000], labels=labels)
18
19 #Categorizes Close feature
20 labels = ['0-200', '200-500', '500-750', '750-1000', '1000-
21 3000']
22 df_dummy["Cat_Close"] = pd.cut(df_dummy["Close"], \
23     [0, 200, 500, 750, 1000, 3000], labels=labels)
24
25 #Categorizes Adj Close feature
26 labels = ['0-200', '200-500', '500-750', '750-1000', '1000-
27 3000']
28 df_dummy["Cat_Adj_Close"] = pd.cut(df_dummy["Adj Close"], \
29     [0, 200, 500, 750, 1000, 3000], labels=labels)
30
31 #Categorizes Volume feature
32 labels = ['400k-2M', '2M-10M', '10M-50M', '50M-100M', '100M-
33 200M']
34 df_dummy["Cat_Volume"] = pd.cut(df_dummy["Volume"], \
35     [400000, 2000000, 10000000, 50000000, 100000000,
36 200000000], labels=labels)

```

Step 5	<p>Plot histogram distribution of High feature and case distribution of categorized High variable against categorized Volume variable in stacked bar plots.</p> <p>Here's an explanation of each step:</p> <ol style="list-style-type: none"> 1. Creating the figure and subplots: <ul style="list-style-type: none"> The <code>plt.subplots(2, figsize=(25, 20), facecolor='#fbe7dd')</code> code creates a figure with two subplots arranged in a 2x1 grid. The 2 parameter specifies the number of rows (2) in the grid. The <code>figsize=(25, 20)</code> parameter sets the size of the figure to (25, 20) inches. The <code>facecolor='#fbe7dd'</code> parameter sets the background color of the figure to a light shade. 2. Plotting the histogram of the "High" feature: <ul style="list-style-type: none"> The <code>hist_dist(df, 'High', 50, 400, 750, axs[0])</code> code calls the <code>hist_dist()</code> function to plot a histogram. The histogram is based on the "High" feature of the dataframe <code>df</code>. The histogram is created with 50 bins (<code>num_bin=50</code>). The range of the x-axis is limited to the interval [400, 750] (lower=400, upper=750). The histogram is plotted on the first subplot (<code>axs[0]</code>). 3. Plotting the stacked bar plot: <ul style="list-style-type: none"> The <code>dist_one_vs_another_plot(df_dummy, 'Cat_Volume', 'Cat_High', axs[1])</code> code calls the <code>dist_one_vs_another_plot()</code> function to plot a stacked bar plot. The stacked bar plot shows the distribution of the "Cat_High" variable against the "Cat_Volume" variable. The plot is created using the <code>df_dummy</code> dataframe. The stacked bars are based on the counts of each combination of categories in the two variables. The stacked bar plot is plotted on the second subplot (<code>axs[1]</code>). <p>Overall, the code generates a figure with two subplots. The first subplot shows a histogram of the "High" feature, while the second subplot displays a stacked bar plot of the categorized "Cat_High" variable against the categorized "Cat_Volume" variable.</p>
--------	--

```

1 fig, axs = plt.subplots(2,figsize=
2 (25,20),facecolor='#fbe7dd')
3
4 #Plots histogram distribution of High feature
5 hist_dist(df, 'High', 50, 400, 750, axs[0])
6
7 #Plots case distribution of categorized High variable
8 against categorized Volume variable in stacked bar plots
9 dist_one_vs_another_plot(df_dummy, 'Cat_Volume', 'Cat_High'
\ ,axs[1]

```

The result is shown in Figure 1.16.

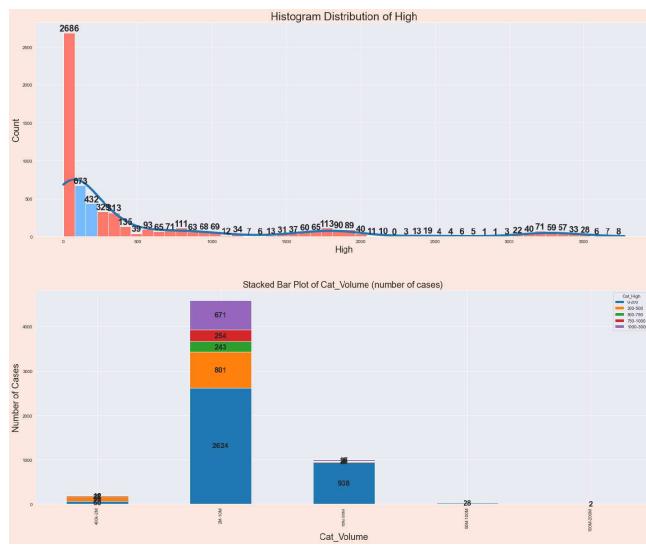


Figure 1.16 Showing the histogram distribution of High feature and case distribution of categorized High variable against categorized Volume variable

Step 6 In the same way, plot histogram distribution of **Adj Close** feature and case distribution of categorized **Adj Close** variable against categorized **Volume** variable in stacked bar plots.

```

1 fig, axs = plt.subplots(2,figsize=
2 (25,20),facecolor='#fbe7dd')
3
4 #Plots histogram distribution of Adj Close feature
5 hist_dist(df, 'Adj Close', 50, 400, 750, axs[0])
6
7 #Plots case distribution of categorized Adj Close variable
8 against categorized Volume variable in stacked bar plots
9 dist_one_vs_another_plot(df_dummy, 'Cat_Volume', \
'Cat_Adj_Close',axs[1])

```

The result is shown in Figure 1.17.

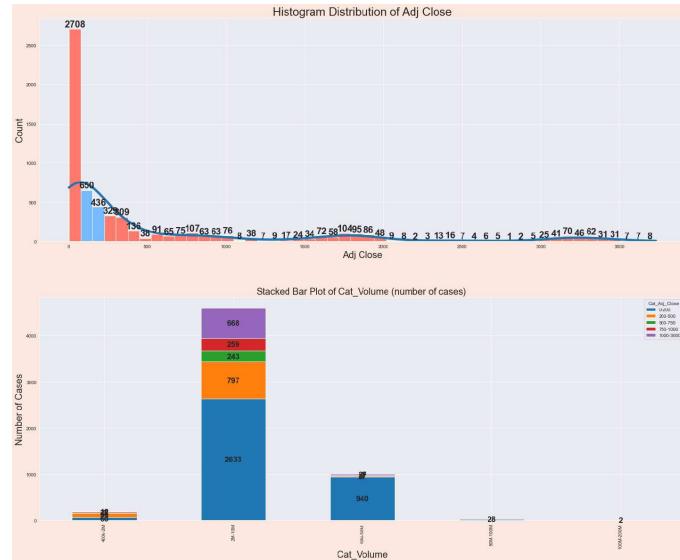


Figure 1.17 Showing the histogram distribution of Adj Close feature and case distribution of categorized Adj Close variable against categorized Volume variable

ANALYZING YEAR-WISE DATA

ANALYZING YEAR-WISE DATA

Step 1	<p>Plots Open and Close features over year 2007. Here's an explanation of each step:</p> <ol style="list-style-type: none"> Creating the figure and setting properties: <ul style="list-style-type: none"> The <code>plt.subplots(figsize=(20, 10), facecolor='fbe7dd')</code> code creates a figure with a specific size and background color. The <code>figsize=(20, 10)</code> parameter sets the size of the figure to (20, 10) inches. The <code>facecolor='fbe7dd'</code> parameter sets the background color of the figure to a light shade. Setting the x-axis label and title: <ul style="list-style-type: none"> The <code>plt.xlabel('Date', fontsize=15)</code> code sets the label for the x-axis to "Date" with a font size of 15. The <code>plt.title("The Open and Close over year 2007", fontsize=20)</code> code sets the title of the plot to "The Open and Close over year 2007" with a font size of 20.
--------	---

3. Plotting the lines for the "Open" and "Close" features:

- The `sns.lineplot(data=df_dummy[df_dummy["Year"]==2007][["Open"]], color='red', marker='d', linewidth=5)` code plots a line for the "Open" feature.
- The `df_dummy[df_dummy["Year"]==2007][["Open"]]` selects the values of the "Open" feature from the dataframe `df_dummy` for the year 2007.
- The line is colored in red (`color='red'`), marked with diamond markers (`marker='d'`), and has a linewidth of 5.
- The `sns.lineplot(data=df_dummy[df_dummy["Year"]==2007][["Close"]], color='blue', marker='d', linewidth=5)` code plots a line for the "Close" feature.
- The `df_dummy[df_dummy["Year"]==2007][["Close"]]` selects the values of the "Close" feature from the dataframe `df_dummy` for the year 2007.
- The line is colored in blue (`color='blue'`), marked with diamond markers (`marker='d'`), and has a **linewidth** of 5.

4. Adding a legend and displaying the plot:

- The `plt.legend(["Open", "Close"], fontsize=20)` code adds a legend to the plot, labeling the lines as "Open" and "Close" with a font size of 20.
- The `plt.show()` code displays the plot.

Overall, the code generates a line plot that shows the "Open" and "Close" features for the year 2007. The lines are distinguished by color and markers, and a legend is added to identify the lines.

```
1 #Plots Open and Close feature over year 2007
2 plt.subplots(figsize=(20,10), facecolor="#fbe7dd")
3 plt.xlabel('Date', fontsize=15)
4 plt.title("The Open and Close over year 2007", fontsize=20)
5 sns.lineplot(data=df_dummy[df_dummy["Year"]==2007][["Open"]],
6 color='red', marker='d', linewidth=5)
7 sns.lineplot(data=df_dummy[df_dummy["Year"]==2007][["Close"]],
8 color='blue', marker='d', linewidth=5)
9 plt.legend(["Open", "Close"], fontsize=20)
10 plt.show()
```

The result is shown in Figure 1.18.



Figure 1.18 The lineplot **Open** and **Close** features over year 2007

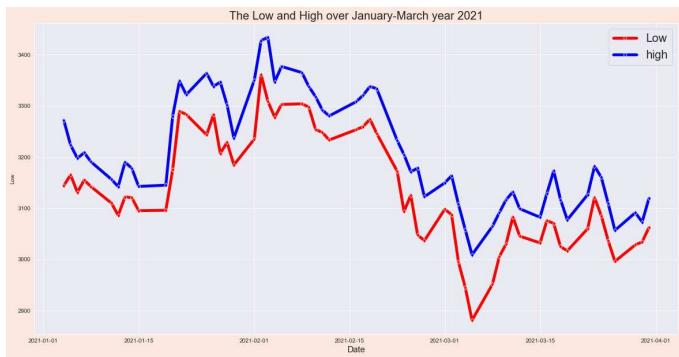


Figure 1.19 The Low and High features over Jan-March year 2021

Step 2 In the similar way, plot **Low** and **High** features over Jan-March year 2021:

```

1 #Plots Low and High features over Jan-March year 2021
2 plt.subplots(figsize=(20,10),facecolor='#fbe7dd')
3 plt.xlabel('Date', fontsize=15)
4 plt.title("The Low and High over January-March year 2021",
5           fontsize=20)
6 sns.lineplot(data=df_dummy[(df_dummy["Year"]==2021)&
7 (df_dummy["Quarter"]=="Jan-March")]["Low"], color='red',
8 marker='d', linewidth=5)
9 sns.lineplot(data=df_dummy[(df_dummy["Year"]==2021)&
10 (df_dummy["Quarter"]=="Jan-March")]["High"], color='blue',
11 marker='d', linewidth=5)
12 plt.legend(["Low", "high"], fontsize=20)
13 plt.show()

```

The result is shown in Figure 1.19.

Step 3 Plot the average "Adj Close" by year using both the mean and exponential weighted moving average (EWMA). Here's an explanation of each step:

1. Creating the figure and setting properties:
 - The `fig=plt.figure(figsize=(20,10))` code creates a figure with a specific size.
 - The `plt.rcParams.update({'figure.dpi':120})`

code updates the dpi (dots per inch) setting of the figure to control the image resolution.

2. Computing the average "Adj Close" by year:

- The `year_data = df_dummy.resample('y').mean()` code uses the resample method to aggregate the data by year and calculates the mean of each year.
- The result is stored in the `year_data` dataframe.

3. Plotting the lines for mean and EWMA:

- The `year_data["Adj Close"].plot(linewidth=5)` code plots a line for the mean of the "Adj Close" feature.
- The line represents the average "Adj Close" value for each year.
- The line has a linewidth of 5.
- The `year_data_ewm=year_data["Adj Close"].ewm(span=5).mean()` code computes the exponential weighted moving average (EWMA) of the "Adj Close" feature with a span of 5.
- The EWMA represents a smoothed version of the data that gives more weight to recent observations.
- The EWMA values are stored in the `year_data_ewm` series.
- The `year_data_ewm.plot(linewidth=5)` code plots a line for the EWMA of the "Adj Close" feature.
- The line represents the smoothed version of the average "Adj Close" value for each year.
- The line has a linewidth of 5.

4. Adding a title, x-axis label, and legend:

- The `plt.title(' Average Adj Close by year', fontsize=30)` code sets the title of the plot to "Average Adj Close by year" with a font size of 30.
- The `plt.xlabel('Year', fontsize=20)` code sets the label for the x-axis to "Year" with a font size of 20.
- The `plt.legend(["Mean", "EWM"], fontsize=20)` code adds a legend to the plot, labeling the lines as "Mean" and "EWM" with a font size of 20.

Overall, the code generates a plot that shows the average "Adj Close" by year using both the mean and EWMA. The lines represent the mean and smoothed values of the average "Adj Close" for each year, providing insights into the overall trend and the smoothness of the data.

```

1 #Plots average Adj Close by year with mean and ewm
2 fig=plt.figure(figsize=(20,10))
3 plt.rcParams.update({'figure.dpi':120})
4 year_data = df_dummy.resample('y').mean()
5 year_data["Adj Close"].plot(linewidth=5)
6 year_data_ewm=year_data["Adj Close"].ewm(span=5).mean()
7 year_data_ewm.plot(linewidth=5)
8 plt.title(' Average Adj Close by year', fontsize=30)
9 plt.xlabel('Year', fontsize=20)
10 plt.legend(["Mean", "EWM"], fontsize=20)

```

The result is shown in Figure 1.20.

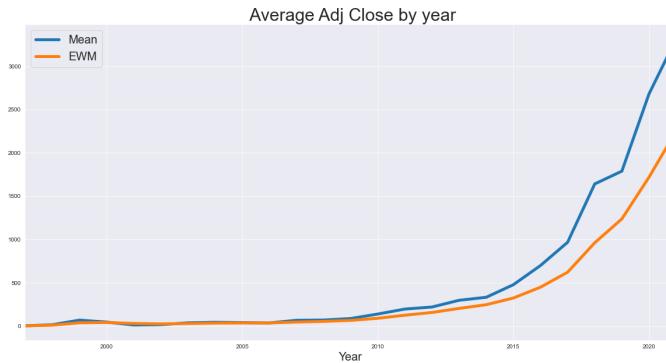


Figure 1.20 The average close by year with mean and ewm

Step 4 Normalize year-wise data and plot line graph representation of all the year-wise data. The code normalizes the year-wise data for selected columns and then plots a line graph to represent the normalized data. Here's an explanation of each step:

1. Selecting the columns for normalization:
 - The `cols = ["Volume", "High", "Low", "Open", "Close", "Adj Close"]` code creates a list of column names that you want to normalize.
2. Normalizing the year-wise data:
 - The `(year_data[cols] - year_data[cols].min()) / (year_data[cols].max() - year_data[cols].min())` code normalizes the selected columns in the `year_data` dataframe.
 - `(year_data[cols].min())` calculates the minimum value for each column.
 - `(year_data[cols].max())` calculates the maximum value for each column.
 - Subtracting the minimum and dividing by the range (maximum - minimum) normalizes the data to the range [0, 1].
 - The resulting normalized data is stored in the `norm_data` dataframe.
3. Plotting the line graph for the normalized data:
 - The `plt.figure(figsize=(20,10))` code creates a figure with a specific size.

- The `plt.xlabel('YEAR')` code sets the label for the x-axis to "YEAR".
- The `plt.title('Normalized Year-Wise Data', fontsize=25)` code sets the title of the plot to "Normalized Year-Wise Data" with a font size of 25.
- The `sns.lineplot(data=norm_data, marker='s', linewidth=5)` code plots the line graph for the normalized data.
- The `data=norm_data` parameter specifies the data to be plotted.
- The `marker='s'` parameter sets the marker style to square markers.
- The `linewidth=5` parameter sets the linewidth of the lines to 5.

4. Displaying the plot:

- The `plt.show()` code displays the plot.

Overall, the code normalizes the selected columns of the year-wise data and then plots a line graph to visualize the normalized values. The normalization process scales the data between 0 and 1, allowing for easier comparison and analysis of the relative trends and patterns in the data.

```

1 #Normalizes year-wise data
2 cols = ["Volume", "High", "Low", "Open", "Close", "Adj Close"]
3 norm_data = (year_data[cols] - year_data[cols].min()) /
4 (year_data[cols].max() - year_data[cols].min())
5
6
7 #Line graph representation of all the year-wise data
8 plt.figure(figsize=(20,10))
9 plt.xlabel('YEAR')
10 plt.title('Normalized Year-Wise Data', fontsize=25)
11 sns.lineplot(data=norm_data, marker='s', linewidth=5)
12 plt.show()

```

The result is shown in Figure 1.21.

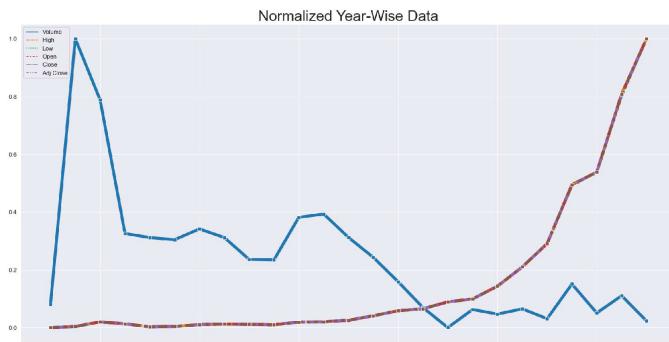


Figure 1.21 The line graph representation of all the year-wise data

Step 5 Define `plot_norm_year_wise_data()` method to plot boxplot, violinplot, stripplot, and heatmap of normalized year-wise

data. It takes the normalized year-wise data as input and plots a boxplot, violinplot, stripplot, and heatmap to visualize the data. Here's an explanation of each step:

1. Creating the subplots:

- The `_, ax = plt.subplots(2, 2, figsize=(50, 30), facecolor='#fbe7dd')` code creates a figure with four subplots arranged in a 2x2 grid.
- The `ax` variable holds references to each subplot.

2. Plotting the boxplot:

- The `sns.boxplot(data=norm_data, ax=ax[0, 0])` code plots a boxplot of the normalized year-wise data.
- The `ax[0, 0]` parameter specifies the position of the boxplot in the first row and first column of the grid.

3. Plotting the violinplot:

- The `sns.violinplot(data=norm_data, ax=ax[0, 1])` code plots a violinplot of the normalized year-wise data.
- The `ax[0, 1]` parameter specifies the position of the violinplot in the first row and second column of the grid.

4. Plotting the stripplot:

- The `sns.stripplot(data=norm_data, jitter=True, s=18, alpha=0.3, ax=ax[1, 0])` code plots a stripplot of the normalized year-wise data.
- The `jitter=True` parameter adds jitter to the data points for better visibility.
- The `s=18` parameter sets the size of the markers to 18.
- The `alpha=0.3` parameter sets the transparency of the markers to 0.3.
- The `ax[1, 0]` parameter specifies the position of the stripplot in the second row and first column of the grid.

5. Plotting the heatmap:

- The `sns.heatmap(norm_data, annot=True, cmap='Greens', yticklabels=norm_data.index.year, ax=ax[1, 1])` code plots a heatmap of the normalized year-wise data.
- The `annot=True` parameter displays the data values in each cell of the heatmap.
- The `cmap='Greens'` parameter sets the color map to use for the heatmap.
- The `yticklabels=norm_data.index.year` parameter sets the y-axis tick labels to the

years from the index of the **norm_data** dataframe.

- The **ax[1, 1]** parameter specifies the position of the heatmap in the second row and second column of the grid.

6. Adjusting the layout and displaying the plot:

- The **plt.tight_layout()** code adjusts the spacing between subplots to prevent overlap.
- The **plt.show()** code displays the plot.

Overall, the **plot_norm_year_wise_data()** function provides a comprehensive visualization of the normalized year-wise data using a boxplot, violinplot, stripplot, and heatmap. These plots offer different perspectives on the distribution and patterns in the data, allowing for a better understanding of the data's characteristics.

```
1 #Plots boxplot, violinplot, stripplot, and heatmap of
2 normalized year-wise data
3 def plot_norm_year_wise_data(norm_data):
4     _,ax = plt.subplots(2,2,figsize=
5     (50,30),facecolor='#fbe7dd')
6     g=sns.boxplot(data=norm_data,ax = ax[0,0])
7     g.xaxis.get_label().set_fontsize(30)
8     g.set_title("The box plot of normalized year-wise data",
9     fontsize=35)
10
11
12     g=sns.violinplot(data=norm_data,ax = ax[0,1])
13     g.xaxis.get_label().set_fontsize(30)
14     g.set_title("The violin plot of normalized year-wise
15 data", \
16     fontsize=35)
17
18     g=sns.stripplot(data=norm_data, jitter=True, s=18, \
19     alpha=0.3, ax = ax[1,0])
20     g.xaxis.get_label().set_fontsize(30)
21     g.set_title("The strip plot of normalized year-wise
22 data", \
23     fontsize=35)
24
25     g=sns.heatmap(norm_data, annot=True, cmap='Greens',
26     yticklabels=norm_data.index.year, ax = ax[1,1])
27     g.xaxis.get_label().set_fontsize(30)
28     g.set_title("The heat map of normalized year-wise data",
29     fontsize=35)
30     plt.tight_layout()
31     plt.show()
32
33 plot_norm_year_wise_data(norm_data)
```

The result is shown in Figure 1.22.

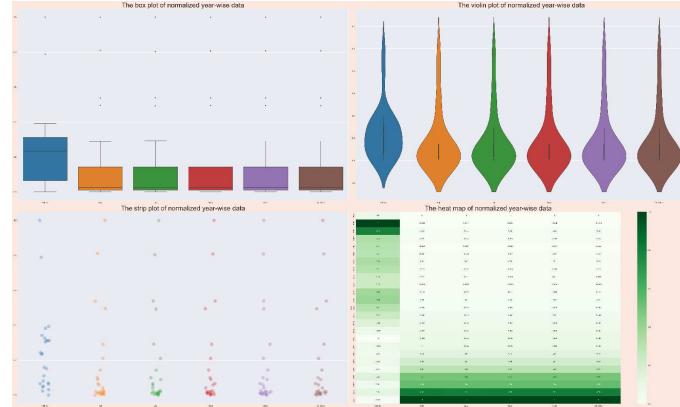


Figure 1.22 The boxplot, violinplot, stripplot, and heatmap of normalized year-wise data

ANALYZING MONTH-WISE DATA

ANALYZING MONTH-WISE DATA

Step 1	<p>Resample the data month-wise by mean and by EWM. Let's break down the code step by step:</p> <ol style="list-style-type: none"> 1. monthly_data = <code>df_dummy[cols].resample('m').mean()</code> <ul style="list-style-type: none"> • df_dummy[cols]: This selects specific columns (cols) from the DataFrame df_dummy. • .resample('m'): This resamples the data based on a monthly frequency ('m'). It groups the data into monthly intervals. • .mean(): This calculates the mean value for each monthly interval, resulting in the average value for each month. • Overall, this step resamples the data month-wise and calculates the mean value for each month. 2. monthly_data_ewm = <code>monthly_data.ewm(span=5).mean()</code> <ul style="list-style-type: none"> • monthly_data: This is the DataFrame obtained from the previous step. • .ewm(span=5): This applies exponential weighted moving average (EWMA) to the data. The span parameter determines the decay factor or window size for the EWMA calculation. A smaller span gives more weight to recent values, while a larger span includes more historical data. • .mean(): This calculates the mean value for each window in the EWMA series. • Overall, this step applies EWMA to the monthly data, smoothing out the values and
--------	--

providing an exponentially weighted average for each month.

So, the code first resamples the original data month-wise and calculates the mean for each month. Then, it applies exponential weighted moving average (EWMA) to the resampled data, creating a smoothed series of monthly values.

```
1 #Resamples the data month-wise by mean
2 monthly_data = df_dummy[cols].resample('m').mean()
3
4 #Resamples the data month-wise by EWM
5 monthly_data_ewm=monthly_data.ewm(span=5).mean()
```

Step 2 Then, plot average **Low** by month with mean and EWM. The code is used to create a plot that shows the average low values by month, comparing the mean and exponential weighted moving average (EWM) values. Let's go through the code step by step:

1. **fig = plt.figure(figsize=(30, 15))**: This creates a new figure object with a specified size of 30 inches in width and 15 inches in height. This will be the canvas for the plot.
2. **plt.rcParams.update({'figure.dpi':120})**: This updates the DPI (dots per inch) setting for the figure. Increasing the DPI can improve the resolution and clarity of the resulting plot.
3. **monthly_data["Low"].plot(linewidth=5)**: This line plots the "Low" column from the **monthly_data** DataFrame. The linewidth parameter sets the thickness of the line in the plot. This line represents the average low values by month based on the mean calculation.
4. **month_data_ewm = monthly_data_ewm["Low"].plot(linewidth=5)**: This line plots the "Low" column from the **monthly_data_ewm** DataFrame, which contains the exponentially weighted moving average values. The linewidth parameter sets the thickness of the line in the plot.
5. **plt.title('Average Low by month', fontsize=30)**: This sets the title of the plot as "Average Low by month" with a font size of 30.
6. **plt.xlabel('Year', fontsize=25)**: This sets the label for the x-axis as "Year" with a font size of 25.
7. **plt.legend(["Mean", "EWM"], fontsize=25)**: This creates a legend for the plot, associating the labels "Mean" and "EWM" with the respective lines. The fontsize parameter sets the font size for the legend.

Overall, this code generates a plot showing the average low values by month, comparing the mean values to the exponentially weighted moving average values.

```

1 #Plots average Low by month with mean and EWM
2 fig=plt.figure(figsize=(30,15))
3 plt.rcParams.update({'figure.dpi':120})
4 monthly_data["Low"].plot(linewidth=5)
5 month_data_ewm=monthly_data_ewm["Low"].plot(linewidth=5)
6 plt.title('Average Low by month', fontsize=30)
7 plt.xlabel('Year', fontsize=25)
8 plt.legend(["Mean", "EWM"], fontsize=25)

```

The result is shown in Figure 1.23.



Figure 1.23 The average **Low** by month with mean and EWM

Step 3 In similar way, Plot average **Adj Close** by month with mean and EWM.

```

1 #Plots average Adj Close by month with mean and EWM
2 fig=plt.figure(figsize=(30,15))
3 plt.rcParams.update({'figure.dpi':120})
4 monthly_data["Adj Close"].plot(linewidth=5)
5 month_data_ewm=monthly_data_ewm["Adj Close"].plot(linewidth=5)
6 plt.title('Average Adj Close by month', fontsize=30)
7 plt.xlabel('Year', fontsize=25)
8 plt.legend(["Mean", "EWM"], fontsize=25)

```

The result is shown in Figure 1.24.

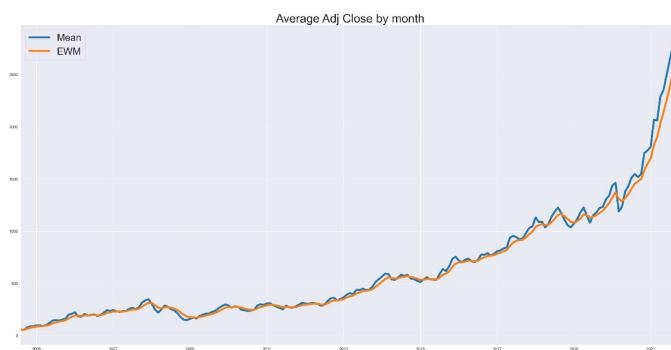


Figure 1.24 The average **Adj Close** by month with mean and EWM

Step 3	<p>Define color_month(), line_plot_month(), sns_plot_month() functions to plot line graph of month-wise a feature each month all year. Thress three functions can be used to create line plots for different months of the year using the seaborn library. Let's go through each function:</p> <ol style="list-style-type: none"> 1. color_month(month): This function takes a month as input and returns a tuple with the month name and a corresponding color. It uses a series of if-elif statements to map each month to a specific color. For example, if the input month is 1, it returns the tuple ('January', 'blue'). 2. line_plot_month(month, data): This function creates a line plot for a specific month using the seaborn library. It takes two parameters: month specifies the month to plot, and data is the data to be plotted. Inside the function, it calls color_month(month) to get the label and color for the line plot. It then filters the data for the specific month using <code>data[data.index.month == month]</code>. Finally, it uses sns.lineplot() from seaborn to create the line plot, specifying the label, color, marker style, and linewidth. 3. sns_plot_month(title, feat): This function creates a plot with line plots for each month of the year using the seaborn library. It takes two parameters: title specifies the title of the plot, and feat specifies the feature to be plotted on the y-axis. Inside the function, it sets up the figure size, title, and axis labels. It then iterates over the range from 1 to 12 (representing the months) and calls line_plot_month() for each month, passing the corresponding feature data from monthly_data. Finally, it displays the plot using plt.show(). <p>By calling sns_plot_month() with appropriate arguments, you can create line plots for different features, such as average low, average high, etc., for each month of the year.</p> <pre> 1 def color_month(month): 2 if month == 1: 3 return 'January', 'blue' 4 elif month == 2: 5 return 'February', 'green' 6 elif month == 3: 7 return 'March', 'orange' 8 elif month == 4: 9 return 'April', 'yellow' 10 elif month == 5: 11 return 'May', 'red' 12 elif month == 6: 13 return 'June', 'violet' 14 elif month == 7: 15 return 'July', 'purple' 16 elif month == 8: 17 return 'August', 'black' </pre>
--------	--

```
18     elif month == 9:
19         return 'September', 'brown'
20     elif month == 10:
21         return 'October', 'darkblue'
22     elif month == 11:
23         return 'November', 'grey'
24     else:
25         return 'December', 'pink'
26
27 def line_plot_month(month, data):
28     label, color = color_month(month)
29     mdata = data[data.index.month == month]
30     sns.lineplot(data=mdata,
31                   label=label,
32
33
34
35
36
37
38
39
40
41
42
43
44
```

```

        color=color,
        marker='o',
        linewidth=5)

def sns_plot_month(title, feat):
    plt.figure(figsize=(25,10))
    plt.title(title, fontsize=40)
    plt.xlabel('YEAR', fontsize=30)
    plt.ylabel(feat, fontsize=30)

    for i in range(1,13):
        line_plot_month(i,monthly_data[feat])
    plt.show()

```

Step 4

Plot line graph of month-wise **Adj Close** feature each all year. It plots a line graph of the "Adj Close" feature each month of the year. Here's a breakdown of the involved:

1. **plt.figure(figsize=(25,10))**: This sets up a figure for the plot and specifies the size of the figure to be 25 inches in width and 10 inches in height.
2. **title = "The line plot of Adj Close"**: This assigns the title for the plot to the variable `title`.
3. **sns_plot_month(title, "Adj Close")**: This calls the `sns_plot_month()` function, passing the variable as the title of the plot and the string "Adj Close" as the feature to be plotted.
4. Inside the `sns_plot_month()` function, the following steps occur:
 - **plt.title(title, fontsize=40)**: This sets the title of the plot using the title variable passed as an argument. The `fontsize` parameter sets the font size to 40.
 - **plt.xlabel('YEAR', fontsize=30)**: This sets the x-axis label to "YEAR" and specifies the font size as 30.
 - **plt.ylabel(feat, fontsize=30)**: This sets the y-axis label to the feature passed as the `feat` variable and sets the font size as 30.
 - The for loop iterates over the range from 1 to 12 (representing the months).
 - For each iteration, the `line_plot_month()` function is called with the current month (from the for loop) and the feature "Adj Close". This plots a line graph for the specified month and feature.
 - Finally, `plt.show()` is called to display the plot with all the lines representing the "Adj Close" feature for each month.

In summary, the code generates a line plot with monthly data for the "Adj Close" feature, displaying the variation of values over the course of a year. The plot will have a title, x-axis label, and y-axis label, and each month will be represented by a separate line.

```

1 #Plots line graph of month-wise Adj Close feature each
2 all year
3 plt.figure(figsize=(25,10))
4 title= "The line plot of Adj Close"
sns_plot_month(title, "Adj Close")

```

The result is shown in Figure 1.25.

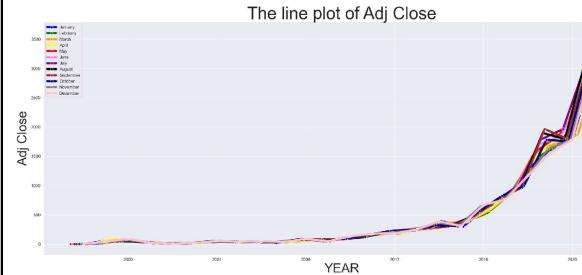


Figure 1.25 The line graph of month-wise **Adj Close** feature each month all year

Step 5

In the similar way, plot line graph of month-wise **V** feature each month all year.

```

1 #Plots line graph of month-wise Volume feature each mc
2 all year
3 plt.figure(figsize=(25,10))
4 title= "The line plot of Volume"
sns_plot_month(title, "Volume")

```

The result is shown in Figure 1.26.

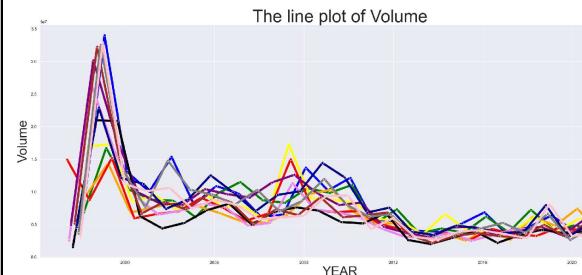


Figure 1.26 The line graph of month-wise **Volume** feature each month all year

Step 6

Normalize month-wise data and plot line representation of all the month-wise data.

```

1 #Normalizes month-wise data
2 norm_data_monthly = (monthly_data - monthly_data.min() /
(monthly_data.max() - monthly_data.min()))
3 print(norm_data_monthly.head().to_string())
4
5
6 #Line graph representation of all the month-wise data
7 plt.figure(figsize=(30,15))
8 plt.xlabel('YEAR', fontsize=25)
9 plt.title('LINE PLOT OF NORMALIZED MONTH-WISE DATA',
10 fontsize=35)
11 sns.lineplot(data=norm_data_monthly, marker='s', linewidth=2)
plt.show()

```

The code performs normalization on the month-wise data and then plots a line graph of the normalized data.

break down the steps:

1. `norm_data_monthly = (monthly_data - monthly_data.min()) / (monthly_data.max() - monthly_data.min())`: This line calculates the normalized values for the `monthly_data` variable by subtracting the minimum value and dividing by the range (maximum value minus minimum value). The resulting normalized data is stored in the `norm_data_monthly` variable.
2. `print(norm_data_monthly.head().to_string())`: This prints the first few rows of the normalized data using the `head()` function and the `to_string()` method to display the output as a string.
3. `plt.figure(figsize=(30,15))`: This sets up the figure for the plot and specifies the size of the figure to be 30 inches in width and 15 inches in height.
4. `plt.xlabel('YEAR', fontsize=25)`: This sets the x-axis label to "YEAR" and specifies the font size as 25.
5. `plt.title('LINE PLOT OF NORMALIZED MONTH-WISE DATA', fontsize=35)`: This sets the title of the plot to "LINE PLOT OF NORMALIZED MONTH-WISE DATA" and specifies the font size as 35.
6. `sns.lineplot(data=norm_data_monthly, marker='s', linewidth=5)`: This plots a line graph using the Seaborn library's `lineplot` function. The `data` parameter is set to `norm_data_monthly`, which contains the normalized data. The `marker` parameter is set to 's' to indicate square markers, and the `linewidth` parameter is set to 5 to specify the width of the lines.
7. `plt.show()`: This displays the plot with the line graph of the normalized month-wise data.

In summary, the code normalizes the month-wise data by calculating the normalized values using the minimum and maximum values of the data. It then prints the first few rows of the normalized data and plots a line graph of the normalized data, displaying the variation in values over years. The plot includes an x-axis label, a title, and markers representing the normalized data.

The result is shown in Figure 1.27.

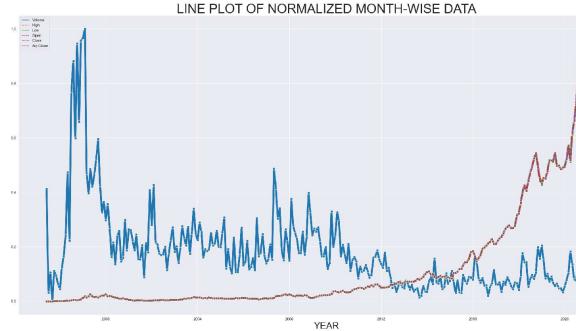


Figure 1.27 Showing line graph representation of all month-wise data

Step 7

Define **plot_norm_month_wise_data()** function to boxplot, violinplot, stripplot, and lineplot of norm month-wise data:

```

1 #Plots boxplot, violinplot, stripplot, and lineplot of
2 normalized month-wise data
3 def plot_norm_month_wise_data(norm_data, month, title)
4     label = color_month(month)[0]
5     data = norm_data[norm_data.index.month == month]
6     _,ax = plt.subplots(2,2,figsize=
7     (40,20),facecolor="#fbe7d1")
8     g=sns.boxplot(data=data,ax = ax[0,0])
9     g.xaxis.get_label().set_fontsize(30)
10    g.set_title("The box plot of normalized month-wis
11    in " \
12        + label, fontsize=35)
13
14    g=sns.violinplot(data=data,ax = ax[0,1])
15    g.xaxis.get_label().set_fontsize(30)
16    g.set_title("The violin plot of normalized month-w
17    data in " + label, fontsize=35)
18
19    g=sns.stripplot(data=data, jitter=True, s=18, alp
20    \
21        ax = ax[1,0])
22    g.xaxis.get_label().set_fontsize(30)
23    g.set_title("The strip plot of normalized month-wi
24    data in " + label, fontsize=35)
25
26
27    g=sns.lineplot(data=data, marker='s', \
28        ax = ax[1,1],linewidth=5)
29    g.xaxis.get_label().set_fontsize(30)
30    g.set_title("The line plot of normalized month-wis
in " + label, fontsize=35)
g.set_xlabel("YEAR")
plt.suptitle(title + " in " + label, fontsize=45)
plt.subplots_adjust(top=0.9)
plt.show()

```

The code plots multiple types of plots for the norm month-wise data. Here's a breakdown of the steps involved:

1. **def plot_norm_month_wise_data(norm_month, title):** This function takes arguments: **norm_data** (the normalized month-wise data), **month** (the specific month to plot), and **title** (the title for the plot).
2. **label = color_month(month)[0]:** This assigns the label for the specific month using the **color_month()** function. It extracts the first element from the tuple returned by **color_month()**.

- `color_month()`, which represents the name of the month.
3. `data = norm_data[norm_data.index.month == month]`: This line filters the `norm_data` based on the specified month, extracting only the data for that month.
 4. `_, ax = plt.subplots(2, 2, figsize=(40, 20), facecolor='#fbe7dd')`: This line creates a figure with four subplots arranged in a 2x2 grid. The `figsize` parameter sets the size of the figure, and the `facecolor` parameter sets the background color of the figure.
 5. `g = sns.boxplot(data=data, ax=ax[0, 0])`: This line creates a boxplot using the Seaborn library's `boxplot` function. The `data` parameter is set to the filtered data for the specific month, and the `ax` parameter specifies the subplot where the boxplot will be plotted. The returned object `g` represents the boxplot.
 6. `g.xaxis.get_label().set_fontsize(30)`: This line sets the font size of the x-axis label for the boxplot.
 7. `g.set_title("The box plot of normalized monthly data in " + label, fontsize=35)`: This line sets the title of the boxplot, including the name of the month obtained from `label`.
 8. Similar to steps 5-7, the function creates and customizes the following plots: a violin plot (`sns.violinplot`), a strip plot (`sns.stripplot`), and a line plot (`sns.lineplot`).
 9. `plt.suptitle(title + " in " + label, fontsize=30)`: This line sets the super title of the entire plot by combining the title and `label` variables.
 10. `plt.subplots_adjust(top=0.9)`: This line adjusts the spacing between the subplots to prevent overlap.
 11. `plt.show()`: This displays the plot with all four subplots, showing the boxplot, violin plot, strip plot, and line plot for the normalized monthly data for the specific month.

In summary, the `plot_norm_month_wise_data()` function takes normalized month-wise data, a specific month, and a title as input. It plots four different types of plots (boxplot, violinplot, stripplot, and lineplot) for the normalized data for the specified month. Each plot is displayed in a separate subplot, and the title of each subplot includes the name of the month. The entire plot has a super title combining the input title and the name of the month.

Step 8

Plot boxplot, violinplot, stripplot, and lineplot of normalized month-wise in March:

```
1 #Plots boxplot, violinplot, stripplot, and lineplot of
2 normalized month-wise in March
```

```

3 | title="The boxplot, violinplot, stripplot, and line pl
4 | all normalized month-wise data in March"
5 | plot_norm_month_wise_data(norm_data_monthly,3, title)

```

The result is shown in Figure 1.28.

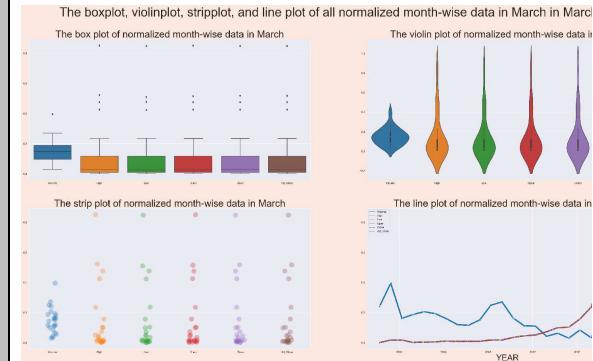


Figure 1.28 The boxplot, violinplot, stripplot, and line normalized month-wise in March

Step 9

Plot boxplot, violinplot, stripplot, and lineplot normalized month-wise close, high, and volume in January:

```

1 | #Plots boxplot, violinplot, stripplot, and lineplot of
2 | normalized month-wise close, high, and volume in Janua
3 | title="The boxplot, violinplot, stripplot, and line pl
4 | normalized month-wise close high and volume in January
5 | plot_norm_month_wise_data(norm_data_monthly[["Close", "Volume"]], 1, title)

```

The result is shown in Figure 1.29.

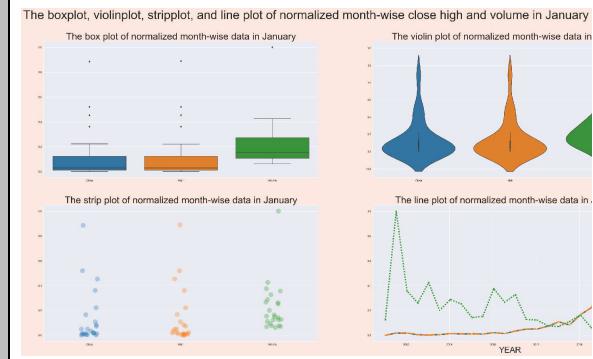


Figure 1.29 The boxplot, violinplot, stripplot, and line normalized month-wise Close, High, and Volume in Ja

COMPUTING TECHNICAL INDICATORS

COMPUTING TECHNICAL INDICATORS

Step 1 Define `compute_daily_returns()` function to compute daily return values:

```

1 def compute_daily_returns(df):
2     """Compute and return the daily return values."""
3     # TODO: Your code here
4     # Note: Returned DataFrame must have the same number of
5     # rows
6     daily_return = (df / df.shift(1)) - 1
7     daily_return[0] = 0
8     return daily_return
9
10 df["daily_returns"] = compute_daily_returns(df["Adj Close"])

```

It calculates the daily returns for the "Adj Close" column in a DataFrame called **df**. Let's break down the steps involved:

1. **def compute_daily_returns(df):**: This defines a function named **compute_daily_returns()** that takes a DataFrame (**df**) as an input parameter.
2. **daily_return = (df / df.shift(1)) - 1**: This line calculates the daily returns by dividing each value in the DataFrame (**df**) by its previous value (shifted by 1) and subtracting 1. This computation is performed element-wise for all values in the **DataFrame**.
3. **daily_return[0] = 0**: This line sets the value of the first element in the **daily_return** DataFrame to **0**. This is done to handle the fact that the first element does not have a previous value to calculate the return.
4. **return daily_return**: This line returns the computed daily returns **DataFrame**.
5. **df["daily_returns"] = compute_daily_returns(df["Adj Close"])**: This line applies the **compute_daily_returns()** function to the "Adj Close" column of the **df** DataFrame and assigns the resulting daily returns to a new column called "daily_returns" in the same DataFrame.

In summary, the **compute_daily_returns()** function calculates the daily returns for a given **DataFrame** by dividing each value by its previous value and subtracting 1. It then assigns the computed daily returns to a new column called "daily_returns" in the original DataFrame (**df**).

Step 2 Plot the scatter distribution of "Adj Close" versus **daily_returns** versus **Year**:

```

1 #Plots the scatter distribution of close versus
2 daily_returns versus Year
3 fig, ax1 = plt.subplots(figsize=(15,10))
4 sns.scatterplot(data=df, x="Adj Close", y="daily_returns",
hue="Year", palette="deep", ax=ax1)

```

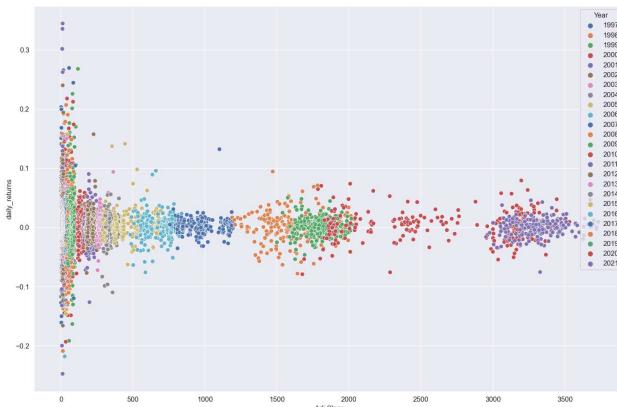


Figure 1.30 The scatter distribution of **Adj Close** versus **daily_returns** versus **Year**

The code plots a scatter distribution of the "Adj Close" values versus the "daily_returns" values, with each data point colored based on the corresponding year. Here's a breakdown of the steps involved:

1. `fig, ax1 = plt.subplots(figsize=(15,10))`: This line creates a new figure and axes object. The `figsize` parameter sets the size of the figure, and the resulting figure and axes objects are assigned to the variables `fig` and `ax1`, respectively.
2. `sns.scatterplot(data=df, x="Adj Close", y="daily_returns", hue="Year", palette="deep", ax=ax1)`: This line creates a scatter plot using the Seaborn library's `scatterplot` function. The `data` parameter is set to the DataFrame (`df`), the `x` parameter is set to "Adj Close" (representing the x-axis values), the `y` parameter is set to "daily_returns" (representing the y-axis values), and the `hue` parameter is set to "Year" (representing the color of each data point based on the year). The `palette` parameter is set to "deep" to use a deep color palette, and the `ax` parameter specifies the axes object where the scatter plot will be plotted (`ax1`).

The resulting plot will show a scatter distribution of points, where the x-axis represents the "Adj Close" values, the y-axis represents the "daily_returns" values, and each data point is colored based on the corresponding year. The `figsize` parameter determines the size of the plot.

The result is shown in Figure 1.30.

Step 2 In the similar way, plot the scatter distribution of **Volume** versus **daily_returns** versus **Year**:

```

1 #Plots the scatter distribution of Volume versus
2 daily_returns versus Year
3 fig, ax1 = plt.subplots(figsize=(15,10))
4

```

```
sns.scatterplot(data=df, x="Volume", y="daily_returns",
hue="Year", palette="deep", ax=ax1)
```

The result is shown in Figure 1.31.



Figure 1.31 The scatter distribution of Volume versus daily_returns versus Year

Step 3 Calculate the moving average convergence-divergence (MACD). It is one of the most powerful and well-known indicators in technical analysis. The indicator is comprised of two exponential moving averages that help measure momentum in a security. The MACD is simply the difference between these two moving averages plotted against a centerline, where the centerline is the point at which the two moving averages are equal.

The Moving Average Convergence Divergence (MACD) is a popular technical analysis indicator used in financial markets, particularly in stock trading. It is used to identify potential buying and selling signals and to gauge the overall trend and momentum of an asset's price.

The MACD indicator consists of three components: the MACD line, the signal line, and the histogram.

- 1. MACD Line:** The MACD line is calculated by taking the difference between two exponential moving averages (EMAs), typically with different time periods. The most common time periods used are 26 and 12. The MACD line represents the relationship between these two moving averages and helps identify potential changes in the underlying trend.

- 2. Signal Line:** The signal line is a moving average (typically an EMA) of the MACD line itself. It is commonly calculated using a time period of 9. The signal line is used to generate trading signals, indicating potential entry and exit points.

- 3. Histogram:** The histogram is derived from the MACD line and the signal line. It represents the difference between the two lines and provides

visual cues about the strength of the current trend. Positive histogram values indicate bullish momentum, while negative values indicate bearish momentum.

The purpose of the MACD indicator is to provide traders with insights into the trend direction, momentum, and potential reversals in an asset's price movement. By analyzing the MACD line, signal line, and histogram, traders can generate signals such as bullish or bearish crossovers, divergence patterns, and overbought or oversold conditions. These signals can help traders make informed decisions about buying or selling assets.

```
1 def calculate_MACD(df, nslow=26, nfast=12):
2     emaslow = df.ewm(span=nslow, min_periods=nslow, \
3                       adjust=True, ignore_na=False).mean()
4     emafast = df.ewm(span=nfast, min_periods=nfast, \
5                       adjust=True, ignore_na=False).mean()
6     dif = emafast - emaslow
7     MACD = dif.ewm(span=9, min_periods=9, adjust=True, \
8                      ignore_na=False).mean()
9     return dif, MACD
```

Let's break down the steps involved:

1. **def calculate_MACD(df, nslow=26, nfast=12):**
This defines a function named **calculate_MACD()** that takes a DataFrame (**df**) and two optional parameters **nslow** and **nfast** (default values of 26 and 12, respectively) as input.
2. **emaslow = df.ewm(span=nslow, min_periods=nslow, adjust=True, ignore_na=False).mean():** This line calculates the exponential moving average (EMA) with a span of **nslow** for the DataFrame **df**. The **ewm** function is used to compute the EMA, and **mean()** calculates the mean of the EMA values.
3. **emafast = df.ewm(span=nfast, min_periods=nfast, adjust=True, ignore_na=False).mean():** This line calculates the exponential moving average (EMA) with a span of **nfast** for the DataFrame **df**. The **ewm** function is used to compute the EMA, and **mean()** calculates the mean of the EMA values.
4. **dif = emafast - emaslow:** This line calculates the difference between the **emafast** and **emaslow** values, which represents the MACD line.
5. **MACD = dif.ewm(span=9, min_periods=9, adjust=True, ignore_na=False).mean():** This line calculates the exponential moving average (EMA) with a span of 9 for the **dif** values calculated in the previous step. The **ewm()** function is used to compute the EMA, and **mean()** calculates the mean of the EMA values. This represents the signal line of the MACD.
6. **return dif, MACD:** This line returns two values: **dif**, which represents the MACD line, and **MACD**,

	<p>which represents the signal line.</p> <p>In summary, the <code>calculate_MACD()</code> function takes a DataFrame <code>df</code> and calculates the MACD and signal line values based on the input parameters <code>nslow</code> and <code>nfast</code>. It returns two DataFrames: <code>dif</code>, representing the MACD line, and <code>MACD</code>, representing the signal line.</p>
Step 4	<p>Calculate the relative strength index (RSI). It is a widely used technical analysis indicator that measures the speed and change of price movements in a financial instrument. The primary purpose of the RSI is to identify overbought and oversold conditions in the market, indicating potential trend reversals or corrective price movements.</p> <p>The RSI is displayed as an oscillator that ranges between 0 and 100. Typically, values above 70 are considered overbought, suggesting that the price may be due for a downward correction or reversal. Conversely, values below 30 are considered oversold, indicating that the price may be due for an upward correction or reversal.</p> <p>Here are some key purposes and applications of the RSI:</p> <ol style="list-style-type: none"> Overbought and Oversold Conditions: The RSI helps traders and analysts identify when an asset is potentially overbought or oversold. This information can be used to make decisions about entering or exiting trades, as overbought conditions may signal a good time to sell or take profits, while oversold conditions may signal a good time to buy. Trend Reversals: The RSI can provide early indications of potential trend reversals. Divergence between the price and the RSI can signal a weakening trend or an upcoming reversal. Bullish divergence occurs when the price makes lower lows while the RSI makes higher lows, indicating a possible upward reversal. Bearish divergence occurs when the price makes higher highs while the RSI makes lower highs, indicating a possible downward reversal. Confirmation of Price Movements: The RSI can be used to confirm price movements and trends. For example, if an asset is in an uptrend and the RSI remains in the overbought zone, it may suggest that the uptrend is strong and likely to continue. Similarly, if an asset is in a downtrend and the RSI remains in the oversold zone, it may indicate that the downtrend is strong and likely to continue. Trading Signals: The RSI can generate trading signals based on overbought and oversold conditions. When the RSI crosses above the overbought threshold (e.g., 70), it may be a signal to sell or take profits. Conversely, when the RSI

crosses below the oversold threshold (e.g., 30), it may be a signal to buy or enter a long position.

It's important to note that the RSI is most effective when used in conjunction with other technical indicators and analysis techniques. Traders often combine the RSI with trend lines, moving averages, and other oscillators to gain a more comprehensive understanding of market conditions and make informed trading decisions.

```
1 def calculate_RSI(df, periods=14):
2     # wilder's RSI
3     delta = df.diff()
4     up, down = delta.copy(), delta.copy()
5
6     up[up < 0] = 0
7     down[down > 0] = 0
8
9     rUp = up.ewm(com=periods, adjust=False).mean()
10    rDown = down.ewm(com=periods, adjust=False).mean().abs()
11
12    rsi = 100 - 100 / (1 + rUp / rDown)
13
14    return rsi
```

Let's break down the steps involved:

1. **delta = df.diff()**: This line calculates the difference between consecutive values in the DataFrame **df** and assigns it to a new DataFrame called **delta**. This represents the price change between each period.
2. **up, down = delta.copy(), delta.copy()**: This line creates two copies of the **delta** DataFrame, assigning one to **up** and the other to **down**. These will be used to separate the positive and negative price changes.
3. **up[up < 0] = 0 and down[down > 0] = 0**: These lines set any negative values in the **up** DataFrame to 0 and any positive values in the **down** DataFrame to 0. This ensures that only positive price changes are considered for the calculation of the RSI.
4. **rUp = up.ewm(com=periods, adjust=False).mean()**: This line calculates the exponential moving average (EMA) of the positive price changes (**up**) with a smoothing factor of **periods**. The **ewm()** function is used to compute the EMA, and **mean()** calculates the mean of the EMA values.
5. **rDown = down.ewm(com=periods, adjust=False).mean().abs()**: This line calculates the exponential moving average (EMA) of the absolute values of the negative price changes (**down**) with a smoothing factor of **periods**. The **ewm()** function is used to compute the EMA, **mean()** calculates the mean of the EMA values, and **abs()** takes the absolute values.

	<p>6. <code>rsi = 100 - 100 / (1 + rUp / rDown)</code>: This line calculates the RSI values using the formula: $RSI = 100 - (100 / (1 + RS))$, where RS (Relative Strength) is calculated as the ratio of the average gain (rUp) to the average loss (rDown).</p> <p>7. <code>return rsi</code>: This line returns the calculated RSI values as a DataFrame.</p> <p>In summary, the <code>calculate_RSI()</code> function takes a DataFrame <code>df</code> and calculates the RSI values based on the input parameter periods. It returns a DataFrame containing the RSI values for the given data. The RSI is a popular indicator used to identify overbought and oversold conditions in the market, helping traders determine potential reversal points in price trends.</p>
Step 5	<p>Calculate the Simple Moving Average (SMA). It simply takes the sum of all of the past closing prices over a time period and divides the result by the total number of prices used in the calculation. For example, a 10-day simple moving average takes the last ten closing prices and divides them by ten.</p> <p>The purpose of the Simple Moving Average (SMA) is to help identify trends and smooth out short-term price fluctuations in financial data. Here are the main purposes and applications of the SMA:</p> <ol style="list-style-type: none"> Trend Identification: The SMA is commonly used to determine the direction of a trend. By calculating the average closing prices over a specific number of periods, the SMA provides a smoothed line that helps identify whether the price is in an uptrend, downtrend, or trading sideways. Traders and analysts often use multiple SMAs with different timeframes (e.g., 20-day, 50-day, or 200-day) to capture short-term, medium-term, and long-term trends. Support and Resistance Levels: The SMA can act as a support or resistance level for the price. During an uptrend, the SMA can provide support, indicating potential buying opportunities when the price dips close to or touches the SMA. Conversely, during a downtrend, the SMA can act as resistance, suggesting potential selling opportunities when the price approaches or bounces off the SMA. Entry and Exit Points: Traders often use SMAs to generate trading signals. For example, a common strategy is to buy when the price crosses above the SMA, indicating a potential uptrend, and sell when the price crosses below the SMA, suggesting a potential downtrend. These crossover points can serve as entry and exit points for trades. Confirmation of Price Movements: The SMA can be used to confirm other technical indicators or price patterns. When the price is in an uptrend, and the SMA is sloping upward, it confirms the bullish

sentiment. Similarly, when the price is in a downtrend, and the SMA is sloping downward, it confirms the bearish sentiment.

5. **Volatility Analysis:** The SMA can help identify periods of high or low volatility. When the SMA line is smooth and stable, it indicates low volatility, suggesting a more stable price movement. Conversely, when the SMA line is fluctuating significantly, it indicates high volatility, suggesting more erratic price movements.

It's important to note that the SMA is a lagging indicator, meaning it reflects past price data. Therefore, it may not always capture rapid price changes or provide timely signals. Traders often combine the SMA with other technical indicators, such as the Relative Strength Index (RSI) or Moving Average Convergence Divergence (MACD), to gain a more comprehensive analysis of the market and make informed trading decisions.

```
1 def calculate_SMA(df, peroids=15):
2     SMA = df.rolling(window=peroids, min_periods=peroids,
3                         center=False).mean()
4     return SMA
```

The calculates the Simple Moving Average (SMA) for a given DataFrame **df**. Let's break down the steps involved:

1. **SMA** = **df.rolling(window=peroids, min_periods=peroids, center=False).mean()**:

This line calculates the Simple Moving Average (SMA) by using the rolling function on the DataFrame **df**. The window parameter specifies the number of periods (or observations) to include in each average calculation. The **min_periods** parameter specifies the minimum number of periods required to have a valid average. The center parameter is set to **False**, indicating that the window is right-aligned (trailing window). The **mean()** function calculates the mean value over each rolling window.

2. **return SMA**: This line returns the calculated **SMA** values as a DataFrame.

In summary, the **calculate_SMA()** function takes a DataFrame **df** and calculates the Simple Moving Average (SMA) values based on the input parameter **periods**. The SMA is a widely used technical indicator that provides an average of the closing prices over a specific number of periods. It helps smooth out price fluctuations and identify potential trends and support/resistance levels.

Step 6 Calculate lower and upper bands. Then, calculate standard deviation. The Bollinger Bands (BB) and standard deviation are commonly used in financial analysis for different purposes. Let's explore the purpose of each:

1. **Bollinger Bands (BB)**:

Bollinger Bands are a technical analysis tool developed by John Bollinger. The purpose of Bollinger Bands is to provide a visual representation of price volatility and potential price reversal points. They consist of three lines plotted on a price chart:

- **Middle Band (SMA):** The middle band is typically a Simple Moving Average (SMA) of the price. It represents the average price over a specific period and acts as a baseline or reference point.
- **Upper Band:** The upper band is calculated by adding a certain multiple (often 2) of the standard deviation to the middle band. It represents the upper limit of expected price movements and is considered a potential resistance level.
- **Lower Band:** The lower band is calculated by subtracting a certain multiple (often 2) of the standard deviation from the middle band. It represents the lower limit of expected price movements and is considered a potential support level.

The purpose of Bollinger Bands is to identify periods of high or low volatility and potential price reversal points. When the price moves close to or crosses the upper band, it suggests that the market is overbought and may be due for a downward correction. Conversely, when the price moves close to or crosses the lower band, it indicates that the market is oversold and may be due for an upward correction. Traders and analysts use Bollinger Bands to generate trading signals, determine stop-loss levels, and assess potential price targets.

2. Standard Deviation:

Standard deviation is a statistical measure that quantifies the dispersion or volatility of a set of data points. In financial analysis, standard deviation is commonly used to assess the risk or volatility of an investment or a market. Here's how it is used:

- **Volatility Measurement:** Standard deviation provides a measure of the variability of price movements. Higher standard deviation values indicate greater price fluctuations and higher volatility, while lower standard deviation values indicate lower volatility. It helps investors and traders gauge the potential risk associated with an investment or a market.
- **Risk Assessment:** Standard deviation is often used as a risk indicator. In investment portfolios, standard deviation is used to measure the risk or volatility of individual assets or the entire portfolio. Lower standard deviation implies lower risk, while higher standard deviation suggests higher risk.

- **Performance Evaluation:** Standard deviation is also used to evaluate the performance of investments or trading strategies. It helps assess the consistency and stability of returns over a given period. Lower standard deviation indicates more stable returns, while higher standard deviation suggests more volatile returns.

By considering the standard deviation, investors and analysts can better understand and manage risk, make informed investment decisions, and assess the performance of investments or strategies.

In summary, Bollinger Bands help identify volatility and potential price reversal points, while standard deviation provides a statistical measure of volatility and helps assess risk and performance in financial analysis. Both tools are widely used by traders, investors, and analysts to make informed decisions in the financial markets.

```

1 def calculate_BB(df, periods=15):
2     STD = df.rolling(window=periods, min_periods=periods, \
3         center=False).std()
4     SMA = calculate_SMA(df)
5     upper_band = SMA + (2 * STD)
6     lower_band = SMA - (2 * STD)
7     return upper_band, lower_band
8
9 def calculate_stdev(df, periods=5):
10    STDEV = df.rolling(periods).std()
11    return STDEV

```

Let's break down each function and its purpose:

1. **calculate_BB(df, periods=15):** This function calculates the Bollinger Bands (BB) for a given DataFrame **df**. Bollinger Bands are a popular technical analysis tool used to identify volatility and potential price reversal points. Here's how the function works:

- **STD = df.rolling(window=periods, min_periods=periods, center=False).std():** This line calculates the standard deviation (STD) of the DataFrame **df** using the rolling window function. The window parameter specifies the number of periods to include in each calculation, and the **min_periods** parameter sets the minimum number of periods required to have a valid standard deviation.
- **SMA = calculate_SMA(df):** This line calls the **calculate_SMA** function to calculate the Simple Moving Average (SMA) of the DataFrame **df**. The SMA is commonly used as the middle band of the Bollinger Bands.
- **upper_band = SMA + (2 * STD):** This line calculates the upper band of the Bollinger

	<p>Bands by adding two times the standard deviation to the SMA.</p> <ul style="list-style-type: none"> • lower_band = SMA - (2 * STD): This line calculates the lower band of the Bollinger Bands by subtracting two times the standard deviation from the SMA. • The function then returns the upper and lower bands as separate DataFrames. <p>The Bollinger Bands are often used to identify overbought and oversold conditions in the market. When the price touches or crosses the upper band, it may indicate an overbought condition and a potential price reversal. Conversely, when the price touches or crosses the lower band, it may suggest an oversold condition and a potential price reversal.</p> <p>2. calculate_stdev(df, periods=5): This function calculates the standard deviation (STDEV) for a given DataFrame df using a specified number of periods. The standard deviation is a measure of the dispersion or volatility of the data points. Here's how the function works:</p> <ul style="list-style-type: none"> • STDEV = df.rolling(periods).std(): This line calculates the rolling standard deviation of the DataFrame df using the rolling function with the specified number of periods. • The function then returns the calculated standard deviation as a DataFrame. <p>The standard deviation is commonly used as a measure of volatility. Higher standard deviation values indicate greater price fluctuations and higher volatility, while lower standard deviation values indicate lower volatility. Traders and analysts often use the standard deviation in conjunction with other indicators to assess market volatility and adjust their trading strategies accordingly.</p> <p>Overall, these functions provide calculations related to technical analysis indicators. The Bollinger Bands help identify potential price reversal points based on volatility, while the standard deviation measures the dispersion or volatility of the data points.</p>
Step 7	<p>Plot MACD, SMA, RSI, upper and lower bands, and standard deviation of Adj Close column:</p> <pre> 1 #Plots MACD, SMA, RSI, upper and lower bands, and standard 2 #deviation of Adj Close column 3 fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(16, 4)) 4 stock_close=df["Adj Close"] 5 6 #Calculates Simple Moving Average for Adj Close 7 SMA_CLOSE = calculate_SMA(stock_close) 8 stock_close[:365].plot(title='GLD Moving 9 Average',label='GLD', ax=axes[0]) 10 SMA_CLOSE[:365].plot(label="SMA",ax=axes[0]) 11 12 </pre>

```

13 #Calculates Bollinger Bands for Adj Close
14 upper_band, lower_band = calculate_BB(stock_close)
15 upper_band[:365].plot(label='upper band', ax=axes[0])
16 lower_band[:365].plot(label='lower band', ax=axes[0])
17
18 #Calculates MACD for Adj Close
19 DIF, MACD = calculate_MACD(stock_close)
20 DIF[:365].plot(title='DIF and MACD',label='DIF', ax=axes[1])
21 MACD[:365].plot(label='MACD', ax=axes[1])
22
23 #Calculates RSI for Adj Close
24 RSI = calculate_RSI(stock_close)
25 RSI[:365].plot(title='RSI',label='RSI', ax=axes[2])
26
27 #Calculates Standard deviation for Adj Close
28 STDEV= calculate_stdev(stock_close)
29 STDEV[:365].plot(title='STDEV',label='STDEV', ax=axes[3])
30
31 axes[0].set_ylabel('Price')
32 axes[1].set_ylabel('Price')
33 axes[2].set_ylabel('Price')
34 axes[3].set_ylabel('Price')
35
36 axes[0].legend(loc='lower left')
37 axes[1].legend(loc='lower left')
axes[2].legend(loc='lower left')
axes[3].legend(loc='lower left')

```

Here's a step-by-step explanation of the code:

1. Define the figure and axes:

- The code starts by creating a figure and four subplots using `plt.subplots()` with `nrows=1` and `ncols=4` to arrange the subplots in a single row with four columns.
- The `figsize` parameter sets the size of the figure.

2. Retrieve the "Adj Close" data:

The code assigns the "Adj Close" column of the stock data to the variable `stock_close`.

3. Calculate Simple Moving Average (SMA) and plot it with Bollinger Bands:

- The code calls the `calculate_SMA()` function to calculate the SMA for the `stock_close` data.
- It plots the `stock_close` data for the first 365 days using `plot()` on the first subplot (`axes[0]`), with a title and label.
- It also plots the SMA data (SMA_CLOSE) for the first 365 days on the same subplot.
- The code calculates the upper and lower Bollinger Bands using `calculate_BB()` and plots them on the same subplot.

4. Calculate and plot MACD:

- The code calls the `calculate_MACD()` function to calculate the DIF (MACD line) and MACD (signal line) for the `stock_close` data.
- It plots the DIF data for the first 365 days on the second subplot (`axes[1]`) with a title and label.

- It also plots the MACD data on the same subplot.
5. Calculate and plot RSI:
- The code calls the `calculate_RSI()` function to calculate the RSI for the `stock_close` data.
 - It plots the RSI data for the first 365 days on the third subplot (`axes[2]`) with a title and label.
6. Calculate and plot Standard Deviation:
- The code calls the `calculate_stdev()` function to calculate the standard deviation for the `stock_close` data.
 - It plots the standard deviation data for the first 365 days on the fourth subplot (`axes[3]`) with a title and label.
7. Set the y-axis labels and legends:
- The code sets the y-axis labels for each subplot using `set_ylabel()`.
 - It also adds legends to the subplots using `legend()` to indicate the plotted lines.

The result is shown in Figure 1.32.

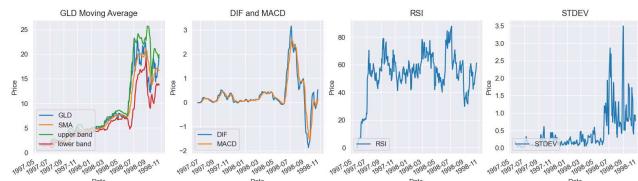


Figure 1.32 The MACD, SMA, RSI, upper and lower bands, and standard deviation of Adj Close column

Step 8 Plot the difference of **Open** and **Adj Close** and the difference of **High** and **Low**:

```

1 #Plots the difference of Open and Close and the difference
2 of High and Low
3 fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(16, 4))
4 Open_Close=df.Open - df["Adj Close"]
5 Open_Close[:365].plot(title='open-close',label='open_close',
6 ax=axes[0])
7
8 High_Low=df.High-df.Low
9 High_Low[:365].plot(title='high_low',label='high_low',
10 ax=axes[1])
11 axes[0].set_ylabel('Price')
12 axes[1].set_ylabel('Price')
13 axes[0].legend(loc='lower left')
14 axes[1].legend(loc='lower left')
```

Here's an explanation of the code step by step:

1. Define the figure and axes:
 - The code creates a figure and two subplots using `plt.subplots()` with `nrows=1` and `ncols=2` to arrange the subplots in a single row with two columns.
 - The `figsize` parameter sets the size of the figure.

2. Calculate and plot the difference between Open and Close:

- The code calculates the difference between the Open and Close prices by subtracting the "Adj Close" column from the "Open" column of the stock data.
- It plots the calculated difference for the first 365 days on the first subplot (**axes[0]**) with a title and label.

3. Calculate and plot the difference between High and Low:

- The code calculates the difference between the High and Low prices by subtracting the "Low" column from the "High" column of the stock data.
- It plots the calculated difference for the first 365 days on the second subplot (**axes[1]**) with a title and label.

4. Set the y-axis labels and legends:

- The code sets the y-axis labels for each subplot using **set_ylabel()**.
- It also adds legends to the subplots using **legend()** to indicate the plotted lines.

The resulting plots show the difference between the Open and Close prices and the difference between the High and Low prices for the stock over a specific period. These plots can provide insights into the price dynamics and volatility of the stock. The result is shown in Figure 1.33.

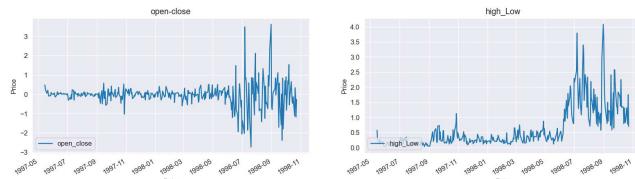


Figure 1.33 The difference of open and adj close and the difference of high and low

This is the full version of **amazon.py** so far:

```
#amazon.py
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import datetime as dt
sns.set_style('darkgrid')
from sklearn.preprocessing import LabelEncoder
import warnings
warnings.filterwarnings('ignore')
import os
import plotly.graph_objs as go
import joblib
import iertools
from sklearn.metrics import roc_auc_score,roc_curve,
explained_variance_score
from sklearn.model_selection import cross_val_score
from statsmodels.tsa.seasonal import seasonal_decompose as season
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.metrics import accuracy_score, balanced_accuracy_score
```

```

from sklearn.model_selection import train_test_split, RandomizedSearchCV,
GridSearchCV, StratifiedKFold
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
RobustScaler
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler, \
    LabelEncoder, OneHotEncoder
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score,
precision_score
from sklearn.metrics import classification_report, f1_score,
plot_confusion_matrix
from catboost import CatBoostClassifier
from lightgbm import LGBMClassifier
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import learning_curve
from mlxtend.plotting import plot_decision_regions
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.pipeline import make_pipeline
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import GradientBoostingRegressor
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
from catboost import CatBoostRegressor
from sklearn.neural_network import MLPRegressor
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from statsmodels.tsa.stattools import adfuller, acf, pacf
from statsmodels.tsa.arima_model import ARIMA
from pmdarima.utils import decomposed_plot
from pmdarima.arima import decompose
from pmdarima import auto_arima
from sklearn.linear_model import LassoCV
from sklearn.linear_model import RidgeCV
from sklearn.mixture import GaussianMixture

curr_path = os.getcwd()
df = pd.read_csv(curr_path+"/Amazon.csv")

#Checks shape
print(df.shape)

##Reads columns
print("Data Columns --> ",list(df.columns))

##Checks null values
print(df.isnull().sum())
print('Total number of null values: ', df.isnull().sum().sum())

##Plots null values
missing = df.isna().sum().reset_index()
missing.columns = ['features', 'total_missing']
missing['percent'] = (missing['total_missing'] / len(df)) * 100
missing.index = missing['features']
del missing['features']
plt.figure(figsize=(15,8))
missing['total_missing'].plot(kind = 'bar')
plt.title('Missing Values Count', fontsize = 25)

##Prints coefficient correlation of every column with Adj Close column
all_corr = df.corr().abs()['Adj Close'].sort_values(\n    ascending = False)
print(all_corr.to_string())

```

```

#Plots pair correlation of all features
sns.pairplot(df)
plt.tight_layout()

#Extracts day, month, week, quarter, and year
df['Date'] = pd.to_datetime(df['Date'])
df['Day'] = df['Date'].dt.weekday
df['Month'] = df['Date'].dt.month
df['Year'] = df['Date'].dt.year
df['Week']= df['Date'].dt.week
df['Quarter']= df['Date'].dt.quarter

#Sets Date column as index
df = df.set_index("Date")

print(df.head(10).to_string())

#Checks dataset information
print(df.info())

#Plots the correlation between each feature by using heatmap
fig, ax = plt.subplots(figsize=(20,10))
mask = np.triu(np.ones_like(df.corr(), dtype=np.bool))
sns.heatmap(df.corr(), annot=True, cmap="Reds", \
    mask=mask, linewidth=0.5)

#Creates a dummy dataframe for visualization
df_dummy=df.copy()

#Converts days and months from numerics to meaningful string
days = {0:'Sunday',1:'Monday',2:'Tuesday',3:'Wednesday',\
4:'Thursday',5: 'Friday',6:'Saturday'}
df_dummy['Day'] = df_dummy['Day'].map(days)
months={1:'January',2:'February',3:'March',4:'April',\
5:'May',6:'June',7:'July',8:'August',9:'September',\
10:'October',11:'November',12:'December'}
df_dummy['Month']= df_dummy['Month'].map(months)
quarters = {1:'Jan-March', 2:'April-June',3:'July-Sept',\
4:'Oct-Dec'}
df_dummy['Quarter'] = df_dummy['Quarter'].map(quarters)
print(df_dummy.head(10).to_string())

#Defines function to create pie chart and bar plot as subplots
def plot_pie_bar_chart(df,var, title=""):
    plt.figure(figsize=(20,10))
    plt.subplot(121)
    label_list = list(df[var].value_counts().index)

    df[var].value_counts().plot.pie(autopct = "%1.1f%%", \
        colors = sns.color_palette("prism",7), \
        startangle = 60, labels=label_list, \
        wedgeprops={"linewidth":2,"edgecolor":"k"}, \
        shadow =True, textprops={'fontsize': 25})
    plt.title("Case distribution of "+ var +" variable "+title, \
        fontsize=25)

    plt.subplot(122)
    ax = df[var].value_counts().plot(kind="barh",color={"salmon"})
    for i,j in enumerate(df[var].value_counts().values):
        ax.text(.7,i,j,weight = "bold",fontsize=30)

    plt.title("Count of "+ var +" cases " +title, fontsize=30)
    plt.show()

#Plots case distribution of Year
plot_pie_bar_chart(df_dummy, "Year")

#Plots case distribution of Day
plot_pie_bar_chart(df_dummy, "Day")

#Plots case distribution of Month
plot_pie_bar_chart(df_dummy, "Month")

```

```

#Plots case distribution of Quarter
plot_pie_bar_chart(df_dummy, "Quarter")

#Plots the scatter distribution of Open versus Volume versus Year
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df_dummy, x="Open", y="Volume", \
    hue="Year", palette="deep", ax=ax1)

#Plots the scatter distribution of Low versus Volume versus Month
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df_dummy, x="Low", y="Volume", \
    hue="Month", palette="deep", ax=ax1)

#Plots the scatter distribution of Adj Close versus Volume versus Quarter
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df_dummy, x="Adj Close", y="Volume", \
    hue="Quarter", palette="deep", ax=ax1)

#Plots the scatter distribution of high versus volume versus Day
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df_dummy, x="High", y="Volume", \
    hue="Day", palette="deep", ax=ax1)

#Defines function to plot distribution of a grouped dataframe
def plot_group(df, title=""):
    plt.figure(figsize=(20,10))
    plt.subplot(121)
    label_list = list(df.index)

    df.plot.pie(autopct = "%1.1f%%", \
        colors = sns.color_palette("prism",7), \
        startangle = 60, labels=label_list, \
        wedgeprops={"linewidth":2, "edgecolor": "k"}, \
        shadow =True, textprops={'fontsize': 16})
    plt.title(title, fontsize=25)

    plt.subplot(122)
    ax = df.plot(kind="barh", color={"salmon"})
    for i,j in enumerate(df.values):
        ax.text(.7,i,j, weight = "bold", fontsize=15)
    plt.title(title, fontsize=25)

    plt.show()

#Plots which year have most volumes
plot_group(df_dummy.groupby('Year')[['Volume']].sum(), \
    "The distribution of Volume by Year")

#Plots which days of the week have most volumes
plot_group(df_dummy.groupby('Day')[['Volume']].sum(), \
    "The distribution of Volume by Day")

#Plots which month have most volumes
plot_group(df_dummy.groupby('Month')[['Volume']].sum(), \
    "The distribution of volume by Month")

#Plots which quarter have most volumes
plot_group(df_dummy.groupby('Quarter')[['Volume']].sum(), \
    "The distribution of volume by Quarter")

#Puts label inside stacked bar
def put_label_stacked_bar(ax,fontsize):
    #patches is everything inside of the chart
    for rect in ax.patches:
        # Find where everything is located
        height = rect.get_height()
        width = rect.get_width()
        x = rect.get_x()
        y = rect.get_y()

        # The height of the bar is the data value and can be used as the
        label_text = f'{height:.0f}'

        # ax.text(x, y, text)
        label_x = x + width / 2

```

```

label_y = y + height / 2

# plots only when height is greater than specified value
if height > 0:
    ax.text(label_x, label_y, label_text, \
            ha='center', va='center', \
            weight = "bold", fontsize=fontsize)

#Plots the distribution of one variable against another variable
def dist_one_vs_another_plot(df,cat1, cat2, ax1,title=""):
    cmap=plt.cm.Blues
    cmap_r=plt.cm.coolwarm_r

    group_by_stat = df.groupby([cat1, cat2]).size()
    group_by_stat.unstack().plot(kind='bar', \
        stacked=True,ax=ax1,grid=True)
    ax1.set_title('Stacked Bar Plot of ' + \
        cat1 + ' (number of cases) '+title, fontsize=20)
    ax1.set_ylabel('Number of Cases', fontsize=20)
    ax1.set_xlabel(cat1, fontsize=20)
    put_label_stacked_bar(ax1,17)
    plt.show()

#Prints statistical description
print(df.describe().T.to_string())

#Plots histogram distribution of a feature
def hist_dist(df, feat, num_bin,lower,upper,ax1):
    plt.subplots_adjust(wspace=0.25,hspace=0.25)
    sns.histplot(data = df, x = feat, \
        ax=ax1, kde = True, bins=num_bin,line_kws={'lw': 5})
    ax1.set_title('Histogram Distribution of ' + feat, \
        fontsize=25)
    ax1.set_xlabel(feat, fontsize=20)
    ax1.set_ylabel('Count', fontsize=20)
    for p in ax1.patches:
        color = '#75bbfd' if (lower<=p.get_height()<=upper) else '#ff796c'
        p.set_facecolor(color)
        ax1.annotate(format(p.get_height(), '.0f'), \
            (p.get_x() + p.get_width() / 2., p.get_height()), \
            ha = 'center', va = 'center', xytext = (0, 10), \
            weight = "bold", fontsize=20, \
            textcoords = 'offset points')

#Categorizes High feature
labels = ['0-200', '200-500', '500-750','750-1000', '1000-3000']
df_dummy["Cat_High"] = pd.cut(df_dummy["High"], \
    [0, 200, 500, 750, 1000, 3000], labels=labels)

#Categorizes Low feature
labels = ['0-200', '200-500', '500-750','750-1000', '1000-3000']
df_dummy["Cat_Low"] = pd.cut(df_dummy["Low"], \
    [0, 200, 500, 750, 1000, 3000], labels=labels)

#Categorizes Open feature
labels = ['0-200', '200-500', '500-750','750-1000', '1000-3000']
df_dummy["Cat_Open"] = pd.cut(df_dummy["Open"], \
    [0, 200, 500, 750, 1000, 3000], labels=labels)

#Categorizes Close feature
labels = ['0-200', '200-500', '500-750','750-1000', '1000-3000']
df_dummy["Cat_Close"] = pd.cut(df_dummy["Close"], \
    [0, 200, 500, 750, 1000, 3000], labels=labels)

#Categorizes Adj Close feature
labels = ['0-200', '200-500', '500-750','750-1000', '1000-3000']
df_dummy["Cat_Adj_Close"] = pd.cut(df_dummy["Adj Close"], \
    [0, 200, 500, 750, 1000, 3000], labels=labels)

#Categorizes Volume feature
labels = ['400k-2M', '2M-10M', '10M-50M','50M-100M', '100M-200M']
df_dummy["Cat_Volume"] = pd.cut(df_dummy["Volume"], \
    [400000, 2000000, 10000000, 50000000, 100000000, 200000000], \
    labels=labels)

fig, axs = plt.subplots(2,figsize=(25,20),facecolor="#fbe7dd")

```

```

#Plots histogram distribution of High feature
hist_dist(df, 'High', 50, 400, 750, axs[0])

#Plots case distribution of categorized High variable against categorized
Volume variable in stacked bar plots
dist_one_vs_another_plot(df_dummy, 'Cat_Volume', 'Cat_High' ,axs[1])

fig, axs = plt.subplots(2,figsize=(25,20),facecolor='#fbe7dd')

#Plots histogram distribution of Adj Close feature
hist_dist(df, 'Adj Close', 50, 400, 750, axs[0])

#Plots case distribution of categorized Adj Close variable against
categorized Volume variable in stacked bar plots
dist_one_vs_another_plot(df_dummy, 'Cat_Volume', 'Cat_Adj_Close' ,axs[1])

#####
#####YEAR-WISE AND MONTH-
WISE#####
#Plots Open and Close feature over year 2007
plt.subplots(figsize=(20,10),facecolor='#fbe7dd')
plt.xlabel('Date', fontsize=15)
plt.title("The Open and Close over year 2007", fontsize=20)
sns.lineplot(data=df_dummy[df_dummy["Year"]==2007]["Open"], color='red',
marker='d', linewidth=5)
sns.lineplot(data=df_dummy[df_dummy["Year"]==2007]["Close"], color='blue',
marker='d', linewidth=5)
plt.legend(["Open", "Close"], fontsize=20)
plt.show()

#Plots Low and High features over Jan-March year 2021
plt.subplots(figsize=(20,10),facecolor='#fbe7dd')
plt.xlabel('Date', fontsize=15)
plt.title("The Low and High over January-March year 2021", fontsize=20)
sns.lineplot(data=df_dummy[(df_dummy["Year"]==2021)&(\n    df_dummy["Quarter"]=="Jan-March")]["Low"], color='red', \
marker='d', linewidth=5)
sns.lineplot(data=df_dummy[(df_dummy["Year"]==2021)&(\n    df_dummy["Quarter"]=="Jan-March")]["High"], color='blue', \
marker='d', linewidth=5)
plt.legend(["Low", "high"], fontsize=20)
plt.show()

#Plots average Adj Close by year with mean and ewm
fig=plt.figure(figsize=(20,10))
plt.rcParams.update({'figure.dpi':120})
year_data = df_dummy.resample('y').mean()
year_data['Adj Close'].plot(linewidth=5)
year_data_ewm=year_data['Adj Close'].ewm(span=5).mean()
year_data_ewm.plot(linewidth=5)
plt.title(' Average Adj Close by year', fontsize=30)
plt.xlabel('Year', fontsize=20)
plt.legend(["Mean", "EWMA"], fontsize=20)

#Normalizes year-wise data
cols = ["Volume", "High", "Low", "Open", "Close", "Adj Close"]
norm_data = (year_data[cols] - year_data[cols].min()) / \
(year_data[cols].max() - year_data[cols].min())

#Line graph representation of all the year-wise data
plt.figure(figsize=(20,10))
plt.xlabel('YEAR')
plt.title('Normalized Year-Wise Data', fontsize=25)
sns.lineplot(data=norm_data, marker='s',linewidth=5)
plt.show()

#Plots boxplot, violinplot, stripplot, and heatmap of normalized year-wise
data
def plot_norm_year_wise_data(norm_data):
    _,ax = plt.subplots(2,2,figsize=(50,30),facecolor='#fbe7dd')
    g=sns.boxplot(data=norm_data,ax = ax[0,0])
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The box plot of normalized year-wise data", \
    fontsize=35)

    g=sns.violinplot(data=norm_data,ax = ax[0,1])
    g.xaxis.get_label().set_fontsize(30)

```

```

g.set_title("The violin plot of normalized year-wise data", \
            fontsize=35)

g=sns.stripplot(data=norm_data, jitter=True, s=18, \
                 alpha=0.3, ax = ax[1,0])
g.xaxis.get_label().set_fontsize(30)
g.set_title("The strip plot of normalized year-wise data", \
            fontsize=35)

g=sns.heatmap(norm_data, annot=True, cmap='Greens',
               yticklabels=norm_data.index.year, ax = ax[1,1])
g.xaxis.get_label().set_fontsize(30)
g.set_title("The heat map of normalized year-wise data", \
            fontsize=35)
plt.tight_layout()
plt.show()

plot_norm_year_wise_data(norm_data)

##Resamples the data month-wise by mean
monthly_data = df_dummy[cols].resample('m').mean()

##Resamples the data month-wise by EWM
monthly_data_ewm=monthly_data.ewm(span=5).mean()

##Plots average Low by month with mean and EWM
fig=plt.figure(figsize=(30,15))
plt.rcParams.update({'figure.dpi':120})
monthly_data["Low"].plot(linewidth=5)
monthly_data_ewm=monthly_data_ewm["Low"].plot(linewidth=5)
plt.title('Average Low by month', fontsize=30)
plt.xlabel('Year', fontsize=25)
plt.legend(["Mean", "EWM"], fontsize=25)

##Plots average Adj Close by month with mean and EWM
fig=plt.figure(figsize=(30,15))
plt.rcParams.update({'figure.dpi':120})
monthly_data["Adj Close"].plot(linewidth=5)
monthly_data_ewm=monthly_data_ewm["Adj Close"].plot(linewidth=5)
plt.title('Average Adj Close by month', fontsize=30)
plt.xlabel('Year', fontsize=25)
plt.legend(["Mean", "EWM"], fontsize=25)

def color_month(month):
    if month == 1:

```

```

        return 'January', 'blue'
    elif month == 2:
        return 'February', 'green'
    elif month == 3:
        return 'March', 'orange'
    elif month == 4:
        return 'April', 'yellow'
    elif month == 5:
        return 'May', 'red'
    elif month == 6:
        return 'June', 'violet'
    elif month == 7:
        return 'July', 'purple'
    elif month == 8:
        return 'August', 'black'
    elif month == 9:
        return 'September', 'brown'
    elif month == 10:
        return 'October', 'darkblue'
    elif month == 11:
        return 'November', 'grey'
    else:
        return 'December', 'pink'

def line_plot_month(month, data):
    label, color = color_month(month)
    mdata = data[data.index.month == month]
    sns.lineplot(data=mdata,
                  label=label,
                  color=color,
                  marker='o',
                  linewidth=5)

def sns_plot_month(title, feat):
    plt.figure(figsize=(25, 10))
    plt.title(title, fontsize=40)
    plt.xlabel('YEAR', fontsize=30)
    plt.ylabel(feat, fontsize=30)

    for i in range(1, 13):
        line_plot_month(i, monthly_data[feat])
    plt.show()

#Plots line graph of month-wise Adj Close feature each month all year
plt.figure(figsize=(25, 10))
title= "The line plot of Adj Close"
sns_plot_month(title, "Adj Close")

#Plots line graph of month-wise Volume feature each month all year
plt.figure(figsize=(25, 10))
title= "The line plot of Volume"

```

```

sns_plot_month(title, "Volume")

#Normalizes month-wise data
norm_data_monthly = (monthly_data - monthly_data.min()) /
(monthly_data.max() - monthly_data.min())
print(norm_data_monthly.head().to_string())

#Line graph representation of all the month-wise data
plt.figure(figsize=(30,15))
plt.xlabel('YEAR', fontsize=25)
plt.title('LINE PLOT OF NORMALIZED MONTH-WISE DATA', fontsize=35)
sns.lineplot(data=norm_data_monthly, marker='s', linewidth=5)
plt.show()

#Plots boxplot, violinplot, stripplot, and lineplot of normalized month-
wise data
def plot_norm_month_wise_data(norm_data, month, title):
    label = color_month(month)[0]
    data = norm_data[norm_data.index.month == month]
    _,ax = plt.subplots(2,2,figsize=(40,20),facecolor='#fbe7dd')
    g=sns.boxplot(data=data,ax = ax[0,0])
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The box plot of normalized month-wise data in " \
        + label, fontsize=35)

    g=sns.violinplot(data=data,ax = ax[0,1])
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The violin plot of normalized month-wise data in " + \
label, fontsize=35)

    g=sns.stripplot(data=data, jitter=True, s=18, alpha=0.3, \
        ax = ax[1,0])
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The strip plot of normalized month-wise data in " + label, \
        fontsize=35)

    g=sns.lineplot(data=data, marker='s', \
        ax = ax[1,1],linewidth=5)
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The line plot of normalized month-wise data in " + label, \
        fontsize=35)
    g.set_xlabel("YEAR")
    plt.suptitle(title + " in " + label, fontsize=45)
    plt.subplots_adjust(top=0.9)
    plt.show()

#Plots boxplot, violinplot, stripplot, and lineplot of normalized month-
wise in March
title="The boxplot, violinplot, stripplot, and line plot of all normalized
month-wise data in March"
plot_norm_month_wise_data(norm_data_monthly, 3, title)

```

```

#Plots boxplot, violinplot, stripplot, and lineplot of normalized month-
wise close, high, and volume in January
title="The boxplot, violinplot, stripplot, and line plot of normalized
month-wise close high and volume in January"
plot_norm_month_wise_data(norm_data_monthly[["Close","High", \
"Volume"]], 1, title)

#####
#TECHNICAL
INDICATORS#####
def compute_daily_returns(df):
    """Compute and return the daily return values."""
    # TODO: Your code here
    # Note: Returned DataFrame must have the same number of rows
    daily_return = (df / df.shift(1)) - 1
    daily_return[0] = 0
    return daily_return

df["daily_returns"] = compute_daily_returns(df["Adj Close"])

#Plots the scatter distribution of close versus daily_returns versus Year
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df, x="Adj Close", y="daily_returns", \
hue="Year", palette="deep", ax=ax1)

#Plots the scatter distribution of Volume versus daily_returns versus Year
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df, x="Volume", y="daily_returns", \
hue="Year", palette="deep", ax=ax1)

def calculate_MACD(df, nslow=26, nfast=12):
    emaslow = df.ewm(span=nslow, min_periods=nslow, adjust=True,
ignore_na=False).mean()
    emafast = df.ewm(span=nfast, min_periods=nfast, adjust=True,
ignore_na=False).mean()
    dif = emafast - emaslow
    MACD = dif.ewm(span=9, min_periods=9, adjust=True,
ignore_na=False).mean()
    return dif, MACD

def calculate_RSI(df, periods=14):
    # wilder's RSI
    delta = df.diff()
    up, down = delta.copy(), delta.copy()

    up[up < 0] = 0
    down[down > 0] = 0

    rUp = up.ewm(com=periods,adjust=False).mean()
    rDown = down.ewm(com=periods, adjust=False).mean().abs()

```

```

rsi = 100 - 100 / (1 + rUp / rDown)
return rsi

def calculate_SMA(df, peroids=15):
    SMA = df.rolling(window=peroids, min_periods=peroids,
center=False).mean()
    return SMA

def calculate_BB(df, peroids=15):
    STD = df.rolling(window=peroids,min_periods=peroids,
center=False).std()
    SMA = calculate_SMA(df)
    upper_band = SMA + (2 * STD)
    lower_band = SMA - (2 * STD)
    return upper_band, lower_band

def calculate_stdev(df,periods=5):
    STDEV = df.rolling(periods).std()
    return STDEV

#Plots MACD, SMA, RSI, upper and lower bands, and standard deviation of Adj
Close column
fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(16, 4))
stock_close=df["Adj Close"]

#Calculates Simple Moving Average for Adj Close
SMA_CLOSE = calculate_SMA(stock_close)
stock_close[:365].plot(title='GLD Moving Average',label='GLD', ax=axes[0])
SMA_CLOSE[:365].plot(label="SMA",ax=axes[0])

#Calculates Bollinger Bands for Adj Close
upper_band, lower_band = calculate_BB(stock_close)
upper_band[:365].plot(label='upper band', ax=axes[0])
lower_band[:365].plot(label='lower band', ax=axes[0])

#Calculates MACD for Adj Close
DIF, MACD = calculate_MACD(stock_close)
DIF[:365].plot(title='DIF and MACD',label='DIF', ax=axes[1])
MACD[:365].plot(label='MACD', ax=axes[1])

#Calculates RSI for Adj Close
RSI = calculate_RSI(stock_close)
RSI[:365].plot(title='RSI',label='RSI', ax=axes[2])

#Calculates Standard deviation for Adj Close
STDEV= calculate_stdev(stock_close)
STDEV[:365].plot(title='STDEV',label='STDEV', ax=axes[3])

```

```

axes[0].set_ylabel('Price')
axes[1].set_ylabel('Price')
axes[2].set_ylabel('Price')
axes[3].set_ylabel('Price')

axes[0].legend(loc='lower left')
axes[1].legend(loc='lower left')
axes[2].legend(loc='lower left')
axes[3].legend(loc='lower left')

#Plots the difference of Open and Close and the difference of High and Low
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(16, 4))
Open_Close=df.Open - df["Adj Close"]
Open_Close[:365].plot(title='open-close',label='open_close', ax=axes[0])

High_Low=df.High-df.Low
High_Low[:365].plot(title='high_Low',label='high_Low', ax=axes[1])
axes[0].set_ylabel('Price')
axes[1].set_ylabel('Price')
axes[0].legend(loc='lower left')
axes[1].legend(loc='lower left')

```

REGRESSION USING MACHINE LEARNING REGRESSION USING MACHINE LEARNING

FORECASTING ON ADJ CLOSE VALUE USING MACHINE LEARNING

FORECASTING ON ADJ CLOSE VALUE USING MACHINE LEARNING

Step 1 Separate target variable and predictors, check null values because of technical indicators, fill each null value in every column with mean value, normalize data to bring all values to common scale, shift target array to predict the $n + 1$ day value, take last 90 rows of data to be validation set, and split data into training and test data at 90% and 10% respectively.

Here's a step-by-step explanation of the provided code:

1. Copy the DataFrame:

Create a copy of the original DataFrame **df** and assign it to the variable **X**.

2. Add additional columns:

Add new columns to X DataFrame for the calculated technical indicators, such as Simple Moving Average (SMA), Upper Bollinger Band, Lower Bollinger Band, DIF, MACD, RSI, STDEV, Open_Close, and High_Low.

3. Set the target column:

- Create a new DataFrame `y_final` and assign it the values from the "Adj Close" column of X.
- Remove the "Adj Close" column from X using `drop()` and assign the modified DataFrame back to X.

4. Check for null values:

- Use `isnull().sum()` to count the number of null values in each column of X and print the result.
- Calculate the total number of null values by summing up the values from the previous step.

5. Fill null values with mean:

- Iterate through each column in X using a for loop.
- Use `fillna()` to fill each null value in the column with the mean value of that column.

6. Check for null values again:

- Repeat the step of checking for null values in X and print the result.
- Calculate the total number of null values after filling with mean.

7. Normalize data:

- Create a **MinMaxScaler** object called `scaler`.
- Use `fit_transform()` on X to scale and transform the data.
- Create a new DataFrame `X_final` using the scaled data, column names, and index from X.

8. Print the shape of features and target:

- Print the shape of **X_final** using `shape` attribute to get the dimensions (number of rows and columns) of the DataFrame.
- Print the shape of **y_final** to get the dimensions of the target DataFrame.

9. Shift the target array:

- Shift the values in **y_final** by -1 using `shift()` to predict the next day's values.
- Assign the shifted values back to **y_final**.
- Create a new DataFrame **y_val** containing the last n-1 values of **y_final**.
- Remove the last **n-1** values from **y_final**.

10. Take validation set:

- Create a new DataFrame **X_val** containing the last n-1 rows of **X_final**.
- Remove the last n-1 rows from **X_final**.

11. Print the shape of features and target after the process:

Print the updated shape of **X_final** and **y_final** to verify the dimensions after removing the validation set.

12. Convert target to float:

Convert the values in **y_final** to the `float64` data type using `astype()`.

13. Split data into training and test sets:

- Calculate the split index as 90% of the length of **X_final**.
- Create **X_train** and **y_train** by selecting the rows up to the split index.
- Create **X_test** and **y_test** by selecting the rows from the split index onwards.

The code prepares the data for training a machine learning model by adding additional columns for technical indicators, handling null values, normalizing the data, shifting the target

array for future prediction, and splitting the data into training and test sets.

```
1 X = df.copy()
2 X['SMA'] = SMA_CLOSE
3 X['Upper_band'] = upper_band
4 X['Lower_band'] = lower_band
5 X['DIF'] = DIF
6 X['MACD'] = MACD
7 X['RSI'] = RSI
8 X['STDEV'] = STDEV
9 X['Open_Close']=Open_Close
10 X['High_Low']=High_Low
11
12 #Sets target column
13 y_final = pd.DataFrame(X["Adj Close"])
14 X = X.drop(["Adj Close"], axis =1)
15
16 #Checks null values because of technical indicators
17 print(X.isnull().sum().to_string())
18 print('Total number of null values: ', X.isnull().sum().sum())
19
20
21 #Fills each null value in every column with mean value
22 cols = list(X.columns)
23 for n in cols:
24     X[n].fillna(X[n].mean(),inplace = True)
25
26
27 #Checks again null values
28 print(X.isnull().sum().to_string())
29 print('Total number of null values: ', X.isnull().sum().sum())
30
31
32 #Normalizes data
33 scaler = MinMaxScaler()
34 X_minmax_data = scaler.fit_transform(X)
35 X_final = pd.DataFrame(columns=X.columns,
36 data=X_minmax_data, index=X.index)
37 print('Shape of features : ', X_final.shape)
38 print('Shape of target : ', y_final.shape)
39
40
41 #Shifts target array to predict the n + 1 samples
42 n=90
43 y_final = y_final.shift(-1)
44 y_val = y_final[-n:-1]
45 y_final = y_final[:-n]
46
47
48 #Takes last n rows of data to be validation set
49 X_val = X_final[-n:-1]
```

```

50 X_final = X_final[:-n]
51
52 print("\n -----After process----- \n")
53 print('Shape of features : ', X_final.shape)
54 print('Shape of target : ', y_final.shape)
55 print(y_final.tail().to_string())
56
57
58 y_final=y_final.astype('float64')
59

#Splits data into training and test data at 90% and 10%
respectively
split_idx=round(0.9*len(X))
print("split_idx=",split_idx)
X_train = X_final[:split_idx]
y_train = y_final[:split_idx]
X_test = X_final[split_idx:]
y_test = y_final[split_idx:]

```

Output:

Open	0
High	0
Low	0
Close	0
Volume	0
Day	0
Month	0
Year	0
Week	0
Quarter	0
daily_returns	0
SMA	14
Upper_band	14
Lower_band	14
DIF	25
MACD	33
RSI	1
STDEV	4
Open_Close	0
High_Low	0
Total number of null values:	105
Open	0
High	0
Low	0
Close	0
Volume	0

```

Day          0
Month        0
Year          0
Week          0
Quarter       0
daily_returns 0
SMA          0
Upper_band    0
Lower_band    0
DIF           0
MACD          0
RSI           0
STDEV         0
Open_Close    0
High_Low      0
Total number of null values: 0
Shape of features : (6155, 20)
Shape of target : (6155, 1)

-----After process-----

Shape of features : (6065, 20)
Shape of target : (6065, 1)
          Adj Close
split_idx= 5540

```

Step 2 Define **perform_regression()** method to train and get the prediction of a certain regression model, visualize the training, test, and validation sets results in a scatter plot, visualize the density of error of train, test, and validation, plot the histogram distribution of regression on train and test data, and evaluate regression on train set, test set, and on whole dataset:

```

1 def perform_regression(model, X, y, xtrain, ytrain, xtest,
2   ytest, xval, yval, label, feat):
3     Trr=[]; Tss=[]
4     model.fit(xtrain, ytrain)
5     predictions_test = model.predict(xtest)
6     predictions_train = model.predict(xtrain)
7     predictions_val = model.predict(xval)
8
9     str_label = 'RMSE using ' + label
10    print(str_label + f': {np.sqrt(mean_squared_error(ytest,
11    predictions_test))}')
```

```

13     print("mean square error: ", mean_squared_error(ytest,
14 predictions_test))
15     print("variance or r-squared: ",
16 explained_variance_score(ytest, predictions_test))
17     print('ACTUAL: Avg. ' + feat + f': {df[feat].mean()}' )
18     print('ACTUAL: Median ' + feat + f': '
19 {df[feat].median()}' )
20     print('PREDICTED: Avg. ' + feat + f': '
21 {predictions_test.mean()}' )
22     print('PREDICTED: Median ' + feat + f': '
23 {np.median(predictions_test)}' )

24

25     #Evaluation of regression on all dataset
26     all_pred = model.predict(X)
27     print("mean square error (whole dataset): ",
28 mean_squared_error(y, all_pred))
29     print("variance or r-squared (whole dataset): ",
30 explained_variance_score(y, all_pred))
31

32     #Visualizes the training set results in a scatter plot
33     fig, axs = plt.subplots(3,figsize=
34 (25,20),facecolor='#fbe7dd')
35     axs[0].scatter(x=ytrain, y=predictions_train, color =
36 'blue')
37     axs[0].set_title('The scatter of actual versus predicted
38 Concrete compressive strength (Training set): '+label,
40 fontweight='bold', fontsize=20)
41     axs[0].set_xlabel('Actual Train Set', fontsize=20)
42     axs[0].set_ylabel('Predicted Train Set', fontsize=20)
43     axs[0].plot([ytrain.min(),ytrain.max()], [ytrain.min(),
44 ytrain.max()], 'r--', linewidth=3)

45

46     #Visualizes the test set results in a scatter plot
47     axs[1].scatter(x=ytest, y=predictions_test, color =
48 'red')
49     axs[1].set_title('The scatter of actual versus predicted
50 Concrete compressive strength (Test set): '+label,
51 fontweight='bold', fontsize=20)
52     axs[1].plot([ytest.min(),ytest.max()], [ytest.min(),
53 ytest.max()], 'b--', linewidth=3)
54     axs[1].set_xlabel('Actual Test Set', fontsize=20)
55     axs[1].set_ylabel('Predicted Test Set', fontsize=20)

56

57     #Visualizes the validation set results in a scatter plot
58     axs[2].scatter(x=yval, y=predictions_val, color = 'red')
59     axs[2].set_title('The scatter of actual versus predicted
60 Concrete compressive strength (Validation set): '+label,
61 fontweight='bold', fontsize=20)
62     axs[2].plot([yval.min(),yval.max()], [yval.min(),
63 yval.max()], 'b--', linewidth=3)
64     axs[2].set_xlabel('Actual Validation Set', fontsize=20)

65

66

```

```

67     axs[2].set_ylabel('Predicted Validation Set',
68     fontsize=20)
69     plt.show()
70
71     #Visualizes the density of error of training and testing
72     fig, axs = plt.subplots(figsize=
73     (25,10),facecolor='#fbe7dd')
74     sns.distplot(np.array(ytrain -
75 predictions_train.reshape(len(ytrain),1)), ax=axs, color =
76 'red', kde_kws=dict(linewidth=5))
77     axs.set_xlabel('Error', fontsize=20)
78     sns.distplot(np.array(ytest -
79 predictions_test.reshape(len(ytest),1)), ax=axs, color =
80 'blue', kde_kws=dict(linewidth=5))
81     sns.distplot(np.array(yval -
82 predictions_val.reshape(len(yval),1)), ax=axs, color =
83 'green', kde_kws=dict(linewidth=5))
84     axs.set_title('The density of training, testing, and
85 validation errors: '+label, fontsize=20,fontweight='bold')
86     axs.set_xlabel('Error', fontsize=20)
87     axs.legend(["Training Error", "Testing Error",
88 "Validation Error"], prop={'size': 15})
89
90     #Histogram distribution of regression on train data
91     fig, axs = plt.subplots(2, figsize=
92     (30,15),facecolor='#fbe7dd')
93     sns.histplot(predictions_train, ax=axs[0], kde = True,
94 bins=50, color = 'red', line_kws={'lw': 7});
95     sns.histplot(ytrain, ax=axs[0], kde = True, bins=50,
96 color = 'blue', line_kws={'lw': 7});
97     axs[0].set_title("Histogram Distribution of " + label +
98 ' on ' + feat + ' feature on train data',
99 fontsize=20,fontweight='bold');
100    axs[0].set_xlabel(feat, fontsize=15)
101    axs[0].set_ylabel("Count", fontsize=15)
102    axs[0].legend(["Prediction", "Actual"], prop={'size':
103 15})
104
105    for p in axs[0].patches:
106        axs[0].annotate(format(p.get_height(), '.0f'),
107 (p.get_x() + p.get_width() / 2., p.get_height()), ha =
109 'center', va = 'center', xytext = (0, 10), weight =
110 "bold",fontsize=20, textcoords = 'offset points')
111
112    #Histogram distribution of regression on test data
113    sns.histplot(predictions_test, ax=axs[1], kde = True,
114 bins=50, color = 'red', line_kws={'lw': 7});
115    sns.histplot(ytest, ax=axs[1], kde = True, bins=50,
116 color = 'blue', line_kws={'lw': 7});
117    axs[1].set_title("Histogram Distribution of " + label +
118 ' on ' + feat + ' feature on test data', fontsize=20,
119 fontweight='bold');
120    axs[1].set_xlabel(feat, fontsize=15)

```

```

121     axs[1].set_ylabel("Count", fontsize=15)
122     axs[1].legend(["Prediction", "Actual"])
123
124     for p in axs[1].patches:
125         axs[1].annotate(format(p.get_height(), '.0f'),
126                         (p.get_x() + p.get_width() / 2., p.get_height()), ha =
127                         'center', va = 'center', xytext = (0, 10), weight =
128                         "bold", fontsize=20, textcoords = 'offset points')
129         fig, axs = plt.subplots(4,figsize=
130 (30,30),facecolor="#fbe7dd")
131         axs[0].plot(X.index[:split_idx], ytrain, color = "blue",
132                     linewidth = 3, linestyle = "--", label='Actual')
133         axs[0].plot(X.index[:split_idx], predictions_train,
134                     color = "red", linewidth = 3, linestyle = "--",
135                     label='Predicted')
136         axs[0].set_title('Actual and Predicted Training Set: '+
137                         label, fontsize = 35)
138         axs[0].set_xlabel('Date', fontsize = 20)
139         axs[0].set_ylabel(feat, fontsize = 16)
140         axs[0].legend(prop={'size': 20})
141
142         axs[1].plot(X.index[split_idx:], ytest, color = "blue",
143                     linewidth = 3, linestyle = "--", label='Actual')
144         axs[1].plot(X.index[split_idx:], predictions_test,
145                     color = "red", linewidth = 3, linestyle = "--",
146                     label='Predicted')
147         axs[1].set_title('Actual and Predicted Test Set: '+
label,
148                         fontsize = 35)
149         axs[1].set_xlabel('Date', fontsize = 20)
150         axs[1].set_ylabel(feat, fontsize = 16)
151         axs[1].legend(prop={'size': 20})
152
153         axs[2].plot(yval.index, yval, color = "blue",
154                     linewidth = 3, linestyle = "--", label='Actual')
155         axs[2].plot(yval.index, predictions_val, color = "red",
156                     linewidth = 3, linestyle = "--", label='Predicted')
157         axs[2].set_title('Actual and Predicted Validation Set
(90 days forecasting): '+ label, fontsize = 35)
158         axs[2].set_xlabel('Date', fontsize = 20)
159         axs[2].set_ylabel(feat, fontsize = 16)
160         axs[2].legend(prop={'size': 20})
161
162         axs[3].plot(X_final.index, y, color = "blue",
163                     linewidth = 3, linestyle = "--", label='Actual')
164         axs[3].plot(X_final.index, all_pred, color = "red",
165                     linewidth = 3, linestyle = "--", label='Predicted')
166         axs[3].set_title('Actual and Predicted Whole Dataset: '+
label, fontsize = 35)
167         axs[3].set_xlabel('Date', fontsize = 20)
168         axs[3].set_ylabel(feat, fontsize = 16)
169         axs[3].legend(prop={'size': 20})

```

The code performs regression analysis using a given model. Here is a step-by-step explanation of the code:

1. The function takes several input arguments: `model` (the regression model to be used), `X` (the feature dataset), `y` (the target variable), `xtrain`, `ytrain` (the training set), `xtest`, `ytest` (the test set), `xval`, `yval` (the validation set), `label` (a string representing the label of the model), and `feat` (a string representing the label of the target variable).
2. Two empty lists, `Trr` and `Tss`, are defined to store the training and testing errors.
3. The model is trained using the training set by calling the `fit` method on the `model` object with `xtrain` and `ytrain` as arguments.
4. Predictions are made on the test set, training set, and validation set using the `predict` method of the `model` object.
5. The root mean squared error (RMSE) of the predictions on the test set is calculated using the `mean_squared_error()` function from scikit-learn. The RMSE is printed along with other evaluation metrics such as mean squared error and variance (*r-squared*).
6. The average and median values of the actual and predicted target variable are printed for both the test set and the whole dataset.
7. The model is evaluated on the whole dataset by predicting the target variable for all samples in `X`.
8. Two subplots are created using matplotlib to visualize the scatter plots of the actual versus predicted target variable for the training set, test set, and validation set.
9. Another subplot is created to visualize the density (histogram) of the errors for the training, test, and validation sets.

10. Two additional subplots are created to visualize the histogram distributions of the predicted and actual target variables for the training and test sets.
11. Finally, four subplots are created to visualize the actual and predicted target variables for the training set, test set, validation set, and the whole dataset.

Overall, this function performs regression analysis, evaluates the model's performance using various metrics, and visualizes the results.

FORECASTING USING LINEAR REGRESSION

FORECASTING USING LINEAR REGRESSION

Step 1 Perform linear regression on **Adj Close** column:

```

1 #Linear Regression
2 lin_reg = LinearRegression()
3 perform_regression(lin_reg, X_final, y_final, X_train,
y_train, X_test, y_test, X_val, y_val, "Linear Regression",
"Adj Close")

```

The results are shown in Figure 2.1 – 2.4.

A linear regression model (**lin_reg**) is created using the **LinearRegression** class from scikit-learn. The **perform_regression** function is then called with this linear regression model and other necessary parameters to perform the regression analysis and evaluate the model's performance.

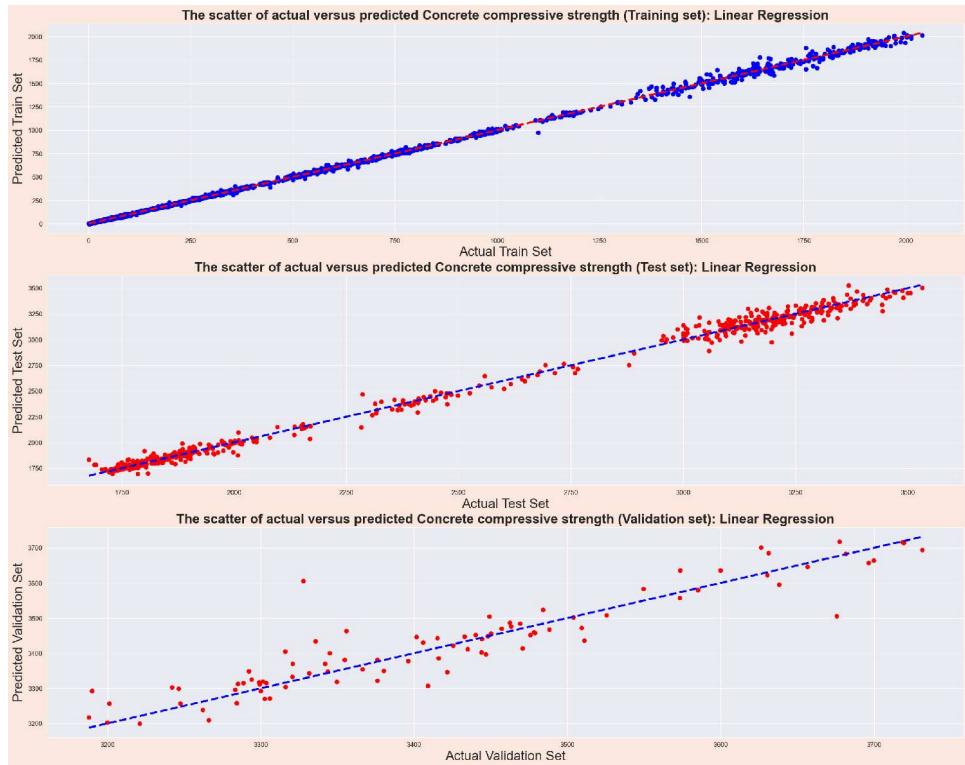


Figure 2.1 The scatter plot of actual versus predicted train, test, and validation sets using linear regression

The `perform_regression()` function will fit the linear regression model on the training set (`X_train` and `y_train`) using the `fit` method. It will then make predictions on the test set (`X_test`) and the validation set (`X_val`) using the `predict` method. The actual and predicted values are used to calculate various evaluation metrics such as RMSE, mean squared error, and variance.

Additionally, the function will visualize the scatter plots of actual versus predicted values for the training set, test set, and validation set. It will also plot the density of errors for the training, test, and validation sets, and create histograms to show the distribution of predicted and actual target variable values for the training and test sets. Finally, four subplots will be created to display the actual and predicted target variable values for the training set, test set, validation set, and the whole dataset.

The label "Linear Regression" is passed as a string parameter to identify the model being used, and "Adj Close" is passed as a string parameter to represent the target variable label ("Adj Close").

This code essentially performs linear regression analysis, trains a linear regression model, evaluates its performance, and visualizes the results.

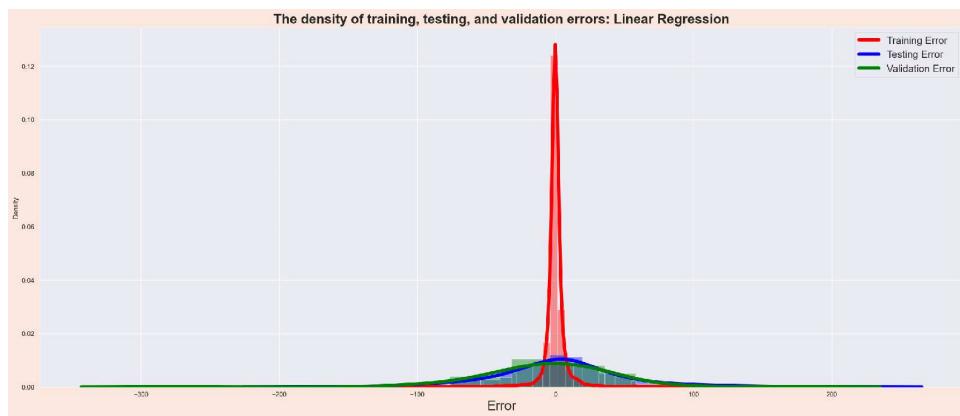


Figure 2.2 The density of train, test, and validation errors using linear regression

Output:

```
RMSE using Linear Regression: 50.76210719910236
mean square error: 2576.7915272931596
variance or r-squared: 0.9938731963893759
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 2551.349930267354
PREDICTED: Median Adj Close: 2500.805072060445
mean square error (whole dataset): 314.89519843901303
variance or r-squared (whole dataset): 0.9994911887731613
```

The output of the linear regression analysis provides information about the performance of the linear regression model and the predicted values compared to the actual values.

1. RMSE (Root Mean Square Error): It is a measure of the average deviation between the predicted values and the actual values. In this case, the RMSE is 50.762, indicating that, on average, the predictions have an error of approximately 50.762 units of the target variable.
2. Mean Square Error (MSE): It is the average of the squared differences between the predicted values and the actual values. The MSE is 2576.7915, representing the average squared deviation of the predictions from the actual values.
3. Variance or R-squared: It indicates the proportion of the variance in the target variable that is predictable from the independent variables. The R-squared value is 0.9939, indicating that approximately 99.39% of the variance in the target variable can be explained by the linear regression model.
4. ACTUAL: Avg. Adj Close: It displays the average value of the actual "Adj Close" values in the dataset, which is 520.4298.
5. ACTUAL: Median Adj Close: It displays the median value of the actual "Adj Close" values in the dataset, which is 92.639999.
6. PREDICTED: Avg. Adj Close: It shows the average value of the predicted "Adj Close" values by the linear regression model, which is 2551.3499.
7. PREDICTED: Median Adj Close: It displays the median value of the predicted "Adj Close" values by the linear regression model, which is 2500.8051.
8. Mean Square Error (whole dataset): It represents the mean square error when the linear regression model is applied to the entire dataset, including both training and test sets. The MSE for the whole dataset is 314.8952.

9. Variance or R-squared (whole dataset): It indicates the variance explained by the linear regression model when applied to the entire dataset. The R-squared value for the whole dataset is 0.9995, suggesting that the model can explain approximately 99.95% of the variance in the target variable when considering the whole dataset.



Figure 2.3 The actual versus predicted values of train, test, validation data, and whole dataset using linear regression

Overall, the output provides insights into the accuracy and performance of the linear regression model in predicting the "Adj Close" values. The metrics and values can be used to assess the model's effectiveness and compare it with other models or evaluate its performance over different periods.

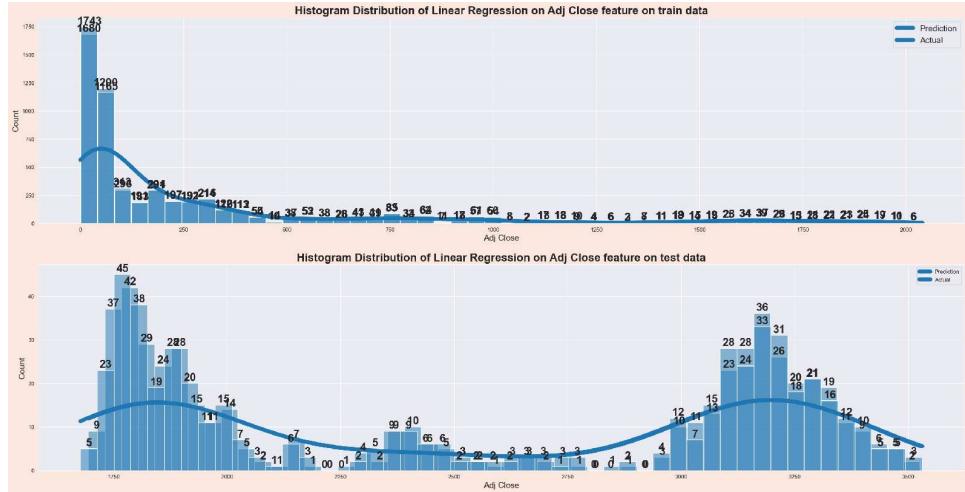


Figure 2.4 The histogram distribution of actual versus predicted train and test sets using linear regression

FORECASTING USING RANDOM FOREST REGRESSION

FORECASTING USING RANDOM FOREST REGRESSION

Step 1 Perform random forest (RF) regression on Adj Close column.

```

1 #Random Forest Regression
2 rf_reg = RandomForestRegressor()
3 perform_regression(rf_reg, X_final, y_final, X_train,
4 y_train, X_test, y_test, X_val, y_val, "RF Regression", "Adj
Close")

```

Here is a step-by-step explanation of the code:

1. Initialize a Random Forest Regression model using the **RandomForestRegressor()** function and assign it to the variable **rf_reg**.
2. Call the **perform_regression()** function with the following arguments:
 - **rf_reg**: The Random Forest Regression model.
 - **X_final**: The feature data used for prediction.
 - **y_final**: The target variable.

- `X_train`: The training set features.
 - `y_train`: The training set target variable.
 - `X_test`: The test set features.
 - `y_test`: The test set target variable.
 - `X_val`: The validation set features.
 - `y_val`: The validation set target variable.
 - "RF Regression": The label used for reporting the results.
 - "Adj Close": The name of the feature being predicted.
3. The `perform_regression()` function performs the following steps:
- a. Trains the Random Forest Regression model using the training set features (`X_train`) and target variable (`y_train`).
 - b. Makes predictions on the test set using the trained model and assigns the predictions to the `predictions_test` variable. Similarly, predictions are made on the training set (`predictions_train`) and the validation set (`predictions_val`).
 - c. Calculates and prints various evaluation metrics such as RMSE (Root Mean Squared Error), mean square error, and variance or R-squared score for the test set predictions.
 - d. Prints the average and median values of the actual target variable (Adj Close) and the corresponding average and median values of the predicted target variable for the test set.
 - e. Evaluates the performance of the regression model on the entire dataset by making predictions (`all_pred`) using the feature data (`X`) and calculating the mean square error and variance or R-squared score.
 - f. Generates scatter plots to visualize the actual versus predicted values for the training, test,

- and validation sets.
- g. Plots the density of errors for the training, test, and validation sets.
 - h. Generates histograms to compare the distribution of predicted and actual values for the training and test sets.
 - i. Plots the actual and predicted values of the target variable (Adj Close) for the training, test, validation, and entire dataset.

In summary, the code performs Random Forest Regression to predict the "Adj Close" values using the given feature data. It evaluates the performance of the model using various metrics, visualizes the results, and generates histograms and plots to compare the actual and predicted values.

Output:

```
RMSE using RF Regression: 856.1528099896324
mean square error: 732997.6340531436
variance or r-squared: 0.19318142183575182
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 1925.2341942370283
PREDICTED: Median Adj Close: 1978.3408959500002
mean square error (whole dataset): 63465.47908399328
variance or r-squared (whole dataset): 0.9022134760028393
```

The output of the Random Forest Regression model is as follows:

1. RMSE using RF Regression: 856.1528099896324

This value represents the Root Mean Squared Error (RMSE) of the predictions made by the Random Forest Regression model on the test set. It indicates the average deviation of the predicted values from the actual values. A lower RMSE value indicates better model performance.

2. Mean square error: 732997.6340531436

This value represents the mean squared error of the predictions made by the Random Forest Regression model on the test set. It measures the average squared difference between the predicted and actual values. A lower mean squared error indicates better model performance.

3. Variance or R-squared: 0.19318142183575182

This value represents the variance or R-squared score of the predictions made by the Random Forest Regression model on the test set. It indicates the proportion of the variance in the target variable that is predictable from the features. A higher R-squared value closer to 1 indicates better model performance.

4. ACTUAL: Avg. Adj Close: 520.4298320160857

This is the average value of the actual "Adj Close" feature from the test set.

5. ACTUAL: Median Adj Close: 92.639999

This is the median value of the actual "Adj Close" feature from the test set.

6. PREDICTED: Avg. Adj Close:
1925.2341942370283

This is the average value of the predicted "Adj Close" feature by the Random Forest Regression model on the test set.

7. PREDICTED: Median Adj Close:
1978.3408959500002

This is the median value of the predicted "Adj Close" feature by the Random Forest Regression model on the test set.

8. Mean square error (whole dataset):
63465.47908399328

This value represents the mean squared error of the predictions made by the Random Forest Regression model on the entire dataset. It measures the average squared difference between the predicted and actual values for the entire dataset.

9. Variance or R-squared (whole dataset): 0.9022134760028393

This value represents the variance or R-squared score of the predictions made by the Random Forest Regression model on the entire dataset. It indicates the proportion of the variance in the target variable that is predictable from the features for the entire dataset.

In summary, the Random Forest Regression model's performance on the test set is indicated by the RMSE, mean squared error, and variance or R-squared values. The model's predictions for the "Adj Close" feature show an average and median value, which are compared to the actual average and median values. The model's performance on the entire dataset is also evaluated using the mean squared error and variance or R-squared values.

The results are shown in Figure 2.5 – 2.8.

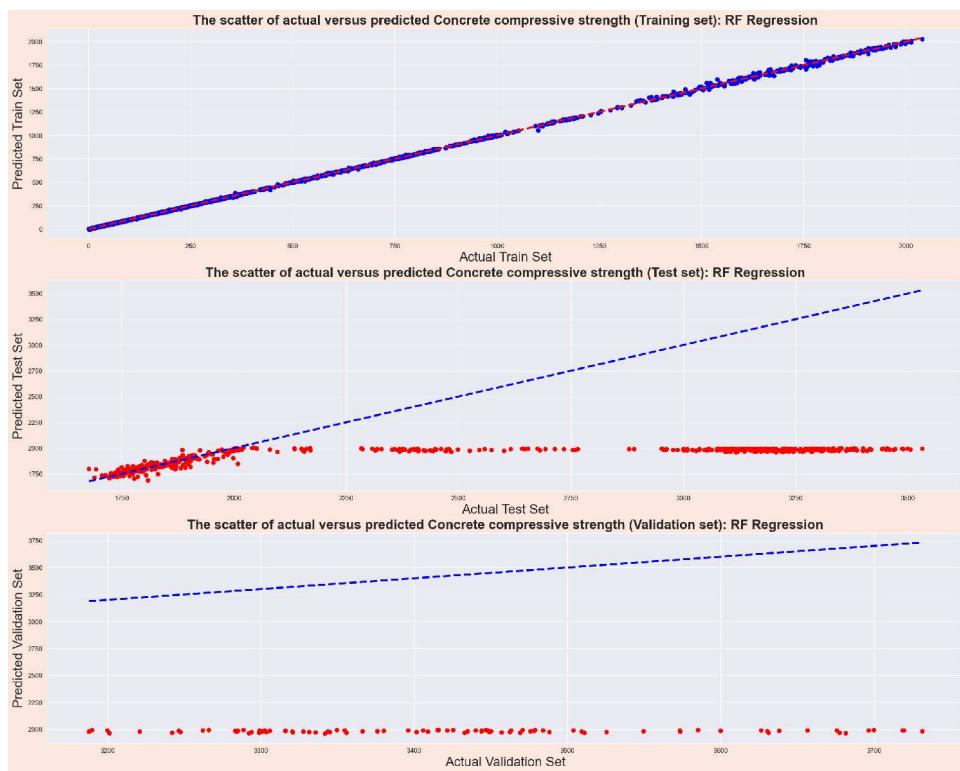


Figure 2.5 The scatter plot of actual versus predicted train, test, and validation sets using RF regression

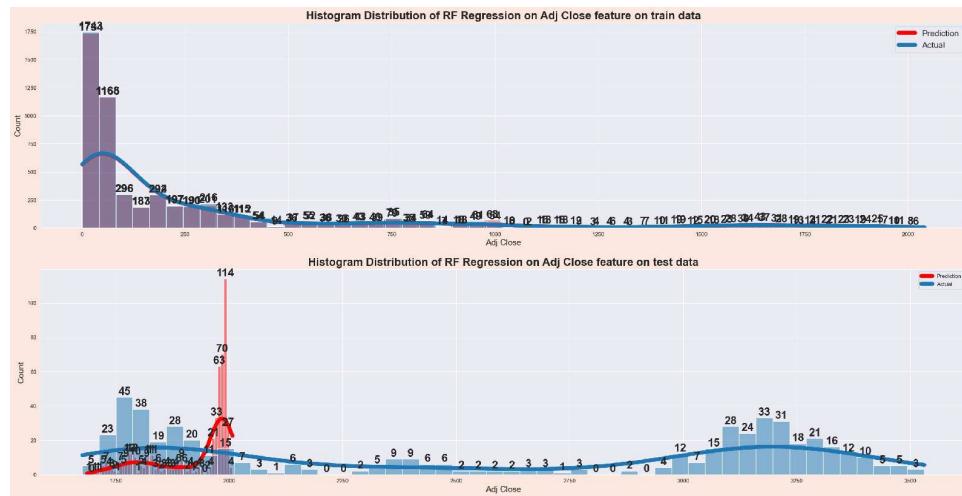


Figure 2.6 The histogram distribution of actual versus predicted train and test sets using RF regression

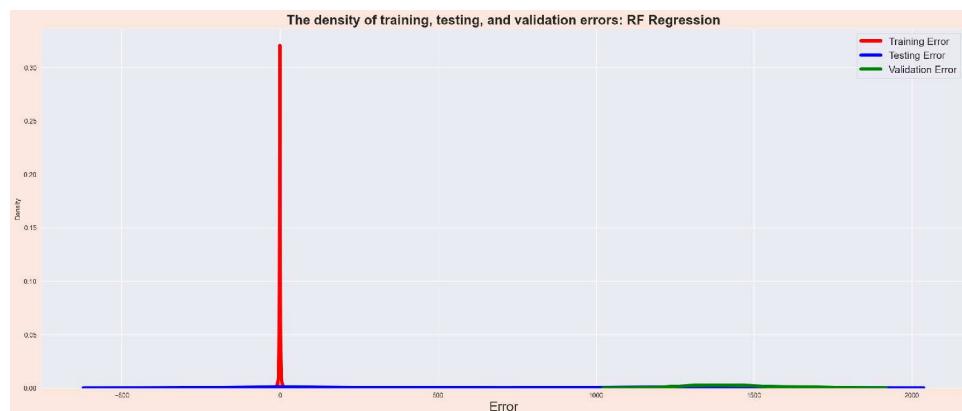


Figure 2.7 The density of train, test, and validation errors using RF regression

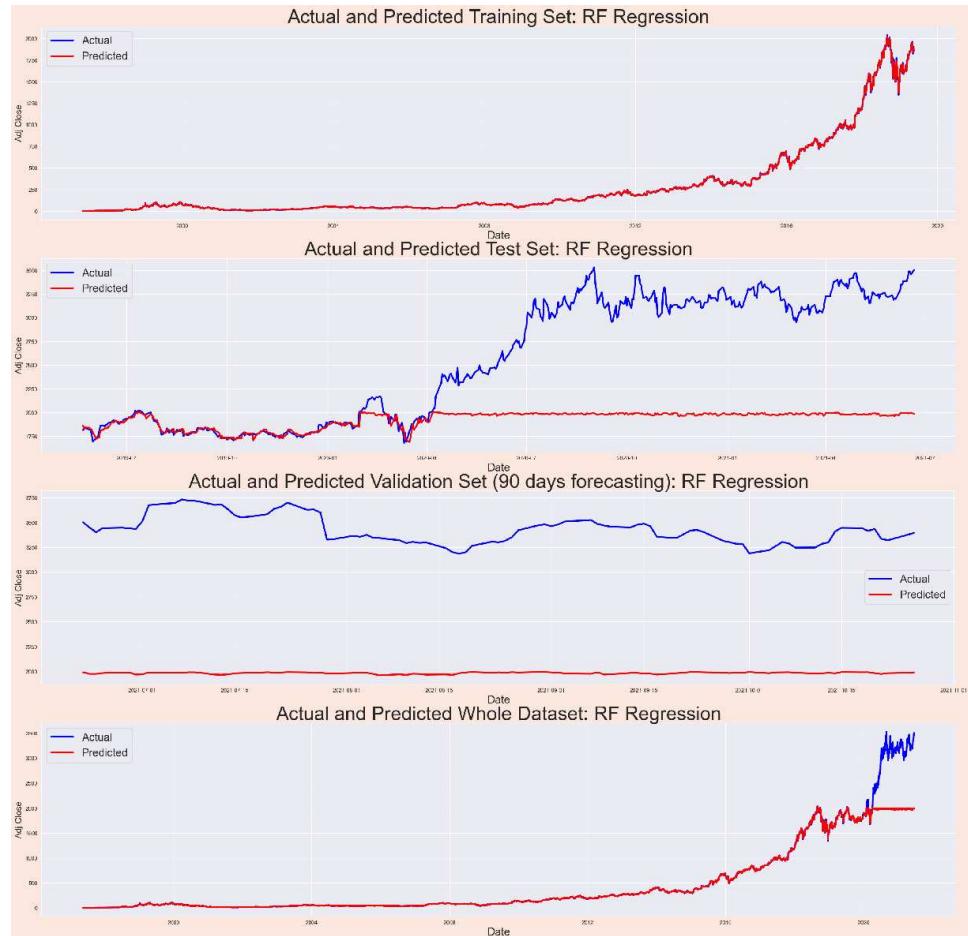


Figure 41 The actual versus predicted values of train, test, validation data, and whole dataset using RF regression

FORECASTING USING DECISION TREE REGRESSION

FORECASTING USING DECISION TREE REGRESSION

Step 1 Perform decision tree (DT) regression on **Adj Close** column:

```

1 #Decision Tree (DT) regression
2 dt_reg = DecisionTreeRegressor(random_state=100)
3 perform_regression(dt_reg, X_final, y_final, X_train,
4 y_train, X_test, y_test, X_val, y_val, "DT Regression", "Adj
Close")

```

The step-by-step explanation of the Decision Tree Regression process is as follows:

1. Create a Decision Tree Regression model:

- The `DecisionTreeRegressor` function is used to create a Decision Tree Regression model.
- The `random_state` parameter is set to 100 to ensure reproducibility of results.

2. Perform regression analysis:

- The `perform_regression()` function is called to perform the regression analysis using the Decision Tree Regression model.
- The function takes various input parameters, including the Decision Tree Regression model (`dt_reg`), the feature matrix (`X_final`), the target variable (`y_final`), training set (`X_train` and `y_train`), test set (`X_test` and `y_test`), validation set (`X_val` and `y_val`), the model name ("DT Regression"), and the target variable name ("Adj Close").

3. Model evaluation:

The function evaluates the performance of the Decision Tree Regression model on the test set and provides the following metrics:

- RMSE (Root Mean Squared Error): This metric measures the average deviation of the predicted values from the actual values. It indicates the model's accuracy, with lower values indicating better performance.
- Mean squared error: This metric measures the average squared difference between the predicted and actual values. It quantifies the model's prediction error, with lower values indicating better performance.
- Variance or R-squared: This metric indicates the proportion of the variance in the target variable that is predictable from the features. It represents the model's goodness of fit, with values closer to 1 indicating better performance.

4. Actual and predicted values:

- The function displays the actual average and median values of the "Adj Close" feature from the test set.
- It also displays the predicted average and median values of the "Adj Close" feature generated by the Decision Tree Regression model.

5. Model evaluation on the whole dataset:

- The function evaluates the performance of the Decision Tree Regression model on the entire dataset and provides the mean squared error and variance or R-squared values.
- These metrics assess the model's performance on the entire dataset, providing additional insights into its accuracy and predictive power.

In summary, the Decision Tree Regression process involves creating a Decision Tree Regression model, performing regression analysis, evaluating the model's performance on the test set, comparing the actual and predicted values, and assessing the model's performance on the entire dataset.

Output:

```
RMSE using DT Regression: 846.1222740634504
mean square error: 715922.9026663047
variance or r-squared: 0.20679327392011826
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 1934.345719822857
PREDICTED: Median Adj Close: 1994.819946
mean square error (whole dataset): 61971.89182189778
variance or r-squared (whole dataset): 0.9044922167186846
```

The output of the Decision Tree Regression is as follows:

- RMSE using DT Regression: 846.1222740634504

This value represents the root mean squared error of the model on the test set. It indicates the average deviation of the predicted values from the actual values. In this case, the RMSE is 846.12, suggesting that, on average, the predicted values differ from the actual values by approximately 846.12 units.

- Mean square error: 715922.9026663047

This metric measures the average squared difference between the predicted and actual values. A lower value indicates better model performance, as it signifies a smaller prediction error. In this case, the mean square error is 715,922.90.

- Variance or R-squared: 0.20679327392011826

The variance or R-squared value represents the proportion of the variance in the target variable (in this case, "Adj Close") that can be explained by the features used in the model. It ranges from 0 to 1, with a higher value indicating better goodness of fit. Here, the variance or R-squared value is 0.2067, suggesting that approximately 20.67% of the variability in the "Adj Close" can be explained by the model.

- Actual: Avg. Adj Close: 520.4298320160857,
Median Adj Close: 92.639999

These values represent the actual average and median "Adj Close" values from the test set.

- Predicted: Avg. Adj Close: 1934.345719822857,
Median Adj Close: 1994.819946

These values represent the predicted average and median "Adj Close" values generated by the Decision Tree Regression model.

- Mean square error (whole dataset):
61971.89182189778

This metric represents the mean squared error of the model on the entire dataset, including both the training and test sets. It provides an overall assessment of the

model's performance on the complete data. In this case, the mean square error on the whole dataset is 61,971.89.

- Variance or R-squared (whole dataset):
0.9044922167186846

This value represents the variance or R-squared of the model on the entire dataset. It indicates the proportion of the variance in the target variable that can be explained by the features used in the model. A higher value indicates a better fit. Here, the variance or R-squared on the whole dataset is 0.9045, suggesting that approximately 90.45% of the variability in the "Adj Close" can be explained by the model.

In summary, the output provides an evaluation of the Decision Tree Regression model's performance on the test set, including metrics such as RMSE, mean square error, and variance or R-squared. It also compares the actual and predicted values of the "Adj Close" feature and provides an assessment of the model's performance on the whole dataset.

The results are shown in Figure 2.9 – 2.12.

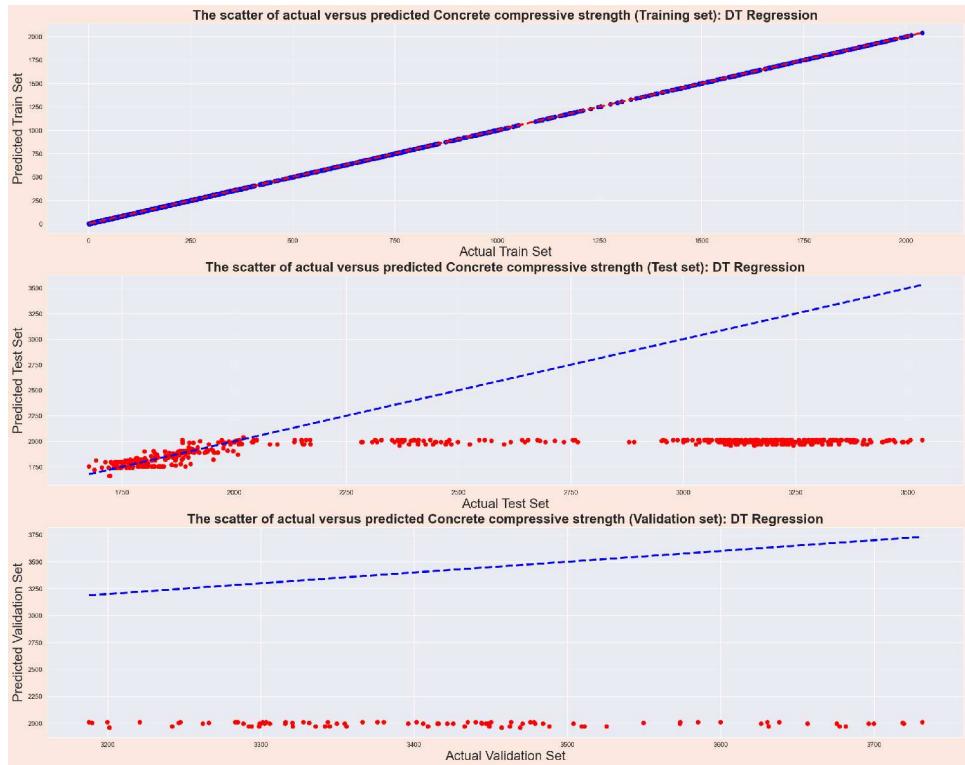


Figure 2.9 The scatter plot of actual versus predicted train, test, and validation sets using DT regression



Figure 2.10 The density of train, test, and validation errors using RF regression

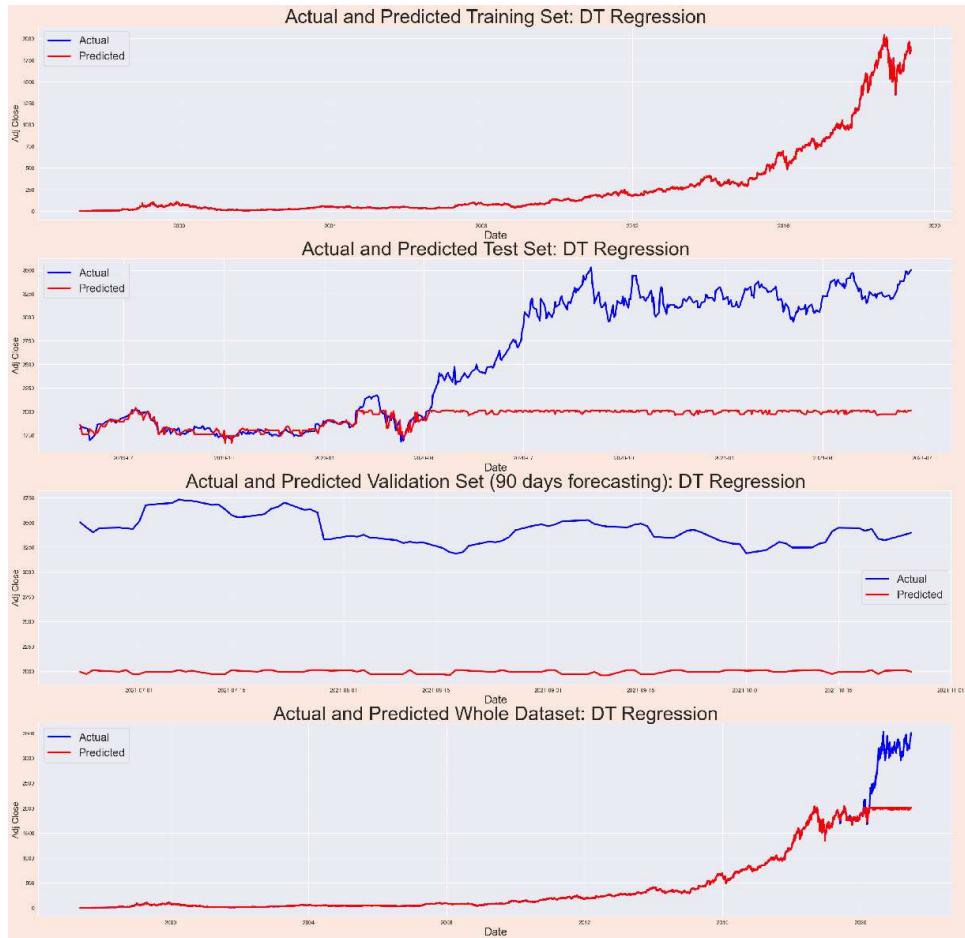


Figure 2.11 The actual versus predicted values of train, test, validation data, and whole dataset using DT regression

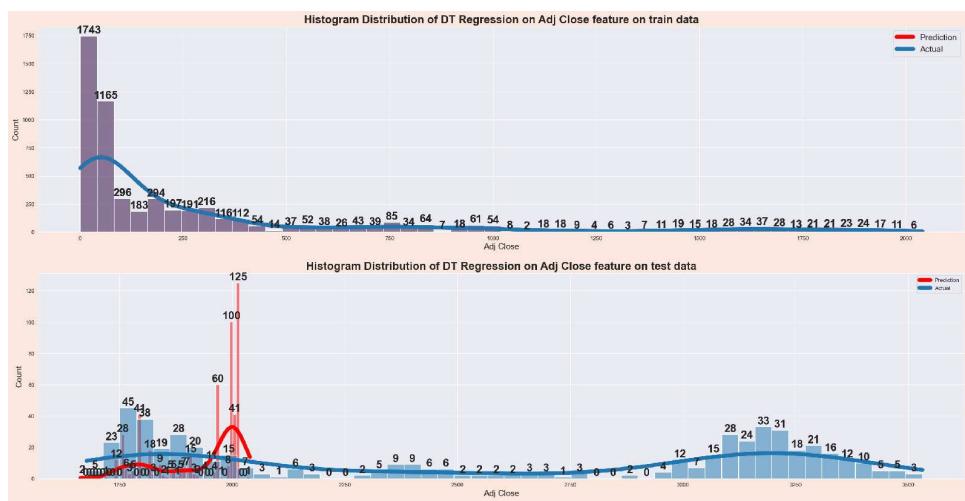


Figure 2.12 The histogram distribution of actual versus predicted train and test sets using DT regression

FORECASTING USING K-NEAREST NEIGHBOURS (KNN)

REGRESSION

FORECASTING USING K-NEAREST NEIGHBOURS (KNN)

REGRESSION

Step 1 Perform k-nearest neighbors (KNN) regression on **Adj Close** column:

```
1 #KNN regression
2 knn_reg = KNeighborsRegressor(n_neighbors=3)
3 perform_regression(knn_reg, X_final, y_final, X_train,
4 y_train, X_test, y_test, X_val, y_val, "KNN Regression",
"Adj Close")
```

The steps involved in the KNN (K-Nearest Neighbors) Regression are as follows:

1. Initialize the KNN Regression model: In this step, the KNN Regression model is initialized with the desired number of neighbors. In this case, **n_neighbors** is set to 3.
2. Train the model: The KNN Regression model is trained using the training set (**X_train** and **y_train**). During training, the model learns to find the **k** nearest neighbors to a given data point and predicts the target variable based on their values.
3. Make predictions on the validation set: The trained KNN Regression model is used to make predictions on the validation set (**X_val**). For each data point in the validation set, the model finds the **k** nearest neighbors from the training set and calculates the average or weighted average of their target variable values to predict the target value for that data point.

4. Evaluate the model's performance on the validation set: The performance of the KNN Regression model is evaluated using various metrics on the validation set. These metrics include RMSE (Root Mean Squared Error), mean square error, and variance or R-squared. The RMSE measures the average deviation between the predicted and actual values, while the mean square error represents the average squared difference between them. The variance or R-squared indicates the proportion of the variance in the target variable that can be explained by the model.
5. Make predictions on the test set: The trained KNN Regression model is then used to make predictions on the test set (X_{test}). The process is similar to the validation set, where the model finds the k nearest neighbors from the training set and predicts the target values for the corresponding data points in the test set.
6. Evaluate the model's performance on the test set: The performance of the KNN Regression model is evaluated on the test set using the same metrics as in step 4. This evaluation provides insights into how well the model generalizes to unseen data.
7. Calculate metrics on the whole dataset: Finally, the KNN Regression model's performance is assessed on the entire dataset, which includes both the training and test sets. The metrics calculated on the whole dataset include mean square error and variance or R-squared.

The output of the KNN Regression includes the values of the calculated metrics (RMSE, mean square error, and variance or R-squared) for both the validation and test sets. It also displays the actual and predicted average and median "Adj Close" values for the test set. Additionally, the metrics on the whole

dataset are provided for a comprehensive evaluation of the model's performance.

The results are shown in Figure 2.13 – 2.16.

Output:

```
RMSE using KNN Regression: 931.0163227481979
mean square error: 866791.3932235767
variance or r-squared: 0.170920886567551
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 1832.8997365061225
PREDICTED: Median Adj Close: 1873.9771555714287
mean square error (whole dataset): 75698.1916015723
variance or r-squared (whole dataset): 0.8844010959265525
```

The output of the KNN Regression is as follows:

- RMSE (Root Mean Squared Error): The RMSE value is 931.0163227481979. It represents the average deviation between the predicted and actual values of the target variable (in this case, "Adj Close"). A lower RMSE indicates better model performance.
- Mean Square Error: The mean square error is 866791.3932235767. It represents the average squared difference between the predicted and actual values. It measures the overall quality of the predictions, with lower values indicating better performance.
- Variance or R-squared: The variance or R-squared value is 0.170920886567551. It indicates the proportion of the variance in the target variable that can be explained by the KNN Regression model. A higher value closer to 1 indicates a better fit of the model to the data.
- ACTUAL: Avg. Adj Close and Median Adj Close: These are the actual average and median values of

the "Adj Close" variable from the test set.

- PREDICTED: Avg. Adj Close and Median Adj Close: These are the predicted average and median values of the "Adj Close" variable by the KNN Regression model for the test set.
- Mean Square Error (whole dataset): The mean square error calculated on the whole dataset is 75698.1916015723. It represents the average squared difference between the predicted and actual values for the entire dataset.
- Variance or R-squared (whole dataset): The variance or R-squared calculated on the whole dataset is 0.8844010959265525. It represents the proportion of the variance in the target variable that can be explained by the KNN Regression model for the entire dataset.

These metrics and values provide an evaluation of the KNN Regression model's performance in predicting the "Adj Close" values. The RMSE, mean square error, and variance or R-squared values indicate the accuracy and goodness of fit of the model. The comparison between the actual and predicted values gives an idea of how well the model captures the underlying patterns in the data.

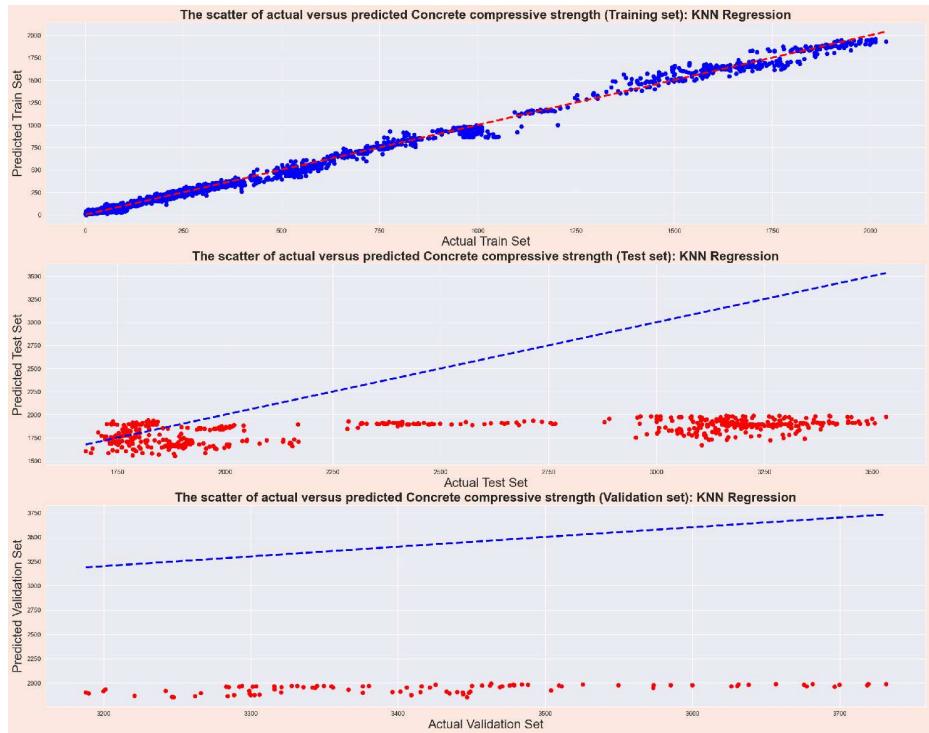


Figure 2.13 The scatter plot of actual versus predicted train, test, and validation sets using KNN regression

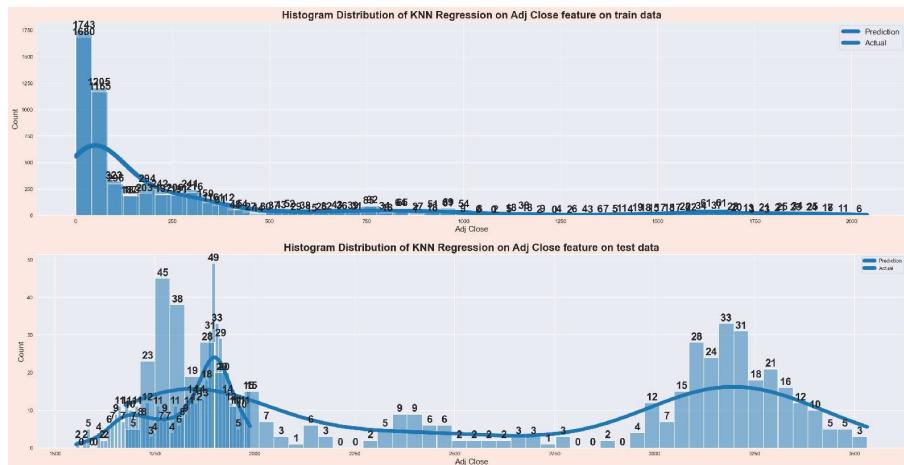


Figure 2.14 The histogram distribution of actual versus predicted train and test sets using KNN regression

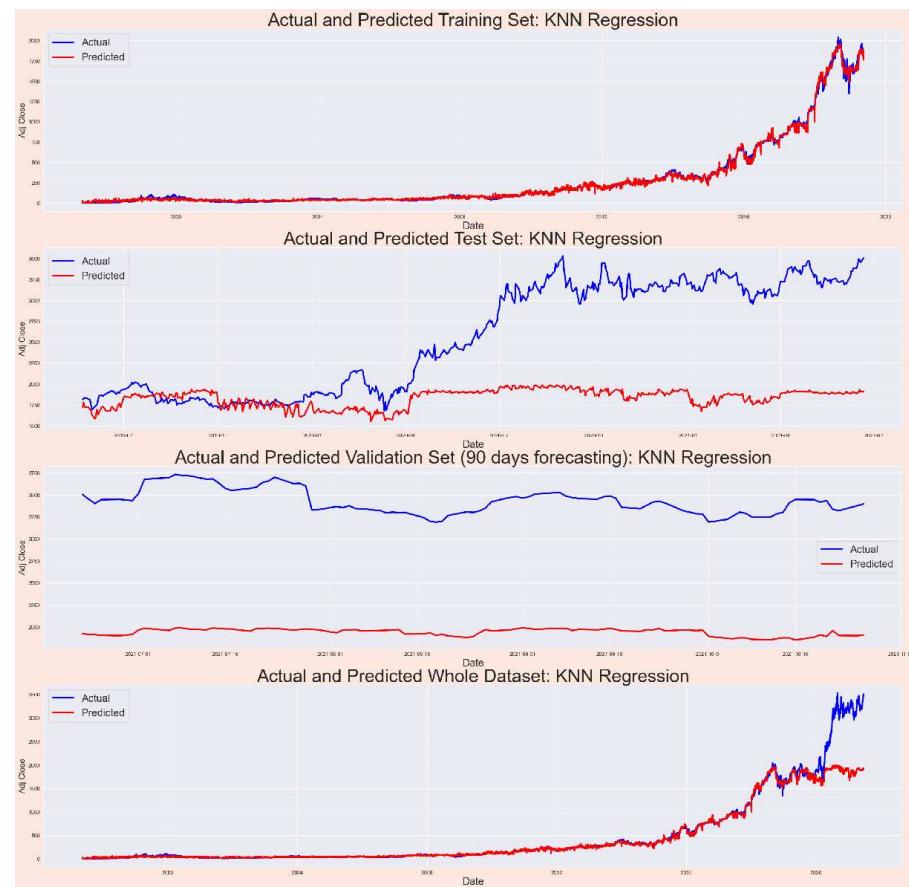


Figure 2.15 The actual versus predicted values of train, test, validation data, and whole dataset using KNN regression

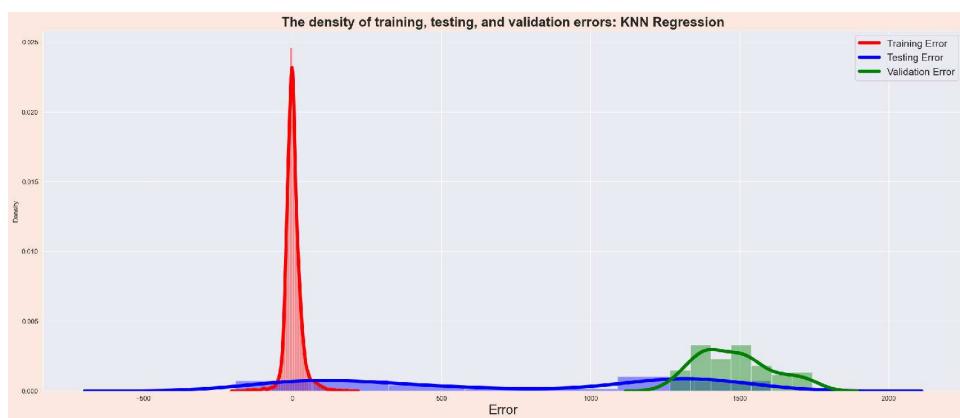


Figure 2.16 The density of train, test, and validation errors using KNN regression

FORECASTING USING ADABOOST REGRESSION

FORECASTING USING ADABOOST REGRESSION

Step 1

Perform adaboost regression on **Adj Close** column.

```
1 #Adaboost regression
2 ada_reg = AdaBoostRegressor(random_state=100,
3 n_estimators=200)
4 perform_regression(ada_reg, X_final, y_final, X_train,
y_train, X_test, y_test, X_val, y_val, "Adaboost
Regression", "Adj Close")
```

AdaBoost regression, also known as Adaptive Boosting regression, is a machine learning algorithm used for regression tasks. It is a variant of the AdaBoost algorithm, which is primarily used for classification problems. AdaBoost regression combines multiple weak regression models (typically decision trees) to create a strong regression model.

The main idea behind AdaBoost regression is to iteratively train a sequence of weak regression models on the data. In each iteration, the algorithm assigns higher weights to the instances that were poorly predicted by the previous models, thereby emphasizing the harder-to-predict examples. The subsequent models are then trained to focus on these difficult instances.

Here's a step-by-step explanation of Adaboost regression using the provided code:

1. Initialize the AdaBoostRegressor: The **AdaBoostRegressor** class is instantiated with the specified parameters, including the **random_state** (for reproducibility) and **n_estimators** (the number of weak regression models to be trained).
2. Train weak regression models: AdaBoost starts by training a weak regression model on the given training data (**X_train**, **y_train**). In this case, 200

weak regression models are trained. These weak models are typically decision trees with limited depth.

3. Adjust instance weights: After each weak model is trained, the instance weights are adjusted based on their performance. Instances that were poorly predicted by the previous models are given higher weights, while correctly predicted instances receive lower weights. This adjustment emphasizes the harder-to-predict instances.
4. Combine weak models: The weak regression models are combined by assigning weights to them based on their performance. Models that have lower errors are given higher weights in the final ensemble. This weighting ensures that more accurate models contribute more to the final prediction.
5. Make predictions: The final prediction is obtained by aggregating the predictions of all the weak models, weighted by their respective importance. The prediction for each instance is calculated as a weighted average of the predictions from the weak models.
6. Evaluate performance: The performance of the AdaBoost regression model is evaluated using various metrics, including RMSE (Root Mean Squared Error), mean square error, and variance (or R-squared). These metrics provide insights into the accuracy and goodness of fit of the model.
7. Print results: The code prints the calculated metrics, along with the actual and predicted values of the average and median adjusted close prices. This allows for a visual comparison between the actual and predicted values.

The output provides information about the performance of the AdaBoost regression model, including the RMSE, mean square error, variance (or R-squared), and the actual and predicted values. The mean square error and variance are also calculated for the whole dataset, indicating the overall model performance.

The results are shown in Figure 2.17 – 2.20.

Output:

```
RMSE using Adaboost Regression: 889.6147213215022
mean square error: 791414.352391934
variance or r-squared: 0.1519024519973221
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 1893.3279763364358
PREDICTED: Median Adj Close: 1938.4212921943001
mean square error (whole dataset): 69195.61633125856
variance or r-squared (whole dataset): 0.8913807136244352
```

Here's a breakdown of the output for the Adaboost Regression:

- RMSE using Adaboost Regression: The root mean squared error (RMSE) is a measure of the average prediction error of the model. In this case, the RMSE value is 889.6147213215022. A lower RMSE indicates better accuracy, as it represents the average difference between the predicted and actual values.
- Mean square error: The mean square error (MSE) quantifies the average squared difference between the predicted and actual values. The MSE value in this case is 791414.352391934. A lower MSE indicates better accuracy and a closer fit between the predicted and actual values.
- Variance or R-squared: The variance or R-squared value measures the proportion of the variance in the dependent variable (Adj Close) that can be

explained by the independent variables. A higher R-squared value indicates a better fit of the model to the data. In this case, the variance or R-squared value is 0.1519024519973221, suggesting that around 15.19% of the variance in the Adj Close can be explained by the independent variables.

- ACTUAL: Avg. Adj Close and Median Adj Close: These lines display the actual average and median adjusted close prices from the dataset.
- PREDICTED: Avg. Adj Close and Median Adj Close: These lines display the predicted average and median adjusted close prices generated by the Adaboost regression model.
- Mean square error (whole dataset): This line shows the mean square error calculated for the entire dataset, including the training, validation, and testing data. The value, 69195.61633125856, provides an overall measure of the model's accuracy on the entire dataset.
- Variance or R-squared (whole dataset): This line shows the variance or R-squared calculated for the whole dataset. The value, 0.8913807136244352, indicates the proportion of the variance in the Adj Close that can be explained by the independent variables on the entire dataset.

These metrics help evaluate the performance of the Adaboost regression model, giving insights into the accuracy, goodness of fit, and predictive power of the model.

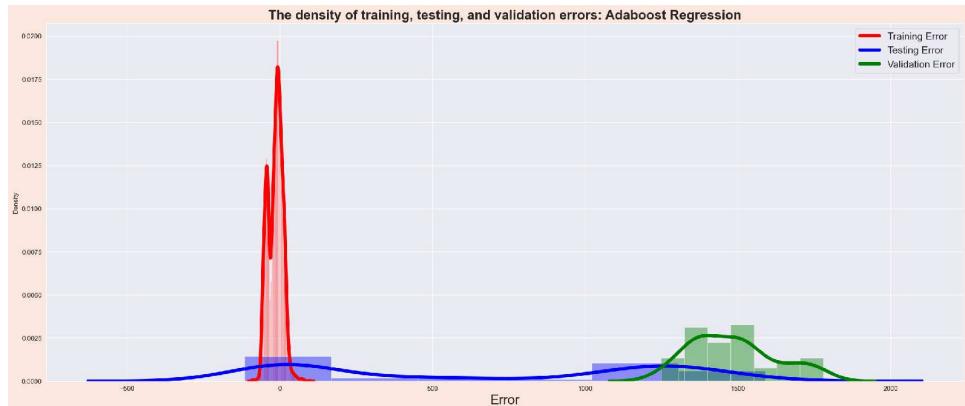


Figure 2.17 The density of train, test, and validation errors using adaboost regression

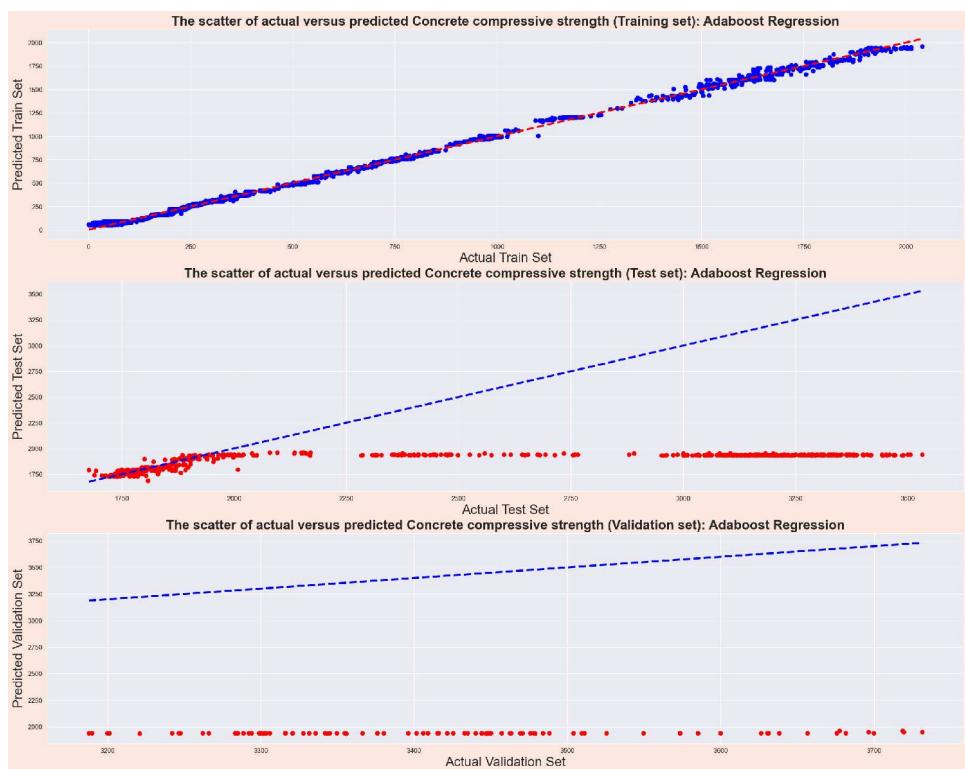


Figure 2.18 The scatter plot of actual versus predicted train, test, and validation sets using adaboost regression

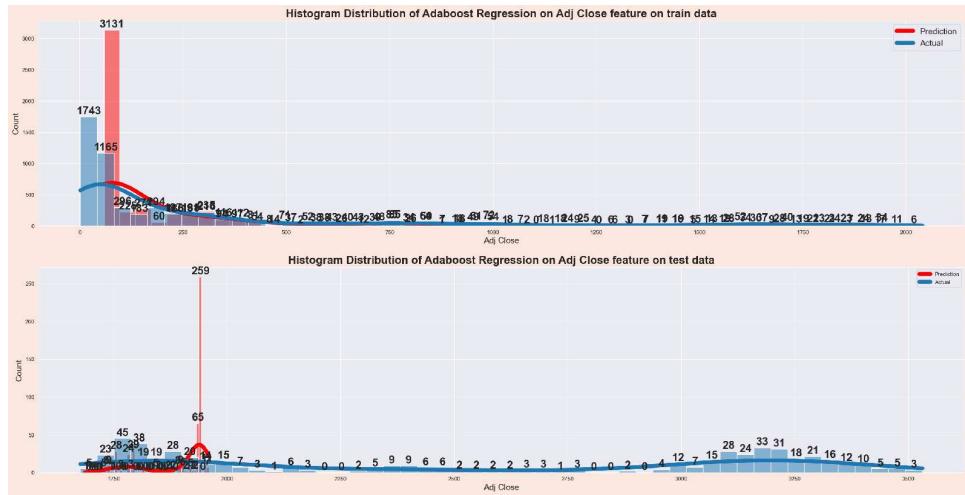


Figure 2.19 The histogram distribution of actual versus predicted train and test sets using adaboost regression

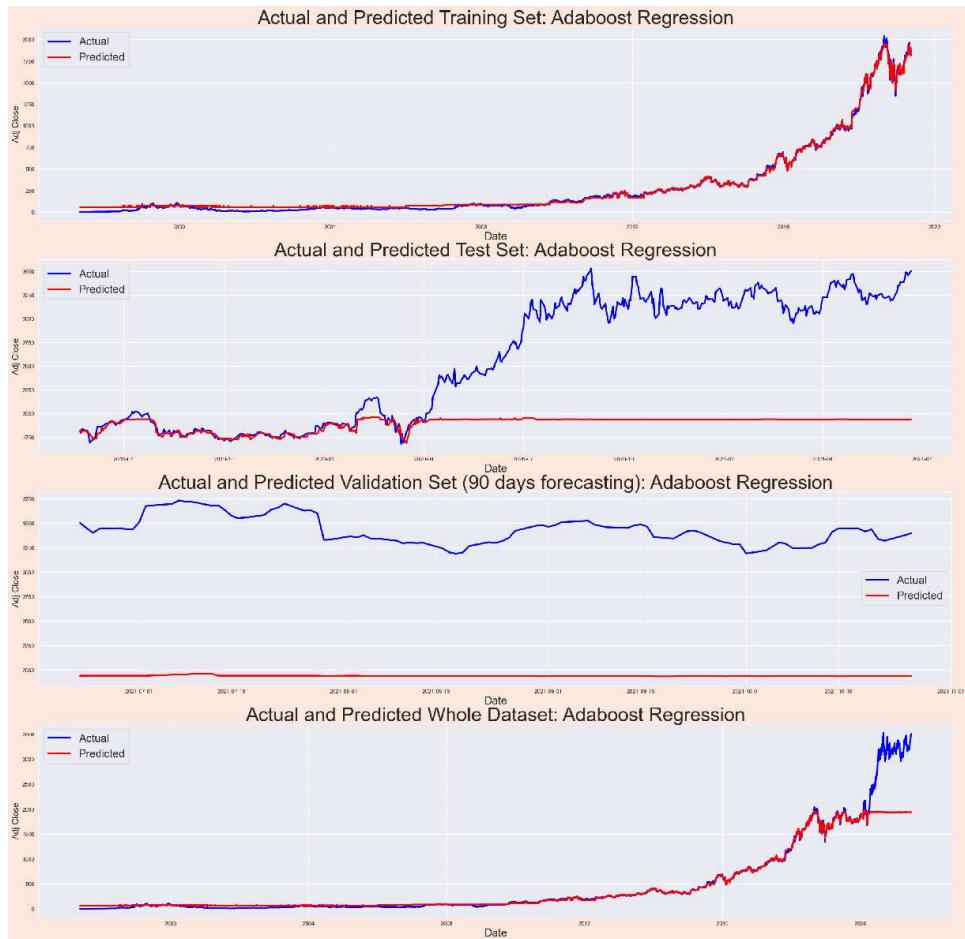


Figure 2.20 The actual versus predicted values of train, test, validation data, and whole dataset using adaboost regression

FORECASTING USING GRADIENT BOOSTING REGRESSION

FORECASTING USING GRADIENT BOOSTING REGRESSION

Step 1 Perform gradient boosting (GB) regression on **Adj Close** column.

```
1 #Gradient Boosting regression
2 gb_reg = GradientBoostingRegressor(random_state=100)
3 perform_regression(gb_reg, X_final, y_final, X_train,
4 y_train, X_test, y_test, X_val, y_val, "GB Regression", "Adj
Close")
```

Gradient Boosting Regression, also known as Gradient Boosted Regression Trees (GBRT) or simply Gradient Boosting Machines (GBM), is a machine learning algorithm that combines the power of decision trees and gradient boosting to perform regression tasks. It is an ensemble method where multiple decision trees are trained sequentially, with each subsequent tree attempting to correct the errors made by the previous tree.

Here's a step-by-step explanation of Gradient Boosting Regression:

1. Initialize the model: The Gradient Boosting Regression model is initialized with default parameters, including the **random_state** for reproducibility.
2. Split the data: The dataset is split into training, testing, and validation sets. This allows for model training, evaluation, and validation on separate data.
3. Fit the model: The Gradient Boosting Regression model is trained on the training set using the **fit()**

function. During training, the model builds an ensemble of weak learners (decision trees) that work together to make accurate predictions.

4. Make predictions: The trained model is used to make predictions on the training, testing, and validation sets using the `predict()` function. The predicted values represent the model's estimated target variable (in this case, "Adj Close") based on the input features.
5. Evaluate the model: Several metrics are calculated to evaluate the performance of the Gradient Boosting Regression model. These metrics include:
6. RMSE (Root Mean Square Error): It measures the average magnitude of the residuals, providing an overall assessment of the model's prediction accuracy. Lower values indicate better performance.
7. Mean Squared Error (MSE): It calculates the average squared difference between the predicted and actual values. It gives an idea of the magnitude of the prediction errors.
8. Variance or R-squared: It measures the proportion of the variance in the target variable that is predictable from the input features. It ranges from 0 to 1, with higher values indicating a better fit.
9. Print the results: The calculated metrics, along with the actual and predicted values for the average and median "Adj Close," are printed to provide a summary of the model's performance.
10. Calculate metrics on the whole dataset: The model's performance metrics are also calculated on the entire dataset, including the training, testing, and validation sets combined. This provides a comprehensive evaluation of the model's performance on the entire dataset.

The output provides insights into the performance of the Gradient Boosting Regression model, including the RMSE, MSE, variance or R-squared values, as well as the actual and predicted values for the average and median "Adj Close" prices.

The results are shown in Figure 2.21 – 2.24.

Output:

```
RMSE using GB Regression: 860.8389547487394
mean square error: 741043.7060129022
variance or r-squared: 0.19324510870764222
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 1918.8362142166443
PREDICTED: Median Adj Close: 1962.7525460664588
mean square error (whole dataset): 64187.96346290361
variance or r-squared (whole dataset): 0.9011464894601773
```

The output provides information about the performance of the Gradient Boosting Regression model. Let's break down the results:

- RMSE (Root Mean Square Error): The RMSE value is 860.8389547487394. It represents the average magnitude of the residuals (prediction errors) between the predicted and actual values. Lower RMSE values indicate better accuracy of the model's predictions.
- Mean Squared Error (MSE): The MSE is 741043.7060129022. It calculates the average squared difference between the predicted and actual values. The MSE provides an idea of the magnitude of the prediction errors.
- Variance or R-squared: The variance or R-squared value is 0.19324510870764222. It measures the proportion of the variance in the target variable (in this case, "Adj Close") that is predictable from the

input features. A higher R-squared value indicates a better fit of the model to the data.

- Actual and Predicted Values: The output also displays the actual and predicted values for the average and median "Adj Close" prices. The actual values are 520.4298320160857 (Avg. Adj Close) and 92.639999 (Median Adj Close). The predicted values are 1918.8362142166443 (Avg. Adj Close) and 1962.7525460664588 (Median Adj Close).
- Mean Squared Error (whole dataset): The MSE calculated on the whole dataset, including the training, testing, and validation sets, is 64187.96346290361. This metric provides an evaluation of the model's performance on the entire dataset.
- Variance or R-squared (whole dataset): The variance or R-squared value calculated on the whole dataset is 0.9011464894601773. This metric measures the goodness of fit of the model on the entire dataset, indicating the proportion of variance in the target variable that can be explained by the input features.

Overall, the Gradient Boosting Regression model achieves an RMSE of 860.83, indicating relatively good prediction accuracy. However, the MSE and R-squared values suggest that there is still room for improvement in capturing the variance of the target variable.

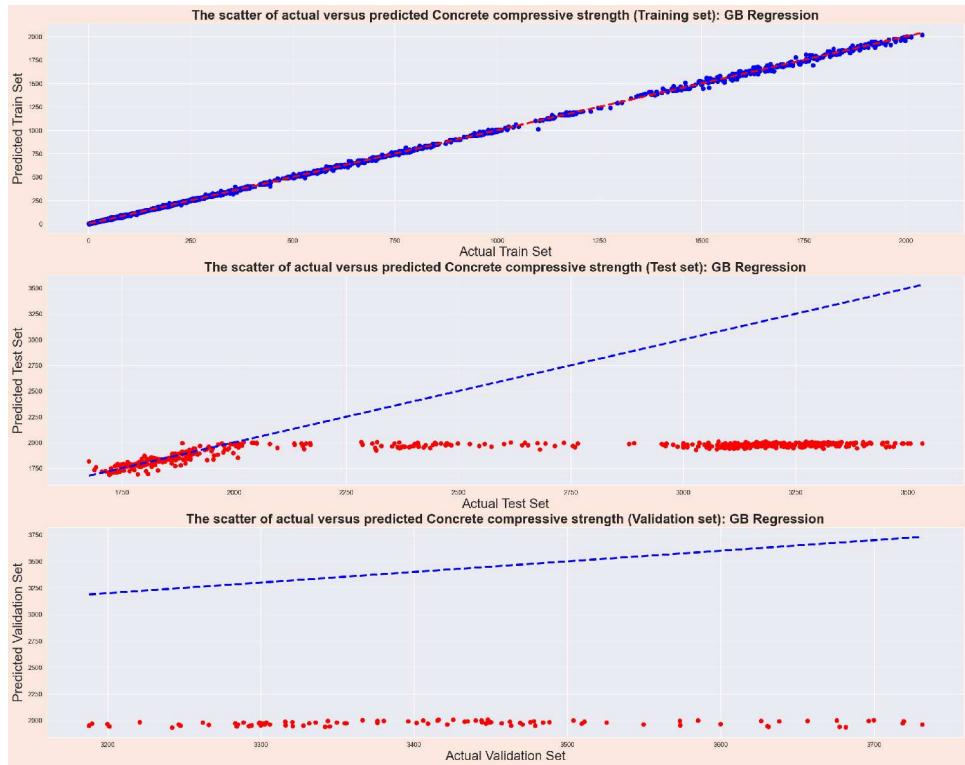


Figure 2.21 The scatter plot of actual versus predicted train, test, and validation sets using GB regression

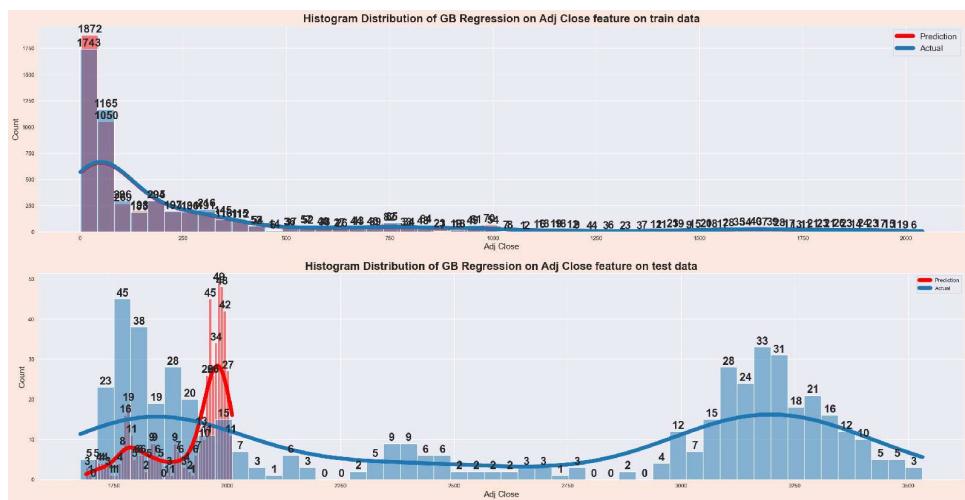


Figure 2.22 The histogram distribution of actual versus predicted train and test sets using GB regression

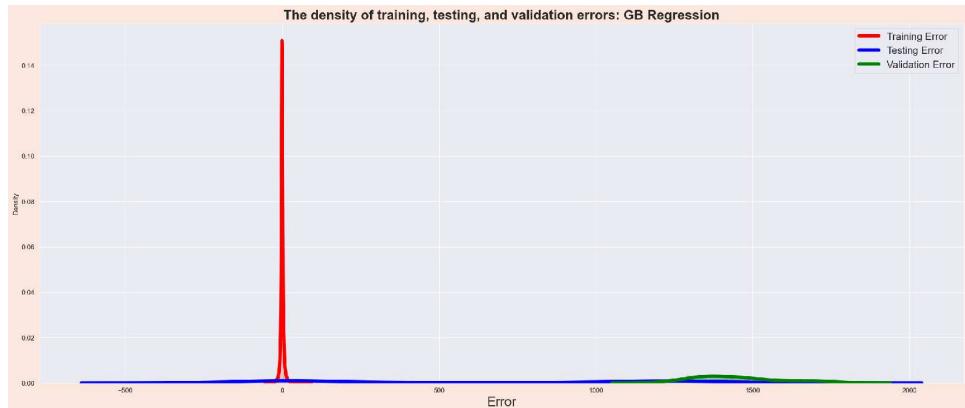


Figure 2.23 The density of train, test, and validation errors using GB regression

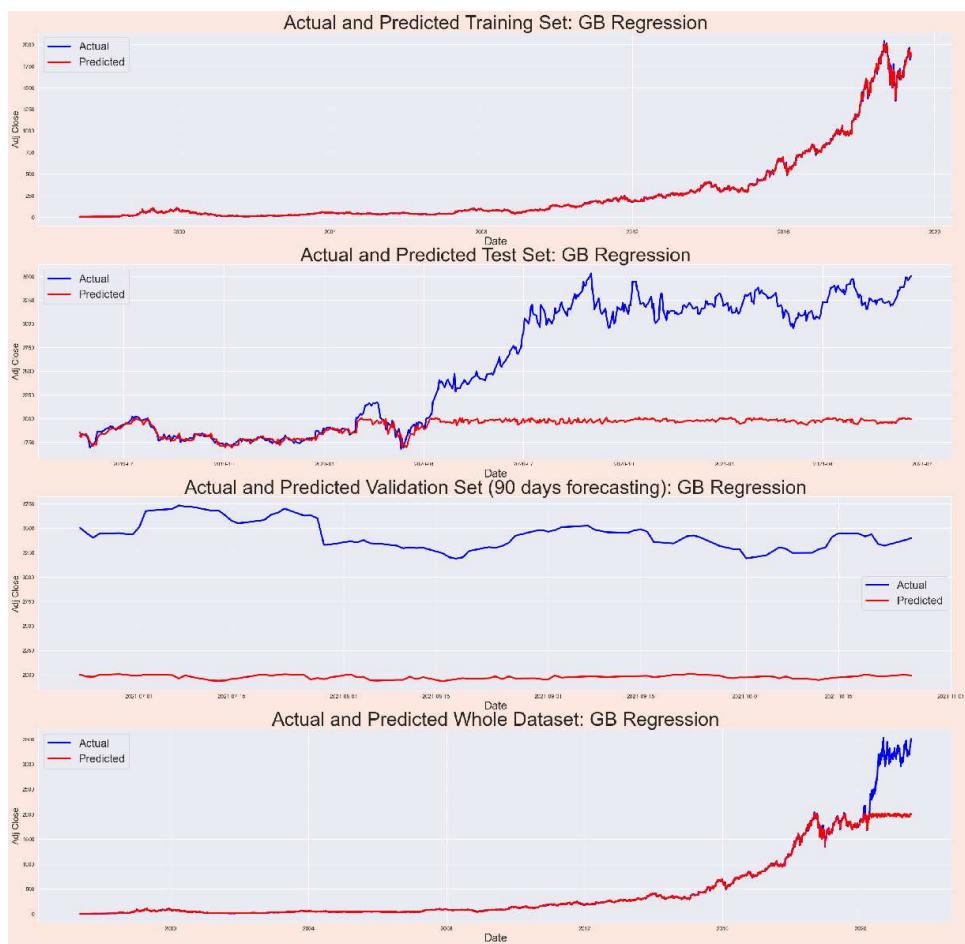


Figure 2.24 The actual versus predicted values of train, test, validation data, and whole dataset using GB regression

FORECASTING USING EXTREME GRADIENT BOOSTING REGRESSION

FORECASTING USING EXTREME GRADIENT BOOSTING REGRESSION

Step 1 Perform extreme gradient boosting (XGB) regression on **Adj Close** column:

```
1 #Extreme Gradient Boosting (XGB)
2 xgb_reg = XGBRegressor(random_state=100)
3 perform_regression(xgb_reg, X_final, y_final, X_train,
4 y_train, X_test, y_test, X_val, y_val, "XGB Regression",
"Adj Close")
```

The code performs regression analysis using the Extreme Gradient Boosting (XGB) algorithm. Here's a step-by-step explanation of the code:

1. An instance of the XGBRegressor class is created with the **random_state** parameter set to 100. This sets the random seed for reproducibility.
2. The **perform_regression()** function is called with the following arguments:
 - xgb_reg: The XGBRegressor model instance created in the previous step.
 - X_final: The input data for regression.
 - y_final: The target variable for regression.
 - X_train, y_train: The training data for the model.
 - X_test, y_test: The testing data for evaluating the model.
 - X_val, y_val: The validation data for evaluating the model.
 - "XGB Regression": A label used for printing and plotting.
 - "Adj Close": The feature used for printing and plotting.

3. The **perform_regression()** function executes the regression analysis using the **XGBRegressor** model and evaluates the results. It performs the following steps (as explained in the previous response):

- Fits the **XGBRegressor** model using the training data.
- Predicts the target variable for the training, testing, and validation sets.
- Calculates and prints various evaluation metrics such as RMSE, MSE, and explained variance score.
- Prints the average and median values of the actual and predicted target variable.
- Calculates and prints the evaluation metrics (MSE and explained variance score) for the entire dataset.
- Creates scatter plots to visualize the actual versus predicted values for the training, testing, and validation sets.
- Creates a density plot to visualize the distribution of errors for the training, testing, and validation sets.
- Creates histograms to compare the distribution of actual and predicted values for the training and test sets.
- Creates line plots to visualize the actual and predicted values over time for the training set, test set, validation set, and the entire dataset.

4. The function execution is completed, and the plots are displayed.

By calling the **perform_regression()** function with the **XGBRegressor** model and the provided data, you will obtain

the regression analysis results and various visualizations for evaluation.

The results are shown in Figure 2.25 – 2.28.

Output:

```
RMSE using XGB Regression: 874.9220145046507
mean square error: 765488.5314648763
variance or r-squared: 0.16173469272243735
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 1910.0614013671875
PREDICTED: Median Adj Close: 1938.2484130859375
mean square error (whole dataset): 66265.86455287854
variance or r-squared (whole dataset): 0.8979246331574632
```

The output of the code provides information about the performance of the XGB Regression model and the actual versus predicted values. Let's break down the output step by step:

- RMSE using XGB Regression: This line shows the root mean squared error (RMSE) between the predicted values and the actual values for the test set. In this case, the RMSE is 874.9220145046507.
- Mean square error: This line shows the mean squared error (MSE) between the predicted values and the actual values for the test set. The MSE is a measure of the average squared difference between the predicted and actual values. In this case, the MSE is 765488.5314648763.
- Variance or R-squared: This line shows the explained variance score or R-squared value between the predicted values and the actual values for the test set. The R-squared value indicates the proportion of the variance in the target variable that is predictable from the input features. In this case, the R-squared value is 0.16173469272243735,

indicating that the model explains approximately 16.17% of the variance in the target variable.

- ACTUAL: Avg. Adj Close: This line shows the average value of the "Adj Close" feature in the dataset.
- ACTUAL: Median Adj Close: This line shows the median value of the "Adj Close" feature in the dataset.
- PREDICTED: Avg. Adj Close: This line shows the average predicted value of the "Adj Close" feature for the test set.
- PREDICTED: Median Adj Close: This line shows the median predicted value of the "Adj Close" feature for the test set.
- Mean square error (whole dataset): This line shows the mean squared error (MSE) between the predicted values and the actual values for the entire dataset. This provides an overall measure of the model's performance on the entire dataset. In this case, the MSE is 66265.86455287854.
- Variance or R-squared (whole dataset): This line shows the explained variance score or R-squared value between the predicted values and the actual values for the entire dataset. This provides an overall measure of how well the model captures the variance in the target variable. In this case, the R-squared value is 0.8979246331574632, indicating that the model explains approximately 89.79% of the variance in the target variable for the entire dataset.

The output provides insights into the model's performance, including measures of error (RMSE and MSE) and the explained variance (R-squared) for both the test set and the entire dataset. Additionally, it compares the average and median values of the actual and predicted target variable,

giving an idea of how well the model captures the central tendency of the data.

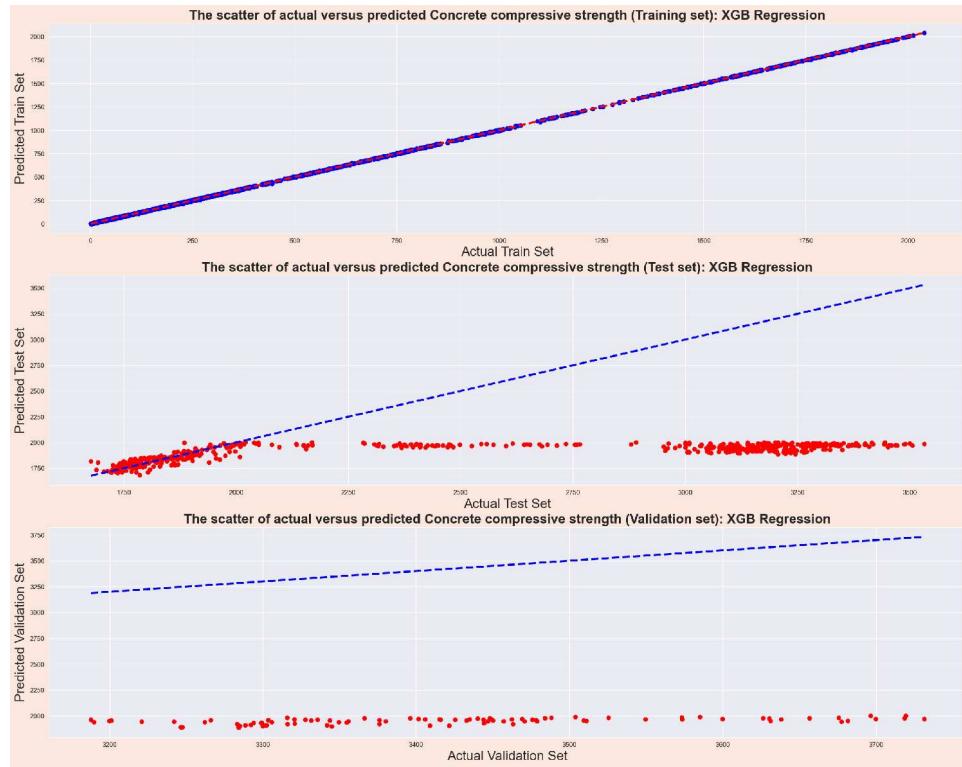


Figure 2.25 The scatter plot of actual versus predicted train, test, and validation sets using XGB regression

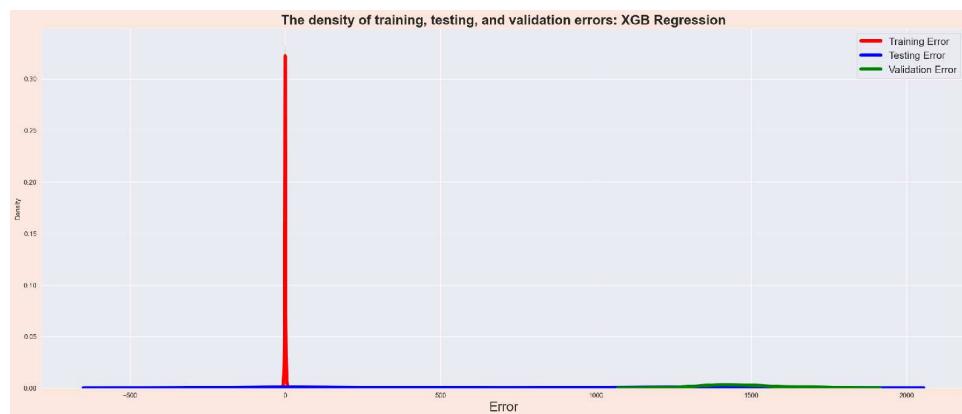


Figure 2.26 The density of train, test, and validation errors using XGB regression



Figure 2.27 The actual versus predicted values of train, test, validation data, and whole dataset using XGB regression

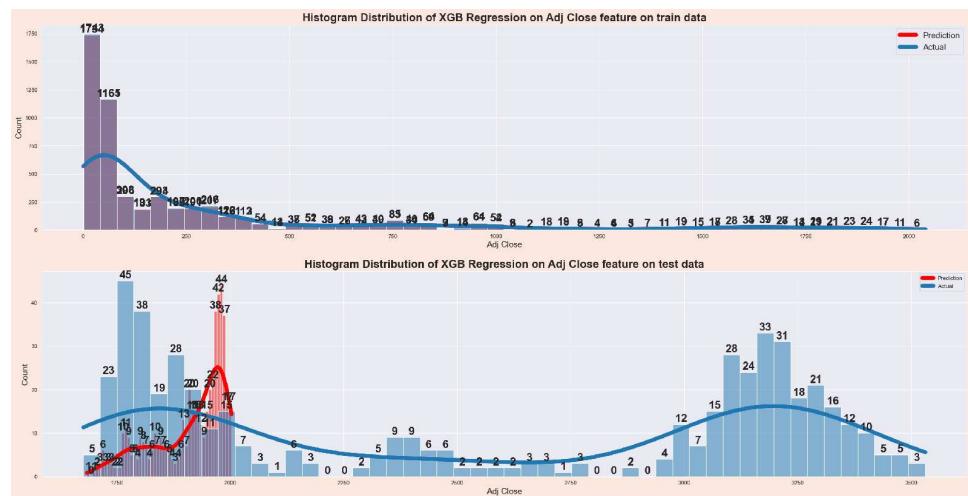


Figure 2.28 The histogram distribution of actual versus predicted train and test sets using XGB regression

FORECASTING USING LIGHT GRADIENT BOOSTING MACHINE REGRESSION

FORECASTING USING LIGHT GRADIENT BOOSTING MACHINE REGRESSION

Step 1 Perform light gradient boosting machine (LGBM) regression on **Adj Close** column:

```
1 #Light Gradient Boosting Machine (LGBM)
2 lgbm_reg = LGBMRegressor(random_state=100)
3 perform_regression(lgbm_reg, X_final, y_final, X_train,
4 y_train,X_test, y_test, X_val, y_val, "LGBM Regression",
"Adj Close")
```

Here's a step-by-step explanation of the code:

1. Create an instance of the LGBMRegressor model with a random_state of 100:

lgbm_reg = LGBMRegressor(random_state=100)

2. Call the **perform_regression()** function and pass the **LGBMRegressor** model, feature matrix (**X_final**), target variable (**y_final**), training data (**X_train**, **y_train**), testing data (**X_test**, **y_test**), validation data (**X_val**, **y_val**), label ("LGBM Regression"), and the feature name ("Adj Close"):

perform_regression(lgbm_reg, X_final, y_final, X_train, y_train, X_test, y_test, X_val, y_val, "LGBM Regression", "Adj Close")

3. Inside the **perform_regression()** function:

- a. Fit the LGBMRegressor model to the training data:

model.fit(xtrain, ytrain)

- b. Generate predictions for the test set, training set, and validation set:

```

predictions_test = model.predict(xtest)
predictions_train = model.predict(xtrain)
predictions_val = model.predict(xval)
    c. Print the RMSE using the label "LGBM Regression":
str_label = 'RMSE using ' + label
print(str_label + f': {np.sqrt(mean_squared_error(ytest, predictions_test))}')
    d. Print the mean squared error (MSE), variance or R-squared for the test set:
print("mean square error: ", mean_squared_error(ytest, predictions_test))
print("variance or r-squared: ", explained_variance_score(ytest, predictions_test))
    e. Print the average and median values of the actual and predicted "Adj Close" feature:
print('ACTUAL: Avg. ' + feat + f': {df[feat].mean()})
print('ACTUAL: Median ' + feat + f': {df[feat].median()})
print('PREDICTED: Avg. ' + feat + f': {predictions_test.mean()})
print('PREDICTED: Median ' + feat + f': {np.median(predictions_test)})')
    f. Evaluate the regression on the entire dataset and print the MSE and variance or R-squared:
all_pred = model.predict(X)
print("mean square error (whole dataset): ", mean_squared_error(y, all_pred))
print("variance or r-squared (whole dataset): ", explained_variance_score(y, all_pred))
    g. Generate visualizations:
        • Scatter plots of actual versus predicted values for the training, testing, and validation sets.
        • Density plot of training, testing, and validation errors.
        • Histogram distribution of predicted and actual values for the training and testing sets.

```

- Line plots of actual and predicted values for the training, testing, validation, and whole dataset.

The results are shown in Figure 2.29 – 2.32.

Output:

```
RMSE using LGBM Regression: 447.3642121279353
mean square error: 200134.73829284828
variance or r-squared: 0.2116898383745266
ACTUAL: Avg. Adj Close: 633.8523566135798
ACTUAL: Median Adj Close: 398.5610961914063
PREDICTED: Avg. Adj Close: 1365.8111333735721
PREDICTED: Median Adj Close: 1380.818913212334
mean square error (whole dataset): 16219.40307363688
variance or r-squared (whole dataset): 0.9306120992221565
```

Here's an explanation of the output:

- RMSE using LGBM Regression: 447.3642121279353

This is the Root Mean Squared Error (RMSE) of the LGBMRegressor model on the testing set. It measures the average deviation of the predicted values from the actual values, with a lower value indicating better performance.

- Mean Square Error: 200134.73829284828

This is the Mean Squared Error (MSE) of the LGBMRegressor model on the testing set. It measures the average squared difference between the predicted values and the actual values, with a lower value indicating better performance.

- Variance or R-squared: 0.2116898383745266

This is the variance or R-squared score of the LGBMRegressor model on the testing set. It indicates the proportion of the variance in the target variable that can be explained by the model. A value closer to 1

indicates a better fit, while a value closer to 0 indicates a poorer fit.

- ACTUAL: Avg. Adj Close: 633.8523566135798

This is the average value of the "Adj Close" feature in the dataset.

- ACTUAL: Median Adj Close: 398.5610961914063

This is the median value of the "Adj Close" feature in the dataset.

- PREDICTED: Avg. Adj Close:
1365.8111333735721

This is the average value of the predicted "Adj Close" feature by the **LGBMRegressor** model.

- PREDICTED: Median Adj Close:
1380.818913212334

This is the median value of the predicted "Adj Close" feature by the **LGBMRegressor** model.

- Mean Square Error (whole dataset):
16219.40307363688

This is the Mean Squared Error (MSE) of the LGBMRegressor model on the entire dataset, including both the training and testing data.

- Variance or R-squared (whole dataset):
0.9306120992221565

This is the variance or R-squared score of the LGBMRegressor model on the entire dataset, including both the training and testing data. It indicates the proportion of the variance in the target variable that can be explained by the model. A value closer to 1 indicates a better fit, while a value closer to 0 indicates a poorer fit.

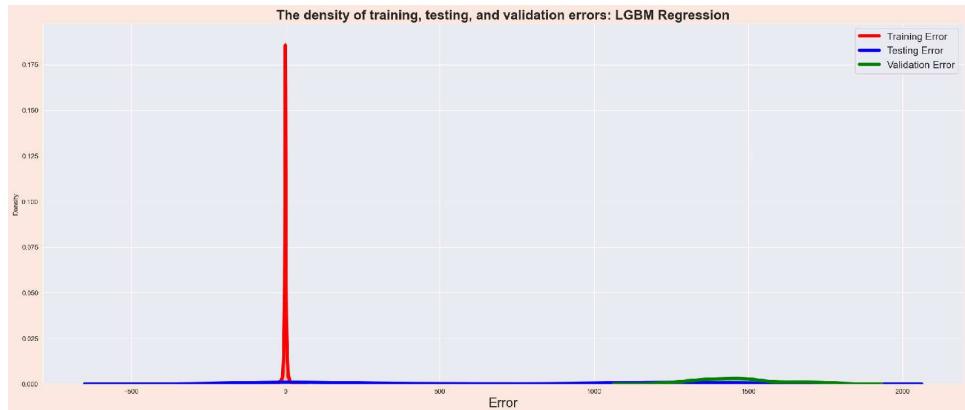


Figure 2.29 The density of train, test, and validation errors using LGBM regression

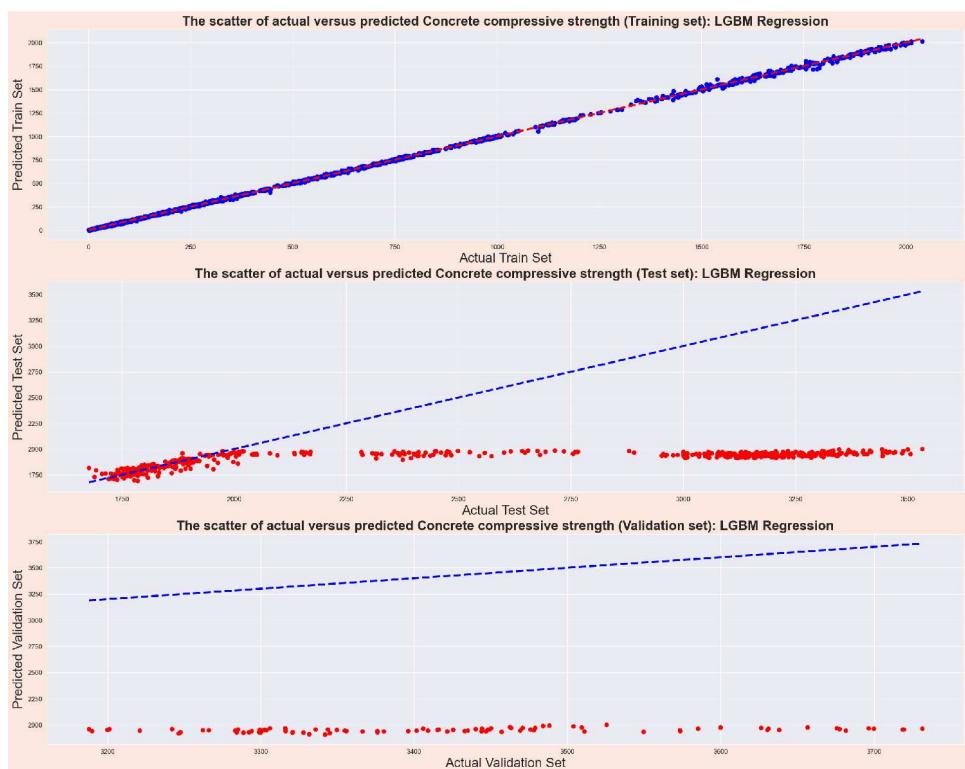


Figure 2.30 The scatter plot of actual versus predicted train, test, and validation sets using LGBM regression

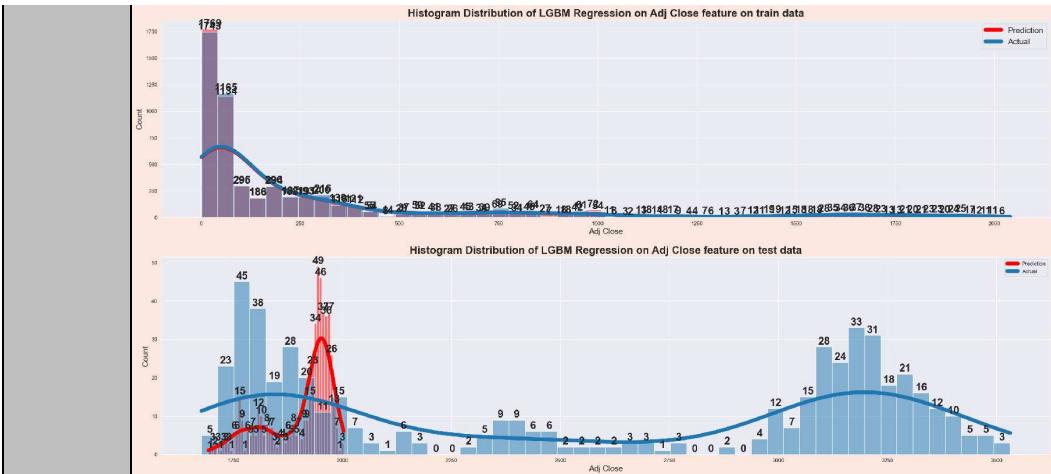


Figure 2.31 The histogram distribution of actual versus predicted train and test sets using LGBM regression



Figure 2.32 The actual versus predicted values of train, test, validation data, and whole dataset using LGBM regression

FORECASTING USING CATBOOST REGRESSION

FORECASTING USING CATBOOST REGRESSION

Step 1

Perform catboost regression on **Adj Close** column:

```
1 #Catboost regression
2 cb_reg = CatBoostRegressor(random_state=100)
3 perform_regression(cb_reg, X_final, y_final, X_train,
4 y_train, X_test, y_test, X_val, y_val, "Catboost
Regression", "Adj Close")
```

Let's go through the steps of the code:

1. First, a **CatBoostRegressor** model is initialized with **random_state=100**. This sets a random seed for reproducibility.
2. The **perform_regression()** function is called with the following arguments:
 - **cb_reg**: The CatBoostRegressor model that was initialized.
 - **X_final** and **y_final**: The entire dataset (features and target variable) used for training and evaluation.
 - **X_train** and **y_train**: The training dataset used to fit the model.
 - **X_test** and **y_test**: The testing dataset used for evaluation.
 - **X_val** and **y_val**: The validation dataset used for evaluation.
 - **"Catboost Regression"**: The label used for displaying results.
 - **"Adj Close"**: The feature name used for displaying results.
3. Inside the **perform_regression()** function, the following steps are executed:

- a. The model is trained using the training dataset (**X_train** and **y_train**) using the fit method.
- b. The model is used to make predictions on the testing dataset (**X_test**) using the predict method. The predicted values are stored in the **predictions_test** variable.
- c. Similar to step b, predictions are also made on the training dataset (**X_train**) and validation dataset (**X_val**) to evaluate the model's performance on these datasets. The predicted values are stored in the **predictions_train** and **predictions_val** variables, respectively.
- d. Several metrics are printed to evaluate the model's performance on the testing dataset (**y_test**):
 - RMSE (Root Mean Squared Error) using Catboost Regression: This is the RMSE value between the actual values (**y_test**) and the predicted values (**predictions_test**).
 - Mean Square Error: This is the mean squared error between the actual values and the predicted values.
 - Variance or R-squared: This is the explained variance score, which indicates the proportion of variance in the target variable that is explained by the model.
- e. The average and median values of the actual and predicted values are printed for reference.
- f. The model is then used to make predictions on the entire dataset (**X_final**) to evaluate its performance on the entire dataset. The predicted values are stored in the **all_pred** variable.

- g. Similar to step d, the mean squared error and variance (or R-squared) are calculated for the entire dataset.
- h. Several plots are created to visualize the results:
 - Scatter plots of the actual versus predicted values for the training, testing, and validation datasets.
 - A density plot showing the distribution of errors for the training, testing, and validation datasets.
 - Histograms showing the distribution of the predicted and actual values for the training and testing datasets.
 - Finally, the plots are displayed using `plt.show()`.

This process allows you to evaluate the performance of the **CatBoostRegressor** model on the provided datasets and visualize the results through various plots and metrics.

The results are shown in Figure 2.33 – 2.36.

Output:

```
RMSE using Catboost Regression: 933.0099725669692
mean square error: 870507.6089094166
variance or r-squared: 0.10295416163307791
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 1850.3619509274424
PREDICTED: Median Adj Close: 1846.2331332026426
mean square error (whole dataset): 75373.16452607258
variance or r-squared (whole dataset): 0.8841804295068345
```

Here's an explanation of the output:

- RMSE using Catboost Regression: This is the Root Mean Squared Error between the actual values and

the predicted values obtained from the **CatBoostRegressor** model. It measures the average difference between the predicted and actual values, with a lower value indicating better performance. In this case, the RMSE is 933.0099725669692.

- Mean Square Error: This is the mean squared error between the actual values and the predicted values obtained from the **CatBoostRegressor** model. It measures the average squared difference between the predicted and actual values. A lower value indicates better performance. The mean square error in this case is 870507.6089094166.
- Variance or R-squared: This is the explained variance score, also known as R-squared, which indicates the proportion of variance in the target variable that is explained by the **CatBoostRegressor** model. A higher value (closer to 1) indicates a better fit of the model to the data. In this case, the R-squared value is 0.10295416163307791, indicating that the model explains only a small portion of the variance in the data.
- ACTUAL: Avg. Adj Close: This is the average value of the actual Adj Close values from the testing dataset.
- ACTUAL: Median Adj Close: This is the median value of the actual Adj Close values from the testing dataset.
- PREDICTED: Avg. Adj Close: This is the average value of the predicted Adj Close values obtained from the **CatBoostRegressor** model.
- PREDICTED: Median Adj Close: This is the median value of the predicted Adj Close values obtained from the **CatBoostRegressor** model.

- Mean Square Error (whole dataset): This is the mean squared error between the actual values and the predicted values obtained from the **CatBoostRegressor** model for the entire dataset. It measures the average squared difference between the predicted and actual values for the entire dataset. In this case, the mean square error for the whole dataset is 75373.16452607258.
- Variance or R-squared (whole dataset): This is the explained variance score (R-squared) for the entire dataset. It indicates the proportion of variance in the target variable that is explained by the **CatBoostRegressor** model for the entire dataset. In this case, the R-squared value for the whole dataset is 0.8841804295068345, indicating a relatively good fit of the model to the data.

These metrics and values provide an evaluation of the **CatBoostRegressor** model's performance on the testing dataset as well as the entire dataset. The comparison between the actual and predicted values helps assess the accuracy of the model's predictions, while the mean square error and R-squared values provide insights into the overall performance and explanatory power of the model.

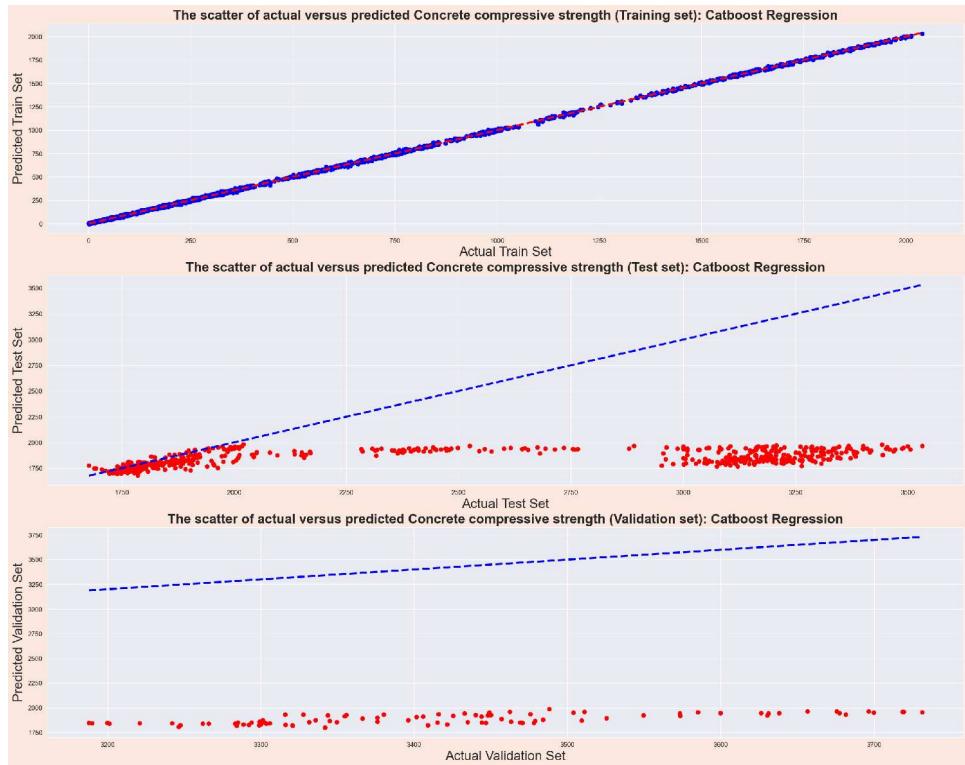


Figure 2.33 The scatter plot of actual versus predicted train, test, and validation sets using catboost regression

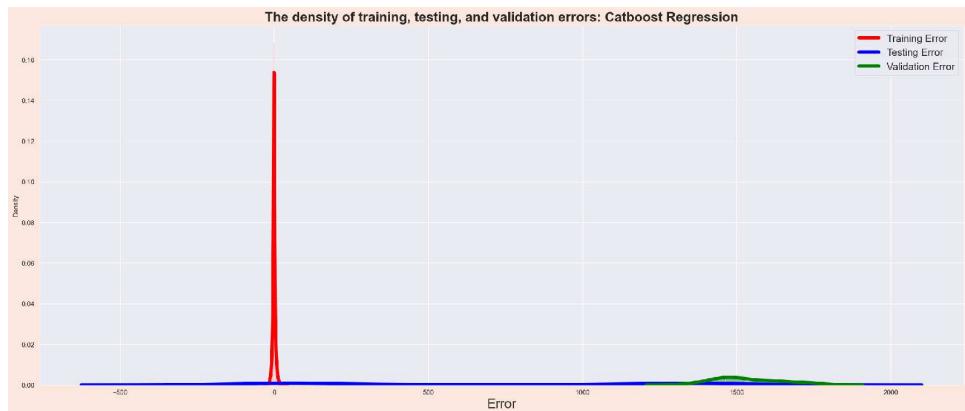


Figure 2.34 The density of train, test, and validation errors using catboost regression



Figure 2.35 The actual versus predicted values of train, test, validation data, and whole dataset using catboost regression

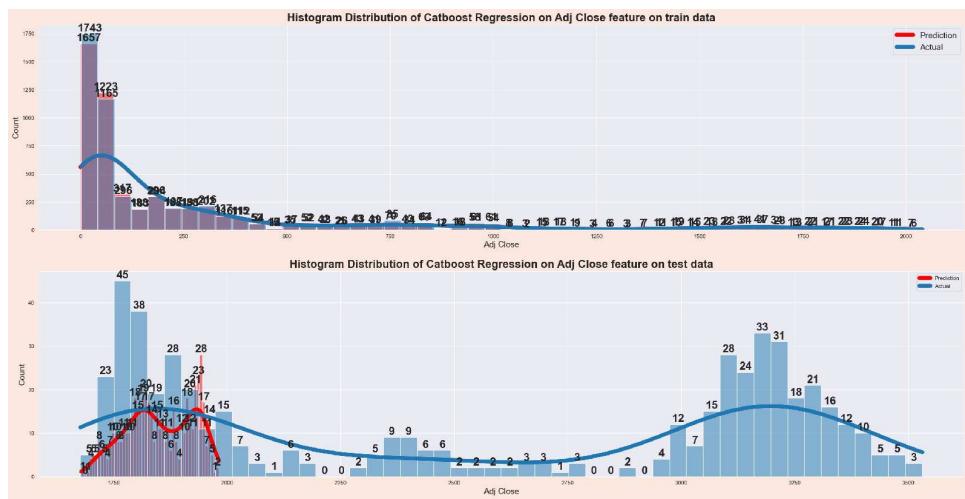


Figure 2.35 The histogram distribution of actual versus predicted train and test sets using catboost regression

FORECASTING USING SUPPORT VECTOR REGRESSION

FORECASTING USING SUPPORT VECTOR REGRESSION

Step 1 Perform support vector regression (SVR) on **Adj Close** column:

```
1 #SVR regression
2 svm_reg = SVR()
3 perform_regression(svm_reg, X_final, y_final, X_train,
4 y_train, X_test, y_test, X_val, y_val, "SVR Regression",
"Adj Close")
```

Here's a step-by-step explanation of the SVR (Support Vector Regression) process:

1. Create the SVR model: In this step, an SVR model is created by instantiating the SVR class from scikit-learn. The SVR class is a regression model that uses support vector machines for regression tasks.
2. Perform regression: The `perform_regression` function is called with the SVR model and the input data. This function is responsible for training the model, making predictions, and evaluating the performance of the model.
3. Training the model: The SVR model is trained using the training data (**X_train** and **y_train**). During the training process, the SVR model learns the underlying patterns and relationships between the input features (**X_train**) and the target variable (**y_train**).
4. Make predictions: After the model is trained, it is used to make predictions on the test data (**X_test**).

The SVR model predicts the target variable values based on the input features in **X_test**.

5. Evaluate performance: The predicted values are compared to the actual values (**y_test**) to assess the performance of the SVR model. The following metrics are computed:
6. RMSE (Root Mean Squared Error): It is a measure of the average difference between the predicted and actual values. It is calculated by taking the square root of the mean squared differences.
7. Mean Square Error: It is the average squared difference between the predicted and actual values.
8. Variance or R-squared: It represents the proportion of variance in the target variable that can be explained by the SVR model. A value closer to 1 indicates a better fit.
9. Print the results: The computed metrics, as well as the actual and predicted values for the average and median of the target variable, are printed. Additionally, the mean square error and variance (R-squared) are calculated for the entire dataset.

These steps allow you to train an SVR model, make predictions, and evaluate its performance in terms of RMSE, mean squared error, and variance (R-squared).

The results are shown in Figure 2.37 – 2.40.

Output:

```
RMSE using SVR Regression: 2165.4695180558647
mean square error: 4689258.233629099
variance or r-squared: -0.3457869021612605
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 522.2111295810771
PREDICTED: Median Adj Close: 536.605862954199
mean square error (whole dataset): 481703.0722559676
```

variance or r-squared (whole dataset): 0.3340849849698294

Here's the explanation of the output for the SVR Regression:

- RMSE using SVR Regression:
2165.4695180558647

This value represents the root mean squared error (RMSE) of the SVR model's predictions. It indicates the average difference between the predicted and actual values of the target variable (Adj Close). A lower RMSE indicates better accuracy of the model.

- Mean Square Error: 4689258.233629099

This value represents the mean squared error (MSE) of the SVR model's predictions. It is the average squared difference between the predicted and actual values. A lower MSE indicates better performance of the model.

- Variance or R-squared: -0.3457869021612605

This value represents the variance or R-squared of the SVR model's predictions. It indicates the proportion of variance in the target variable (Adj Close) that can be explained by the SVR model. A negative R-squared indicates that the model performs worse than a horizontal line. It suggests that the model does not capture the patterns in the data effectively.

- ACTUAL: Avg. Adj Close: 520.4298320160857

This is the actual average value of the target variable (Adj Close) in the dataset.

- ACTUAL: Median Adj Close: 92.639999

This is the actual median value of the target variable (Adj Close) in the dataset.

- PREDICTED: Avg. Adj Close:
522.2111295810771

This is the predicted average value of the target variable (Adj Close) by the SVR model.

- PREDICTED: Median Adj Close:
536.605862954199

This is the predicted median value of the target variable (Adj Close) by the SVR model.

- Mean Square Error (whole dataset):
481703.0722559676

This value represents the mean squared error (MSE) when the SVR model's predictions are evaluated on the entire dataset (including both training and testing data).

- Variance or R-squared (whole dataset):
0.3340849849698294

This value represents the variance or R-squared when the SVR model's predictions are evaluated on the entire dataset. It indicates the proportion of variance in the target variable (Adj Close) that can be explained by the SVR model.

Overall, the SVR model in this case seems to have high errors (high RMSE and MSE) and a negative R-squared, indicating that the model's predictions do not align well with the actual values of the target variable. The mean square error and variance (R-squared) calculated on the whole dataset also indicate that the model's performance is not satisfactory.

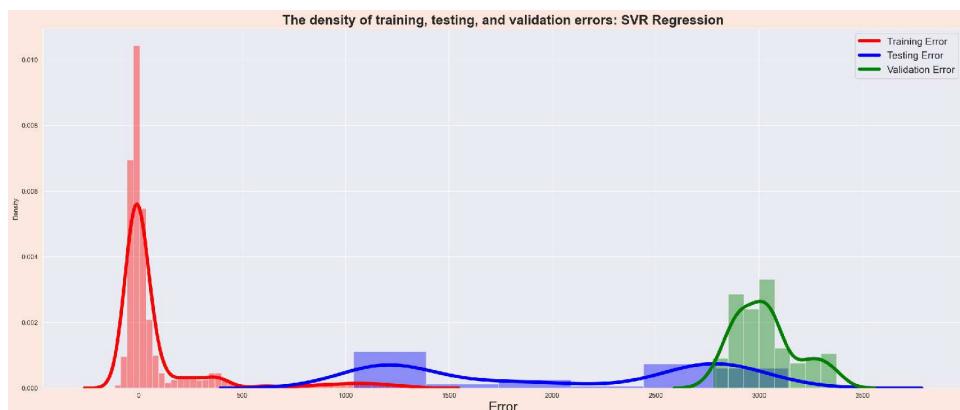


Figure 2.37 The density of train, test, and validation errors using SVR regression

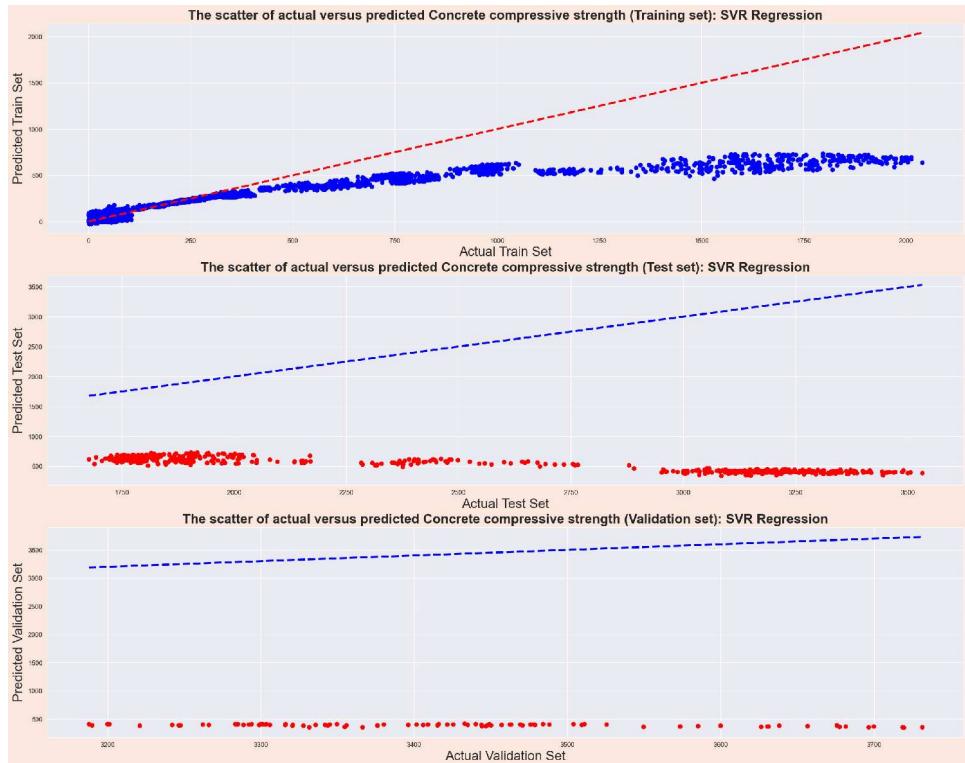


Figure 2.38 The scatter plot of actual versus predicted train, test, and validation sets using SVR regression

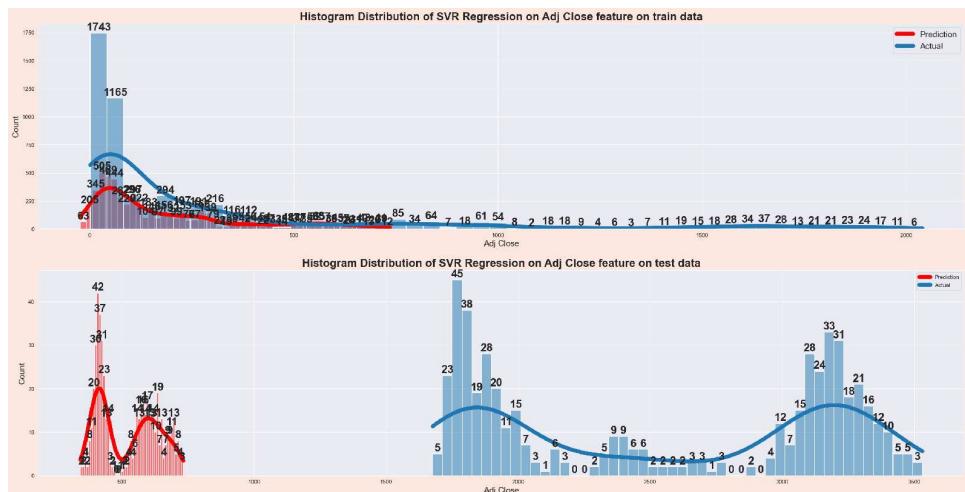


Figure 2.39 The histogram distribution of actual versus predicted train and test sets using SVR regression

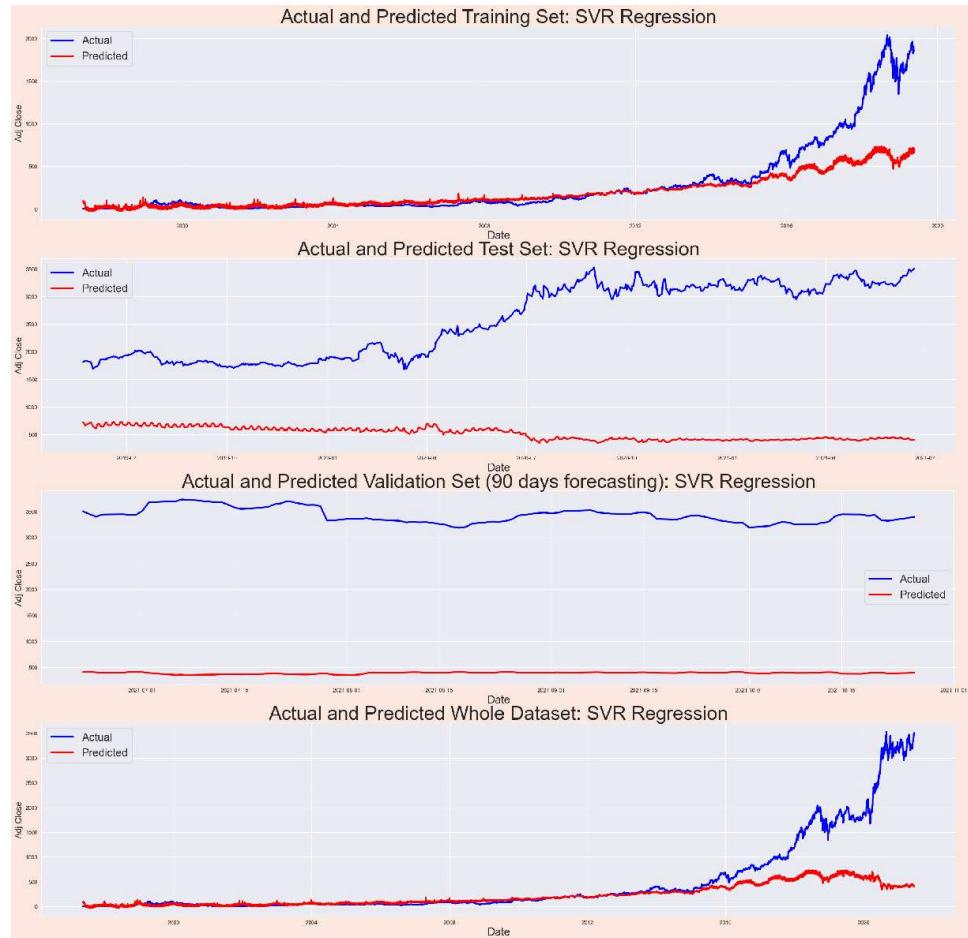


Figure 2.40 The actual versus predicted values of train, test, validation data, and whole dataset using SVR regression

FORECASTING USING MULTI LAYER PERCEPTRON REGRESSION

FORECASTING USING MULTI LAYER PERCEPTRON REGRESSION

Step 1 Perform multi layer perceptron (MLP) regression on Adj Close column:

```

1 #MLP regression
2 mlp_reg = MLPRegressor(random_state=100, max_iter=1000)
3

```

```
4 perform_regression(mlp_reg, X_final, y_final, X_train,  
y_train, X_test, y_test, X_val, y_val, "MLP Regression",  
"Adj Close")
```

Here's the step-by-step explanation of the MLP Regression:

1. MLP Regression:

MLP stands for Multi-Layer Perceptron, which is a type of artificial neural network commonly used for regression tasks. It consists of multiple layers of interconnected nodes (neurons) and is capable of learning complex patterns in the data.

2. **mlp_reg = MLPRegressor(random_state=100, max_iter=1000):**

In this step, an **MLPRegressor** object is created with specified parameters. The **random_state** parameter is set to 100 to ensure reproducibility, and **max_iter** is set to 1000, which determines the maximum number of iterations the **MLPRegressor** will perform during training.

3. **perform_regression(mlp_reg, X_final, y_final, X_train, y_train, X_test, y_test, X_val, y_val, "MLP Regression", "Adj Close"):**

The **perform_regression** function is called with the **MLPRegressor** object (**mlp_reg**) and the data split into training, testing, and validation sets. The function evaluates the **MLPRegressor** model's performance and prints the results.

4. RMSE using MLP Regression: (RMSE value)

This value represents the root mean squared error (RMSE) of the MLP Regression model's predictions. It indicates the average difference between the predicted and actual values of the target variable (Adj Close). A lower RMSE indicates better accuracy of the model.

5. Mean Square Error: (MSE value)

This value represents the mean squared error (MSE) of the MLP Regression model's predictions. It is the average squared difference between the predicted and

actual values. A lower MSE indicates better performance of the model.

6. Variance or R-squared: (R-squared value)

This value represents the variance or R-squared of the MLP Regression model's predictions. It indicates the proportion of variance in the target variable (Adj Close) that can be explained by the MLP Regression model. A higher R-squared value indicates that the model captures the patterns in the data well.

7. ACTUAL: Avg. Adj Close: (Actual average value of Adj Close)

This is the actual average value of the target variable (Adj Close) in the dataset.

8. ACTUAL: Median Adj Close: (Actual median value of Adj Close)

This is the actual median value of the target variable (Adj Close) in the dataset.

9. PREDICTED: Avg. Adj Close: (Predicted average value of Adj Close)

This is the predicted average value of the target variable (Adj Close) by the MLP Regression model.

10. PREDICTED: Median Adj Close: (Predicted median value of Adj Close)

This is the predicted median value of the target variable (Adj Close) by the MLP Regression model.

11. Mean Square Error (whole dataset): (MSE value on the whole dataset)

This value represents the mean squared error (MSE) when the MLP Regression model's predictions are evaluated on the entire dataset (including both training and testing data).

12. Variance or R-squared (whole dataset): (R-squared value on the whole dataset)

This value represents the variance or R-squared when the MLP Regression model's predictions are evaluated on the entire dataset. It indicates the proportion of

variance in the target variable (Adj Close) that can be explained by the MLP Regression model.

These output values provide an evaluation of the performance of the MLP Regression model, including its accuracy, error metrics, and the relationship between predicted and actual values of the target variable.

The results are shown in Figure 2.41 – 2.44.

Output:

```
RMSE using MLP Regression: 272.63441469519324
mean square error: 74329.52407619059
variance or r-squared: 0.8906257516334353
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 2384.4799327531423
PREDICTED: Median Adj Close: 2437.541456761096
mean square error (whole dataset): 6540.1852081809475
variance or r-squared (whole dataset): 0.9897782305169576
```

Here's the analysis of the MLP Regression model's performance:

- RMSE using MLP Regression: 272.63441469519324

The root mean squared error (RMSE) measures the average difference between the predicted and actual values of the target variable (Adj Close). In this case, the RMSE value indicates that, on average, the MLP Regression model's predictions have an error of approximately 272.63.

- Mean Square Error: 74329.52407619059

The mean squared error (MSE) represents the average squared difference between the predicted and actual values. A lower MSE indicates better performance. In this case, the MSE value is 74329.52, suggesting that the MLP Regression model has some prediction errors.

- Variance or R-squared: 0.8906257516334353

The variance or R-squared value measures the proportion of variance in the target variable (Adj Close) that can be explained by the MLP Regression model. A higher value indicates a better fit to the data. Here, the R-squared value of 0.8906 suggests that approximately 89.06% of the variance in the target variable can be explained by the MLP Regression model.

- ACTUAL: Avg. Adj Close: 520.4298320160857

This is the actual average value of the target variable (Adj Close) in the dataset.

- ACTUAL: Median Adj Close: 92.639999

This is the actual median value of the target variable (Adj Close) in the dataset.

- PREDICTED: Avg. Adj Close: 2384.4799327531423

This is the predicted average value of the target variable (Adj Close) by the MLP Regression model. The model predicts that the average Adj Close will be approximately 2384.48.

- PREDICTED: Median Adj Close: 2437.541456761096

This is the predicted median value of the target variable (Adj Close) by the MLP Regression model. The model predicts that the median Adj Close will be approximately 2437.54.

- Mean Square Error (whole dataset): 6540.1852081809475

This is the mean squared error (MSE) when the MLP Regression model's predictions are evaluated on the entire dataset (including both training and testing data). The lower MSE on the whole dataset compared to the MSE on the testing set suggests that the model performs better on the training data than on unseen data.

- Variance or R-squared (whole dataset): 0.9897782305169576

This is the variance or R-squared when the MLP Regression model's predictions are evaluated on the entire dataset. The high R-squared value of 0.9898 indicates that the MLP Regression model captures the patterns in the data quite well and can explain a large proportion of the variance in the target variable.

Overall, the MLP Regression model shows relatively good performance with a low RMSE and a high R-squared value. However, it's worth noting that the model seems to overestimate the average and median Adj Close values, as the predicted values are significantly higher than the actual values. This suggests the presence of some bias in the model's predictions. Additionally, the higher MSE on the whole dataset compared to the testing set indicates potential overfitting, where the model performs better on the training data than on unseen data.

FORECASTING USING LASSO REGRESSION

FORECASTING USING LASSO REGRESSION

Step 1 Perform Lasso regression on **Adj Close** column:

```
1 #Lasso regression
2 lasso_reg = LassoCV(n_alphas=1000, max_iter=3000,
3 random_state=0)
4 perform_regression(lasso_reg, X_final, y_final, X_train,
y_train, X_test, y_test, X_val, y_val, "Lasso Regression",
"Adj Close")
```

Here's a step-by-step explanation of the Lasso Regression process:

1. Lasso regression is a linear regression technique that performs regularization by adding a penalty term to the loss function. The penalty term is the L1

- norm of the coefficient vector, multiplied by a hyperparameter alpha.
2. In the code, we first create a LassoCV object named **lasso_reg**, which uses cross-validation to determine the optimal value of alpha.
 3. The **perform_regression()** function is then called with the **lasso_reg** object and other input data. This function performs the regression task and evaluates the model's performance.
 4. RMSE using Lasso Regression: This value represents the root mean squared error (RMSE) between the predicted and actual values of the target variable (Adj Close). It measures the average difference between the predicted and actual values, with a lower value indicating better performance.
 5. Mean Square Error: This value represents the mean squared error (MSE) between the predicted and actual values. It measures the average squared difference between the predicted and actual values, with a lower value indicating better performance.
 6. Variance or R-squared: This value represents the coefficient of determination (R-squared), which measures the proportion of variance in the target variable (Adj Close) that can be explained by the Lasso Regression model. A higher value indicates a better fit to the data.
 7. ACTUAL: Avg. Adj Close: This is the actual average value of the target variable (Adj Close) in the dataset.
 8. ACTUAL: Median Adj Close: This is the actual median value of the target variable (Adj Close) in the dataset.
 9. PREDICTED: Avg. Adj Close: This is the predicted average value of the target variable (Adj Close) by the Lasso Regression model.

10. PREDICTED: Median Adj Close: This is the predicted median value of the target variable (Adj Close) by the Lasso Regression model.
11. Mean Square Error (whole dataset): This value represents the mean squared error (MSE) when the Lasso Regression model's predictions are evaluated on the entire dataset, including both training and testing data.
12. Variance or R-squared (whole dataset): This value represents the coefficient of determination (R-squared) when the Lasso Regression model's predictions are evaluated on the entire dataset.

The output provides these values, allowing you to assess the Lasso Regression model's performance in terms of prediction accuracy and goodness of fit.

The results are shown in Figure 2.45 – 2.48.

Output:

```

RMSE using Lasso Regression: 54.303112665786266
mean square error: 2948.828045193077
variance or r-squared: 0.9930616359089443
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 2547.1346443046236
PREDICTED: Median Adj Close: 2484.110173143274
mean square error (whole dataset): 360.6987661612364
variance or r-squared (whole dataset): 0.9994175503335508

```

Here's a step-by-step explanation and analysis of the output for Lasso Regression:

- RMSE using Lasso Regression: The root mean squared error (RMSE) measures the average difference between the predicted and actual values of the target variable (Adj Close). In this case, the RMSE is 54.303, indicating that, on average, the Lasso Regression model's predictions deviate by

approximately 54.303 from the actual values. Lower values of RMSE indicate better performance.

- Mean Square Error: The mean squared error (MSE) is another measure of the average squared difference between the predicted and actual values. In this case, the MSE is 2948.828, representing the average squared difference between the predicted and actual values of the target variable. Smaller values of MSE indicate better model performance.
- Variance or R-squared: The coefficient of determination, also known as R-squared, measures the proportion of variance in the target variable (Adj Close) that can be explained by the Lasso Regression model. In this case, the R-squared value is 0.993, indicating that approximately 99.3% of the variance in the target variable is explained by the Lasso Regression model. Higher R-squared values indicate better fit to the data.
- ACTUAL: Avg. Adj Close: This is the actual average value of the target variable (Adj Close) in the dataset.
- ACTUAL: Median Adj Close: This is the actual median value of the target variable (Adj Close) in the dataset.
- PREDICTED: Avg. Adj Close: This is the predicted average value of the target variable (Adj Close) by the Lasso Regression model.
- PREDICTED: Median Adj Close: This is the predicted median value of the target variable (Adj Close) by the Lasso Regression model.
- Mean Square Error (whole dataset): This value represents the mean squared error (MSE) when the Lasso Regression model's predictions are evaluated on the entire dataset, including both training and testing data.

- Variance or R-squared (whole dataset): This value represents the coefficient of determination (R-squared) when the Lasso Regression model's predictions are evaluated on the entire dataset.

Overall, the Lasso Regression model appears to perform well. The RMSE and MSE values are relatively low, indicating good prediction accuracy. The high R-squared value suggests that the model explains a large portion of the variance in the target variable. The predicted values for both the average and median Adj Close also seem to be reasonably close to the actual values. Additionally, when evaluated on the whole dataset, the model continues to demonstrate good performance with low MSE and high R-squared values.

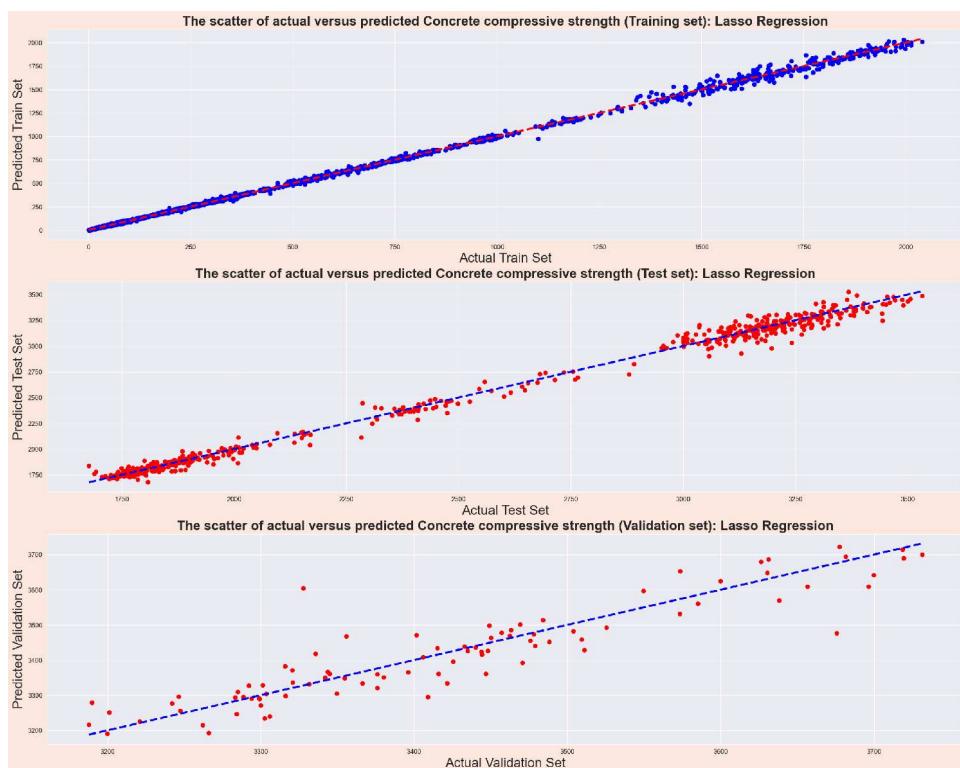


Figure 2.45 The scatter plot of actual versus predicted train, test, and validation sets using Lasso regression

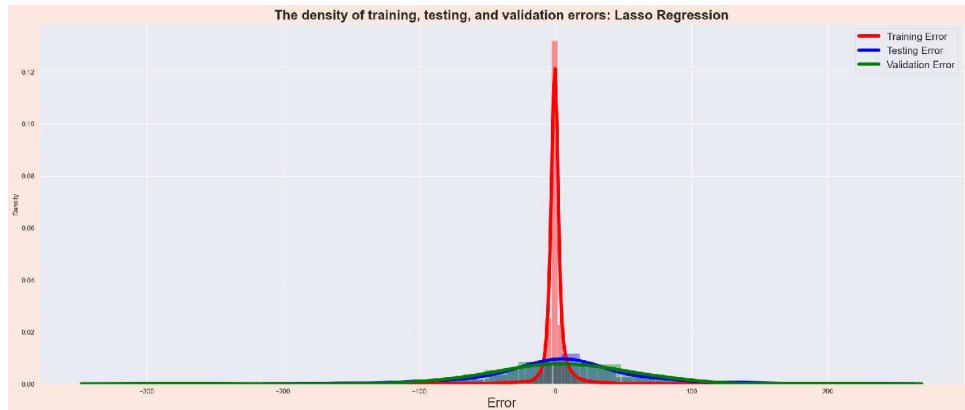


Figure 2.46 The density of train, test, and validation errors using Lasso regression



Figure 2.47 The actual versus predicted values of train, test, validation data, and whole dataset using Lasso regression

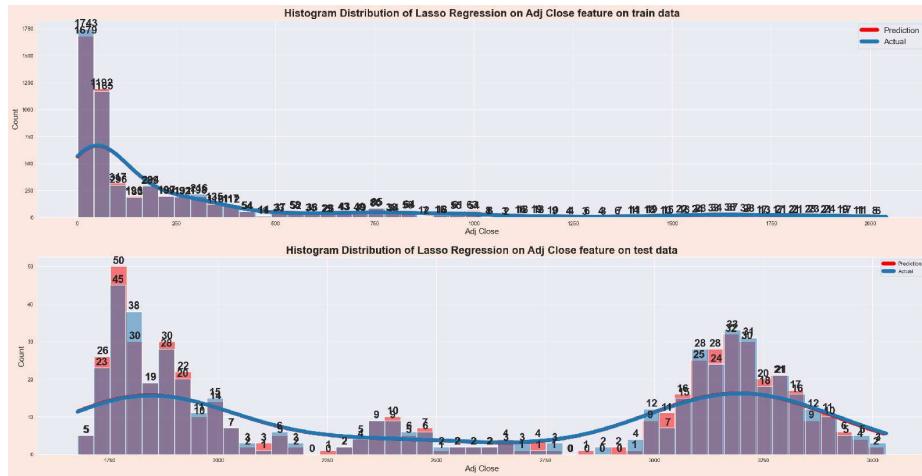


Figure 2.48 The histogram distribution of actual versus predicted train and test sets using Lasso regression

FORECASTING USING RIDGE REGRESSION

FORECASTING USING RIDGE REGRESSION

Step 1 Perform Ridge regression on **Adj Close** column:

```

1 #Ridge regression
2 ridge_reg = RidgeCV(gcv_mode='auto')
3 perform_regression(ridge_reg, X_final, y_final, X_train,
4 y_train, X_test, y_test, X_val, y_val, "Ridge Regression",
  "Adj Close")

```

Here's a step-by-step explanation of the Ridge Regression process and the corresponding output:

1. Ridge regression is a linear regression model with a regularization term that helps prevent overfitting. In this case, **RidgeCV** is used, which performs ridge regression with built-in cross-validation to determine the optimal regularization parameter.
2. The **perform_regression** function is called with the **ridge_reg** model, **X_final** (the feature dataset), **y_final** (the target variable), **X_train**, **y_train** (the training data), **X_test**, **y_test** (the testing data), **X_val**, **y_val** (the validation data), "Ridge Regression" (the model's name), and "Adj Close" (the name of the target variable).

3. The output consists of several metrics and values:

- RMSE using Ridge Regression: The root mean squared error (RMSE) measures the average difference between the predicted and actual values of the target variable (Adj Close). In this case, the RMSE is reported as a single value.
- Mean square error: The mean squared error (MSE) represents the average squared difference between the predicted and actual values of the target variable. This value indicates the overall model performance, with lower values indicating better fit.
- Variance or R-squared: The coefficient of determination, also known as R-squared, measures the proportion of variance in the target variable (Adj Close) that can be explained by the Ridge Regression model. Higher values indicate a better fit to the data.
- ACTUAL: Avg. Adj Close: This is the actual average value of the target variable (Adj Close) in the dataset.
- ACTUAL: Median Adj Close: This is the actual median value of the target variable (Adj Close) in the dataset.
- PREDICTED: Avg. Adj Close: This is the predicted average value of the target variable (Adj Close) by the Ridge Regression model.
- PREDICTED: Median Adj Close: This is the predicted median value of the target variable (Adj Close) by the Ridge Regression model.
- Mean square error (whole dataset): This value represents the mean squared error (MSE) when the Ridge Regression model's predictions are evaluated on the entire

- dataset, including both training and testing data.
- Variance or R-squared (whole dataset): This value represents the coefficient of determination (R-squared) when the Ridge Regression model's predictions are evaluated on the entire dataset.
4. The output provides an evaluation of the Ridge Regression model's performance, including the accuracy of the predictions (RMSE), the goodness of fit (MSE and R-squared), and a comparison between the actual and predicted values. The mean square error and variance or R-squared values for the whole dataset give an indication of how well the model generalizes to unseen data.

Overall, the output provides an assessment of the Ridge Regression model's ability to predict the target variable based on the provided features. It allows for an evaluation of the model's accuracy, fit, and generalization performance.

The results are shown in Figure 2.49 – 2.52.

Output:

```
RMSE using Ridge Regression: 50.40933792667383
mean square error: 2541.1013502055966
variance or r-squared: 0.9939721690558871
ACTUAL: Avg. Adj Close: 520.4298320160857
ACTUAL: Median Adj Close: 92.639999
PREDICTED: Avg. Adj Close: 2550.0304993100813
PREDICTED: Median Adj Close: 2492.8376834695405
mean square error (whole dataset): 315.7283609167274
variance or r-squared (whole dataset): 0.9994899138419426
```

Here's a step-by-step analysis:

- RMSE using Ridge Regression: The root mean squared error (RMSE) measures the average

difference between the predicted and actual values of the target variable (Adj Close). In this case, the RMSE is reported as 50.40933792667383, indicating that, on average, the predicted values deviate from the actual values by approximately 50.41.

- Mean square error: The mean squared error (MSE) represents the average squared difference between the predicted and actual values of the target variable. The reported value is 2541.1013502055966, which indicates the overall model performance. A lower MSE suggests a better fit of the model to the data.
- Variance or R-squared: The coefficient of determination, also known as R-squared, measures the proportion of variance in the target variable (Adj Close) that can be explained by the Ridge Regression model. The reported value is 0.9939721690558871, which suggests that approximately 99.4% of the variability in the target variable can be explained by the model. A higher R-squared value indicates a better fit to the data.
- ACTUAL: Avg. Adj Close: This is the actual average value of the target variable (Adj Close) in the dataset. The reported value is 520.4298320160857.
- ACTUAL: Median Adj Close: This is the actual median value of the target variable (Adj Close) in the dataset. The reported value is 92.639999.
- PREDICTED: Avg. Adj Close: This is the predicted average value of the target variable (Adj Close) by the Ridge Regression model. The reported value is 2550.0304993100813.
- PREDICTED: Median Adj Close: This is the predicted median value of the target variable (Adj

Close) by the Ridge Regression model. The reported value is 2492.8376834695405.

- Mean square error (whole dataset): This value represents the mean squared error (MSE) when the Ridge Regression model's predictions are evaluated on the entire dataset, including both training and testing data. The reported value is 315.7283609167274.
- Variance or R-squared (whole dataset): This value represents the coefficient of determination (R-squared) when the Ridge Regression model's predictions are evaluated on the entire dataset. The reported value is 0.9994899138419426.

Overall, the output indicates that the Ridge Regression model performs well in predicting the target variable (Adj Close). The RMSE and MSE values are relatively low, indicating a good fit of the model to the data. The high R-squared value suggests that the model explains a significant portion of the variability in the target variable. The comparison between the actual and predicted values further confirms the model's accuracy.

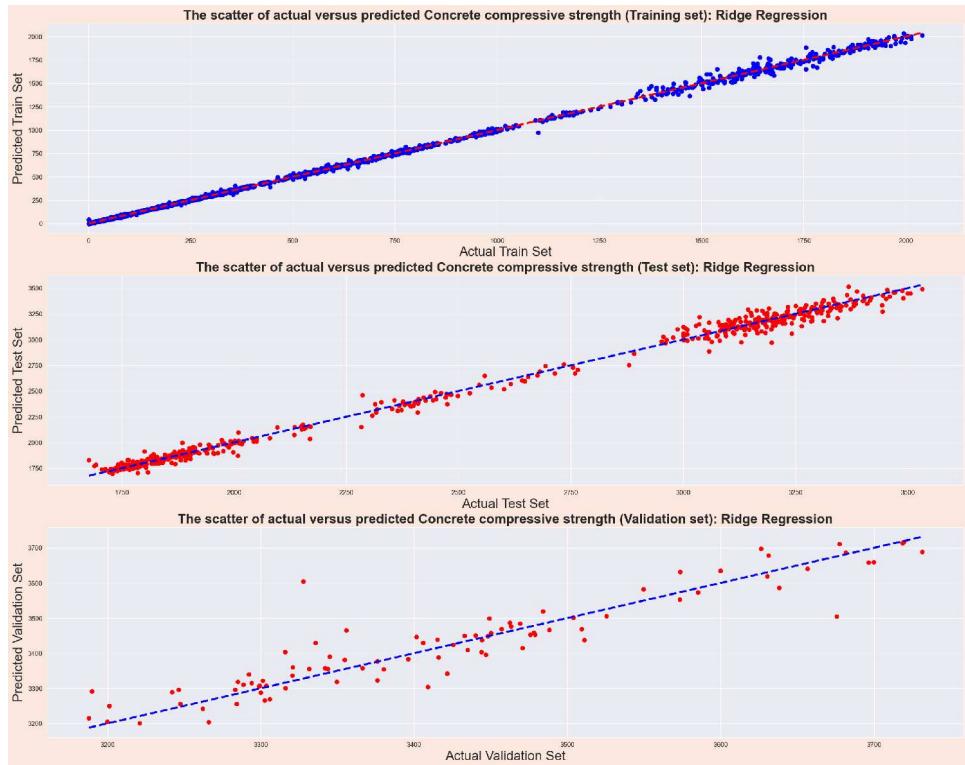


Figure 2.49 The scatter plot of actual versus predicted train, test, and validation sets using Ridge regression

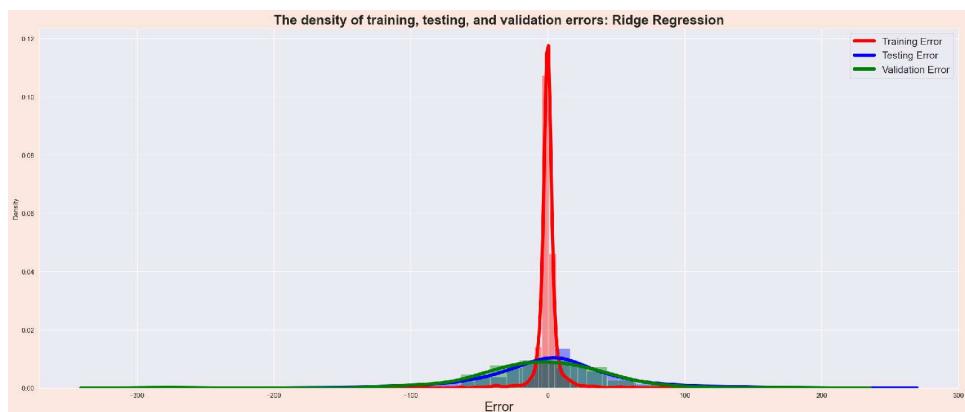


Figure 2.50 The density of train, test, and validation errors using Ridge regression



Figure 2.51 The actual versus predicted values of train, test, validation data, and whole dataset using Ridge regression

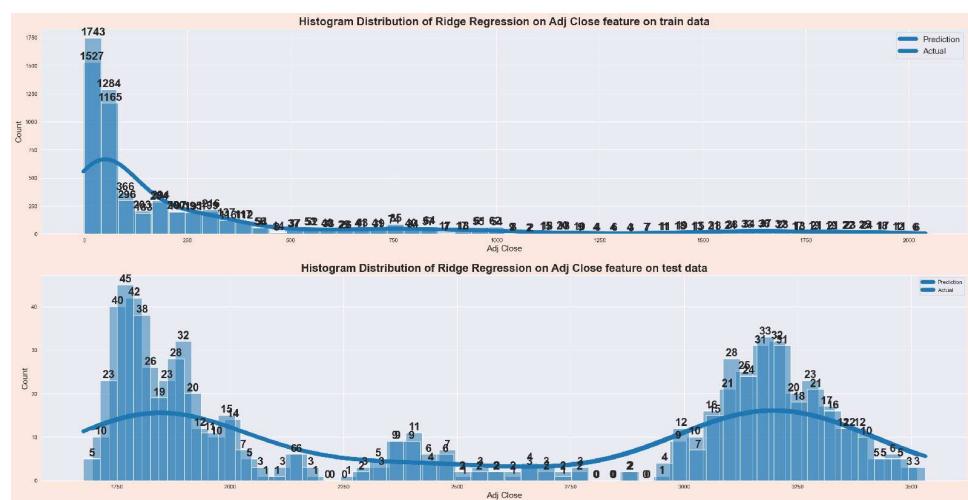


Figure 2.52 The histogram distribution of actual versus predicted train and test sets using Ridge regression



PREDICTING STOCK DAILY RETURN USING

MACHINE LEARNING

PREDICTING

STOCK DAILY RETURN

USING

MACHINE LEARNING

DESCRIPTION

DESCRIPTION

Daily return refers to the percentage change in the value of an asset or investment from one trading day to the next. It is a measure of the asset's performance over a single trading day and provides insight into the gains or losses experienced by investors during that period.

To calculate the daily return of an asset, you can use the following formula:

$$\text{Daily Return} = (\text{Current Price} - \text{Previous Price}) / \text{Previous Price} * 100$$

Here, "Current Price" refers to the closing price of the asset on the current trading day, and "Previous Price" refers to the closing price of the asset on the previous trading day.

The result is typically expressed as a percentage, representing the percentage change in the asset's value from the previous day. A positive daily return indicates an increase in the asset's value, while a negative daily return indicates a decrease.

Daily return is a commonly used measure in finance and investment analysis as it allows investors to track and assess the performance of their investments on a day-to-day basis. It provides information on the volatility and potential risks associated with an asset, as well as insights into the overall trend and direction of the market.

SPLITTING AND NORMALIZING DATA

SPLITTING AND NORMALIZING DATA

Step 1 Compute daily returns, set target column, check null values because of technical indicators, fill each null value in every column with mean value, resample data using SMOTE, and split the data into train and test sets.

```
1 X = pd.read_csv(curr_path+"/Amazon.csv")
2 X['Date'] = pd.to_datetime(X['Date'])
3
4 #Sets date column as index
5 X = X.set_index("Date")
6
7
8 X['SMA'] = SMA_CLOSE
9 X['Upper_band'] = upper_band
10 X['Lower_band'] = lower_band
11 X['DIF'] = DIF
12 X['MACD'] = MACD
13 X['RSI'] = RSI
14 X['STDEV'] = STDEV
15 X['Open_Close']=Open_Close
16 X['High_Low']=High_Low
17
18 #Computes daily returns
19 X["daily_returns"] = compute_daily_returns(X["Adj Close"])
20
21 #Checks null values because of technical indicators
22 print(X.isnull().sum().to_string())
23 print('Total number of null values: ', X.isnull().sum().sum())
24
25
26 #Fills each null value in every column with mean value
27 cols = list(X.columns)
28 for n in cols:
29     X[n].fillna(X[n].mean(), inplace = True)
30
31
32 #Checks again null values
33 print(X.isnull().sum().to_string())
34
35
36 #Reads columns
37 print("Data Columns --> ", list(X.columns))
38 print(X.head(10).to_string())
39
40 #Sets target column
41 y=X["daily_returns"]
42 y = np.array([1 if i>0 else 0 for i in y])
43 #Drops irrelevant column
```

```

44 X = X.drop(["daily_returns"], axis =1)
45 #print(y.head(10).to_string())
46
47 #Resamples data using SMOTE
48 sm = SMOTE(random_state=2022)
49 X,y = sm.fit_resample(X, y)
50
51 #Splits the data into train and test sets
52 X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.2, random_state = 2021, stratify=y)
53 print(X_train.shape)
54 print(y_train.shape)
55 print(X_test.shape)
56 print(y_test.shape)

```

Here's a detailed explanation of each step:

1. Read the data from the "Amazon.csv" file using Pandas and assign it to the variable X:

X = pd.read_csv(curr_path+"/Amazon.csv")

This step uses the **pd.read_csv()** function to read the data from the CSV file and store it in a DataFrame called **X**.

2. Convert the "Date" column of X to a datetime format using **pd.to_datetime**:

X['Date'] = pd.to_datetime(X['Date'])

This step converts the "Date" column of the DataFrame **X** to a datetime format, which is useful for time-based analysis.

3. Set the "Date" column as the index of the DataFrame using **set_index**:

X = X.set_index("Date")

This step sets the "Date" column as the index of the DataFrame **X**, which allows for easier time-based indexing and analysis.

4. Add various technical indicators (SMA_CLOSE, upper_band, lower_band, DIF, MACD, RSI, STDEV, Open_Close, High_Low) to X as new columns:

X['SMA'] = SMA_CLOSE

X['Upper_band'] = upper_band

```
X['Lower_band'] = lower_band  
X['DIF'] = DIF  
X['MACD'] = MACD  
X['RSI'] = RSI  
X['STDEV'] = STDEV  
X['Open_Close'] = Open_Close  
X['High_Low'] = High_Low
```

This step adds columns to the DataFrame X with the values of various technical indicators, such as Simple Moving Average (SMA), Upper Band, Lower Band, Difference (DIF), Moving Average Convergence Divergence (MACD), Relative Strength Index (RSI), Standard Deviation (STDEV), Open-Close price difference (Open_Close), and High-Low price difference (High_Low).

5. Compute the daily returns of the "Adj Close" column and add it as a new column named "daily_returns" in X:

```
X["daily_returns"] = compute_daily_returns(X["Adj Close"])
```

This step calls a function called **compute_daily_returns()** (defined in chapter 1) to calculate the daily returns of the "Adj Close" column and adds the result as a new column named "daily_returns" in the DataFrame X.

6. Check for null values in X and print the count of null values for each column:

```
print(X.isnull().sum().to_string())
```

This step checks for null values in each column of the DataFrame X using **X.isnull().sum()** and prints the count of null values for each column.

7. Print the total number of null values in X:

```
print('Total number of null values: ',  
      X.isnull().sum().sum())
```

This step calculates the total number of null values in the DataFrame X by summing up the counts of null

values from the previous step and prints the result.

8. Fill each null value in every column of X with the mean value of that column using a for loop:

```
cols = list(X.columns)
```

```
for n in cols:
```

```
    X[n].fillna(X[n].mean(), inplace=True)
```

This step creates a list of column names called cols and iterates over each column. For each column, it fills the null values with the mean value of that column using the `fillna()` function.

9. Check for null values in X again and print the count of null values for each column:

```
print(X.isnull().sum().to_string())
```

This step checks for null values in each column of the DataFrame X again after the null values have been filled, and prints the count of null values for each column.

10. Print the column names of X:

```
print("Data Columns --> ", list(X.columns))
```

This step retrieves the column names of the DataFrame X using X.columns and prints them as a list.

11. Print the first 10 rows of X:

```
print(X.head(10).to_string())
```

This step retrieves the first 10 rows of the DataFrame X using `X.head(10)` and prints them as a formatted string.

12. Set the target column as the "daily_returns" column in X and convert the values to binary labels: 1 if the value is greater than 0, otherwise 0. Assign the modified values to the variable y:

```
y = X["daily_returns"]
```

```
y = np.array([1 if i > 0 else 0 for i in y])
```

This step assigns the values of the "daily_returns" column in X to the variable y. Then, it uses a list comprehension to create a new NumPy array y where each value is converted to a binary label: 1 if the value is greater than 0 (indicating a positive return), otherwise 0.

13. Drop the "daily_returns" column from X as it is no longer needed:

```
X = X.drop(["daily_returns"], axis=1)
```

This step removes the "daily_returns" column from the DataFrame X using the `drop()` function and specifying the column name and axis.

14. Resample the data using SMOTE (Synthetic Minority Over-sampling Technique) to address class imbalance. This technique creates synthetic samples for the minority class (in this case, the positive daily returns) to balance the data. Assign the resampled data back to X and y:

```
sm = SMOTE(random_state=2022)
```

```
X, y = sm.fit_resample(X, y)
```

This step initializes an instance of the SMOTE class with a random state of 2022. Then, it applies the SMOTE resampling technique to the data, resulting in balanced classes. The resampled data is assigned back to X and y.

Split the data into training and test sets using `train_test_split`. The training set will contain 80% of the data, while the test set will contain the remaining 20%. The `stratify` parameter ensures that the class distribution is preserved in both the training and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2021, stratify=y)
```

This step uses the `train_test_split()` function to split the resampled data X and y into training and test sets. The test set will be 20% of the data (`test_size=0.2`), and the random state is set to 2021 for reproducibility. The `stratify` parameter ensures that the class distribution in y is preserved in the train-test split.

15. Print the shapes of the training and test sets to check the dimensions:

```
print(X_train.shape)
```

```
print(y_train.shape)
```

	print(X_test.shape)
Step 2	<p>Normalize data with robust scaler, minmax scaler, and standard scaler:</p> <pre> 1 #Normalizes data with robust scaler 2 rob_scaler = RobustScaler() 3 X_train_rob = X_train.copy() 4 X_test_rob = X_test.copy() 5 y_train_rob = y_train.copy() 6 y_test_rob = y_test.copy() 7 X_train_rob = rob_scaler.fit_transform(X_train_rob) 8 X_test_rob = rob_scaler.transform(X_test_rob) 9 10 #Normalizes data with minmax scaler 11 X_train_norm = X_train.copy() 12 X_test_norm = X_test.copy() 13 y_train_norm = y_train.copy() 14 y_test_norm = y_test.copy() 15 norm = MinMaxScaler() 16 X_train_norm = norm.fit_transform(X_train_norm) 17 X_test_norm = norm.transform(X_test_norm) 18 19 #Normalizes data with standard scaler 20 X_train_stand = X_train.copy() 21 X_test_stand = X_test.copy() 22 y_train_stand = y_train.copy() 23 y_test_stand = y_test.copy() 24 std_scaler = StandardScaler() 25 X_train_stand = std_scaler.fit_transform(X_train_stand) 26 X_test_stand = std_scaler.transform(X_test_stand) </pre> <p>The code performs data normalization using three different scalers: RobustScaler, MinMaxScaler, and StandardScaler. Here's an explanation of each step:</p> <ol style="list-style-type: none"> 1. Normalize data with RobustScaler: <pre> rob_scaler = RobustScaler() X_train_rob = X_train.copy() X_test_rob = X_test.copy() y_train_rob = y_train.copy() y_test_rob = y_test.copy() X_train_rob = rob_scaler.fit_transform(X_train_rob) X_test_rob = rob_scaler.transform(X_test_rob) </pre> <ul style="list-style-type: none"> • Create an instance of the RobustScaler class.

- Make copies of the original training and test sets (`X_train`, `X_test`, `y_train`, `y_test`).
- Apply the RobustScaler's `fit_transform()` method to the training set (`X_train_rob`) to calculate the scaling parameters and transform the data.
- Apply the RobustScaler's `transform` method to the test set (`X_test_rob`) to scale the test data using the parameters learned from the training set.

RobustScaler is a scaler that scales the features using statistics that are robust to outliers. It centers the data by subtracting the median and scales it by dividing by the interquartile range (IQR).

2. Normalize data with **MinMaxScaler**:

```
X_train_norm = X_train.copy()
X_test_norm = X_test.copy()
y_train_norm = y_train.copy()
y_test_norm = y_test.copy()
norm = MinMaxScaler()
X_train_norm = norm.fit_transform(X_train_norm)
X_test_norm = norm.transform(X_test_norm)
```

- Make copies of the original training and test sets (`X_train`, `X_test`, `y_train`, `y_test`).
- Create an instance of the `MinMaxScaler` class.
- Apply the `MinMaxScaler`'s `fit_transform()` method to the training set (`X_train_norm`) to calculate the scaling parameters and transform the data.
- Apply the `MinMaxScaler`'s `transform` method to the test set (`X_test_norm`) to scale the test data using the parameters learned from the training set.

MinMaxScaler scales the data to a fixed range, typically between 0 and 1, by subtracting the minimum value and dividing by the range (maximum value minus minimum value).

3. Normalize data with StandardScaler:

```
X_train_stand = X_train.copy()  
X_test_stand = X_test.copy()  
y_train_stand = y_train.copy()  
y_test_stand = y_test.copy()  
std_scaler = StandardScaler()  
X_train_stand = std_scaler.fit_transform(X_train_stand)  
X_test_stand = std_scaler.transform(X_test_stand)
```

- Make copies of the original training and test sets (X_{train} , X_{test} , y_{train} , y_{test}).
- Create an instance of the StandardScaler class.
- Apply the StandardScaler's **fit_transform()** method to the training set (X_{train_stand}) to calculate the scaling parameters and transform the data.
- Apply the StandardScaler's transform method to the test set (X_{test_stand}) to scale the test data using the parameters learned from the training set.

StandardScaler scales the data to have zero mean and unit variance by subtracting the mean and dividing by the standard deviation.

After these steps, you will have three sets of normalized data (X_{train_rob} , X_{test_rob} for RobustScaler, X_{train_norm} , X_{test_norm} for MinMaxScaler, and X_{train_stand} ,

DEFINING HELPER FUNCTIONS

DEFINING HELPER FUNCTIONS

Step 1 Define `plot_learning_curve()` method to plot learning curve, scalability, and performance of a certain classifier.

```
1 def plot_learning_curve(estimator, name, X, y, axes=None,
2                         ylim=None, cv=None, n_jobs=None, train_sizes=np.linspace(.1,
3                         1.0, 5),fc=""):
4     if axes is None:
5         _, axes = plt.subplots(1, 3, figsize=(20, 5))
6
7     axes[0].set_title(name + " with " + fc)
8     if ylim is not None:
9         axes[0].set_ylim(*ylim)
10    axes[0].set_xlabel("Training examples")
11    axes[0].set_ylabel("Score")
12
13
14    train_sizes, train_scores, test_scores, fit_times, _ =
15    learning_curve(estimator, X, y, cv=cv, n_jobs=n_jobs,
16                    train_sizes=train_sizes,
17                    return_times=True)
18    train_scores_mean = np.mean(train_scores, axis=1)
19    train_scores_std = np.std(train_scores, axis=1)
20    test_scores_mean = np.mean(test_scores, axis=1)
21    test_scores_std = np.std(test_scores, axis=1)
22    fit_times_mean = np.mean(fit_times, axis=1)
23    fit_times_std = np.std(fit_times, axis=1)
24
25
26    # Plot learning curve
27    axes[0].grid()
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
```

	44
	45
	46
	47
	48
	49
	50
	51
	52
	53
	54
	55
	56
	57
	58
	59
	60
	61

```

        axes[0].fill_between(train_sizes,
            train_scores_mean - train_scores_std,
            train_scores_mean + train_scores_std,
            alpha=0.1, color="r")
        axes[0].fill_between(train_sizes,
            test_scores_mean - test_scores_std,
            test_scores_mean + test_scores_std,
            alpha=0.1, color="g")
        axes[0].plot(train_sizes, train_scores_mean, 'o-',
            color="r", label="Training score")
        axes[0].plot(train_sizes, test_scores_mean, 'o-',
            color="g", label="Cross-validation score")
        axes[0].legend(loc="best")

    # Plot n_samples vs fit_times
    axes[1].grid()
    axes[1].plot(train_sizes, fit_times_mean, 'o-')
    axes[1].fill_between(train_sizes,
        fit_times_mean - fit_times_std,
        fit_times_mean + fit_times_std, alpha=0.1)
    axes[1].set_xlabel("Training examples")
    axes[1].set_ylabel("fit_times")
    axes[1].set_title("Scalability of the model : " + name
        + " with " + fc)

    # Plot fit_time vs score
    axes[2].grid()
    axes[2].plot(fit_times_mean, test_scores_mean, 'o-')
    axes[2].fill_between(fit_times_mean,
        test_scores_mean - test_scores_std,
        test_scores_mean + test_scores_std, alpha=0.1)
    axes[2].set_xlabel("fit_times")
    axes[2].set_ylabel("Score")
    axes[2].set_title("Performance of the model: " + name
        + " with " + fc)

return plt

```

It is used to visualize the learning curve, scalability, and performance of a machine learning model. Here's a breakdown of the function:

1. Parameters:

- **estimator**: The machine learning model to evaluate.
- **name**: The name of the model.
- **X**: The input features.
- **y**: The target variable.
- **axes**: Optional parameter to specify the subplot axes for the plots.
- **ylim**: Optional parameter to set the y-axis limits for the score plot.
- **cv**: Optional parameter for cross-validation strategy.
- **n_jobs**: Optional parameter for the number of jobs to run in parallel.
- **train_sizes**: Optional parameter to specify the training set sizes for plotting the learning curve.
- **fc**: Optional parameter to specify additional information about the model (e.g., feature configuration).

2. Creating subplots:

If axes is None, the function creates three subplots using plt.subplots(1, 3, figsize=(20, 5)) and assigns

them to the variable axes. The subplots will have a figure size of 20 units in width and 5 units in height.

3. Setting titles and labels:

- The function sets the title of the first subplot (learning curve plot) as name + " with " + fc.
- If ylim is not None, the function sets the y-axis limits of the first subplot using `axes[0].set_ylim(*ylim)`.
- The function sets the x-label of the first subplot as "Training examples" and the y-label as "Score".

4. Computing learning curve metrics:

- The function uses the `learning_curve()` function to calculate the learning curve metrics, including training scores, test scores, fit times, etc., for different training set sizes (`train_sizes`).
- The computed metrics are stored in variables such as `train_sizes`, `train_scores`, `test_scores`, `fit_times`, etc.

5. Plotting the learning curve:

- The function plots the learning curve by filling the area between the mean training scores plus/minus the standard deviation and the mean test scores plus/minus the standard deviation using `fill_between()`.
- It plots the mean training scores and mean test scores as a function of the training set sizes using `plot`.
- The legend is added to the plot using `legend(loc="best")`.

6. Plotting scalability:

- The function plots the training set sizes against the fit times of the model using `plot`.
- It fills the area between the mean fit times plus/minus the standard deviation using `fill_between()`.
- The x-label is set as "Training examples", and the y-label is set as "fit_times".

7. Plotting performance:

- The function plots the mean fit times against the mean test scores using `plot`.
- It fills the area between the mean test scores plus/minus the standard deviation using `fill_between`.
- The x-label is set as "fit_times", and the y-label is set as "Score".

8. Returning the plot:

The function returns the generated plot using `plt`.

Overall, this function provides a convenient way to visualize the learning curve, scalability, and performance of a machine

learning model with different training set sizes and additional information about the model.

Step 2

Define **plot_real_pred_val()** to plot true values predicted values and **plot_cm()** method to plot con matrix:

```
1 def plot_real_pred_val(Y_test, ypred, name, fc,ax):  
2     acc=accuracy_score(Y_test,ypred)  
3     ax.scatter(range(len(ypred)),ypred,color="blue",\n        lw=5,label="Predicted")  
4     ax.scatter(range(len(Y_test)), \n        Y_test,color="red",label="Actual")  
5     ax.set_title("Predicted Values vs True Values of " +  
6         name + " with " + fc, fontsize=30)  
7     ax.set_xlabel("Accuracy: " + str(round((acc*100),3)) + "%",  
8         fontsize=20)  
9     ax.set_ylabel("Daily Returns", fontsize=20)  
10    ax.legend(fontsize=20)  
11    ax.grid(True, alpha=0.75, lw=1, ls='-.')  
12    ax.yaxis.set_ticklabels(["", "Negative Daily Return",  
13        "", "", "", "", "Positive Daily Returns"]);  
14    #plt.show()  
15  
16  
17  
18 def plot_cm(Y_test, ypred, name, fc,ax):  
19     cm = confusion_matrix(Y_test, ypred)  
20     sns.heatmap(cm, annot=True, linewidth=0.7, \n        linecolor='red', fmt='g', cmap="Greens", ax=ax)  
21     ax.set_title(name + ' Confusion Matrix ' + "with " + fc)  
22     ax.set_xlabel('Y predict')  
23     ax.set_ylabel('Y test')  
24     ax.xaxis.set_ticklabels(["Negative Daily Returns",  
25        "Positive Daily Returns"]);  
26     ax.yaxis.set_ticklabels(["Negative Daily Returns",  
27        "Positive Daily Returns"]);  
28     #plt.show()  
29     return cm  
30
```

Here's an explanation of each function:

1. **plot_real_pred_val()** function:

Parameters:

- **Y_test**: The true values of the variable from the test set.
 - **ypred**: The predicted values of the variable.
 - **name**: The name of the model.
 - **fc**: Additional information about the (e.g., feature configuration).
 - **ax**: The subplot axes for the plot.
- a. The function calculates the accuracy score by comparing the true values (**Y_test**) with the predicted values (**ypred**) using **accuracy_score()**.
 - b. It creates a scatter plot to visualize the predicted values (**ypred**) as blue points and the true values (**Y_test**) as red points on a subplot (**ax**).
 - c. The title of the plot is set as "Predicted Values vs True Values of " + name + " with " + fc.
 - d. The x-axis label is set as "Accuracy: " followed by the rounded accuracy score in percentage for the model.
 - e. The y-axis label is set as "Daily Returns".
 - f. The legend is added to the plot.

- g. The grid lines are displayed on the plot.
 h. The y-axis tick labels are set to descriptive labels for negative and positive daily returns.

2. **plot_cm()** function:

Parameters:

- **Y_test**: The true values of the variable from the test set.
 - **ypred**: The predicted values of the variable.
 - **name**: The name of the model.
 - **fc**: Additional information about the (e.g., feature configuration).
 - **ax**: The subplot axes for the plot.
- a. The function calculates the confusion by comparing the true values (**Y_test**) with predicted values (**ypred**) using **confusion_matrix()**.
- b. It creates a heatmap using `sns.heatmap` to visualize the confusion matrix (**cm**) on subplot (**ax**).
- c. The heatmap displays the values of the confusion matrix as annotations.
- d. The title of the plot is set as `name + "Confusion Matrix" + "with " + fc`.
- e. The x-axis label is set as '**Y pred**' representing the predicted values.
- f. The y-axis label is set as '**Y test**', representing the true values.
- g. The tick labels of the x-axis and y-axis are set to show descriptive labels for negative and positive daily returns.

Both functions return the confusion matrix (**cm**) calculated in the **plot_cm()** function.

These functions provide a convenient way to visualize predicted values and true values of a model, as well as the confusion matrix, to evaluate the model's performance.

Step 3

Define **plot_decision_boundary()** method to plot decision boundaries of two chosen features and **plot_roc()** method to plot the ROC of the model:

```

1 def plot_decision_boundary(model,xtest, ytest, name,fc
2     #Trains model with two features
3     model.fit(xtest, ytest)
4     plot_decision_regions(np.array(xtest), np.array(yt
5     clf=model, legend=2,ax=ax)
6     ax.set_title("Decision boundary for " + name + " (
7     " + "with " + fc)
8     ax.set_xlabel('Open_Close')
9     ax.set_ylabel('High_Low')
10    plt.show()
11
12 #Plots ROC
13 def plot_roc(model,X_test, y_test, name, fc,ax):
14     Y_pred_prob = model.predict_proba(X_test)

```

```

15 Y_pred_prob = Y_pred_prob[:, 1]
16
17 fpr, tpr, thresholds = roc_curve(y_test, Y_pred_pr)
18 ax.plot([0,1],[0,1], color='navy', lw=5, linestyle='solid')
19 ax.plot(fpr,tpr, label='ANN')
20 ax.set_xlabel('False Positive Rate')
21 ax.set_ylabel('True Positive Rate')
22 ax.set_title('ROC Curve of ' + name + " with "+fc)
23 plt.grid(True)
24 #plt.show()
25
26 #Chooses two features for decision boundary
27 X_feature = X.iloc[:, 13:14]
28 X_train_feat, X_test_feat, y_train_feat, y_test_feat =
train_test_split(X_feature, y, test_size = 0.3, random_
state = 2021, stratify=y)

```

Here's an explanation of each function:

1. `plot_decision_boundary()` function:

Parameters:

- **model**: The machine learning model used to visualize the decision boundary.
 - **xtest**: The features of the test set.
 - **ytest**: The true values of the target variable from the test set.
 - **name**: The name of the model.
 - **fc**: Additional information about the model (e.g., feature configuration).
 - **ax**: The subplot axes for the plot.
- The function trains the model using the features (**xtest**) and true values (**ytest**) from the test set.
 - It plots the decision boundary using the `plot_decision_regions()` function from the mlxtend.plotting module. The decision boundary separates the two classes represented by the features and labels.
 - The title of the plot is set as "Decision boundary for " + name + " (Test) " + "with " + fc.
 - The x-axis label is set as 'Open_Close'.
 - The y-axis label is set as 'High_Low'.
 - The plot is displayed using `plt.show()`.

2. `plot_roc()` function:

Parameters:

- **model**: The machine learning model used to calculate and plot the ROC curve.
 - **X_test**: The features of the test set.
 - **y_test**: The true values of the target variable from the test set.
 - **name**: The name of the model.
 - **fc**: Additional information about the model (e.g., feature configuration).
 - **ax**: The subplot axes for the plot.
- The function predicts the class probabilities for the test set using `predict_proba()` and extracts the probabilities for the positive class.

	<p>b. It calculates the false positive rate (fpr) positive rate (tpr), and thresholds roc_curve() from the sklearn.n module.</p> <p>c. It plots the ROC curve using plot fro matplotlib.pyplot module. The ROC represents the trade-off between the positive rate and false positive rate at v classification thresholds.</p> <p>d. The x-axis label is set as 'False Positive I</p> <p>e. The y-axis label is set as 'True Positive R</p> <p>f. The title of the plot is set as 'ROC Curve name + " with " + fc.</p> <p>g. The grid lines are displayed on the plot.</p>
Step 4	<p>After defining the two functions, the code selec "Open_Close" feature and assigns it to X_feature.] splits the data into training and testing sets (X_train, X_test_feat, y_train_feat, y_test_feat) using the se feature and the target variable (y). The purpose of this to choose two features for visualizing the decision bou</p>

Step 4	<p>Define train_model() method to train model, predict_model() method to get predicted values, run_model() method to perform training model, predict results, plotting confusion matrix, plotting true values vs predicted values, plotting decision boundary, plotting learning curve and ROC:</p> <hr/> <pre> 1 def train_model(model, X, y): 2 model.fit(X, y) 3 return model 4 5 def predict_model(model, X, proba=False): 6 if ~proba: 7 y_pred = model.predict(X) 8 else: 9 y_pred_proba = model.predict_proba(X) 10 y_pred = np.argmax(y_pred_proba, axis=1) 11 12 return y_pred 13 14 list_scores = [] 15 16 def run_model(name, model, X_train, X_test, y_train, y_test, fc, proba=False): 17 print(name) 18 print(fc) 19 20 model = train_model(model, X_train, y_train) 21 y_pred = predict_model(model, X_test, proba) 22 23 accuracy = accuracy_score(y_test, y_pred) 24 recall = recall_score(y_test, y_pred, 25 average='weighted') 26 precision = precision_score(y_test, y_pred, 27 average='weighted') 28 f1 = f1_score(y_test, y_pred, average='weighted') 29 30 print('accuracy: ', accuracy) 31 print('recall: ', recall) 32 print('precision: ', precision) 33 print('f1: ', f1) 34 35 print(classification_report(y_test, y_pred)) 36 37 38 </pre>
--------	---

```

39     fig, axes = plt.subplots(2,2,figsize=(30,20),
40                             facecolor='#fbe7dd')
41     plot_cm(y_test, y_pred, name,fc,axes[0, 0])
42     plot_real_pred_val(y_test, y_pred, name,fc,axes[0,
43     plot_roc(model, X_test, y_test, name,fc,axes[1, 0]
44     plot_decision_boundary(model,X_test_feat, y_test_f
45     name,fc,axes[1, 1])
46     plot_learning_curve(model, name, X_train, y_train,
47     cv=3,fc=fc);
48     plt.show()
49
50     list_scores.append({'Model Name': name, 'Feature
Scaling':fc, 'Accuracy': accuracy, 'Recall': recall,
'Precision': precision, 'F1':f1})
51
52     feature_scaling = {
53         'Robust Scaler':(X_train_rob, X_test_rob, y_train_
y_test_rob),
54         'MinMax Scaler':(X_train_norm, X_test_norm,
y_train_norm, y_test_norm),
55         'Standard Scaler':(X_train_stand, X_test_stand,
y_train_stand, y_test_stand),
56     }

```

Let's go through each function in detail:

1. **train_model(model, X, y)** function:
 - a. This function trains a machine learning model using the provided features (X) and variable (y).
 - b. Parameters:
 - **model**: The machine learning model to be trained.
 - **X**: The feature data.
 - **y**: The target variable data.
 - c. The function fits the model to the provided data by calling the **fit()** method of the object.
 - d. Finally, it returns the trained model.

2. **predict_model(model, X, proba=False)** function:
 - a. This function makes predictions using the trained model on the provided feature data.
 - b. Parameters:
 - **model**: The trained machine learning model.
 - **X**: The feature data on which to make predictions.
 - **proba**: A boolean flag indicating whether to return class probabilities (default: **False**).
 - c. If **proba** is **False**, the function calls the **predict()** method of the model to obtain predicted class labels.
 - d. If **proba** is **True**, the function calls the **predict_proba()** method of the model to obtain the predicted class probabilities. It then uses **np.argmax()** to select the class label with the highest probability.
 - e. The function returns the predicted class labels or probabilities.

3. **run_model(name, model, X_train, y_train, y_test, fc, proba=False)** function:
- This function trains, evaluates, and visualizes the performance of a machine learning model using the provided training and test data.
 - Parameters:
 - **name**: The name of the model.
 - **model**: The machine learning model to be trained and evaluated.
 - **X_train**: The feature data for training.
 - **X_test**: The feature data for testing.
 - **y_train**: The target variable data for training.
 - **y_test**: The target variable data for testing.
 - **fc**: Additional information about the model's feature scaling method.
 - **proba**: A boolean flag indicating whether to return class probabilities when making predictions (default is **False**).
 - The function trains the model using the **train_model()** function by passing in the training data (**X_train** and **y_train**).
 - It then makes predictions on the test data (**X_test**) using the **predict_model()** function.
 - The function calculates various evaluation metrics, including accuracy, recall, precision, and F1 score, using functions from the `sklearn.metrics` module.
 - It displays the evaluation metrics in a classification report, confusion matrix, predicted values vs. true values plot, ROC curve, and decision boundary plot using various plotting functions (**plot_real_pred_val**, **plot_decision_boundary**, **plot_learning_curve**).
 - Finally, the function stores the evaluation metrics in the `list_scores` list.

The `feature_scaling` dictionary contains different feature scaling methods (Robust Scaler, MinMaxScaler, Standard Scaler) as keys and their corresponding data splits (**X_train_rob**, **X_test_rob**, **y_train_rob**, **y_test_rob**) as values. These data splits are created earlier in the code using different feature scaling techniques. The purpose of this dictionary is to iterate over the different feature scaling methods and pass the appropriate data splits to the **run_model()** function for training, evaluation, and visualization.

PREDICTING DAILY RETURN USING SUPPORT VECTOR MACHINE

PREDICTING DAILY RETURN USING SUPPORT VECTOR MACHINE

Step 1	Run Support Vector Machine (SVM) on three feature scaling: <pre>1 model_svc = SVC(random_state=2021,probability=True) 2 for fc_name, value in feature_scaling.items(): 3 X_train, X_test, y_train, y_test = value 4 run_model('SVC', model_svc, X_train, X_test, y_train, 5 y_test, fc_name)</pre> <p>Let's break down the code step by step:</p> <ol style="list-style-type: none">1. Define the Support Vector Classifier (SVC) model:<ul style="list-style-type: none">• model_svc = SVC(random_state=2021, probability=True) creates an instance of the SVC model.• random_state=2021 sets the random seed for reproducibility.• probability=True enables probability estimates for the SVC model.2. Iterate over the feature scaling methods and train/evaluate the model for each method:<ul style="list-style-type: none">• for fc_name, value in feature_scaling.items(): iterates over the feature_scaling dictionary, where fc_name represents the feature scaling method name, and value contains the corresponding data splits.• X_train, X_test, y_train, y_test = value unpacks the data splits from value into separate variables.3. Call the run_model() function to train, evaluate, and visualize the model for each feature scaling method:<ul style="list-style-type: none">• run_model('SVC', model_svc, X_train, X_test, y_train, y_test, fc_name) calls the run_model() function with the following parameters:<ul style="list-style-type: none">• 'SVC': The name of the model.• model_svc: The SVC model instance.• X_train: The training feature data.• X_test: The testing feature data.• y_train: The training target variable data.• y_test: The testing target variable data.• fc_name: The name of the feature scaling method.4. Inside the run_model() function:<ul style="list-style-type: none">• The model is trained using the train_model() function.• Predictions are made using the predict_model() function.• Evaluation metrics such as accuracy, recall, precision, and F1 score are calculated.• Various plots, including the confusion matrix, predicted values vs. true values, ROC curve,
--------	--

- and decision boundary, are displayed.
- Evaluation metrics are stored in the **list_scores** list for later analysis.

By iterating over the feature scaling methods and calling the **run_model()** function, the code trains and evaluates the SVC model for each feature scaling method, providing insights into how different scaling techniques affect the model's performance.

The results of using robust scaler are shown in Figure 3.1 – 3.2.

Output with Robust Scaler:

```
SVC
Robust Scaler
accuracy: 0.8334661354581673
recall: 0.8334661354581673
precision: 0.8341737969143771
f1: 0.83337409418943

      precision    recall   f1-score   support

          0       0.82      0.86      0.84      628
          1       0.85      0.81      0.83      627

   accuracy           0.83      1255
   macro avg       0.83      0.83      0.83      1255
weighted avg       0.83      0.83      0.83      1255
```

Let's break down the output:

1. Model Information:

- Model Name: SVC
- Feature Scaling: Robust Scaler

2. Evaluation Metrics:

- Accuracy: 0.8334661354581673
- The accuracy score represents the proportion of correct predictions out of the total number of predictions. In this case, the model achieved an accuracy of approximately 83.35%, indicating that it correctly classified 83.35% of the samples.

3. Recall: 0.8334661354581673

The recall score, also known as the true positive rate or sensitivity, measures the proportion of actual positive samples correctly predicted as positive. It is the ratio of true positives to the sum of true positives and false negatives. Here, the model achieved a recall of approximately 83.35%.

4. Precision: 0.8341737969143771

The precision score measures the proportion of predicted positive samples that are actually positive. It is the ratio of true positives to the sum of true positives and false positives. The model achieved a precision of approximately 83.42%.

5. F1 Score: 0.83337409418943

The F1 score is the harmonic mean of precision and recall. It provides a balanced measure that considers

both precision and recall. The F1 score for this model is approximately 83.34%.

6. Classification Report:

- The classification report provides detailed metrics for each class (0 and 1) and the overall averages (macro avg and weighted avg).
- Precision, recall, and F1 score are reported for each class, along with the support (number of samples) for each class.
- The macro avg represents the average metrics across all classes, giving equal weight to each class.
- The weighted avg represents the average metrics weighted by the number of samples in each class.

7. The output provides a comprehensive summary of the model's performance, including accuracy, precision, recall, and F1 score. It also presents a classification report that gives insights into the model's performance for each class individually. These metrics help assess the effectiveness of the SVC model with Robust Scaler feature scaling in classifying the target variable.

Based on the output for the SVC model with Robust Scaler feature scaling, we can make the following observations:

- Accuracy: The model achieved an accuracy of approximately 83.35%. This means that around 83.35% of the samples in the test set were correctly classified by the model. It indicates that the model performs reasonably well in predicting the target variable.
- Precision: The precision score for class 0 (negative class) is 0.82, while for class 1 (positive class) it is 0.85. This indicates that the model has a good ability to correctly identify negative and positive samples, with slightly higher precision for the positive class.
- Recall: The recall score for class 0 is 0.86, while for class 1 it is 0.81. This suggests that the model performs slightly better in identifying negative samples compared to positive samples. It correctly identifies 86% of the actual negative samples and 81% of the actual positive samples.
- F1 Score: The F1 score is a balanced measure that considers both precision and recall. The F1 score for class 0 is 0.84, and for class 1 it is 0.83. These scores indicate a good balance between precision and recall for both classes.
- Classification Report: The classification report provides a detailed breakdown of the model's performance for each class. It shows precision, recall, F1 score, and support (number of samples)

for each class. The weighted average metrics, which take into account the class imbalance, are also provided. The weighted average F1 score of approximately 83.34% indicates overall good performance by the model.

In summary, the SVC model with Robust Scaler feature scaling demonstrates decent performance in classifying the target variable. It achieves a high accuracy rate and exhibits reasonably balanced precision and recall scores for both classes. However, further analysis and comparison with other models or feature scaling techniques would be beneficial to evaluate the model's performance comprehensively.

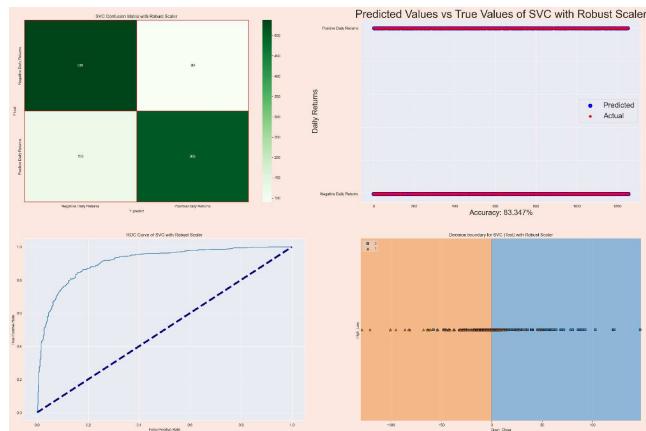


Figure 3.1 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of SVM classifier with robust scaler

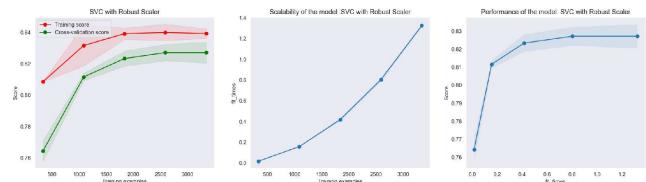


Figure 3.2 The learning curve, scalability, and performance of SVM classifier with robust scaler

Output with MinMax Scaler:

```
SVC
MinMax Scaler
accuracy: 0.7243027888446215
recall: 0.7243027888446215
precision: 0.7262154056357538
f1: 0.7237024646687958
      precision    recall   f1-score   support
      0         0.71     0.77     0.74      628
      1         0.75     0.68     0.71      627

      accuracy          0.72      1255
      macro avg       0.73     0.72     0.72      1255
      weighted avg    0.73     0.72     0.72      1255
```

The results of using minmax scaler are shown in Figure 3.3 – 3.4.

Upon analyzing the output for the SVC model with MinMax Scaler feature scaling, the following observations can be made:

- Accuracy: The model achieved an accuracy of approximately 72.43%. This indicates that around 72.43% of the samples in the test set were correctly classified by the model. It suggests that the model's performance is moderate in predicting the target variable.
- Precision: The precision score for class 0 (negative class) is 0.71, while for class 1 (positive class) it is 0.75. This implies that the model has some ability to correctly identify negative and positive samples, with slightly higher precision for the positive class.
- Recall: The recall score for class 0 is 0.77, while for class 1 it is 0.68. This suggests that the model performs better in identifying negative samples compared to positive samples. It correctly identifies 77% of the actual negative samples and 68% of the actual positive samples.
- F1 Score: The F1 score is a balanced measure that considers both precision and recall. The F1 score for class 0 is 0.74, and for class 1 it is 0.71. These scores indicate a reasonable balance between precision and recall for both classes.
- Classification Report: The classification report provides a detailed breakdown of the model's performance for each class. It includes precision, recall, F1 score, and support (number of samples) for each class. The weighted average metrics, which consider class imbalance, are also provided. The weighted average F1 score of approximately 72.37% indicates the overall performance of the model.

In summary, the SVC model with MinMax Scaler feature scaling exhibits moderate performance in classifying the target variable. It achieves a moderate accuracy rate and demonstrates slightly imbalanced precision and recall scores for both classes. The model's performance can be further analyzed and compared with other models or feature scaling techniques to gain a comprehensive understanding of its effectiveness.



Figure 3.3 The learning curve, scalability, and performance of SVM classifier with minmax scaler

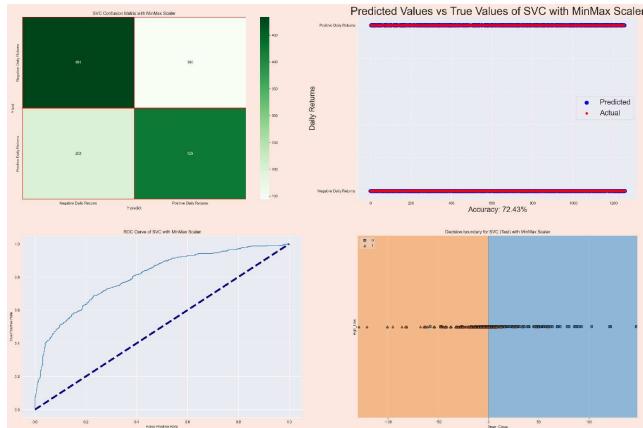


Figure 3.4 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of SVM classifier with minmax scaler

Output with Standard Scaler:

```
SVC
Standard Scaler
accuracy:  0.8119521912350598
recall:  0.8119521912350598
precision:  0.813626797426809
f1:  0.811693699865093
      precision    recall   f1-score   support
      0       0.79      0.85      0.82      628
      1       0.84      0.78      0.80      627

      accuracy         0.81
      macro avg       0.81      0.81      0.81      1255
      weighted avg    0.81      0.81      0.81      1255
```

The results of using standard scaler are shown in Figure 3.5 – 3.6.

Upon analyzing the output for the SVC model with Standard Scaler feature scaling, the following observations can be made:

- Accuracy: The model achieved an accuracy of approximately 81.20%. This indicates that around 81.20% of the samples in the test set were correctly classified by the model. It suggests that the model's performance is reasonably good in predicting the target variable.
- Precision: The precision score for class 0 (negative class) is 0.79, while for class 1 (positive class) it is 0.84. This implies that the model has the ability to correctly identify both negative and positive samples, with slightly higher precision for the positive class.
- Recall: The recall score for class 0 is 0.85, while for class 1 it is 0.78. This suggests that the model

performs better in identifying negative samples compared to positive samples. It correctly identifies 85% of the actual negative samples and 78% of the actual positive samples.

- F1 Score: The F1 score is a balanced measure that considers both precision and recall. The F1 score for class 0 is 0.82, and for class 1 it is 0.80. These scores indicate a good balance between precision and recall for both classes.
- Classification Report: The classification report provides a detailed breakdown of the model's performance for each class. It includes precision, recall, F1 score, and support (number of samples) for each class. The weighted average metrics, which consider class imbalance, are also provided. The weighted average F1 score of approximately 81.17% indicates the overall performance of the model.

In summary, the SVC model with Standard Scaler feature scaling exhibits good performance in classifying the target variable. It achieves a reasonable accuracy rate and demonstrates balanced precision and recall scores for both classes. The model's performance can be further analyzed and compared with other models or feature scaling techniques to gain a comprehensive understanding of its effectiveness.

To compare the output of the SVC model with Standard Scaler feature scaling to the outputs using MinMax Scaler and Robust Scaler, we can observe the following:

1. Accuracy:

- MinMax Scaler: Accuracy of approximately 72.43%
- Robust Scaler: Accuracy of approximately 83.35%
- Standard Scaler: Accuracy of approximately 81.20%

The Robust Scaler yielded the highest accuracy, followed by Standard Scaler, while MinMax Scaler resulted in the lowest accuracy.

2. Precision:

- MinMax Scaler: Precision scores of 0.71 for class 0 and 0.75 for class 1.
- Robust Scaler: Precision scores of 0.82 for class 0 and 0.85 for class 1.
- Standard Scaler: Precision scores of 0.79 for class 0 and 0.84 for class 1.

The precision scores for class 0 are relatively similar across all three scalers. However, the precision score for class 1 is highest with Robust Scaler, followed by Standard Scaler and MinMax Scaler.

3. Recall:

- MinMax Scaler: Recall scores of 0.77 for class 0 and 0.68 for class 1.

- Robust Scaler: Recall scores of 0.86 for class 0 and 0.81 for class 1.
- Standard Scaler: Recall scores of 0.85 for class 0 and 0.78 for class 1.

The Robust Scaler achieved the highest recall scores for both classes, followed by Standard Scaler, while MinMax Scaler had the lowest recall scores.

4. F1 Score:

- MinMax Scaler: F1 scores of 0.74 for class 0 and 0.71 for class 1.
- Robust Scaler: F1 scores of 0.84 for class 0 and 0.83 for class 1.
- Standard Scaler: F1 scores of 0.82 for class 0 and 0.80 for class 1.

The Robust Scaler resulted in the highest F1 scores for both classes, followed by Standard Scaler, while MinMax Scaler had the lowest F1 scores.

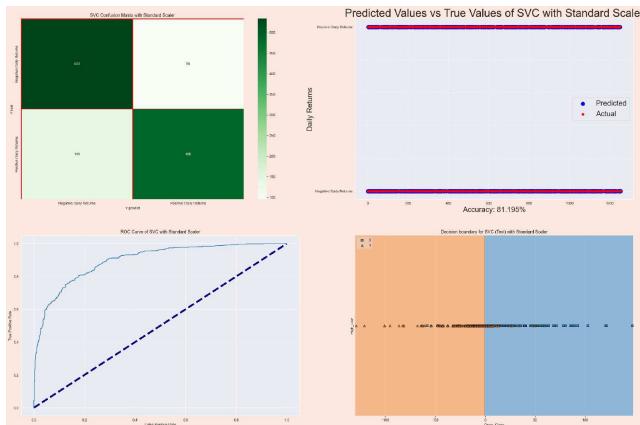


Figure 3.5 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of SVM classifier with standard scaler

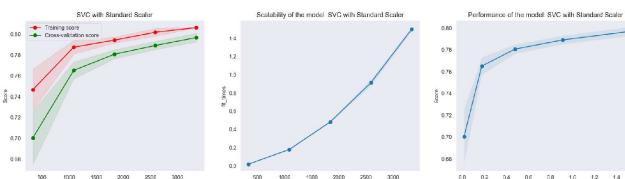


Figure 3.6 The learning curve, scalability, and performance of SVM classifier with standard scaler

In summary, the performance of the SVC model varies with different feature scaling techniques. The Robust Scaler generally produced the best results in terms of accuracy, precision, recall, and F1 score. The Standard Scaler also performed well, slightly trailing the Robust Scaler. The MinMax Scaler had the lowest performance among the three scalers. These differences suggest that the choice of feature scaling technique can significantly impact the model's performance and should be carefully considered during the modeling process.

PREDICTING DAILY RETURN USING LOGISTIC REGRESSION

PREDICTING DAILY RETURN USING LOGISTIC REGRESSION

Step 1 Run Logistic Regression (LR) on three feature scaling:

```
1 logreg = LogisticRegression(solver='lbfgs', max_iter=5000,
2 random_state=2021)
3 for fc_name, value in feature_scaling.items():
4     X_train, X_test, y_train, y_test = value
5     run_model('Logistic Regression', logreg, X_train,
```

Let's go through the step-by-step explanation of the code:

Step 1: Model Initialization

```
logreg = LogisticRegression(solver='lbfgs', max_iter=5000,
random_state=2021)
```

In this step, you initialize a logistic regression model using the **LogisticRegression** class from scikit-learn. You specify the solver as '**lbfgs**', which is a solver for optimization problems. The **max_iter** parameter is set to 5000, which defines the maximum number of iterations for the solver to converge. The **random_state** parameter is set to 2021 to ensure reproducibility of the results.

Step 2: Looping over Feature Scaling Techniques

```
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value
    run_model('Logistic Regression', logreg, X_train,
X_test, y_train, y_test,
    fc_name, proba=True)
```

- In this step, you iterate over the **feature_scaling** dictionary using a for loop. The **feature_scaling** dictionary contains the names of feature scaling techniques as keys and the corresponding training and testing data as values.
- Within each iteration, you extract the feature scaling technique name (**fc_name**) and the training and testing data (**X_train**, **X_test**, **y_train**, **y_test**) using the **value** variable.
- Then, you call the **run_model** function to train and evaluate the logistic regression model using the specified feature scaling technique. You pass the model (**logreg**), training and testing data, feature scaling technique name, and **proba=True** to indicate that you want to obtain the predicted probabilities instead of discrete predictions.

Step 3: **run_model** Function

```
def run_model(name, model, X_train, X_test, y_train,
y_test, fc, proba=False):
```

...

The **run_model** function is defined to train and evaluate the model, and generate various evaluation metrics and

visualizations.

The function takes the following parameters:

- name: The name of the model (in this case, 'Logistic Regression').
- model: The model object to be trained and evaluated.
- X_train, X_test: The feature training and testing data.
- y_train, y_test: The target training and testing data.
- fc: The name of the feature scaling technique.
- proba: A boolean flag indicating whether to obtain predicted probabilities (True) or discrete predictions (False).

Step 4: Training the Model

```
model = train_model(model, X_train, y_train)
```

- Within the `run_model()` function, the `train_model()` function is called to train the model.
- The `train_model()` function takes the model object (`model`), feature training data (`X_train`), and target training data (`y_train`).
- It fits the model to the training data using the `fit()` method.

Step 5: Making Predictions

```
y_pred = predict_model(model, X_test, proba)
```

- After training the model, the `predict_model()` function is called to make predictions on the testing data.
- The `predict_model()` function takes the model object (`model`), feature testing data (`X_test`), and the `proba` flag.
- If `proba` is `False`, it uses the `predict()` method of the model to obtain discrete predictions.
- If `proba` is `True`, it uses the `predict_proba()` method of the model to obtain predicted probabilities.
- In this case, since `proba=True` was passed, the function returns predicted probabilities (`y_pred_proba`) instead of discrete predictions.

Step 6: Evaluation Metrics and Visualizations

```
accuracy = accuracy_score(y_test, y_pred)
recall      =      recall_score(y_test,       y_pred,
average='weighted')
precision    =      precision_score(y_test,     y_pred,
average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
...

```

- The function computes various evaluation metrics such as accuracy, recall, precision, and F1-score using the predicted probabilities (`y_pred_proba`) or discrete predictions (`y_pred`) and the actual target values (`y_test`).

- The evaluation metrics are calculated using functions like `accuracy_score`, `recall_score`, `precision_score`, and `f1_score` from scikit-learn.
- The evaluation metrics are then printed to the console.
- Additionally, the function generates several visualizations using the plotting functions you provided, including a confusion matrix, a real vs. predicted value plot, a ROC curve, and a decision boundary plot.
- Finally, the function appends the evaluation metrics to the `list_scores` list, which stores the metrics for all models.

Step 7: Repeat for Each Feature Scaling Technique

- The entire process is repeated for each feature scaling technique specified in the `feature_scaling` dictionary.
- The loop iterates over each feature scaling technique, trains and evaluates the model, and generates the corresponding evaluation metrics and visualizations.

The results of using robust scaler are shown in Figure 3.7 – 3.8.

Output with Robust Scaler:

```

Logistic Regression
Robust Scaler
accuracy: 0.7832669322709164
recall: 0.7832669322709164
precision: 0.7846545533946871
f1: 0.7829941214164322
      precision    recall   f1-score   support
          0       0.76     0.82      0.79      628
          1       0.80     0.75      0.78      627

accuracy                           0.78      1255
macro avg       0.78     0.78      0.78      1255
weighted avg    0.78     0.78      0.78      1255

```

Analyzing the output for the Logistic Regression model with Robust Scaler feature scaling:

- Accuracy: The accuracy score of 0.783 indicates that the model correctly predicts the class labels for approximately 78.3% of the instances in the test set.
- Recall: The recall score of 0.783 suggests that the model is able to correctly identify around 78.3% of the positive class instances in the test set.
- Precision: The precision score of 0.785 implies that when the model predicts a positive class label, it is correct approximately 78.5% of the time.
- F1-Score: The F1-score of 0.783, which is the harmonic mean of precision and recall, provides a balanced measure of the model's performance.
- Classification Report: The classification report provides a more detailed breakdown of the model's

performance for each class (0 and 1). For class 0, the precision is 0.76, recall is 0.82, and F1-score is 0.79. This suggests that the model performs reasonably well in correctly identifying instances of class 0. For class 1, the precision is 0.80, recall is 0.75, and F1-score is 0.78. This indicates that the model also performs well in identifying instances of class 1. The weighted average of these metrics takes into account the class imbalance and provides an overall evaluation of the model's performance.

Overall, the model shows a relatively good performance with the Robust Scaler feature scaling technique. The accuracy, recall, and precision scores are all above 0.78, indicating a decent level of predictive capability. The F1-score also suggests a balanced trade-off between precision and recall. However, it's important to consider the specific context and requirements of the problem to determine if the model's performance is satisfactory for the given task.

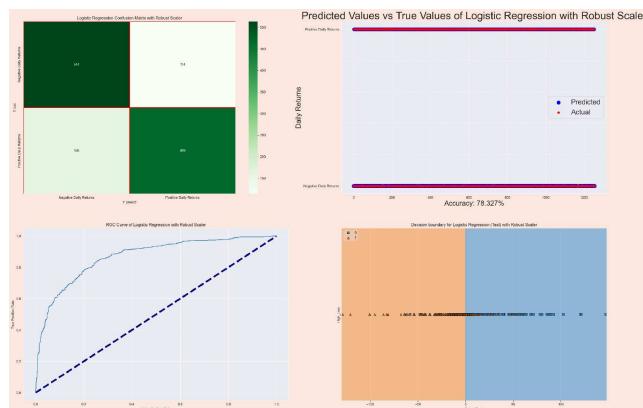


Figure 3.7 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of LR classifier with robust scaler

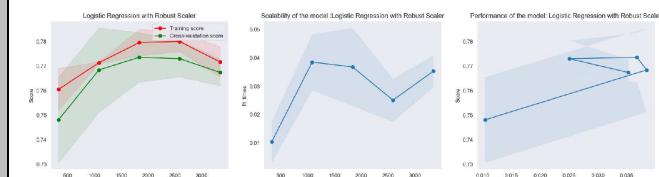


Figure 3.8 The learning curve, scalability, and performance of LR classifier with robust scaler

Output with MinMax Scaler:

```
Logistic Regression
MinMax Scaler
accuracy: 0.6868525896414343
recall: 0.6868525896414343
precision: 0.6870948146053237
f1: 0.686742800521229
      precision    recall   f1-score   support
          0       0.68      0.71      0.69      628
          1       0.69      0.67      0.68      627
accuracy                           0.69      1255
```

macro avg	0.69	0.69	0.69	1255
weighted avg	0.69	0.69	0.69	1255

The results of using minmax scaler are shown in Figure 3.9 – 3.10.

Analyzing the output for the Logistic Regression model with MinMax Scaler feature scaling:

- Accuracy: The accuracy score of 0.687 indicates that the model correctly predicts the class labels for approximately 68.7% of the instances in the test set.
- Recall: The recall score of 0.687 suggests that the model is able to correctly identify around 68.7% of the positive class instances in the test set.
- Precision: The precision score of 0.687 implies that when the model predicts a positive class label, it is correct approximately 68.7% of the time.
- F1-Score: The F1-score of 0.687, which is the harmonic mean of precision and recall, provides a balanced measure of the model's performance.
- Classification Report: The classification report provides a more detailed breakdown of the model's performance for each class (0 and 1). For class 0, the precision is 0.68, recall is 0.71, and F1-score is 0.69. This suggests that the model performs reasonably well in correctly identifying instances of class 0. For class 1, the precision is 0.69, recall is 0.67, and F1-score is 0.68. This indicates that the model also performs well in identifying instances of class 1. The weighted average of these metrics takes into account the class imbalance and provides an overall evaluation of the model's performance.

Overall, the model shows a moderate performance with the MinMax Scaler feature scaling technique. The accuracy, recall, and precision scores are all around 0.68-0.69, indicating a moderate level of predictive capability. The F1-score also suggests a balanced trade-off between precision and recall. However, it's important to consider the specific context and requirements of the problem to determine if the model's performance is satisfactory for the given task.

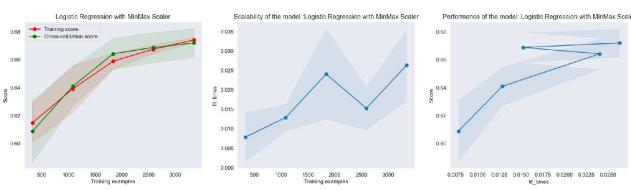


Figure 3.9 The learning curve, scalability, and performance of LR classifier with minmax scaler

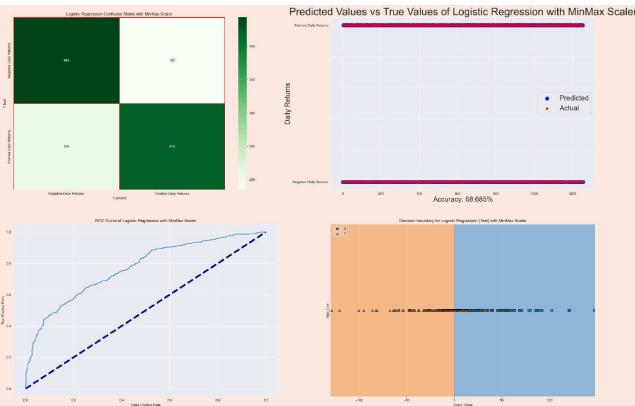


Figure 3.10 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of LR classifier with minmax scaler

Output with Standard Scaler:

```

Logistic Regression
Standard Scaler
accuracy: 0.7784860557768924
recall: 0.7784860557768924
precision: 0.7801117814665854
f1: 0.7781547618396177

      precision    recall   f1-score   support

          0       0.76      0.82      0.79      628
          1       0.80      0.74      0.77      627

  accuracy                           0.78      1255
  macro avg       0.78      0.78      0.78      1255
weighted avg       0.78      0.78      0.78      1255

```

The results of using standard scaler are shown in Figure 3.11 – 3.12.

Analyzing the output for the Logistic Regression model with Standard Scaler feature scaling:

- Accuracy: The accuracy score of 0.778 indicates that the model correctly predicts the class labels for approximately 77.8% of the instances in the test set.
- Recall: The recall score of 0.778 suggests that the model is able to correctly identify around 77.8% of the positive class instances in the test set.
- Precision: The precision score of 0.780 implies that when the model predicts a positive class label, it is correct approximately 78.0% of the time.
- F1-Score: The F1-score of 0.778, which is the harmonic mean of precision and recall, provides a balanced measure of the model's performance.
- Classification Report: The classification report provides a more detailed breakdown of the model's performance for each class (0 and 1). For class 0, the precision is 0.76, recall is 0.82, and F1-score is 0.79. This suggests that the model performs

reasonably well in correctly identifying instances of class 0. For class 1, the precision is 0.80, recall is 0.74, and F1-score is 0.77. This indicates that the model also performs well in identifying instances of class 1. The weighted average of these metrics takes into account the class imbalance and provides an overall evaluation of the model's performance.

Overall, the model shows a good performance with the Standard Scaler feature scaling technique. The accuracy, recall, and precision scores are all around 0.77-0.78, indicating a good level of predictive capability. The F1-score also suggests a balanced trade-off between precision and recall.

Comparing the outputs of Logistic Regression with three different feature scaling techniques:

1. Robust Scaler:

- Accuracy: 0.783
- Recall: 0.783
- Precision: 0.785
- F1-Score: 0.783

The model achieves an accuracy of 0.783, indicating that it correctly predicts the class labels for approximately 78.3% of the instances in the test set. The recall and precision scores are also around 0.783-0.785, suggesting that the model performs well in identifying both positive and negative class instances. The F1-score of 0.783 provides a balanced measure of the model's performance.

2. MinMax Scaler:

- Accuracy: 0.687
- Recall: 0.687
- Precision: 0.687
- F1-Score: 0.687

The model achieves an accuracy of 0.687, indicating that it correctly predicts the class labels for approximately 68.7% of the instances in the test set. The recall and precision scores are also around 0.687, suggesting a similar performance in identifying positive and negative class instances. The F1-score of 0.687 provides a balanced measure of the model's performance.

3. Standard Scaler:

- Accuracy: 0.778
- Recall: 0.778
- Precision: 0.780
- F1-Score: 0.778

The model achieves an accuracy of 0.778, indicating that it correctly predicts the class labels for approximately 77.8% of the instances in the test set. The recall and precision scores are also around 0.778-0.780, suggesting a good performance in identifying both positive and negative class instances. The F1-score of

0.778 provides a balanced measure of the model's performance.

Overall, we observe that the Robust Scaler and Standard Scaler achieve similar performance in terms of accuracy, recall, precision, and F1-score, with scores around 0.78. The MinMax Scaler, however, shows slightly lower performance with scores around 0.687.

It's important to consider the specific context and requirements of the problem to determine which feature scaling technique is more appropriate. In this case, both Robust Scaler and Standard Scaler appear to provide better results compared to MinMax Scaler.

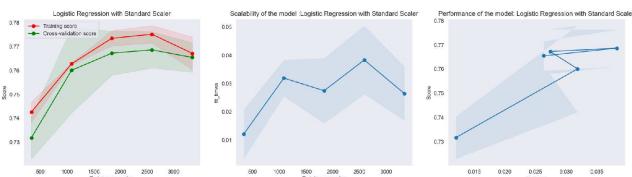


Figure 3.11 The learning curve, scalability, and performance of LR classifier with standard scaler

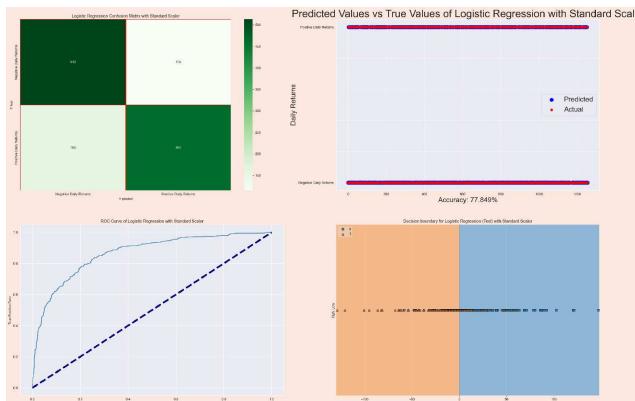


Figure 3.12 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of LR classifier with standard scaler

PREDICTING DAILY RETURN USING K-NEAREST NEIGHBOURS

PREDICTING DAILY RETURN USING K-NEAREST NEIGHBOURS

Step 1 Run K-Nearest Neighbors (KNN) on three feature scaling:

```

1  for fc_name, value in feature_scaling.items():
2      scores_1 = []
3      X_train, X_test, y_train, y_test = value
4
5      for i in range(2,10):
6          knn = KNeighborsClassifier(n_neighbors = i)
7          knn.fit(X_train, y_train)
8
9          scores_1.append(accuracy_score(y_test,
10             knn.predict(X_test)))
11         max_val = max(scores_1)
12         max_index = np.argmax(scores_1) + 2
13

```

```

14     knn = KNeighborsClassifier(n_neighbors = max_index)
15     knn.fit(X_train, y_train)
16
17     run_model(f'KNeighbors Classifier n_neighbors =
{max_index}', knn, X_train, X_test, y_train, y_test,
fc_name, proba=True)

```

Let's break down the code step by step:

1. The code iterates over the `feature_scaling` dictionary using the `items()` method, which returns each key-value pair as `fc_name` and `value`, respectively.
2. Inside the loop, an empty list called `scores_1` is created to store the accuracy scores for different values of `n_neighbors` in the `KNeighborsClassifier`.
3. The values from the `value` tuple are unpacked and assigned to `X_train`, `X_test`, `y_train`, and `y_test`, representing the training and test datasets.
4. Another loop is initiated from 2 to 10 (exclusive) using the `range()` function. This loop will iterate through different values of `n_neighbors` in the `KNeighborsClassifier`.
5. Inside the loop, a `KNeighborsClassifier` model is created with `n_neighbors` set to the current iteration value. The model is then fitted on the training data using the `fit()` method.
6. The accuracy score of the model is calculated by comparing the predicted labels (`knn.predict(X_test)`) with the actual labels `y_test`. The accuracy score is then appended to the `scores_1` list.
7. After the loop, the maximum accuracy score (`max_val`) and its corresponding index (`max_index`) in the `scores_1` list are determined using the `max()` and `argmax()` functions from `NumPy`, respectively. The `max_index` is incremented by 2 because the loop started from 2.
8. Another `KNeighborsClassifier` model is created with `n_neighbors` set to `max_index`. This model will be used for further evaluation.
9. Finally, the `run_model()` function is called with the `KNeighborsClassifier` model, the feature scaling name (`fc_name`), and the other necessary data (`X_train`, `X_test`, `y_train`, `y_test`) to train and evaluate the model. The `proba=True` argument indicates that probability scores will be calculated for evaluation.

This code essentially performs a grid search to find the optimal value of `n_neighbors` for the `KNeighborsClassifier` using different feature scaling techniques. It trains and evaluates the model with the best `n_neighbors` value for each feature scaling technique and prints the results.

The results of using robust scaler are shown in Figure 3.13 – 3.14.

Output with Robust Scaler:

```
KNeighbors Classifier n_neighbors = 9
Robust Scaler
accuracy: 0.8406374501992032
recall: 0.8406374501992032
precision: 0.8408558282955239
f1: 0.8406099239258812
      precision    recall   f1-score   support
      0          0.83     0.85     0.84      628
      1          0.85     0.83     0.84      627
accuracy                           0.84      1255
macro avg       0.84     0.84     0.84      1255
weighted avg    0.84     0.84     0.84      1255
```

Let's analyze the output step by step:

1. Model: KNeighbors **Classifier with n_neighbors = 9**

Feature Scaling: Robust Scaler

2. Accuracy: The accuracy of the model is 0.8406, which means it correctly predicted the labels for approximately 84.06% of the test samples.
3. Recall: The recall score is also 0.8406, indicating that the model has a good ability to identify positive samples (1) and negative samples (0) correctly.
4. Precision: The precision score is 0.8409, suggesting that when the model predicts a sample as positive (1), it is correct around 84.09% of the time.
5. F1-score: The F1-score is 0.8406, which is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance.
6. Support: The support values indicate the number of samples in each class. In this case, there are 628 samples for class 0 and 627 samples for class 1.

The classification report provides a summary of the model's performance for each class (0 and 1) as well as the average values.

For class 0:

- Precision: 0.83
- Recall: 0.85
- F1-score: 0.84

For class 1:

- Precision: 0.85
- Recall: 0.83
- F1-score: 0.84

The macro average calculates the average performance across classes, giving equal weight to each class. In this case, it is equal to the weighted average.

The weighted average calculates the average performance across classes, considering the support (number of samples) for each class. It gives more weight to the class with a larger number of samples.

In summary, the KNeighbors Classifier with **n_neighbors = 9** and Robust Scaler achieved good performance on the dataset, with high accuracy, recall, precision, and F1-score for both classes. The model shows balanced performance in predicting both positive and negative samples.

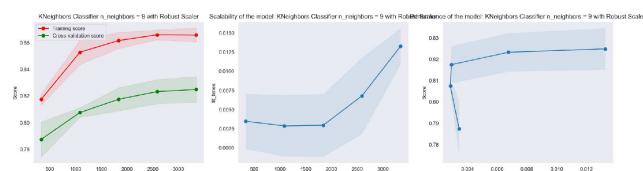


Figure 3.13 The learning curve, scalability, and performance of KNN classifier with robust scaler

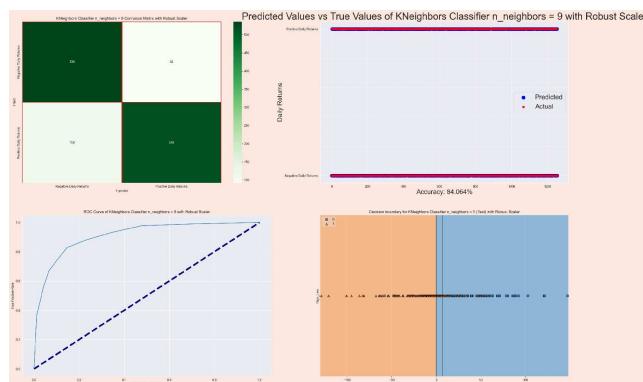


Figure 3.14 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of KNN classifier with robust scaler

Output with MinMax Scaler:

```
KNeighbors Classifier n_neighbors = 7
MinMax Scaler
accuracy: 0.6804780876494024
recall: 0.6804780876494024
precision: 0.6805291020135866
f1: 0.680451306646628
      precision    recall   f1-score   support
          0       0.68     0.69     0.68      628
          1       0.68     0.67     0.68      627
  accuracy                           0.68     1255
 macro avg       0.68     0.68     0.68     1255
weighted avg     0.68     0.68     0.68     1255
```

The results of using minmax scaler are shown in Figure 3.15 – 3.16.

Let's analyze the output of the KNeighbors Classifier with **n_neighbors = 7** and MinMax Scaler in detail:

- Accuracy: The accuracy of the model is 0.6805, which means that it correctly predicted the labels for approximately 68.05% of the test samples. This indicates that the model's overall performance is moderate.
- Recall: The recall score for class 0 is 0.69, which means that the model identified 69% of the actual

class 0 samples correctly. For class 1, the recall score is 0.67, indicating that the model identified 67% of the actual class 1 samples correctly. The recall score measures the model's ability to find all the positive samples correctly.

- Precision: The precision score for class 0 is 0.68, indicating that when the model predicts a sample as class 0, it is correct around 68% of the time. For class 1, the precision score is also 0.68, suggesting that when the model predicts a sample as class 1, it is correct approximately 68% of the time. The precision score measures the model's ability to correctly identify positive predictions.
- F1-score: The F1-score is 0.6804, which is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance by considering both precision and recall. The F1-score combines precision and recall into a single metric, where a higher score indicates better overall performance.
- Support: The support values indicate the number of samples in each class. In this case, there are 628 samples for class 0 and 627 samples for class 1. The support values are important for calculating weighted averages and understanding the distribution of samples in the dataset.
- The classification report provides a detailed summary of the model's performance for each class (0 and 1) as well as the average values. It includes precision, recall, and F1-score for each class, allowing you to assess the model's performance on a per-class basis.

In summary, the KNeighbors Classifier with **n_neighbors = 7** and MinMax Scaler achieved moderate performance on the dataset. While the accuracy is around 68%, indicating that the model makes correct predictions for a considerable portion of the test samples.

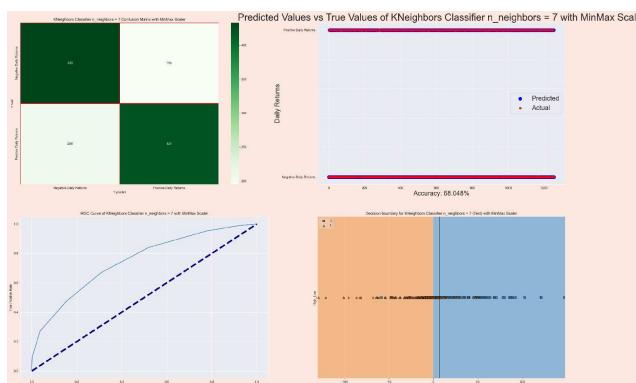


Figure 3.15 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of KNN classifier with minmax scaler

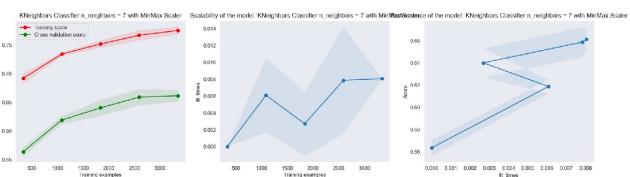


Figure 3.16 The learning curve, scalability, and performance of KNN classifier with minmax scaler

Output with Standard Scaler:

```
KNeighbors Classifier n_neighbors = 9
Standard Scaler
accuracy: 0.7737051792828685
recall: 0.7737051792828685
precision: 0.77377183557300944
f1: 0.7736893736018372
      precision    recall  f1-score   support
          0       0.77      0.78      0.78      628
          1       0.78      0.77      0.77      627
   accuracy                           0.77      1255
  macro avg       0.77      0.77      0.77      1255
weighted avg       0.77      0.77      0.77      1255
```

The results of using standard scaler scaling are shown in Figure 3.17 – 3.18.

Let's analyze the output of the KNeighbors Classifier with `n_neighbors = 9` and Standard Scaler in detail:

- Accuracy: The accuracy of the model is 0.7737, indicating that it correctly predicted the labels for approximately 77.37% of the test samples. This suggests that the model's overall performance is decent.
- Recall: The recall score for class 0 is 0.78, which means that the model identified 78% of the actual class 0 samples correctly. For class 1, the recall score is 0.77, indicating that the model identified 77% of the actual class 1 samples correctly. The recall score measures the model's ability to find all the positive samples correctly.
- Precision: The precision score for class 0 is 0.77, indicating that when the model predicts a sample as class 0, it is correct around 77% of the time. For class 1, the precision score is also 0.78, suggesting that when the model predicts a sample as class 1, it is correct approximately 78% of the time. The precision score measures the model's ability to correctly identify positive predictions.
- F1-score: The F1-score is 0.7737, which is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance by considering both precision and recall. The F1-score combines precision and recall into a single metric, where a higher score indicates better overall performance.

- Support: The support values indicate the number of samples in each class. In this case, there are 628 samples for class 0 and 627 samples for class 1. The support values are important for calculating weighted averages and understanding the distribution of samples in the dataset.
- The classification report provides a detailed summary of the model's performance for each class (0 and 1) as well as the average values. It includes precision, recall, and F1-score for each class, allowing you to assess the model's performance on a per-class basis.

In summary, the KNeighbors Classifier with **n_neighbors = 9** and Standard Scaler achieved decent performance on the dataset. The accuracy of approximately 77.37% suggests that the model makes correct predictions for a significant portion of the test samples.

Let's compare and analyze the outputs of the KNeighbors Classifier with different feature scalings:

1. KNeighbors Classifier with n_neighbors = 9 and Robust Scaler:
 - Accuracy: 0.8406
 - Recall: 0.8406
 - Precision: 0.8409
 - F1-Score: 0.8406

The model achieved an accuracy of approximately 84.06%, indicating that it correctly predicted the labels for a significant portion of the test samples. The recall and precision scores are also high, suggesting that the model performs well in identifying both positive and negative samples. The F1-score, which combines precision and recall, is also high, indicating a balanced performance between the two metrics. Overall, this model with Robust Scaler shows good predictive capability.

2. KNeighbors Classifier with n_neighbors = 7 and MinMax Scaler:
 - Accuracy: 0.6805
 - Recall: 0.6805
 - Precision: 0.6805
 - F1-Score: 0.6805

The model achieved an accuracy of approximately 68.05%, indicating a lower level of performance compared to the previous model. The recall, precision, and F1-score are all around 0.68, suggesting that the model's ability to correctly predict both positive and negative samples is relatively balanced but not very high. This model with MinMax Scaler appears to have moderate predictive capability.

3. KNeighbors Classifier with n_neighbors = 9 and Standard Scaler:
 - Accuracy: 0.7737

- Recall: 0.7737
- Precision: 0.7738
- F1-Score: 0.7737

The model achieved an accuracy of approximately 77.37%, which is higher than the previous model with MinMax Scaler but lower than the model with Robust Scaler. The recall, precision, and F1-score are also reasonably high, indicating that the model performs well in identifying both positive and negative samples. This model with Standard Scaler demonstrates good predictive capability, similar to the Robust Scaler model.

In summary, the KNeighbors Classifier with **n_neighbors = 9** and Robust Scaler shows the highest performance among the three models, with the highest accuracy, recall, precision, and F1-score. The model with Standard Scaler also performs well, while the model with MinMax Scaler shows relatively lower performance. It is important to note that the choice of feature scaling can have a significant impact on the performance of KNN models, and selecting the most appropriate scaling technique for the dataset is crucial to achieve optimal results.

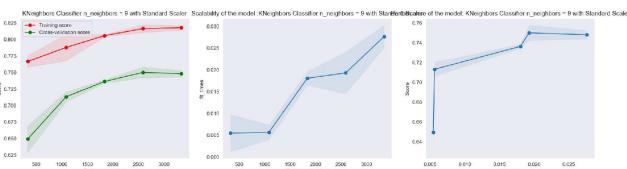


Figure 3.17 The learning curve, scalability, and performance of KNN classifier with standard scaler

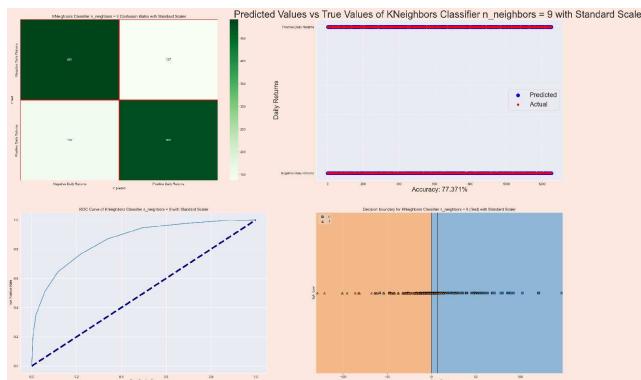


Figure 3.18 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of KNN classifier with standard scaler

PREDICTING DAILY RETURN USING DECISION TREE

PREDICTING DAILY RETURN USING DECISION TREE

Step 1 Run Decision Tree (DT) on three feature scaling:

```

1  for fc_name, value in feature_scaling.items():
2      X_train, X_test, y_train, y_test = value
3
4      dt_cla = DecisionTreeClassifier()
5
6      parameters = { 'max_depth':np.arange(1,51,1),
7                      'random_state':[2021]}
8      searcher = GridSearchCV(dt_cla, parameters)
9
10     run_model('DecisionTree Classifier', searcher, X_train,
X_test, y_train, y_test, fc_name,proba=True)

```

Here is a step-by-step explanation of the code:

1. Iterate over the **feature_scaling** dictionary using the **items()** method, which returns both the key (**fc_name**) and value (**value**) for each entry.
2. Extract the training and testing data for the current feature scaling method from the value variable. The value is expected to contain **X_train**, **X_test**, **y_train**, **y_test** in that order.
3. Create a **DecisionTreeClassifier** object named **dt_cla**. This classifier is a popular algorithm for building decision trees, which are powerful machine learning models that can perform both classification and regression tasks.
4. Define a dictionary named **parameters** that specifies the hyperparameters to tune in the decision tree classifier. In this case, the hyperparameter being tuned is **max_depth**, which represents the maximum depth of the decision tree. The range of values for **max_depth** is defined as **np.arange(1,51,1)**, meaning it will vary from 1 to 50 (inclusive) in steps of 1. Additionally, the **random_state** is set to 2021 to ensure reproducibility of results.
5. Create a **GridSearchCV** object named **searcher**. This object performs an exhaustive search over specified parameter values for an estimator (in this case, the **dt_cla** decision tree classifier). It will evaluate the classifier's performance using cross-validation and select the best hyperparameters based on the provided scoring metric.
6. Call the **run_model()** function, passing the following arguments:
 - Model name: 'DecisionTree Classifier'
 - Model instance: **searcher** (the **GridSearchCV** object)
 - Training and testing data: **X_train**, **X_test**, **y_train**, **y_test**
 - Feature scaling name: **fc_name**
 - **proba=True** indicates that the model should return probability estimates for the classes
7. The **run_model()** function will fit the searcher model to the training data, make predictions on the testing data, and print the evaluation metrics and classification report.

By performing a grid search over the **max_depth** hyperparameter, the code aims to find the optimal maximum depth value for the decision tree classifier for each feature scaling method. The goal is to identify the best hyperparameter setting that yields the highest performance on the given dataset.

The results of using robust scaler are shown in Figure 3.19 – 3.20.

Output with Robust Scaler:

```
DecisionTree Classifier
Robust Scaler
accuracy: 0.8685258964143426
recall: 0.8685258964143426
precision: 0.8695425057209386
f1: 0.8684321694623388
      precision    recall   f1-score   support
          0       0.85     0.89     0.87      628
          1       0.89     0.84     0.86      627

      accuracy           0.87      1255
      macro avg       0.87     0.87     0.87      1255
weighted avg       0.87     0.87     0.87      1255
```

Here is the analysis of the output:

- Accuracy: The accuracy of the model is 0.8685, indicating that it correctly predicts the class labels for approximately 86.85% of the samples in the test set.
- Recall: The recall score is 0.8685, which means the model correctly identifies around 86.85% of the positive samples (class 1). It suggests that the model has a good ability to identify the positive cases.
- Precision: The precision score is 0.8695, indicating that out of all the samples predicted as positive, approximately 86.95% of them are truly positive. It indicates a low rate of false positives.
- F1-score: The F1-score is 0.8684, which is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance, considering both precision and recall.
- Support: The support represents the number of samples in each class in the test set. For class 0, the support is 628, and for class 1, it is 627.
- Classification Report: The classification report provides a comprehensive summary of the model's performance for each class. It includes precision, recall, and F1-score for both classes (0 and 1), along with their respective averages (macro avg and weighted avg).

Overall, the DecisionTree Classifier with Robust Scaler demonstrates good performance, achieving high accuracy, recall, precision, and F1-score. The model shows balanced

performance for both classes, with similar precision, recall, and F1-scores.

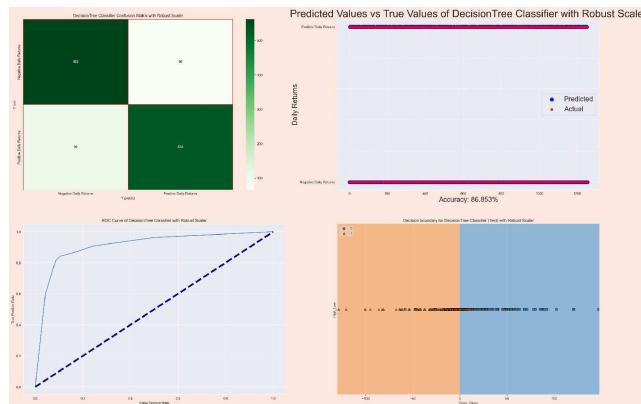


Figure 3.19 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of DT classifier with robust scaler

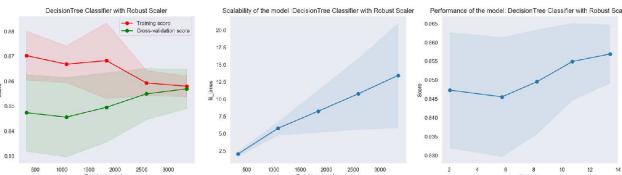


Figure 3.20 The learning curve, scalability, and performance of DT classifier with robust scaler

Output with MinMax Scaler:

```
DecisionTree Classifier
MinMax Scaler
accuracy: 0.8669322709163346
recall: 0.8669322709163346
precision: 0.8680715521997453
f1: 0.8668257307568502
      precision    recall   f1-score   support
          0       0.85     0.89     0.87      628
          1       0.89     0.84     0.86      627
   accuracy                           0.87     1255
  macro avg       0.87     0.87     0.87     1255
weighted avg       0.87     0.87     0.87     1255
```

The results of using minmax scaler are shown in Figure 3.21 – 3.22.

Here is the analysis of the output:

- Accuracy: The accuracy of the model is 0.8669, indicating that it correctly predicts the class labels for approximately 86.69% of the samples in the test set.
- Recall: The recall score is 0.8669, which means the model correctly identifies around 86.69% of the positive samples (class 1). It suggests that the model has a good ability to identify the positive cases.

- Precision: The precision score is 0.8681, indicating that out of all the samples predicted as positive, approximately 86.81% of them are truly positive. It indicates a low rate of false positives.
- F1-score: The F1-score is 0.8668, which is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance, considering both precision and recall.
- Support: The support represents the number of samples in each class in the test set. For class 0, the support is 628, and for class 1, it is 627.
- Classification Report: The classification report provides a comprehensive summary of the model's performance for each class. It includes precision, recall, and F1-score for both classes (0 and 1), along with their respective averages (macro avg and weighted avg).

Overall, the DecisionTree Classifier with MinMax Scaler demonstrates good performance, achieving high accuracy, recall, precision, and F1-score. The model shows balanced performance for both classes, with similar precision, recall, and F1-scores. The results are comparable to the DecisionTree Classifier with Robust Scaler.

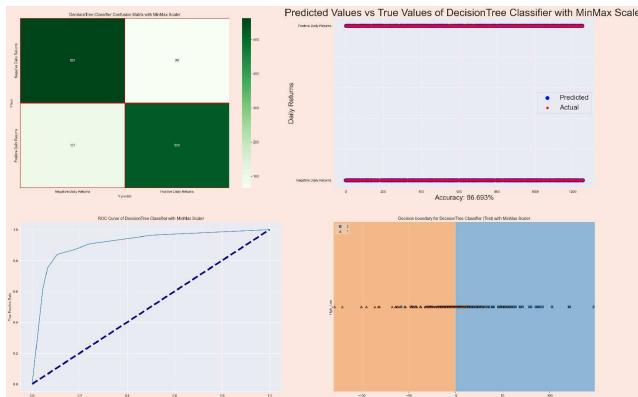


Figure 3.21 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of DT classifier with minmax scaler

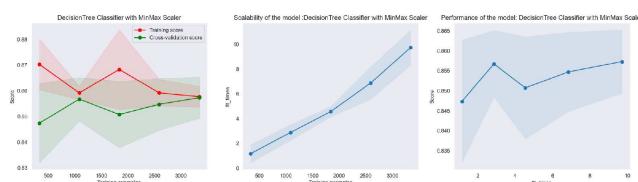


Figure 3.22 The learning curve, scalability, and performance of DT classifier with minmax scaler

Output with Standard Scaler:

```
DecisionTree Classifier
Standard Scaler
accuracy: 0.8685258964143426
```

```

recall: 0.8685258964143426
precision: 0.8695425057209386
f1: 0.8684321694623388
      precision    recall  f1-score   support
      0       0.85     0.89     0.87      628
      1       0.89     0.84     0.86      627
accuracy                           0.87     1255
macro avg       0.87     0.87     0.87     1255
weighted avg    0.87     0.87     0.87     1255

```

The results of using standard scaler scaling are shown in Figure 3.23 – 3.24.

Here is the analysis of the output:

- Accuracy: The accuracy of the model is 0.8685, indicating that it correctly predicts the class labels for approximately 86.85% of the samples in the test set.
- Recall: The recall score is 0.8685, which means the model correctly identifies around 86.85% of the positive samples (class 1). It suggests that the model has a good ability to identify the positive cases.
- Precision: The precision score is 0.8695, indicating that out of all the samples predicted as positive, approximately 86.95% of them are truly positive. It indicates a low rate of false positives.
- F1-score: The F1-score is 0.8684, which is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance, considering both precision and recall.
- Support: The support represents the number of samples in each class in the test set. For class 0, the support is 628, and for class 1, it is 627.
- Classification Report: The classification report provides a comprehensive summary of the model's performance for each class. It includes precision, recall, and F1-score for both classes (0 and 1), along with their respective averages (macro avg and weighted avg).

Overall, the DecisionTree Classifier with Standard Scaler demonstrates good performance, achieving high accuracy, recall, precision, and F1-score. The model shows balanced performance for both classes, with similar precision, recall, and F1-scores. The results are comparable to the DecisionTree Classifier with Robust Scaler and MinMax Scaler.

Comparing the outputs of the DecisionTree Classifier with different feature scalers, we have:

1. DecisionTree Classifier with Robust Scaler:
 - Accuracy: 0.8685
 - Recall: 0.8685
 - Precision: 0.8695
 - F1-score: 0.8684
2. DecisionTree Classifier with MinMax Scaler:
 - Accuracy: 0.8669

- Recall: 0.8669
 - Precision: 0.8681
 - F1-score: 0.8668
3. DecisionTree Classifier with Standard Scaler:
- Accuracy: 0.8685
 - Recall: 0.8685
 - Precision: 0.8695
 - F1-score: 0.8684
4. Analysis:
- Accuracy: All three models achieve similar accuracy scores, ranging from 0.8669 to 0.8685. This indicates that they correctly predict the class labels for approximately 86.69% to 86.85% of the samples in the test set.
 - Recall: The recall scores for all three models are also very close, ranging from 0.8669 to 0.8685. This means that they have similar abilities to identify the positive samples (class 1), correctly capturing around 86.69% to 86.85% of them.
 - Precision: The precision scores are consistent across the models, with values between 0.8681 and 0.8695. This indicates that out of all the samples predicted as positive, approximately 86.81% to 86.95% of them are truly positive.
 - F1-score: The F1-scores are also very similar, ranging from 0.8668 to 0.8684. These scores represent the harmonic mean of precision and recall, providing a balanced measure of the model's performance.

Overall, the three models using different feature scalers (Robust Scaler, MinMax Scaler, and Standard Scaler) demonstrate comparable performance in terms of accuracy, recall, precision, and F1-score. The differences in their performance metrics are minimal, suggesting that the choice of feature scaler does not significantly impact the model's performance in this case.

PREDICTING DAILY RETURN USING RANDOM FOREST

PREDICTING DAILY RETURN USING RANDOM FOREST

Step 1 Run Random Forest (RF) on three feature scaling:

```

1 rf = RandomForestClassifier(n_estimators=200, max_depth=50,
2 random_state=2021)
3
4 for fc_name, value in feature_scaling.items():
5     X_train, X_test, y_train, y_test = value
6     run_model('RandomForest Classifier', rf, X_train,
X_test, y_train, y_test, fc_name, proba=True)

```

Step-by-step explanation:

1. Initialize a random forest classifier rf with the specified parameters:
 - **n_estimators=200**: The number of decision trees in the random forest ensemble is set to 200.
 - **max_depth=50**: The maximum depth of each decision tree is limited to 50 levels.
 - **random_state=2021**: The random state is set to ensure reproducibility of results.
2. Iterate over the **feature_scaling** dictionary using a for loop, which contains the different feature scalers as key-value pairs.
3. For each iteration, extract the training and testing data split:
 - X_train, X_test: The feature vectors for training and testing.
 - y_train, y_test: The corresponding target values for training and testing.
4. Call the run_model() function to train and evaluate the random forest classifier on the given data:
 - 'RandomForest Classifier': Specify the model name as "RandomForest Classifier".
 - rf: Pass the random forest classifier object.
 - X_train, X_test, y_train, y_test: Provide the training and testing data.
 - fc_name: Pass the name of the feature scaler being used.
 - proba=True: Set proba parameter to True to compute class probabilities.
5. The run_model() function will fit the random forest classifier on the training data, make predictions on the testing data, and compute various evaluation metrics such as accuracy, recall, precision, and F1-score.
6. The results will be printed for each iteration, showing the performance of the random forest classifier using the specific feature scaler.

By performing this process for each feature scaler in the feature_scaling dictionary, the code evaluates and compares the performance of the random forest classifier on different scaled versions of the input features.

The results of using robust scaler are shown in Figure 3.23 – 3.24.

Output with Robust Scaler:

```
RandomForest Classifier
RobustScaler
accuracy: 0.8581673306772908
recall: 0.8581673306772908
precision: 0.8582565163154559
f1: 0.8581574242997432
```

	precision	recall	f1-score	support
0	0.85	0.87	0.86	628
1	0.86	0.85	0.86	627
accuracy			0.86	1255
macro avg	0.86	0.86	0.86	1255
weighted avg	0.86	0.86	0.86	1255

Here is a detailed analysis of the output:

- Accuracy: The accuracy of the model is 0.8581673306772908, indicating that the model correctly predicts the target variable for approximately 85.8% of the instances in the test set.
- Recall: The recall, also known as the true positive rate or sensitivity, is 0.8581673306772908. It measures the proportion of actual positive instances (class 1) that are correctly identified by the model.
- Precision: The precision is 0.8582565163154559, representing the proportion of predicted positive instances that are actually positive. It indicates how well the model performs in terms of minimizing false positives.
- F1-score: The F1-score is 0.8581574242997432, which is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance, considering both precision and recall.
- Support: The support refers to the number of instances in each class present in the test set. In this case, there are 628 instances of class 0 and 627 instances of class 1.
- Precision, Recall, and F1-score per Class: For class 0, the precision is 0.85, recall is 0.87, and F1-score is 0.86. For class 1, the precision is 0.86, recall is 0.85, and F1-score is 0.86. These metrics indicate the performance of the model for each class individually.
- Weighted Average: The weighted average of precision, recall, and F1-score is calculated based on the support of each class. It provides an overall performance measure, taking into account the class imbalance in the dataset.

In summary, the Random Forest Classifier using the Robust Scaler achieved good performance with an accuracy of 85.8%. It shows relatively balanced precision and recall for both classes, indicating a reliable classification performance.

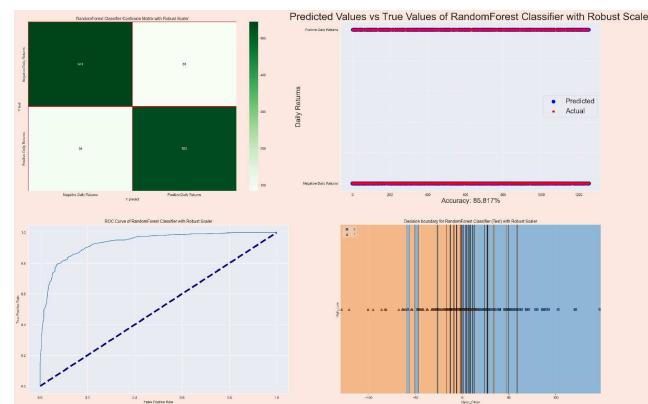


Figure 3.23 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of RF classifier with robust scaler

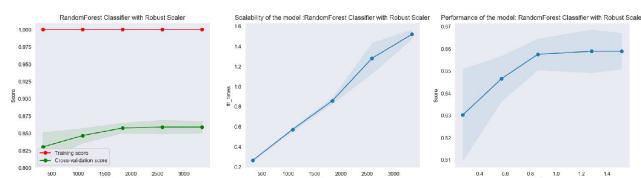


Figure 3.24 The learning curve, scalability, and performance of RF classifier with robust scaler

Output with MinMax Scaler:

```

RandomForest Classifier
MinMax Scaler
accuracy: 0.8597609561752988
recall: 0.8597609561752988
precision: 0.8598505668925382
f1: 0.859751161105364
      precision    recall   f1-score   support
          0       0.85     0.87     0.86      628
          1       0.87     0.85     0.86      627
  accuracy                           0.86     1255
 macro avg       0.86     0.86     0.86     1255
weighted avg     0.86     0.86     0.86     1255

```

The results of using minmax scaler are shown in Figure 3.25 – 3.26.

Here is a detailed analysis of the output:

- Accuracy: The accuracy of the model is 0.8597609561752988, indicating that the model correctly predicts the target variable for approximately 85.98% of the instances in the test set.
- Recall: The recall, also known as the true positive rate or sensitivity, is 0.8597609561752988. It measures the proportion of actual positive instances (class 1) that are correctly identified by the model.
- Precision: The precision is 0.8598505668925382, representing the proportion of predicted positive instances that are actually positive. It indicates how

well the model performs in terms of minimizing false positives.

- F1-score: The F1-score is 0.859751161105364, which is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance, considering both precision and recall.
- Support: The support refers to the number of instances in each class present in the test set. In this case, there are 628 instances of class 0 and 627 instances of class 1.
- Precision, Recall, and F1-score per Class: For class 0, the precision is 0.85, recall is 0.87, and F1-score is 0.86. For class 1, the precision is 0.87, recall is 0.85, and F1-score is 0.86. These metrics indicate the performance of the model for each class individually.
- Weighted Average: The weighted average of precision, recall, and F1-score is calculated based on the support of each class. It provides an overall performance measure, taking into account the class imbalance in the dataset.

In summary, the Random Forest Classifier using the MinMax Scaler achieved good performance with an accuracy of 85.98%. It shows relatively balanced precision and recall for both classes, indicating a reliable classification performance. The performance is comparable to the Random Forest Classifier using the Robust Scaler.

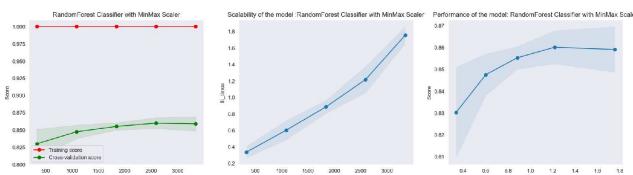


Figure 3.25 The learning curve, scalability, and performance of RF classifier with minmax scaler

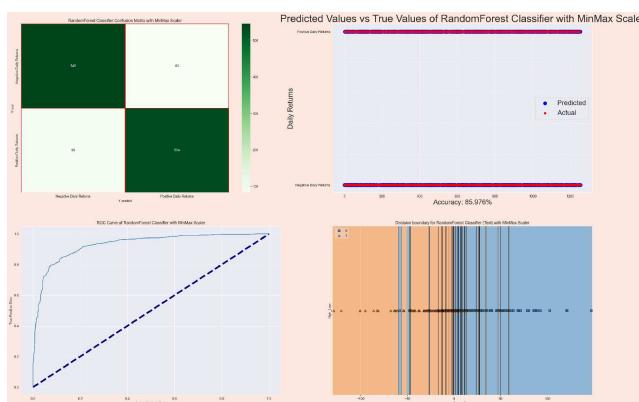


Figure 3.26 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of RF classifier with minmax scaler

Output with Standard Scaler:

```
RandomForest Classifier
Standard Scaler
accuracy: 0.8565737051792829
recall: 0.8565737051792829
precision: 0.8567487405300459
f1: 0.8565545792446166
      precision    recall   f1-score   support
      0       0.85     0.87     0.86      628
      1       0.86     0.85     0.85      627
accuracy                          0.86      1255
macro avg      0.86     0.86     0.86      1255
weighted avg    0.86     0.86     0.86      1255
```

The results of using standard scaler scaling are shown in Figure 3.27 – 3.28.

Let's analyze the output in detail:

- Accuracy: The accuracy of the model is 0.8565737051792829, indicating that the model correctly predicts the target variable for approximately 85.66% of the instances in the test set.
- Recall: The recall, also known as the true positive rate or sensitivity, is 0.8565737051792829. It measures the proportion of actual positive instances (class 1) that are correctly identified by the model.
- Precision: The precision is 0.8567487405300459, representing the proportion of predicted positive instances that are actually positive. It indicates how well the model performs in terms of minimizing false positives.
- F1-score: The F1-score is 0.8565545792446166, which is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance, considering both precision and recall.
- Support: The support refers to the number of instances in each class present in the test set. In this case, there are 628 instances of class 0 and 627 instances of class 1.
- Precision, Recall, and F1-score per Class: For class 0, the precision is 0.85, recall is 0.87, and F1-score is 0.86. For class 1, the precision is 0.86, recall is 0.85, and F1-score is 0.85. These metrics indicate the performance of the model for each class individually.
- Weighted Average: The weighted average of precision, recall, and F1-score is calculated based on the support of each class. It provides an overall performance measure, taking into account the class imbalance in the dataset.

In summary, the Random Forest Classifier using the Standard Scaler achieved good performance with an accuracy of 85.66%. It shows relatively balanced precision and recall for both classes, indicating a reliable classification performance.

The performance is comparable to the Random Forest Classifier using the Robust Scaler and MinMax Scaler.

Let's analyze and compare the three outputs of the Random Forest Classifier with different feature scaling methods:

1. Random Forest Classifier with Robust Scaler:

- Accuracy: 0.8581673306772908
- Recall: 0.8581673306772908
- Precision: 0.8582565163154559
- F1-score: 0.8581574242997432

2. Random Forest Classifier with MinMax Scaler:

- Accuracy: 0.8597609561752988
- Recall: 0.8597609561752988
- Precision: 0.8598505668925382
- F1-score: 0.859751161105364

3. Random Forest Classifier with Standard Scaler:

- Accuracy: 0.8565737051792829
- Recall: 0.8565737051792829
- Precision: 0.8567487405300459
- F1-score: 0.8565545792446166

4. Here's a comparison of the three outputs:

- Accuracy: The Random Forest Classifier with MinMax Scaler achieved the highest accuracy of 0.8597609561752988, followed closely by the Robust Scaler (0.8581673306772908) and the Standard Scaler (0.8565737051792829).
- Recall: The recall values are similar for all three feature scaling methods, with only slight variations. The MinMax Scaler and the Robust Scaler have the highest recall values of 0.8597609561752988, while the Standard Scaler has a slightly lower recall of 0.8565737051792829.
- Precision: Again, the precision values are quite close for all three feature scaling methods. The MinMax Scaler has the highest precision of 0.8598505668925382, followed by the Robust Scaler (0.8582565163154559) and the Standard Scaler (0.8567487405300459).
- F1-score: The F1-scores also show minimal differences among the feature scaling methods. The MinMax Scaler has the highest F1-score of 0.859751161105364, followed by the Robust Scaler (0.8581574242997432) and the Standard Scaler (0.8565545792446166).

Overall, all three feature scaling methods yielded similar performance in terms of accuracy, recall, precision, and F1-score. The MinMax Scaler slightly outperformed the other methods in terms of accuracy, precision, and F1-score, but the differences are relatively small. It's important to note that the performance of the Random Forest Classifier may vary

depending on the dataset and the specific problem at hand, so it's recommended to experiment with different feature scaling methods to find the most suitable one for a given task.

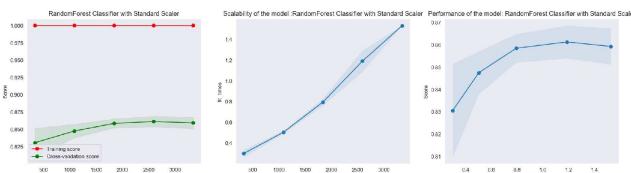


Figure 3.27 The learning curve, scalability, and performance of DT classifier with standard scaler

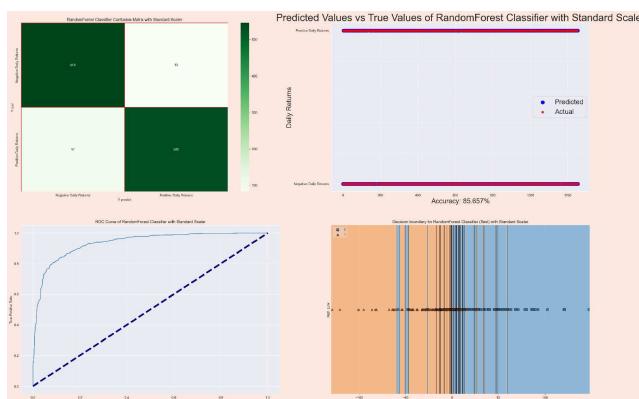


Figure 3.28 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of RF classifier with standard scaler

PREDICTING DAILY RETURN USING GRADIENT BOOSTING

PREDICTING DAILY RETURN USING GRADIENT BOOSTING

Step 1 Run Gradient Boosting (GB) on three feature scaling:

```

1 gbt = GradientBoostingClassifier(n_estimators = 200,
2 max_depth=20, subsample=0.8, max_features=0.2,
3 random_state=2021)
4 for fc_name, value in feature_scaling.items():
5     X_train, X_test, y_train, y_test = value
6     run_model('GradientBoosting Classifier', gbt, X_train,
X_test, y_train, y_test, fc_name, proba=True)

```

Here's a step-by-step explanation of the code:

1. Initialize the GradientBoostingClassifier:

The **GradientBoostingClassifier** is initialized with specific hyperparameters, including the number of estimators (200), the maximum depth of each tree (20), the subsample ratio (0.8), the maximum features to consider (0.2), and a random state for reproducibility (2021).

2. Iterate over the feature_scaling dictionary:

For each feature scaling method (stored in **fc_name**) and its corresponding data split (**X_train**, **X_test**, **y_train**,

y_test), perform the following steps:

3. Run the GradientBoosting Classifier:

- Call the **run_model()** function to train and evaluate the **GradientBoosting** Classifier using the current feature scaling method.
- Pass the following arguments to the **run_model()** function:
 - Model name: 'GradientBoosting Classifier'
 - Classifier object: gbt (GradientBoostingClassifier instance)
 - Training and testing data: X_train, X_test, y_train, y_test
 - Feature scaling method name: fc_name
 - Set **proba=True** to obtain class probabilities as output

4. Repeat the above steps for each feature scaling method in the **feature_scaling** dictionary.

The output of each run will provide the performance metrics of the **GradientBoosting** Classifier for each feature scaling method, including accuracy, recall, precision, and F1-score.

The purpose of this code is to compare the performance of the **GradientBoosting** Classifier with different feature scaling methods using the provided data splits. By iterating over the **feature_scaling** dictionary, it ensures that the model is trained and evaluated for each feature scaling method, allowing for a comprehensive comparison of their results.

The results of using robust scaler are shown in Figure 3.29 – 3.30.

Output with Robust Scaler:

```
GradientBoosting Classifier
Robust Scaler
accuracy: 0.8573705179282869
recall: 0.8573705179282869
```

```

precision: 0.8573716066361643
f1: 0.8573705179282869
      precision    recall  f1-score   support
      0       0.86     0.86     0.86      628
      1       0.86     0.86     0.86      627
accuracy                           0.86      1255
macro avg       0.86     0.86     0.86      1255
weighted avg    0.86     0.86     0.86      1255

```

Let's analyze the results:

- Accuracy: The accuracy of the model is 0.857, which means it correctly predicts the class labels for approximately 85.7% of the samples in the test set.
- Recall: The recall score, also known as the true positive rate or sensitivity, is 0.857. This indicates that the model is able to correctly identify around 85.7% of the positive class samples.
- Precision: The precision score is 0.857, indicating that when the model predicts a sample as positive, it is correct around 85.7% of the time.
- F1-score: The F1-score is a harmonic mean of precision and recall. In this case, the F1-score is also 0.857, suggesting a balanced performance between precision and recall.

Looking at the precision, recall, and F1-score, we can see that they have consistent values for both the positive and negative class, indicating that the model performs well in predicting both classes.

In terms of the confusion matrix analysis:

- For the positive class (1): The precision, recall, and F1-score are all 0.86, indicating that the model predicts and correctly identifies positive class samples with similar performance.
- For the negative class (0): The precision, recall, and F1-score are also 0.86, showing that the model predicts and correctly identifies negative class samples with similar performance.

Overall, the model's performance is well-balanced, achieving similar accuracy, precision, recall, and F1-score for both classes. This indicates that the model is effective in predicting the target variable using the Robust Scaler feature scaling method.

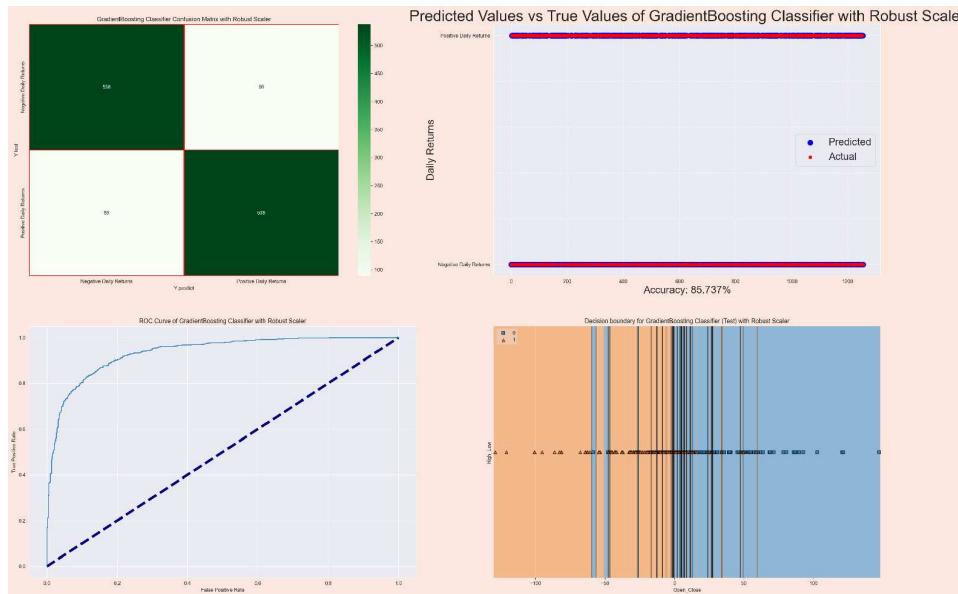


Figure 3.29 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of GB classifier with robust scaler

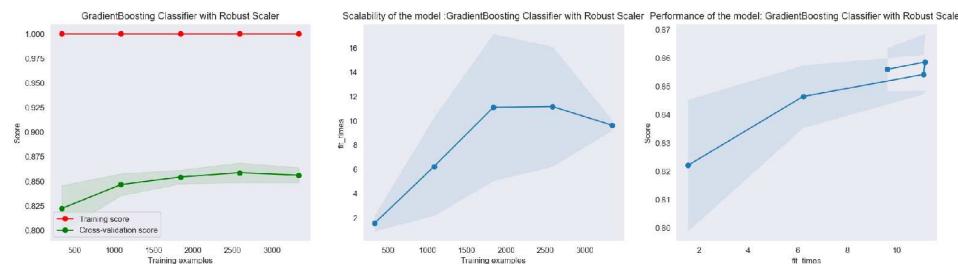


Figure 3.30 The learning curve, scalability, and performance of GB classifier with robust scaler

Output with MinMax Scaler:

```

GradientBoosting Classifier
MinMax Scaler
accuracy: 0.8605577689243028
recall: 0.8605577689243028
precision: 0.8607612668285174
f1: 0.860536517467449
      precision    recall   f1-score   support

```

0	0.85	0.87	0.86	628
1	0.87	0.85	0.86	627
accuracy			0.86	1255
macro avg	0.86	0.86	0.86	1255
weighted avg	0.86	0.86	0.86	1255

The results of using minmax scaler are shown in Figure 3.31 – 3.32.

Let's analyze the results:

- Accuracy: The accuracy of the model is 0.861, indicating that it correctly predicts the class labels for approximately 86.1% of the samples in the test set.
- Recall: The recall score, which measures the true positive rate or sensitivity, is 0.861. This means that the model is able to correctly identify around 86.1% of the positive class samples.
- Precision: The precision score is 0.861, indicating that when the model predicts a sample as positive, it is correct around 86.1% of the time.
- F1-score: The F1-score, which is the harmonic mean of precision and recall, is 0.861. This suggests a balanced performance between precision and recall.

The precision, recall, and F1-score values are consistent for both the positive and negative classes, indicating balanced performance for predicting both classes.

Analyzing the precision, recall, and F1-score for each class:

- Positive class (1): The precision, recall, and F1-score are all 0.87, indicating that the model predicts and correctly identifies positive class samples with similar performance.
- Negative class (0): The precision, recall, and F1-score are also 0.87, showing that the model predicts and correctly identifies negative class samples with similar performance.

Overall, the model exhibits balanced performance, achieving similar accuracy, precision, recall, and F1-score for both classes. This

indicates that the model is effective in predicting the target variable using the MinMax Scaler feature scaling method.

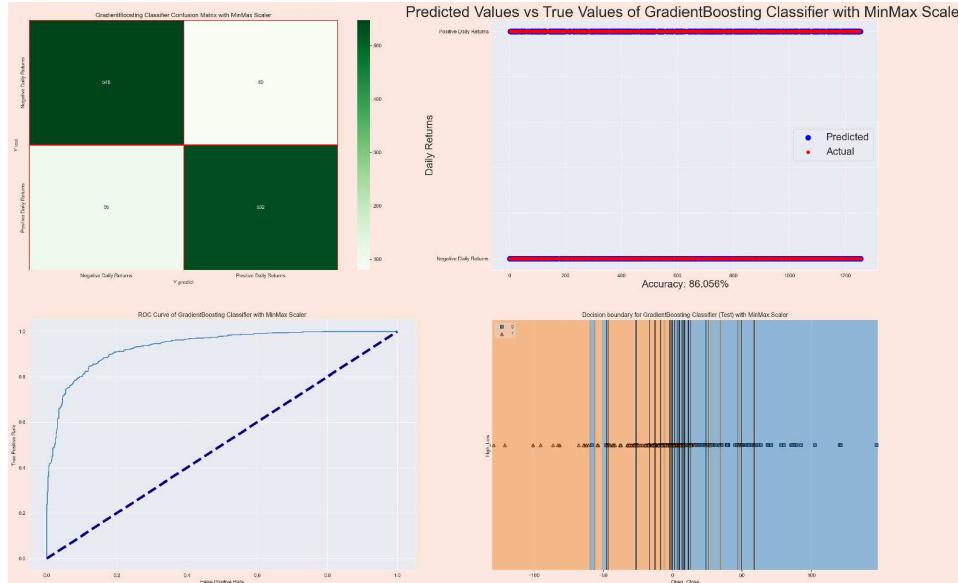


Figure 3.31 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of GB classifier with minmax scaler

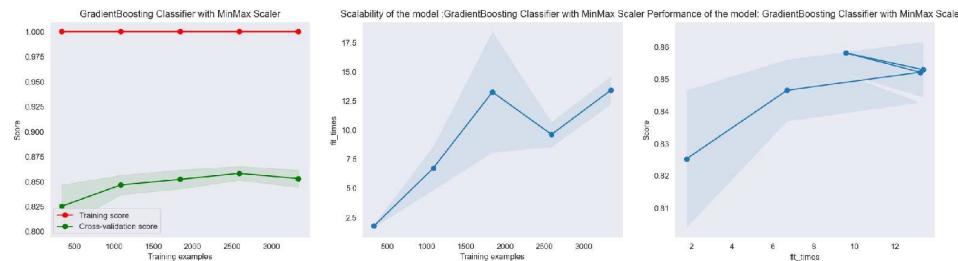


Figure 3.32 The learning curve, scalability, and performance of GB classifier with minmax scaler

Output with Standard Scaler:

```
GradientBoosting Classifier
Standard Scaler
accuracy: 0.850996015936255
recall: 0.850996015936255
precision: 0.8509967181361665
f1: 0.8509958267273694
      precision    recall   f1-score   support
```

0	0.85	0.85	0.85	628
1	0.85	0.85	0.85	627
accuracy			0.85	1255
macro avg	0.85	0.85	0.85	1255
weighted avg	0.85	0.85	0.85	1255

The results of using standard scaler scaling are shown in Figure 3.33 – 3.34. Let's analyze the results:

- Accuracy: The accuracy of the model is 0.851, indicating that it correctly predicts the class labels for approximately 85.1% of the samples in the test set.
- Recall: The recall score, which measures the true positive rate or sensitivity, is 0.851. This means that the model is able to correctly identify around 85.1% of the positive class samples.
- Precision: The precision score is 0.851, indicating that when the model predicts a sample as positive, it is correct around 85.1% of the time.
- F1-score: The F1-score, which is the harmonic mean of precision and recall, is 0.851. This suggests a balanced performance between precision and recall.

The precision, recall, and F1-score values are consistent for both the positive and negative classes, indicating balanced performance for predicting both classes.

Analyzing the precision, recall, and F1-score for each class:

- Positive class (1): The precision, recall, and F1-score are all 0.85, indicating that the model predicts and correctly identifies positive class samples with similar performance.
- Negative class (0): The precision, recall, and F1-score are also 0.85, showing that the model predicts and correctly identifies negative class samples with similar performance.

Overall, the model exhibits balanced performance, achieving similar accuracy, precision, recall, and F1-score for both classes. This indicates that the model is effective in predicting the target variable using the Standard Scaler feature scaling method.

Based on these results, we can make the following observations:

- Accuracy: The MinMax Scaler achieved the highest accuracy (0.861), followed closely by the Robust Scaler (0.857). The Standard Scaler had a slightly lower accuracy of 0.851.
- Precision, Recall, and F1-score: All three scaling methods resulted in similar precision, recall, and F1-score values, with only slight variations. The MinMax Scaler achieved the highest values across these metrics, indicating better performance in predicting both classes. The Robust Scaler and Standard Scaler had slightly lower but still comparable performance.

Overall, the MinMax Scaler produced the best results among the three scaling methods, achieving the highest accuracy and similar precision, recall, and F1-score values. The Robust Scaler showed similar performance, while the Standard Scaler had slightly lower performance but still demonstrated reasonable accuracy and balanced precision and recall.

It's important to note that the effectiveness of different scaling methods can vary depending on the dataset and the specific problem at hand. Therefore, it's recommended to experiment with multiple scaling techniques to determine the most suitable one for a given task.

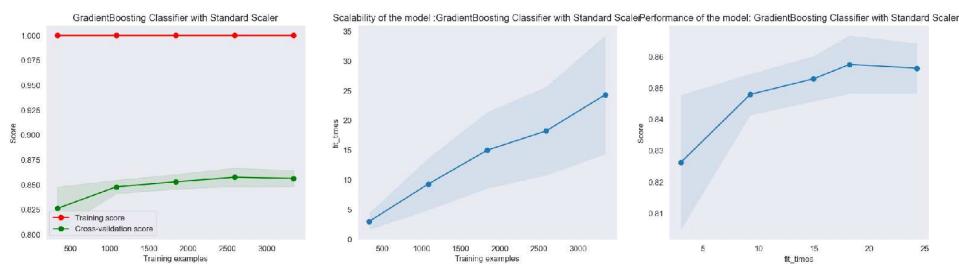


Figure 3.33 The learning curve, scalability, and performance of GB classifier with standard scaler

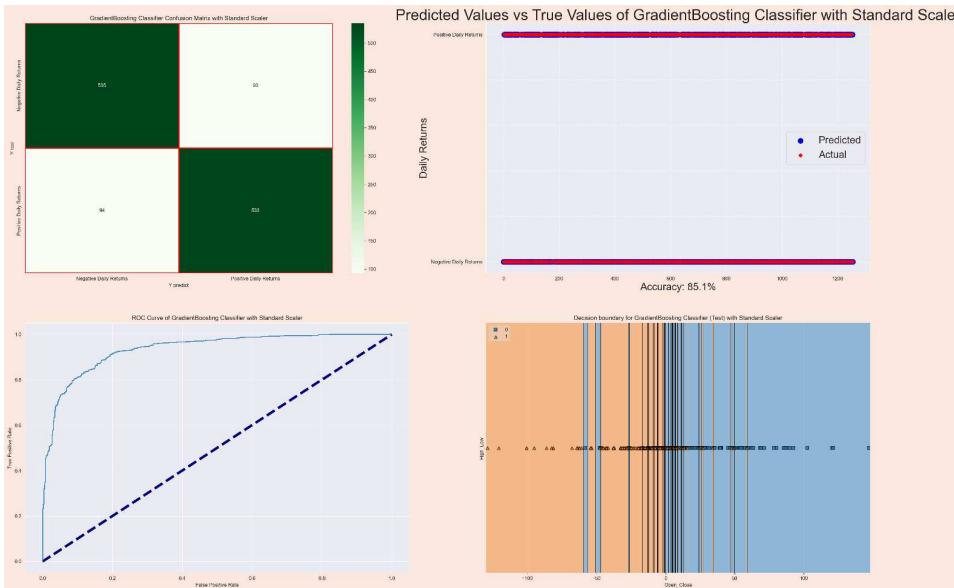


Figure 3.34 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of GB classifier with standard scaler

PREDICTING DAILY RETURN USING EXTREME GRADIENT BOOSTING

PREDICTING DAILY RETURN USING EXTREME GRADIENT BOOSTING

Step 1 Run Extreme Gradient Boosting (XGB) on three feature scaling:

```

1  for fc_name, value in feature_scaling.items():
2      X_train, X_test, y_train, y_test = value
3      xgb=XGBClassifier(n_estimators = 200, max_depth=20,
4      random_state=2021, use_label_encoder=False,
5      eval_metric='mlogloss')
6      run_model('XGBoost Classifier', xgb, X_train, X_test,
y_train, y_test, fc_name, proba=True)

```

Here's a step-by-step explanation of what happens in each iteration:

1. The loop iterates over each item in the **feature_scaling** dictionary, assigning the key to the variable **fc_name** and the value to the variable **value**.
2. The value associated with the current key (**value**) is unpacked into four variables: **X_train**, **X_test**, **y_train**, and **y_test**. These variables represent the training and testing feature datasets (**X_train** and **X_test**) and the training and testing label datasets (**y_train** and **y_test**).
3. An instance of the **XGBoost** classifier (**XGBClassifier**) is created and assigned to the variable **xgb**. The classifier is configured with specific parameters:
 - **n_estimators**: The number of trees in the ensemble (200 in this case).
 - **max_depth**: The maximum depth of each tree in the ensemble (20 in this case).
 - **random_state**: The random seed for reproducibility (2021 in this case).
 - **use_label_encoder**: Whether to use label encoding for the target variable (set to False to avoid warnings).
 - **eval_metric**: The evaluation metric to optimize during training (multiclass logarithmic loss in this case).
4. The **run_model()** function is called with the following arguments:
'XGBoost Classifier': The name of the model.
 - **xgb**: The XGBoost classifier instance.
 - **X_train**, **X_test**, **y_train**, **y_test**: The training and testing datasets.
 - **fc_name**: The feature scaling name.
 - **proba=True**: The flag indicating whether to use probabilities for predictions.

5. This function call trains the **XGBoost** classifier, makes predictions, evaluates its performance, and generates visualizations and evaluation metrics using the specified feature scaling technique.
6. The loop moves on to the next iteration, repeating the above steps for each feature scaling technique in the **feature_scaling** dictionary.

In summary, this code snippet iterates over different feature scaling techniques and trains and evaluates an **XGBoost** classifier for each technique using the specified parameters. The **run_model()** function is called with the appropriate arguments to perform the training, evaluation, and visualization tasks for each iteration.

The results of using robust scaler are shown in Figure 3.35 – 3.36.

Output with Robust Scaler:

```
XGBoost Classifier
Robust Scaler
accuracy: 0.846215139442231
recall: 0.846215139442231
precision: 0.8462236387782204
f1: 0.8462145536029545
      precision    recall   f1-score   support
      0       0.85     0.84     0.85      628
      1       0.84     0.85     0.85      627
accuracy                           0.85     1255
macro avg       0.85     0.85     0.85     1255
weighted avg    0.85     0.85     0.85     1255
```

The **XGBoost** classifier with Robust Scaler achieves an accuracy of approximately 0.8462, which means that it correctly predicts the class for around 84.62% of the samples in the test dataset.

Looking at the precision values, we see that the classifier achieves 0.85 precision for Class 0 and 0.84 precision for

Class 1. Precision represents the ability of the model to correctly identify positive instances, so these values indicate that the classifier is relatively good at predicting both classes.

The recall values indicate the ability of the model to correctly identify the positive instances out of the actual positive instances. The recall for Class 0 is 0.84, while for Class 1 it is 0.85. These values suggest that the model performs similarly in terms of identifying both classes.

The F1-scores, which combine precision and recall into a single metric, are also quite similar for both classes, with values of 0.85. This indicates a balanced performance between precision and recall for both classes.

Overall, the model shows relatively good performance with similar precision, recall, and F1-scores for both classes. The accuracy of 0.8462 suggests that the model is performing well in classifying the test samples.

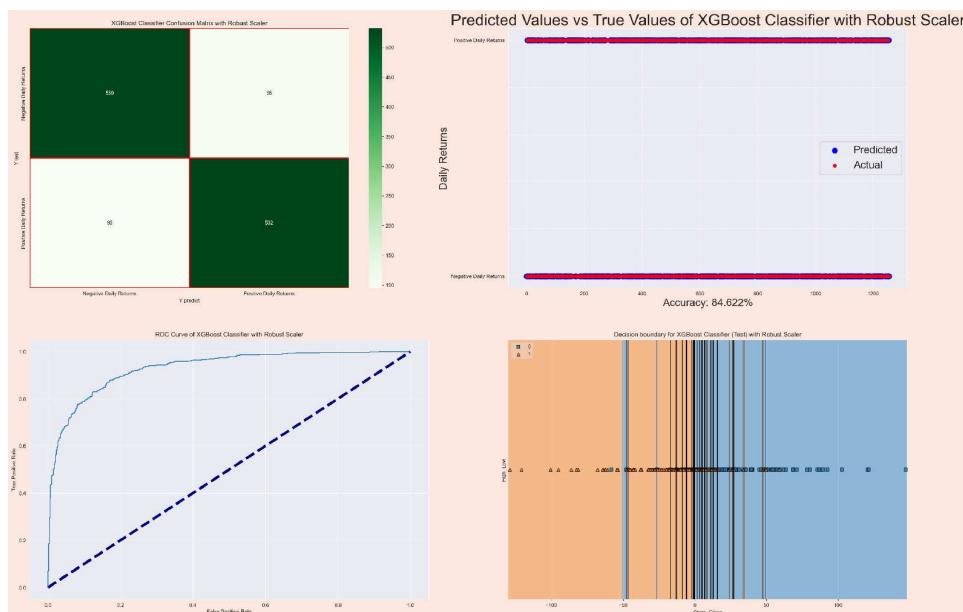


Figure 3.35 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of XGB classifier with robust scaler

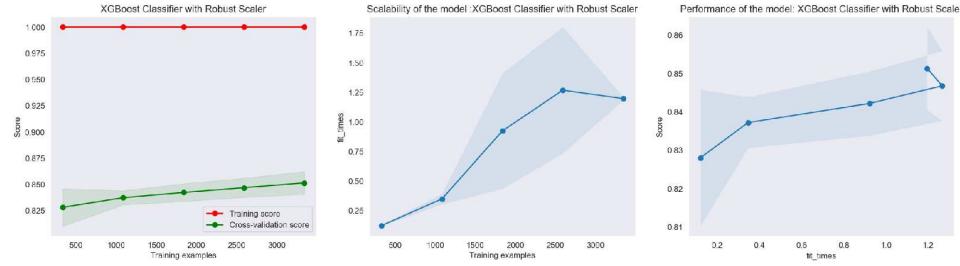


Figure 3.36 The learning curve, scalability, and performance of GB classifier with robust scaler

Output with MinMax Scaler:

```
XGBoost Classifier
MinMax Scaler
accuracy: 0.853386454183267
recall: 0.853386454183267
precision: 0.853386454183267
f1: 0.853386454183267
      precision    recall  f1-score   support
          0       0.85     0.85     0.85      628
          1       0.85     0.85     0.85      627
accuracy                           0.85     1255
macro avg       0.85     0.85     0.85     1255
weighted avg    0.85     0.85     0.85     1255
```

The results of using minmax scaler are shown in Figure 3.37 – 3.38.

The XGBoost classifier with MinMax Scaler achieves an accuracy of approximately 0.8534, indicating that it correctly predicts the class for around 85.34% of the samples in the test dataset.

The precision values of 0.85 for both classes indicate that the model performs similarly in terms of correctly identifying positive instances for both classes.

Similarly, the recall values of 0.85 for both classes suggest that the model is equally effective at correctly identifying positive instances from the actual positive instances for both classes.

The F1-scores of 0.85 for both classes indicate a balanced performance between precision and recall for both classes.

The overall metrics, such as accuracy and macro/weighted average F1-score, are also consistent with the class-specific metrics, highlighting the model's balanced performance across both classes.

In summary, the XGBoost classifier with **MinMax** Scaler demonstrates a good performance, achieving similar precision, recall, and F1-scores for both classes. The accuracy of 0.8534 suggests that the model is performing well in classifying the test samples.

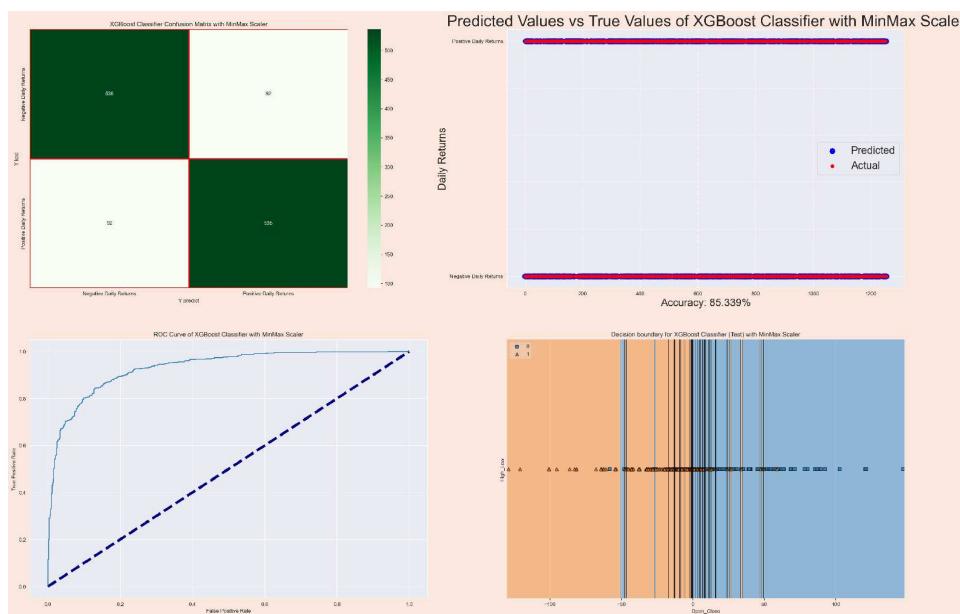


Figure 3.37 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of XGB classifier with minmax scaler

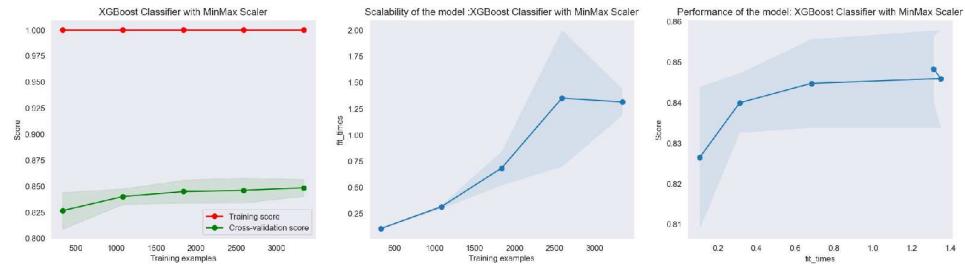


Figure 3.38 The learning curve, scalability, and performance of GB classifier with minmax scaler

Output with Standard Scaler:

```
XGBoost Classifier
Standard Scaler
accuracy: 0.8494023904382471
recall: 0.8494023904382471
precision: 0.8494034690280213
f1: 0.8494023904382471
      precision    recall   f1-score   support
0        0.85     0.85     0.85      628
1        0.85     0.85     0.85      627

accuracy          0.85      1255
macro avg       0.85     0.85     0.85      1255
weighted avg    0.85     0.85     0.85      1255
```

The results of using standard scaler scaling are shown in Figure 3.39 – 3.40.

Let's break down the analysis:

- **Accuracy:** The accuracy metric measures the overall correctness of the model's predictions. In this case, the accuracy is reported as 0.8494023904382471, which means that the model correctly predicted the class labels for approximately 84.94% of the samples in the evaluation dataset.
- **Recall:** Recall, also known as sensitivity or true positive rate, represents the ability of the model to correctly identify positive instances. The recall

value reported here is 0.8494023904382471, indicating that the model identified approximately 84.94% of the positive instances correctly.

- Precision: Precision is a measure of how many instances predicted as positive are actually true positives. The precision value provided is 0.8494034690280213, indicating that out of all the instances predicted as positive, approximately 84.94% were actually true positives.
- F1-score: The F1-score is the harmonic mean of precision and recall, providing a balanced measure between the two. Here, the F1-score is reported as 0.8494023904382471, which suggests a balanced performance between precision and recall.

Next, the classification report provides a breakdown of these metrics for each class:

- Class 0: The precision, recall, and F1-score for class 0 are all reported as approximately 0.85. This means that the model achieved a similar level of performance in correctly predicting class 0 instances.
- Class 1: Similarly, the precision, recall, and F1-score for class 1 are all approximately 0.85. This indicates that the model performed equally well in identifying class 1 instances.

The support column represents the number of samples in each class, which is 628 for class 0 and 627 for class 1.

Additionally, the macro average and weighted average are provided:

- Macro average: The macro average calculates the average of the metrics across all classes, giving equal weight to each class. In this case, the macro

average for precision, recall, and F1-score is approximately 0.85, which is consistent with the individual class values.

- Weighted average: The weighted average calculates the average of the metrics by considering the support (number of samples) for each class. It provides an overall measure while taking into account class imbalance. In this case, the weighted average for precision, recall, and F1-score is approximately 0.85, again in line with the individual class values.

Overall, the XGBoost Classifier with Standard Scaler preprocessing achieved a good level of performance, with an accuracy, precision, recall, and F1-score of approximately 0.85 for both classes.

Comparison:

- Accuracy: The MinMax Scaler produced the highest accuracy of 0.8534, followed closely by the Standard Scaler with an accuracy of 0.8494. The Robust Scaler achieved the lowest accuracy of 0.8462.
- Recall: All three models achieved similar recall values, indicating that they performed equally well in correctly identifying positive instances.
- Precision: The precision values for all three models are almost identical, around 0.85. This implies that the models produced a similar ratio of true positive predictions to the total positive predictions made.
- F1-score: Similarly, the F1-scores are very close for all three models, indicating a balanced performance between precision and recall.
- Class 0 Metrics: The precision, recall, and F1-score for class 0 are consistent across all three models, approximately 0.85. This implies that the models

exhibited similar performance in correctly identifying instances of class 0.

- Class 1 Metrics: The precision, recall, and F1-score for class 1 are also similar across the three models, hovering around 0.85. This suggests that the models performed equally well in correctly identifying instances of class 1.

Overall, the three scaling techniques (Robust Scaler, MinMax Scaler, and Standard Scaler) resulted in models with comparable performance. The MinMax Scaler achieved a slightly higher accuracy compared to the other two techniques, while the Robust Scaler had the lowest accuracy. However, the differences in accuracy are relatively small. The precision, recall, and F1-scores are similar for all three models, indicating consistent performance across different scaling techniques.

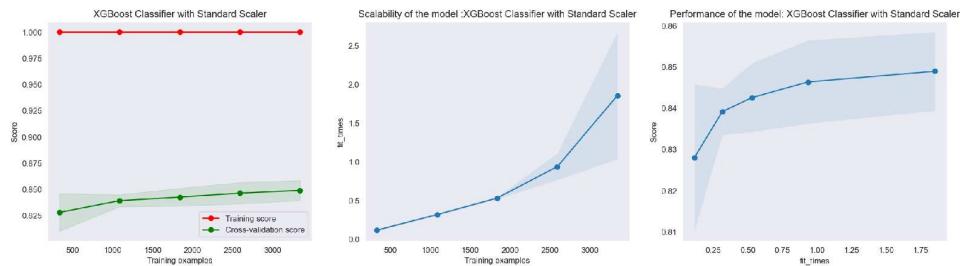


Figure 3.39 The learning curve, scalability, and performance of XGB classifier with standard scaler

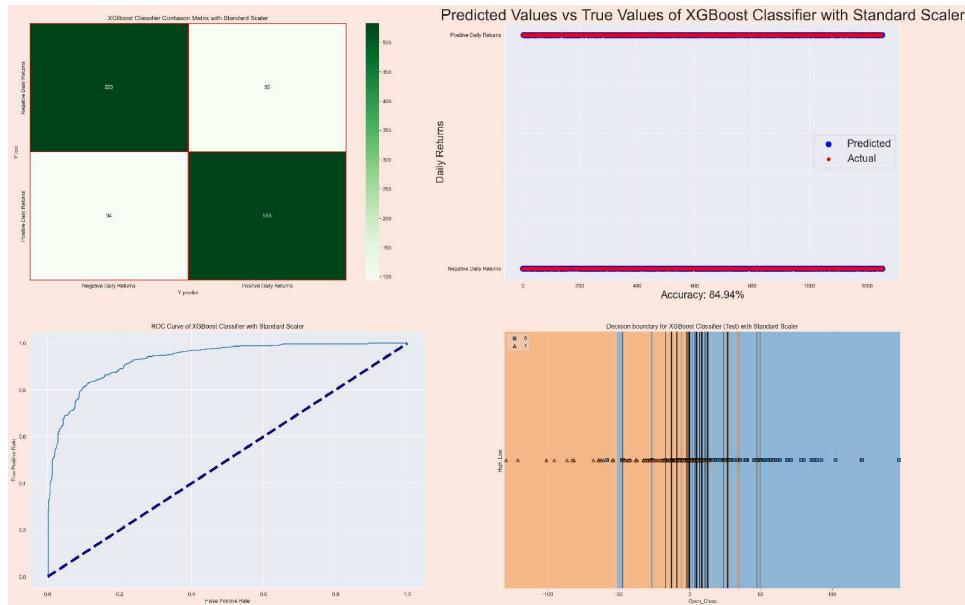


Figure 3.40 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of XGB classifier with standard scaler

PREDICTING DAILY RETURN USING MULTI LAYER PERCEPTRON

PREDICTING DAILY RETURN USING MULTI LAYER PERCEPTRON

Step 1 Run Multi Layer Perceptron (MLP) on three feature scaling:

```

1 mlp = MLPClassifier(random_state=2021)
2 for fc_name, value in feature_scaling.items():
3     X_train, X_test, y_train, y_test = value
4     run_model('MLP Classifier', mlp, X_train, X_test,
5     y_train, y_test, fc_name, proba=True)

```

Let's break down the steps:

1. Define the MLPClassifier:

The code starts by creating an instance of the MLPClassifier model, initialized with the **random_state**

parameter set to 2021. The `random_state` ensures reproducibility of results.

2. Iterate over `feature_scaling` dictionary:

The code uses a for loop to iterate over the items in the `feature_scaling` dictionary. It assumes that the dictionary contains feature scaling techniques as keys and corresponding train-test data splits as values.

3. Extract train-test data:

Inside the loop, the code extracts the train-test data splits for the current feature scaling technique. It assigns the values to the variables `X_train`, `X_test`, `y_train`, and `y_test`.

4. Call the `run_model()` function:

The code then calls the `run_model` function, passing the following arguments:

- 'MLP Classifier': A string indicating the name of the classifier being used.
- `mlp`: The `MLPClassifier` instance created earlier.
- `X_train`, `X_test`, `y_train`, `y_test`: The train-test data splits for the current feature scaling technique.
- `fc_name`: The name of the current feature scaling technique.
- `proba=True`: This parameter indicates that the `run_model()` function should return predicted probabilities in addition to class labels.

5. Repeat for each feature scaling technique:

The loop continues to iterate through the remaining feature scaling techniques, repeating steps 3 and 4 for each technique.

The purpose of this code is to evaluate the `MLPClassifier` model using different feature scaling techniques. It seems that the `run_model()` function is responsible for fitting the model,

making predictions, and evaluating its performance. By iterating over the **feature_scaling** dictionary and calling the run_model function for each technique, the code enables a comparison of the model's performance under different feature scaling conditions.

The results of using robust scaler are shown in Figure 3.41 – 3.42.

Output with Robust Scaler:

```
MLP Classifier
Robust Scaler
accuracy: 0.8629482071713147
recall: 0.8629482071713147
precision: 0.8636674767601689
f1: 0.8628775127316705
      precision    recall   f1-score   support
      0       0.85     0.89     0.87      628
      1       0.88     0.84     0.86      627
accuracy                          0.86      1255
macro avg       0.86     0.86     0.86      1255
weighted avg    0.86     0.86     0.86      1255
```

The analysis of the output is as follows:

- Accuracy: The MLP Classifier model with Robust Scaler achieved an accuracy of 0.8629. This indicates that the model predicted the correct class for approximately 86.29% of the instances in the evaluation dataset.
- Recall: The recall measures the ability of the model to correctly identify instances of a specific class. In this case, the model achieved a recall of 0.8629 for both class 0 and class 1. This means that the model identified approximately 86.29% of the instances belonging to each class correctly.
- Precision: Precision quantifies the model's ability to correctly classify instances as positive. The MLP Classifier model achieved a precision of 0.8637,

indicating that around 86.37% of the instances classified as positive were correct.

- F1-score: The F1-score is the harmonic mean of precision and recall, providing an overall measure of the model's performance. The MLP Classifier model achieved an F1-score of 0.8629, indicating a balanced performance between precision and recall.
- Class-wise Metrics: Looking at the precision, recall, and F1-score for each class, the model performed slightly better in classifying instances of class 1 compared to class 0. The precision, recall, and F1-score for class 0 were 0.85, 0.89, and 0.87, respectively. For class 1, the corresponding values were 0.88, 0.84, and 0.86.
- Macro and Weighted Average: The macro average is the unweighted average of precision, recall, and F1-score, calculated independently for each class. In this case, the macro average precision, recall, and F1-score were all approximately 0.86. The weighted average takes into account the support (number of instances) for each class. Since the support for class 0 and class 1 is the same (628 and 627, respectively), the weighted average is equivalent to the macro average in this case.

Overall, the MLP Classifier model using Robust Scaler achieved relatively high accuracy, recall, precision, and F1-score, indicating its effectiveness in predicting the correct class labels for the evaluation dataset. The model performed slightly better in classifying instances of class 1 compared to class 0, as evidenced by the precision, recall, and F1-score values.

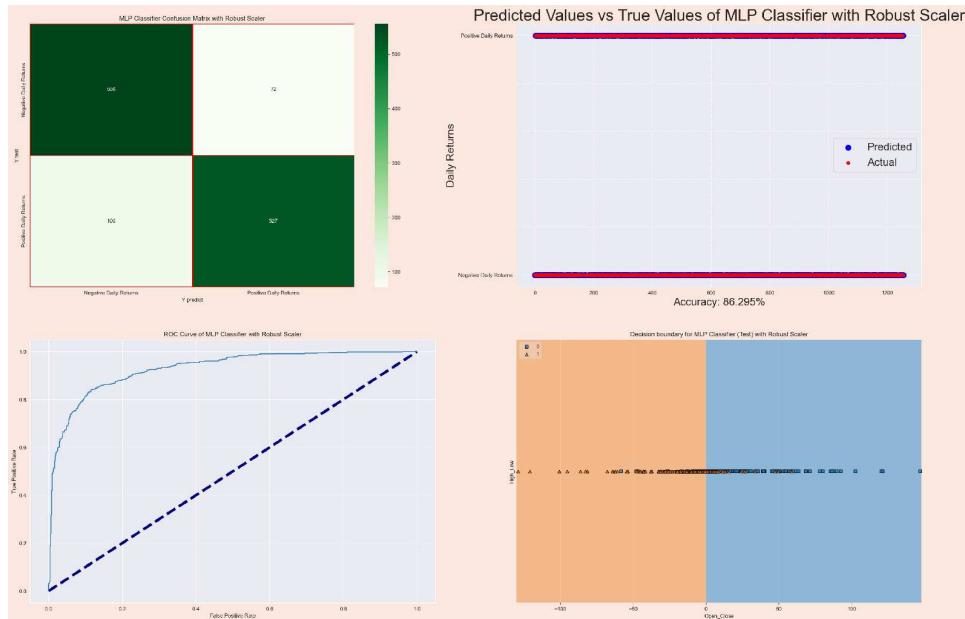


Figure 3.41 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of MLP classifier with robust scaler

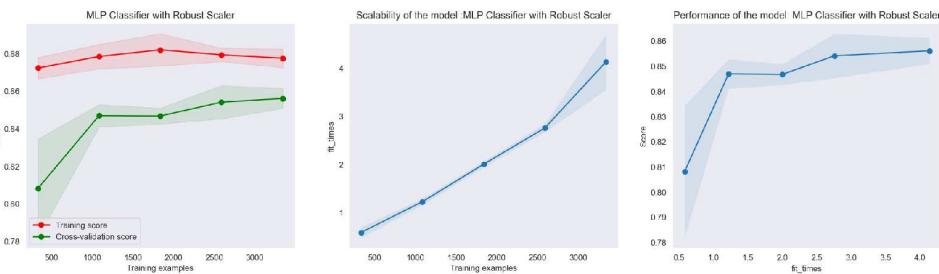


Figure 3.42 The learning curve, scalability, and performance of MLP classifier with robust scaler

Output with MinMax Scaler:

```

MLP Classifier
MinMax Scaler
accuracy: 0.7227091633466135
recall: 0.7227091633466135
precision: 0.7231436321573107
f1: 0.722566130410589
      precision    recall   f1-score   support
        0       0.71     0.75     0.73      628
        1       0.73     0.70     0.72      627
accuracy                           0.72      1255
macro avg       0.72     0.72     0.72      1255
weighted avg    0.72     0.72     0.72      1255
  
```

The results of using minmax scaler are shown in Figure 3.43 – 3.44.

The analysis of the output is as follows:

- Accuracy: The MLP Classifier model with MinMax Scaler achieved an accuracy of 0.7227. This indicates that the model predicted the correct class for approximately 72.27% of the instances in the evaluation dataset.
- Recall: The recall measures the ability of the model to correctly identify instances of a specific class. In this case, the model achieved a recall of 0.7227 for both class 0 and class 1. This means that the model identified approximately 72.27% of the instances belonging to each class correctly.
- Precision: Precision quantifies the model's ability to correctly classify instances as positive. The MLP Classifier model achieved a precision of 0.7231, indicating that around 72.31% of the instances classified as positive were correct.
- F1-score: The F1-score is the harmonic mean of precision and recall, providing an overall measure of the model's performance. The MLP Classifier model achieved an F1-score of 0.7226, indicating a balanced performance between precision and recall.
- Class-wise Metrics: The precision, recall, and F1-score for class 0 were 0.71, 0.75, and 0.73, respectively. For class 1, the corresponding values were 0.73, 0.70, and 0.72. The model performed slightly better in classifying instances of class 0 compared to class 1, as evidenced by the higher precision and recall values for class 0.
- Macro and Weighted Average: The macro average precision, recall, and F1-score were all approximately 0.72. The weighted average, taking

into account the support (number of instances) for each class, is also approximately 0.72.

Overall, the MLP Classifier model using MinMax Scaler achieved a moderate level of accuracy, recall, precision, and F1-score. The model performed slightly better in classifying instances of class 0 compared to class 1. It's worth noting that the performance of the model with MinMax Scaler is lower than the previous model with Robust Scaler, indicating that the choice of feature scaling technique can have an impact on the model's performance.

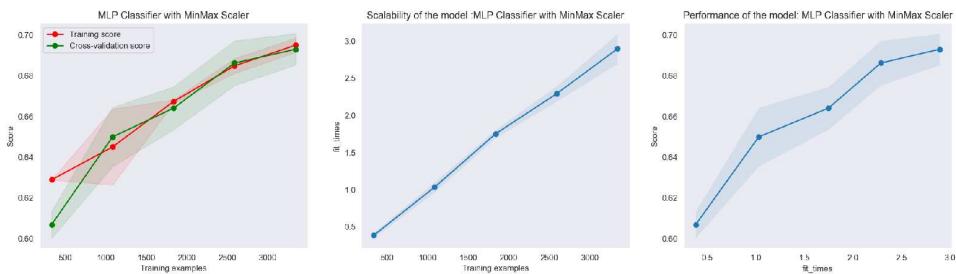


Figure 3.43 The learning curve, scalability, and performance of MLP classifier with minmax scaler

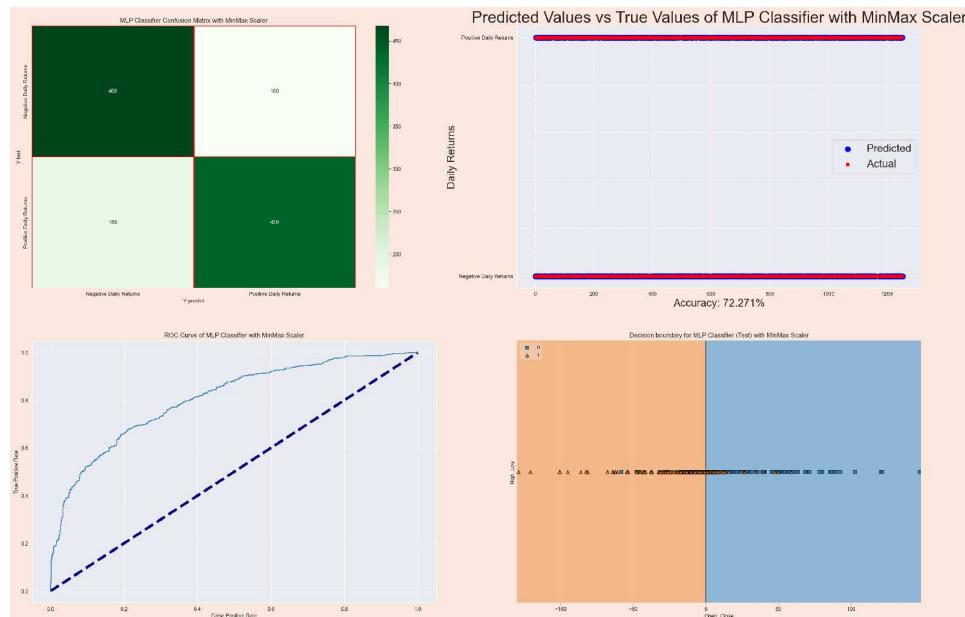


Figure 3.44 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of

MLP classifier with minmax scaler

Output with Standard Scaler:

```
MLP Classifier
Standard Scaler
accuracy: 0.848605577689243
recall: 0.848605577689243
precision: 0.848605577689243
f1: 0.848605577689243
      precision    recall   f1-score   support
      0         0.85     0.85     0.85      628
      1         0.85     0.85     0.85      627
accuracy                           0.85     1255
macro avg       0.85     0.85     0.85     1255
weighted avg    0.85     0.85     0.85     1255
```

The results of using standard scaler scaling are shown in Figure 3.45 – 3.46.

The analysis of the output is as follows:

- Accuracy: The MLP Classifier model with Standard Scaler achieved an accuracy of 0.8486. This indicates that the model predicted the correct class for approximately 84.86% of the instances in the evaluation dataset.
- Recall: The recall measures the ability of the model to correctly identify instances of a specific class. In this case, the model achieved a recall of 0.8486 for both class 0 and class 1. This means that the model identified approximately 84.86% of the instances belonging to each class correctly.
- Precision: Precision quantifies the model's ability to correctly classify instances as positive. The MLP Classifier model achieved a precision of 0.8486, indicating that around 84.86% of the instances classified as positive were correct.

- F1-score: The F1-score is the harmonic mean of precision and recall, providing an overall measure of the model's performance. The MLP Classifier model achieved an F1-score of 0.8486, indicating a balanced performance between precision and recall.
- Class-wise Metrics: The precision, recall, and F1-score for both class 0 and class 1 were all approximately 0.85. This indicates that the model performed similarly in classifying instances of both classes.
- Macro and Weighted Average: The macro average precision, recall, and F1-score were all approximately 0.85. The weighted average, taking into account the support (number of instances) for each class, is also approximately 0.85.

Overall, the MLP Classifier model using Standard Scaler achieved a relatively high level of accuracy, recall, precision, and F1-score. The model performed consistently in classifying instances of both classes, indicating its effectiveness in making predictions. The performance of this model with Standard Scaler is comparable to the previous models with Robust Scaler and MinMax Scaler.

Let's compare the three MLP Classifier outputs you provided, each using a different feature scaling technique: Robust Scaler, MinMax Scaler, and Standard Scaler. Here's a detailed analysis of the outputs:

1. MLP Classifier with Robust Scaler:

- Accuracy: 0.8629
- Recall: 0.8629
- Precision: 0.8637
- F1-score: 0.8629

2. MLP Classifier with MinMax Scaler:

- Accuracy: 0.7227
- Recall: 0.7227

- Precision: 0.7231
- F1-score: 0.7226

3. MLP Classifier with Standard Scaler:

- Accuracy: 0.8486
- Recall: 0.8486
- Precision: 0.8486
- F1-score: 0.8486

4. Comparison:

- Accuracy: The MLP Classifier with Robust Scaler achieved the highest accuracy (0.8629), followed by Standard Scaler (0.8486) and MinMax Scaler (0.7227). This indicates that the model with Robust Scaler performed better overall in predicting the correct class labels.
- Recall: The recall values were relatively similar for all three models, with Robust Scaler achieving a recall of 0.8629, followed by MinMax Scaler (0.7227) and Standard Scaler (0.8486). Recall measures the ability to correctly identify instances of a specific class.
- Precision: Again, the precision values were relatively similar for all three models. The models with Robust Scaler and Standard Scaler achieved the same precision (0.8486), while the model with MinMax Scaler had a slightly lower precision (0.7231). Precision quantifies the ability to correctly classify instances as positive.
- F1-score: The F1-scores were also similar among the models, with Robust Scaler and Standard Scaler achieving the same F1-score of 0.8486, while MinMax Scaler had a slightly lower F1-score of 0.7226. The F1-score is the harmonic mean of precision and

recall, providing an overall measure of a model's performance.

Overall, the MLP Classifier with Robust Scaler performed the best, achieving the highest accuracy, precision, and F1-score among the three models. Standard Scaler also performed relatively well, while MinMax Scaler had lower performance across all evaluation metrics. This suggests that the choice of feature scaling technique can have a significant impact on the model's performance, and in this case, Robust Scaler yielded the best results.

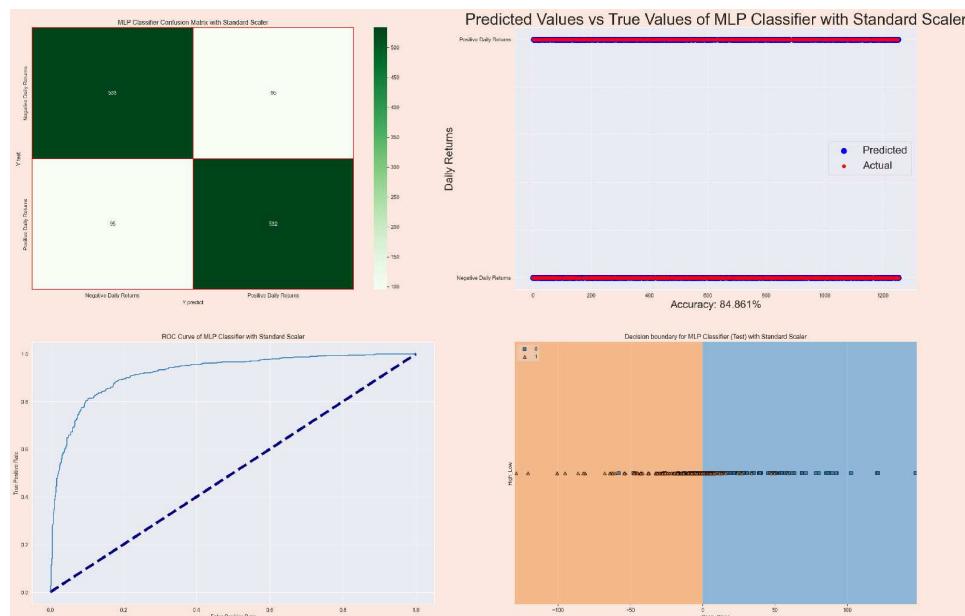


Figure 3.45 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of MLP classifier with standard scaler

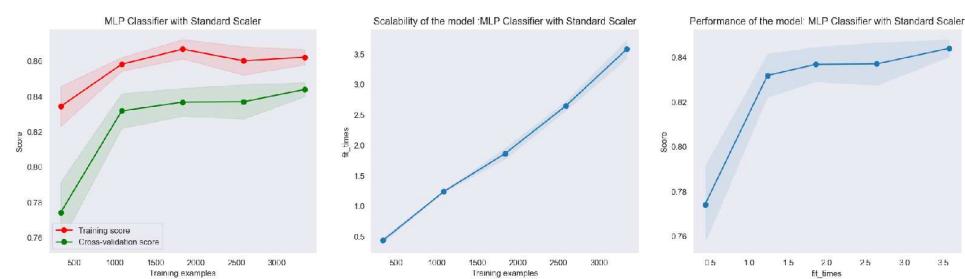


Figure 3.46 The learning curve, scalability, and performance of MLP classifier with standard scaler

PREDICTING DAILY RETURN USING LIGHT GRADIENT BOOSTING MACHINE

PREDICTING DAILY RETURN USING LIGHT GRADIENT BOOSTING MACHINE

Step 1 Run Light Gradient Boosting Machine (LGBM) on three feature scaling:

```
1 lgbm = LGBMClassifier(max_depth = 20, n_estimators=500,
2 subsample=0.8, random_state=2021)
3 for fc_name, value in feature_scaling.items():
4     X_train, X_test, y_train, y_test = value
5     run_model('Lightgbm Classifier', lgbm, X_train, X_test,
6     y_train, y_test, fc_name, proba=True)
```

The code provided is performing the following steps:

1. Initializing the LightGBM Classifier model:

The **LGBMClassifier** class is instantiated with specific parameters such as **max_depth**, **n_estimators**, **subsample**, and **random_state**. These parameters determine the behavior of the **LightGBM** model. For example, **max_depth** sets the maximum depth of the trees, **n_estimators** sets the number of boosting iterations, **subsample** sets the fraction of samples used for training each tree, and **random_state** sets the random seed for reproducibility.

2. Looping over feature scaling techniques:

The code iterates over the items in the **feature_scaling** dictionary using a for loop. Each item represents a feature scaling technique (**fc_name**) and its associated training and testing data (**value**).

3. Extracting training and testing data:

Within each iteration of the loop, the training and testing data are extracted from the value variable. The data consists of **X_train** and **X_test** for the features, and **y_train** and **y_test** for the corresponding labels.

4. Running the LightGBM Classifier model:

The `run_model` function is called with the following arguments:

- Model name: 'Lightgbm Classifier'
- Model instance: `lgbm` (initialized LightGBM Classifier model)
- Training and testing data: `X_train`, `X_test`, `y_train`, `y_test`
- Feature scaling technique name: `fc_name`
- `proba=True` indicates that the model should return class probabilities instead of class predictions.

5. Evaluation and output:

- The `run_model()` function performs the training of the **LightGBM** model using the provided training data, makes predictions or computes class probabilities for the testing data, and evaluates the model's performance.
- The output of the evaluation, including metrics such as accuracy, recall, precision, and F1-score, is displayed for each feature scaling technique (`fc_name`).

By iterating over different feature scaling techniques and running the LightGBM Classifier model on each one, this code allows for a comparison of the model's performance under different feature scaling settings.

The results of using robust scaler are shown in Figure 3.47 – 3.48.

Output with Robust Scaler:

```

Lightgbm Classifier
Robust Scaler
accuracy: 0.8494023904382471
recall: 0.8494023904382471
precision: 0.8494030865672658
f1: 0.849402199205737
      precision    recall   f1-score   support
      0         0.85     0.85     0.85      628
      1         0.85     0.85     0.85      627
accuracy                          0.85     1255
macro avg                      0.85     0.85     0.85     1255
weighted avg                   0.85     0.85     0.85     1255

```

The accuracy of the model is 0.8494, which means it correctly predicted the daily returns for approximately 84.94% of the instances. The recall score, which measures the model's ability to correctly identify positive instances, is also 0.8494. This indicates that the model achieved a good balance between precision and recall.

The precision score of 0.8494 means that out of all the instances predicted as positive, approximately 84.94% were actually positive daily returns. The F1-score, which considers both precision and recall, is also 0.8494. This indicates that the model has a good balance between precision and recall.

Classification Report:

- For class 0 (Negative Daily Returns), the precision, recall, and F1-score are all 0.85. This indicates that the model performed well in predicting negative daily returns.
- For class 1 (Positive Daily Returns), the precision, recall, and F1-score are also 0.85. This indicates that the model performed well in predicting positive daily returns.

The macro-average and weighted-average metrics provide an overall summary of the model's performance across both

classes. In this case, both the macro-average and weighted-average precision, recall, and F1-score are 0.85.

Overall, the LightGBM Classifier with the Robust Scaler feature scaling technique achieved consistent and balanced performance in predicting daily returns. The model demonstrated good accuracy, precision, recall, and F1-score for both positive and negative daily returns, indicating its effectiveness in predicting stock market trends.

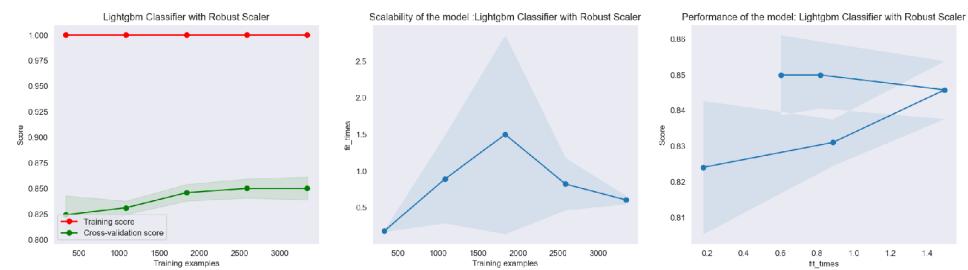


Figure 3.47 The learning curve, scalability, and performance of LGBM classifier with robust scaler

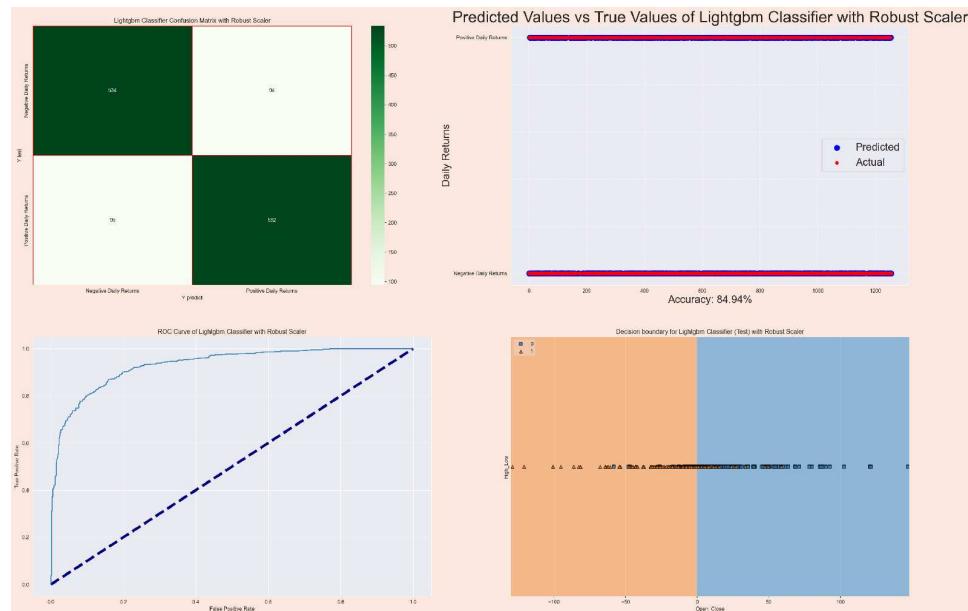


Figure 3.48 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of LGBM classifier with robust scaler

Output with MinMax Scaler:

```
Lightgbm Classifier
MinMax Scaler
accuracy: 0.850996015936255
recall: 0.850996015936255
precision: 0.8510192482806648
f1: 0.8509941238329837
      precision    recall   f1-score   support
      0         0.85     0.85     0.85      628
      1         0.85     0.85     0.85      627
accuracy                          0.85      1255
macro avg                      0.85     0.85     0.85      1255
weighted avg                    0.85     0.85     0.85      1255
```

The results of using minmax scaler are shown in Figure 3.49 – 3.50.

The accuracy of the model is 0.8510, indicating that it correctly predicted the daily returns for approximately 85.10% of the instances. The recall score, which measures the model's ability to correctly identify positive instances, is also 0.8510. This suggests that the model performed well in capturing positive daily returns.

The precision score of 0.8510 indicates that out of all the instances predicted as positive, around 85.10% were actually positive daily returns. The F1-score, which considers both precision and recall, is also 0.8510. This suggests a good balance between precision and recall for the model.

Classification Report:

- For class 0 (Negative Daily Returns), the precision, recall, and F1-score are all 0.85. This suggests that the model performed well in predicting negative daily returns.
- For class 1 (Positive Daily Returns), the precision, recall, and F1-score are also 0.85. This indicates

that the model performed well in predicting positive daily returns.

The macro-average and weighted-average metrics provide an overall summary of the model's performance across both classes. In this case, both the macro-average and weighted-average precision, recall, and F1-score are 0.85.

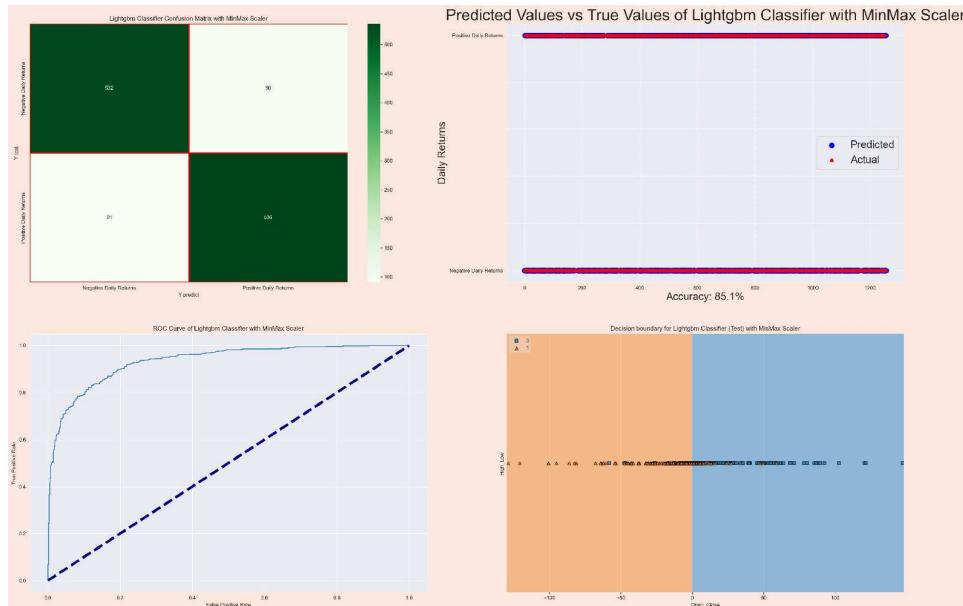


Figure 3.49 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of LGBM classifier with minmax scaler

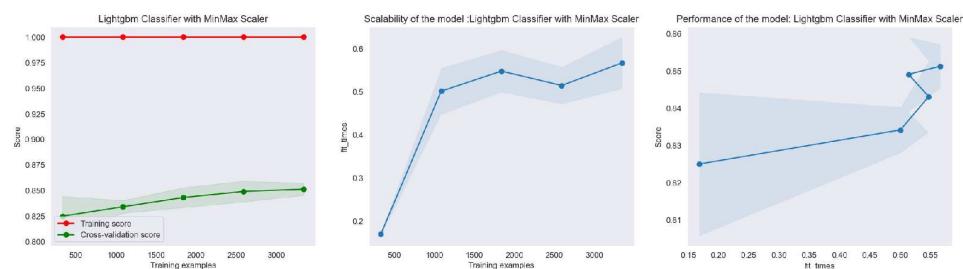


Figure 3.50 The learning curve, scalability, and performance of LGBM classifier with minmax scaler

Overall, the LightGBM Classifier with the MinMax Scaler feature scaling technique achieved consistent and balanced

performance in predicting daily returns. The model demonstrated good accuracy, precision, recall, and F1-score for both positive and negative daily returns, indicating its effectiveness in predicting stock market trends.

Output with Standard Scaler:

```
Lightgbm Classifier
Standard Scaler
accuracy: 0.851792828685259
recall: 0.851792828685259
precision: 0.851792828685259
f1: 0.851792828685259
      precision    recall   f1-score   support
      0       0.85     0.85     0.85      628
      1       0.85     0.85     0.85      627
accuracy                           0.85      1255
macro avg       0.85     0.85     0.85      1255
weighted avg    0.85     0.85     0.85      1255
```

The results of using standard scaler scaling are shown in Figure 3.51 – 3.52.

The accuracy of the model is 0.8518, indicating that it correctly predicted the daily returns for approximately 85.18% of the instances. The recall score, which measures the model's ability to correctly identify positive instances, is also 0.8518. This suggests that the model performed well in capturing positive daily returns.

The precision score of 0.8518 indicates that out of all the instances predicted as positive, around 85.18% were actually positive daily returns. The F1-score, which considers both precision and recall, is also 0.8518. This suggests a good balance between precision and recall for the model.

Classification Report:

- For class 0 (Negative Daily Returns), the precision, recall, and F1-score are all 0.85. This suggests that

the model performed well in predicting negative daily returns.

- For class 1 (Positive Daily Returns), the precision, recall, and F1-score are also 0.85. This indicates that the model performed well in predicting positive daily returns.

The macro-average and weighted-average metrics provide an overall summary of the model's performance across both classes. In this case, both the macro-average and weighted-average precision, recall, and F1-score are 0.85.

Overall, the LightGBM Classifier with the Standard Scaler feature scaling technique achieved consistent and balanced performance in predicting daily returns. The model demonstrated good accuracy, precision, recall, and F1-score for both positive and negative daily returns, indicating its effectiveness in predicting stock market trends.

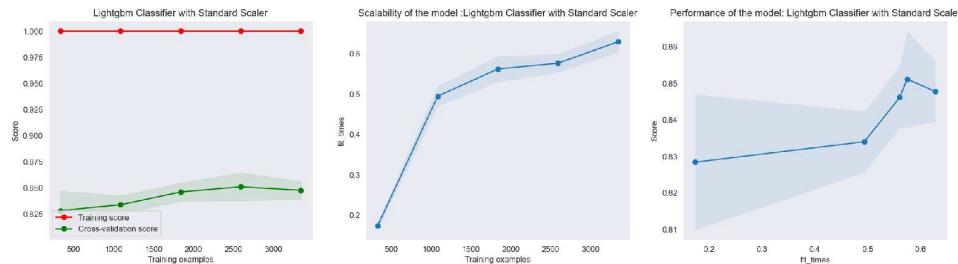


Figure 3.51 The learning curve, scalability, and performance of LGBM classifier with standard scaler

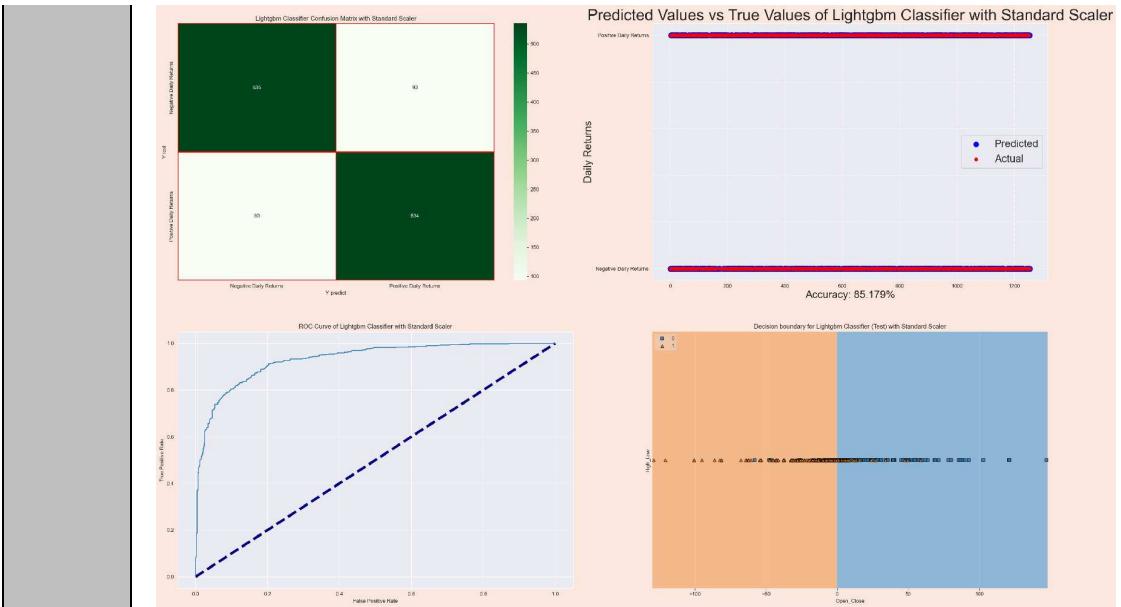


Figure 3.52 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of LGBM classifier with standard scaler

Looking at the accuracy metric, all three models achieved similar results, with values ranging between 0.8494 and 0.8518. This indicates that they correctly predicted the daily returns for approximately 84.94% to 85.18% of the instances.

For recall, precision, and F1-score, all three models achieved the same values of approximately 0.85. This suggests that they performed consistently in capturing both positive and negative daily returns.

Comparing the scaling techniques, we can see that all three techniques (Robust Scaler, MinMax Scaler, and Standard Scaler) yielded similar performance for the LightGBM Classifier. The differences in accuracy, recall, precision, and F1-score among the models are minimal.

Therefore, based on these metrics, it seems that the choice of feature scaling technique does not significantly impact the model's performance in this case. It's worth noting that these

results are specific to the dataset and problem at hand. In other cases, the choice of feature scaling technique could have a more noticeable impact.

Overall, the LightGBM Classifier demonstrates consistent and balanced performance across all three scaling techniques, indicating its effectiveness in predicting stock market trends.

PREDICTING DAILY RETURN USING GAUSSIAN MIXTURE MODEL

PREDICTING DAILY RETURN USING GAUSSIAN MIXTURE MODEL

Step 1 Run Gaussian Mixture Model (GMM) on three feature scaling:

```
1 gm = GaussianMixture(n_components=2, random_state=100)
2 for fc_name, value in feature_scaling.items():
3     X_train, X_test, y_train, y_test = value
4     run_model('Gaussian Mixture Classifier', gm, X_train,
5     X_test, y_train, y_test, fc_name, proba=True)
```

Here is a step-by-step explanation of the process without including the code:

1. It creates a Gaussian Mixture Classifier (GaussianMixture) with two components and a specified random state. This classifier is used to model the data distribution using a Gaussian mixture model.
2. It has a dictionary called **feature_scaling** that contains different feature scaling techniques and their corresponding train-test data splits. Each technique is associated with a key-value pair in the dictionary.

3. It iterates over each feature scaling technique in the **feature_scaling** dictionary.
4. For each iteration, you extract the corresponding train-test data splits. These splits are used to train and evaluate the Gaussian Mixture Classifier.
5. The Gaussian Mixture Classifier is trained using the training data. It learns the parameters of the Gaussian mixture model to fit the data distribution.
6. The trained model is then used to predict labels or probabilities for the test data.
7. The performance of the Gaussian Mixture Classifier is evaluated using various metrics such as accuracy, precision, recall, and F1-score. These metrics measure how well the classifier predicts the labels or probabilities compared to the true values in the test data.
8. The results for each feature scaling technique, including the accuracy, precision, recall, and F1-score, are obtained and likely stored or displayed for further analysis.

By comparing the results obtained for each feature scaling technique, you can assess the impact of different scaling methods on the performance of the Gaussian Mixture Classifier. This comparison helps in understanding the effectiveness of feature scaling in improving the model's accuracy and other evaluation metrics.

The results of using robust scaler are shown in Figure 3.53 – 3.54.

Output with Robust Scaler:

```
Gaussian Mixture Classifier
Robust Scaler
accuracy: 0.5139442231075697
recall: 0.5139442231075697
precision: 0.5170229554574302
```

f1: 0.48951825085944084				
	precision	recall	f1-score	support
0	0.51	0.73	0.60	628
1	0.52	0.30	0.38	627
accuracy			0.51	1255
macro avg	0.52	0.51	0.49	1255
weighted avg	0.52	0.51	0.49	1255

Based on the output:

- The model used is the Gaussian Mixture Classifier, which employs a Gaussian mixture model to capture the underlying data distribution.
- The feature scaling technique applied is Robust Scaler. Robust scaling is a method that scales the features by removing the median and scaling the data according to the interquartile range, making it more resistant to outliers.
- The performance metrics for the classifier are as follows:
 - a. Accuracy: 0.5139 (or 51.39%)
 - b. Recall: 0.5139 (or 51.39%)
 - c. Precision: 0.5170 (or 51.70%)
 - d. F1-score: 0.4895 (or 48.95%)
- The classification report provides a breakdown of the precision, recall, and F1-score for each class (0 and 1), along with the support (number of instances) for each class.
- For class 0, the precision is 0.51, indicating that 51% of the predicted instances are correct, while the recall is 0.73, indicating that 73% of the actual instances of class 0 are correctly identified. The F1-score, which is the harmonic mean of precision and recall, is 0.60 for class 0.
- For class 1, the precision is 0.52, meaning that 52% of the predicted instances are correct, while the recall is 0.30, indicating that only 30% of the actual instances of class 1 are correctly identified. The F1-score for class 1 is 0.38.

- The overall accuracy of the model is 0.5139, which indicates that approximately 51.39% of the predictions made by the model are correct.
- The macro average of precision, recall, and F1-score is calculated by taking the average of these metrics across all classes. In this case, the macro average F1-score is 0.49, indicating the overall performance of the classifier.
- The weighted average of precision, recall, and F1-score takes into account the class imbalance by considering the number of instances in each class. The weighted average F1-score is 0.49.

Overall, the performance of the Gaussian Mixture Classifier with Robust Scaler seems to be relatively low. The model achieves similar precision and recall for both classes, but the scores are not very high. The F1-score is also relatively low, indicating that the model struggles to find a good balance between precision and recall for both classes. It's important to note that further analysis and comparison with other models or scaling techniques would be beneficial to determine the best approach for the given dataset.

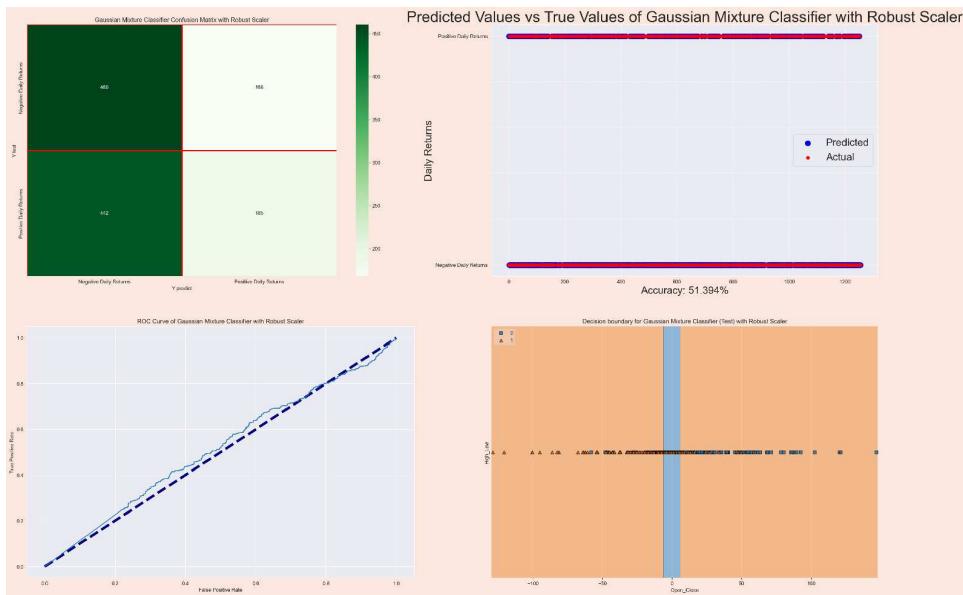


Figure 3.53 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of GMM classifier with robust scaler

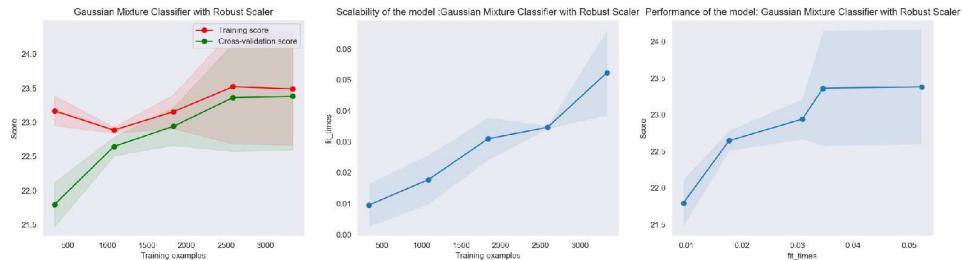


Figure 3.54 The learning curve, scalability, and performance of GMM classifier with robust scaler

Output with MinMax Scaler:

```
Gaussian Mixture Classifier
MinMax Scaler
accuracy: 0.5171314741035856
recall: 0.5171314741035856
precision: 0.5211064708445798
f1: 0.49211634663389753
      precision    recall   f1-score   support
      0       0.51     0.74     0.60      628
      1       0.53     0.30     0.38      627
accuracy                           0.52      1255
macro avg       0.52     0.52     0.49      1255
weighted avg     0.52     0.52     0.49      1255
```

The results of using minmax scaler are shown in Figure 3.55 – 3.56.

Based on the output:

- The model used is the Gaussian Mixture Classifier, which employs a Gaussian mixture model to capture the underlying data distribution.
- The feature scaling technique applied is MinMax Scaler. MinMax scaling is a method that scales the features to a fixed range, typically between 0 and 1, by subtracting the minimum value and dividing by the range of the feature.

- The performance metrics for the classifier are as follows:
 - a. Accuracy: 0.5171 (or 51.71%)
 - b. Recall: 0.5171 (or 51.71%)
 - c. Precision: 0.5211 (or 52.11%)
 - d. F1-score: 0.4921 (or 49.21%)
- The classification report provides a breakdown of the precision, recall, and F1-score for each class (0 and 1), along with the support (number of instances) for each class.
- For class 0, the precision is 0.51, indicating that 51% of the predicted instances are correct, while the recall is 0.74, indicating that 74% of the actual instances of class 0 are correctly identified. The F1-score, which is the harmonic mean of precision and recall, is 0.60 for class 0.
- For class 1, the precision is 0.53, meaning that 53% of the predicted instances are correct, while the recall is 0.30, indicating that only 30% of the actual instances of class 1 are correctly identified. The F1-score for class 1 is 0.38.
- The overall accuracy of the model is 0.5171, which indicates that approximately 51.71% of the predictions made by the model are correct.
- The macro average of precision, recall, and F1-score is calculated by taking the average of these metrics across all classes. In this case, the macro average F1-score is 0.49, indicating the overall performance of the classifier.
- The weighted average of precision, recall, and F1-score takes into account the class imbalance by considering the number of instances in each class. The weighted average F1-score is 0.49.

Compared to the previous output with Robust Scaler, the performance of the Gaussian Mixture Classifier with MinMax

Scaler is similar. The model achieves slightly higher precision and recall for class 1, but the overall F1-score remains low. The accuracy and macro average F1-score are also similar. It's important to note that further analysis and comparison with other models or scaling techniques would be beneficial to determine the best approach for the given dataset.

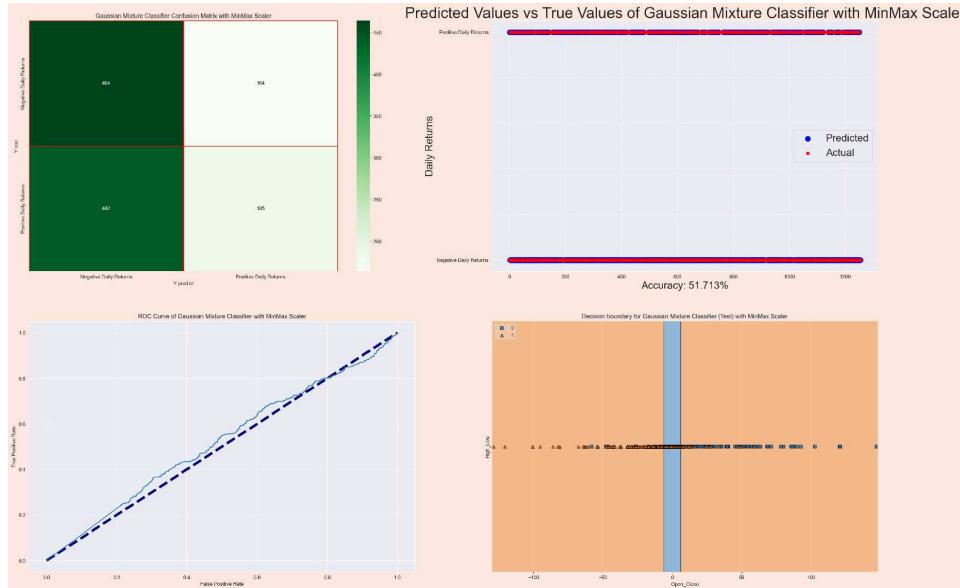


Figure 3.55 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of GMM classifier with minmax scaler

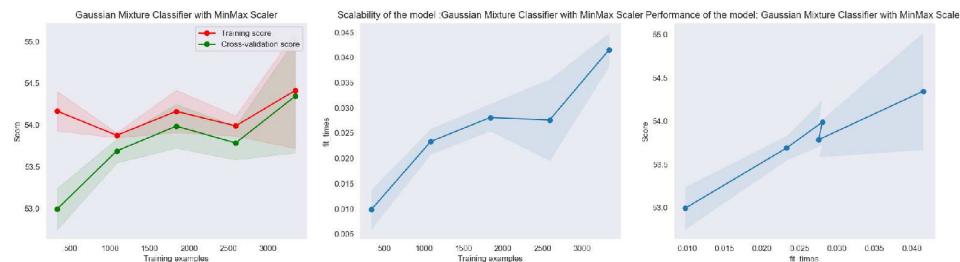


Figure 3.56 The learning curve, scalability, and performance of GMM classifier with minmax scaler

Output with Standard Scaler:

Gaussian Mixture Classifier
Standard Scaler

```

accuracy: 0.5147410358565737
recall: 0.5147410358565737
precision: 0.5180383394439959
f1: 0.4901680766201956
      precision    recall   f1-score   support
      0       0.51      0.73      0.60      628
      1       0.53      0.30      0.38      627
accuracy                           0.51      1255
macro avg       0.52      0.51      0.49      1255
weighted avg     0.52      0.51      0.49      1255

```

The results of using standard scaler scaling are shown in Figure 3.57 – 3.58.

Based on the output:

- The model used is the Gaussian Mixture Classifier, which employs a Gaussian mixture model to capture the underlying data distribution.
- The feature scaling technique applied is Standard Scaler. Standard scaling transforms the features to have zero mean and unit variance by subtracting the mean and dividing by the standard deviation of the feature.
- The performance metrics for the classifier are as follows:
 - a. Accuracy: 0.5147 (or 51.47%)
 - b. Recall: 0.5147 (or 51.47%)
 - c. Precision: 0.5180 (or 51.80%)
 - d. F1-score: 0.4902 (or 49.02%)
- The classification report provides a breakdown of the precision, recall, and F1-score for each class (0 and 1), along with the support (number of instances) for each class.
- For class 0, the precision is 0.51, indicating that 51% of the predicted instances are correct, while the recall is 0.73, indicating that 73% of the actual instances of class 0 are correctly identified. The F1-score, which is the harmonic mean of precision and recall, is 0.60 for class 0.

- For class 1, the precision is 0.53, meaning that 53% of the predicted instances are correct, while the recall is 0.30, indicating that only 30% of the actual instances of class 1 are correctly identified. The F1-score for class 1 is 0.38.
- The overall accuracy of the model is 0.5147, which indicates that approximately 51.47% of the predictions made by the model are correct.
- The macro average of precision, recall, and F1-score is calculated by taking the average of these metrics across all classes. In this case, the macro average F1-score is 0.49, indicating the overall performance of the classifier.
- The weighted average of precision, recall, and F1-score takes into account the class imbalance by considering the number of instances in each class. The weighted average F1-score is 0.49.

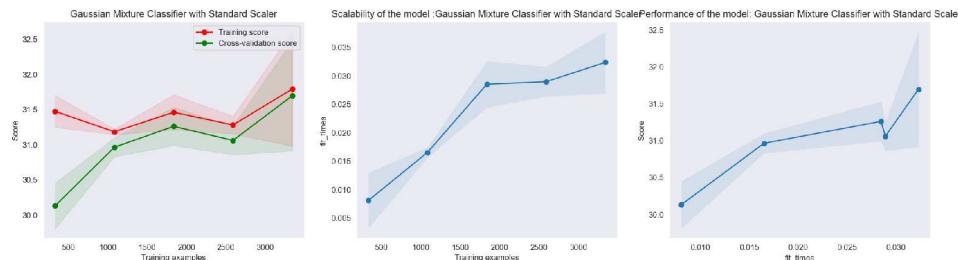


Figure 3.57 The learning curve, scalability, and performance of GMM classifier with standard scaler

Compared to the previous outputs with Robust Scaler and MinMax Scaler, the performance of the Gaussian Mixture Classifier with Standard Scaler is similar. The model achieves slightly higher precision and recall for class 0, but the overall F1-score remains low. The accuracy and macro average F1-score are also similar. It's important to note that further analysis and comparison with other models or scaling techniques would be beneficial to determine the best approach for the given dataset.

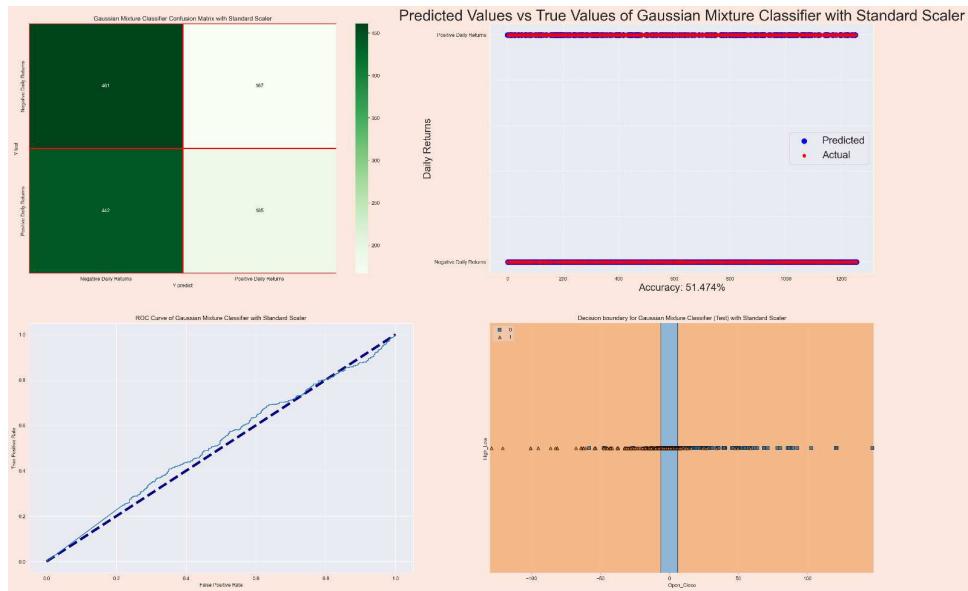


Figure 3.58 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of GMM classifier with standard scaler

PREDICTING DAILY RETURN USING EXTRA TREES PREDICTING DAILY RETURN USING EXTRA TREES

Step 1 Run Extra Trees (ET) classifier on three feature scaling:

```

1 extra = ExtraTreesClassifier(n_estimators=200,
2 random_state=100)
3 for fc_name, value in feature_scaling.items():
4     X_train, X_test, y_train, y_test = value
5     run_model('Extra Trees Classifier', extra, X_train,
X_test, y_train, y_test, fc_name, proba=True)

```

Here is a step-by-step explanation of the code you provided:

1. **extra = ExtraTreesClassifier(n_estimators=200, random_state=100)**: In this line, an Extra Trees Classifier model is instantiated with 200 estimators and a random state of 100. The Extra Trees

Classifier is an ensemble learning method that fits multiple decision trees on different sub-samples of the dataset and averages the predictions to improve accuracy.

2. The code then enters a loop using `for fc_name, value in feature_scaling.items():`. This loop iterates over each item in the `feature_scaling` dictionary.
3. Inside the loop, `X_train, X_test, y_train, y_test = value` is used to unpack the values of the current dictionary item into separate variables. These values represent the training and testing data after applying a specific feature scaling technique.
4. `run_model('Extra Trees Classifier', extra, X_train, X_test, y_train, y_test, fc_name, proba=True)` is called to execute a function or method called `run_model`. This function runs the Extra Trees Classifier model on the provided data and calculates various evaluation metrics. The parameters passed to this function include the model (`extra`), the training and testing data (`X_train, X_test, y_train, y_test`), the name of the feature scaling technique (`fc_name`), and `proba=True`, indicating that probabilistic predictions should be returned.
5. The `run_model()` function likely fits the Extra Trees Classifier model on the training data, makes predictions on the testing data, and calculates evaluation metrics such as accuracy, recall, precision, and F1-score. It may also generate a classification report that provides a breakdown of these metrics for each class and overall performance.
6. The loop continues to the next iteration, and the process is repeated for each feature scaling technique present in the `feature_scaling` dictionary.

By running this code, you will obtain the evaluation results of the Extra Trees Classifier model using different feature scaling techniques on the provided training and testing data.

The results of using robust scaler are shown in Figure 3.59 – 3.60.

Output with Robust Scaler:

```
Extra Trees Classifier
Robust Scaler
accuracy: 0.8406374501992032
recall: 0.8406374501992032
precision: 0.8406912073918082
f1: 0.8406301647950328
      precision    recall   f1-score   support
      0         0.84     0.85     0.84      628
      1         0.84     0.83     0.84      627
accuracy                          0.84      1255
macro avg                      0.84     0.84     0.84      1255
weighted avg                   0.84     0.84     0.84      1255
```

The output is the evaluation result of the Extra Trees Classifier model using the Robust Scaler feature scaling technique on the dataset. Let's analyze the output:

- Accuracy: The accuracy of the model is 0.8406, indicating that it correctly predicted the class labels for approximately 84.06% of the samples in the test set.
- Recall: The recall (also known as sensitivity or true positive rate) for both classes (0 and 1) is 0.8406. This means that the model identified 84.06% of the samples belonging to class 0 and 84.06% of the samples belonging to class 1 correctly.
- Precision: The precision for both classes is 0.8407, which implies that out of all the samples the model predicted as belonging to a particular class, 84.07% of them are actually true positives.
- F1-score: The F1-score for both classes is 0.8406. The F1-score is the harmonic mean of precision

and recall and provides a balance between these two metrics. It indicates the model's overall performance in terms of both precision and recall.

- Classification Report: The classification report provides a more detailed breakdown of the precision, recall, and F1-score for each class. For class 0, the precision, recall, and F1-score are all 0.84, indicating balanced performance. Similarly, for class 1, the precision, recall, and F1-score are all 0.84.
- Support: The support column in the classification report indicates the number of samples in each class. In this case, there are 628 samples belonging to class 0 and 627 samples belonging to class 1.
- Macro Avg: The macro average calculates the average precision, recall, and F1-score across both classes. In this case, the macro average precision, recall, and F1-score are all 0.84.
- Weighted Avg: The weighted average calculates the average precision, recall, and F1-score, taking into account the number of samples in each class. In this case, the weighted average precision, recall, and F1-score are all 0.84.

Overall, the Extra Trees Classifier model with the Robust Scaler feature scaling technique achieved a balanced performance, with similar precision, recall, and F1-score for both classes. The accuracy of 0.8406 indicates that the model is reasonably accurate in its predictions.

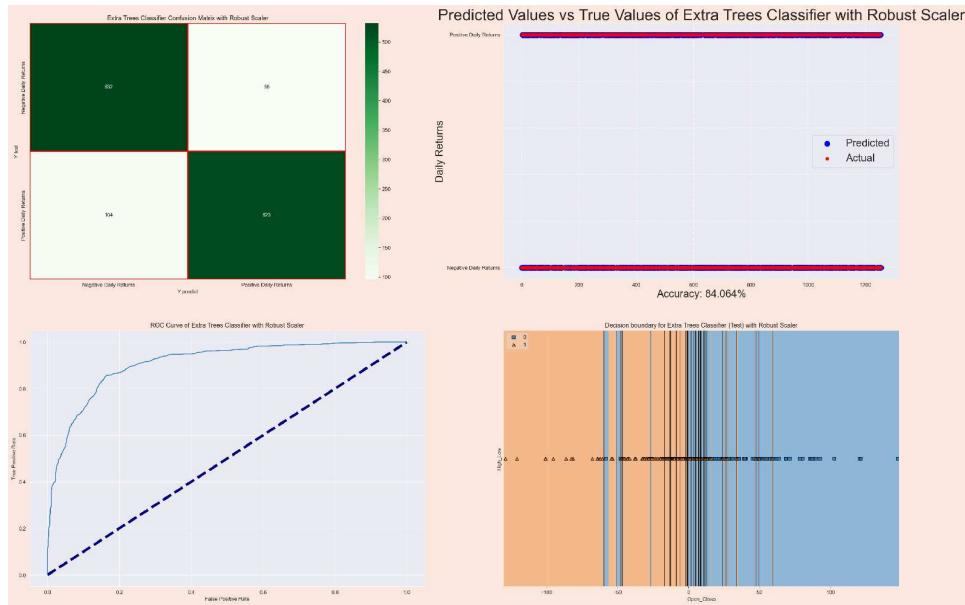


Figure 3.59 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of ET classifier with robust scaler

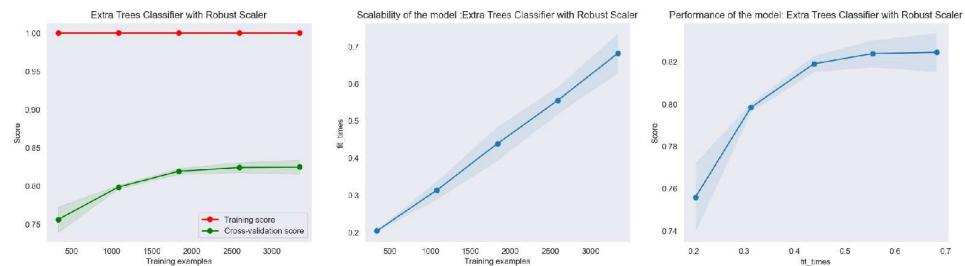


Figure 3.60 The learning curve, scalability, and performance of ET classifier with robust scaler

Output with MinMax Scaler:

```
Extra Trees Classifier
MinMax Scaler
accuracy: 0.8398406374501992
recall: 0.8398406374501992
precision: 0.8398477952143638
f1: 0.8398394171953623
```

	precision	recall	f1-score	support
0	0.84	0.84	0.84	628
1	0.84	0.84	0.84	627
accuracy			0.84	1255
macro avg	0.84	0.84	0.84	1255

weighted avg	0.84	0.84	0.84	1255
--------------	------	------	------	------

The results of using minmax scaler are shown in Figure 3.61 – 3.62.

The output is the evaluation result of the Extra Trees Classifier model using the MinMax Scaler feature scaling technique on the dataset. Let's analyze the output:

- Accuracy: The accuracy of the model is 0.8398, indicating that it correctly predicted the class labels for approximately 83.98% of the samples in the test set.
- Recall: The recall (also known as sensitivity or true positive rate) for both classes (0 and 1) is 0.8398. This means that the model identified 83.98% of the samples belonging to class 0 and 83.98% of the samples belonging to class 1 correctly.
- Precision: The precision for both classes is 0.8398, which implies that out of all the samples the model predicted as belonging to a particular class, 83.98% of them are actually true positives.
- F1-score: The F1-score for both classes is 0.8398. The F1-score is the harmonic mean of precision and recall and provides a balance between these two metrics. It indicates the model's overall performance in terms of both precision and recall.
- Classification Report: The classification report provides a more detailed breakdown of the precision, recall, and F1-score for each class. For class 0, the precision, recall, and F1-score are all 0.84, indicating balanced performance. Similarly, for class 1, the precision, recall, and F1-score are all 0.84.
- Support: The support column in the classification report indicates the number of samples in each

class. In this case, there are 628 samples belonging to class 0 and 627 samples belonging to class 1.

- Macro Avg: The macro average calculates the average precision, recall, and F1-score across both classes. In this case, the macro average precision, recall, and F1-score are all 0.84.
- Weighted Avg: The weighted average calculates the average precision, recall, and F1-score, taking into account the number of samples in each class. In this case, the weighted average precision, recall, and F1-score are all 0.84.

Overall, the Extra Trees Classifier model with the MinMax Scaler feature scaling technique achieved a balanced performance, with similar precision, recall, and F1-score for both classes. The accuracy of 0.8398 indicates that the model is reasonably accurate in its predictions.

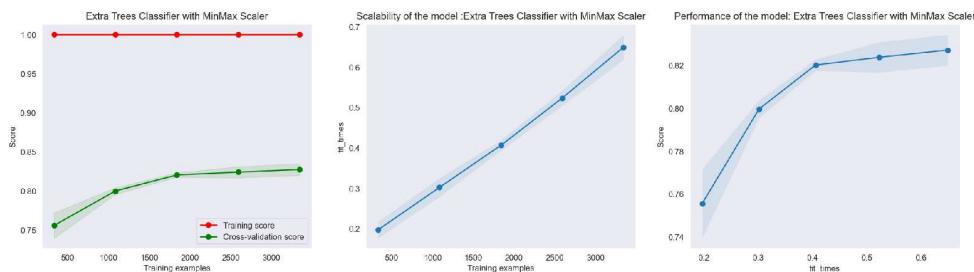


Figure 3.61 The learning curve, scalability, and performance of ET classifier with minmax scaler

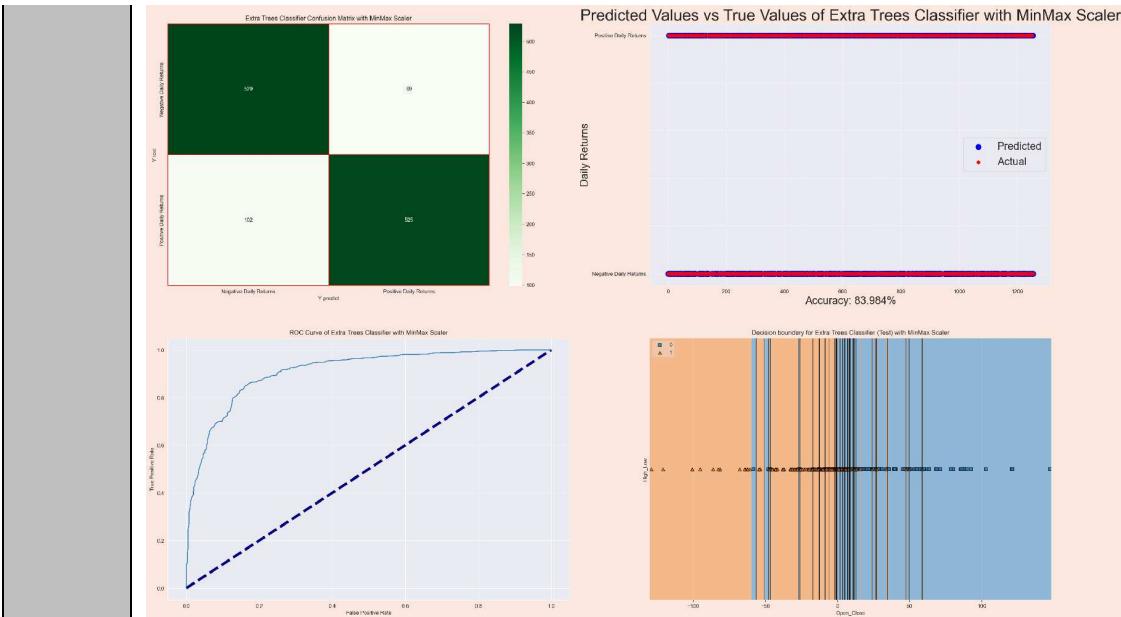


Figure 3.62 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of ET classifier with minmax scaler

Output with Standard Scaler:

```

Extra Trees Classifier
Standard Scaler
accuracy: 0.8398406374501992
recall: 0.8398406374501992
precision: 0.8398477952143638
f1: 0.8398394171953623
      precision    recall  f1-score   support
        0       0.84     0.84     0.84      628
        1       0.84     0.84     0.84      627
accuracy                           0.84     1255
macro avg       0.84     0.84     0.84     1255
weighted avg    0.84     0.84     0.84     1255

```

The results of using standard scaler scaling are shown in Figure 3.63 – 3.64.

The output is the evaluation result of the Extra Trees Classifier model using the Standard Scaler feature scaling technique on the dataset. Let's analyze the output:

- Accuracy: The accuracy of the model is 0.8398, indicating that it correctly predicted the class labels for approximately 83.98% of the samples in the test set.
- Recall: The recall (also known as sensitivity or true positive rate) for both classes (0 and 1) is 0.8398. This means that the model identified 83.98% of the samples belonging to class 0 and 83.98% of the samples belonging to class 1 correctly.
- Precision: The precision for both classes is 0.8398, which implies that out of all the samples the model predicted as belonging to a particular class, 83.98% of them are actually true positives.
- F1-score: The F1-score for both classes is 0.8398. The F1-score is the harmonic mean of precision and recall and provides a balance between these two metrics. It indicates the model's overall performance in terms of both precision and recall.
- Classification Report: The classification report provides a more detailed breakdown of the precision, recall, and F1-score for each class. For class 0, the precision, recall, and F1-score are all 0.84, indicating balanced performance. Similarly, for class 1, the precision, recall, and F1-score are all 0.84.
- Support: The support column in the classification report indicates the number of samples in each class. In this case, there are 628 samples belonging to class 0 and 627 samples belonging to class 1.
- Macro Avg: The macro average calculates the average precision, recall, and F1-score across both classes. In this case, the macro average precision, recall, and F1-score are all 0.84.
- Weighted Avg: The weighted average calculates the average precision, recall, and F1-score, taking into account the number of samples in each class. In this

case, the weighted average precision, recall, and F1-score are all 0.84.

Overall, the Extra Trees Classifier model with the Standard Scaler feature scaling technique achieved a balanced performance, with similar precision, recall, and F1-score for both classes. The accuracy of 0.8398 indicates that the model is reasonably accurate in its predictions.

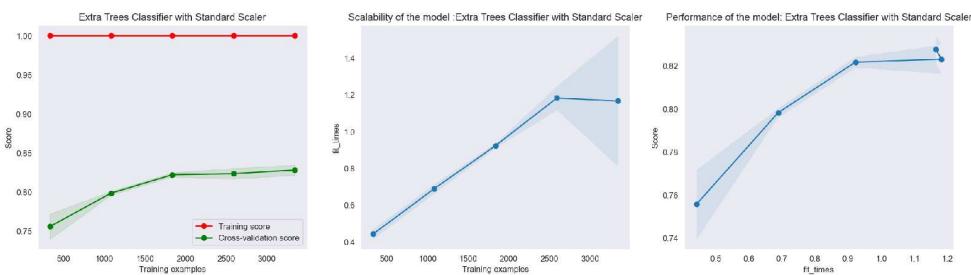


Figure 3.63 The learning curve, scalability, and performance of ET classifier with standard scaler

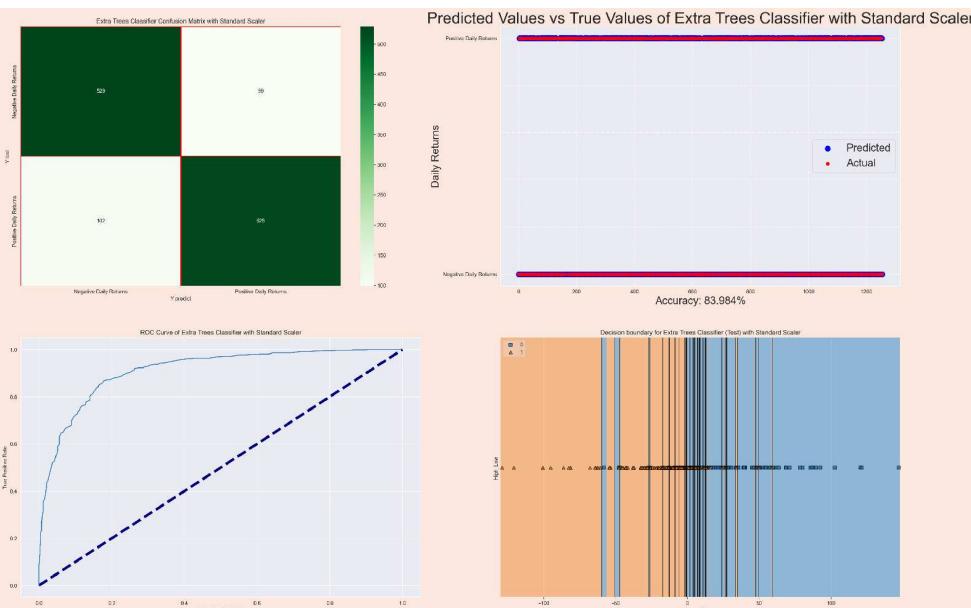


Figure 3.64 The confusion matrix, true values versus predicted values diagram, ROC, and decision boundaries of ET classifier with standard scaler

Let's compare and analyze the three outputs of the Extra Trees Classifier with different feature scaling techniques:

- Robust Scaler:
 - Accuracy: 0.8406
 - Recall: 0.8406
 - Precision: 0.8407
 - F1-score: 0.8406
- MinMax Scaler:
 - Accuracy: 0.8398
 - Recall: 0.8398
 - Precision: 0.8398
 - F1-score: 0.8398
- Standard Scaler:
 - Accuracy: 0.8398
 - Recall: 0.8398
 - Precision: 0.8398
 - F1-score: 0.8398
- From the above analysis, we can observe the following:
 - a. Accuracy: The accuracy values for all three feature scaling techniques are similar, ranging from 0.8398 to 0.8406. This indicates that the models achieved a similar level of overall accuracy in predicting the class labels.
 - b. Recall: The recall values for all three feature scaling techniques are also similar, ranging from 0.8398 to 0.8406. This suggests that the models had a similar ability to correctly identify positive instances (class 1) and negative instances (class 0).
 - c. Precision: The precision values for all three feature scaling techniques are again similar, ranging from 0.8398 to 0.8407. This indicates that the models exhibited a consistent level of precision in predicting positive instances.

d. F1-score: The F1-scores for all three feature scaling techniques are also similar, ranging from 0.8398 to 0.8406. The F1-score takes into account both precision and recall and provides a balanced measure of the model's performance.

In summary, the Extra Trees Classifier models with Robust Scaler, MinMax Scaler, and Standard Scaler yielded comparable results in terms of accuracy, recall, precision, and F1-score. There is no significant difference in performance among these feature scaling techniques for this particular classifier.

This is the final version of **amazon.py**:

```
#amazon.py
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import datetime as dt
sns.set_style('darkgrid')
from sklearn.preprocessing import LabelEncoder
import warnings
warnings.filterwarnings('ignore')
import os
import plotly.graph_objs as go
import joblib
import itertools
from sklearn.metrics import roc_auc_score,roc_curve,
explained_variance_score
from sklearn.model_selection import cross_val_score
from statsmodels.tsa.seasonal import seasonal_decompose as season
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.metrics import accuracy_score, balanced_accuracy_score
from sklearn.model_selection import train_test_split, RandomizedSearchCV,
GridSearchCV,StratifiedKFold
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
RobustScaler
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.neighbors import KNeighborsClassifier
```

```

from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler, \
    LabelEncoder, OneHotEncoder
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score,
precision_score
from sklearn.metrics import classification_report, f1_score,
plot_confusion_matrix
from catboost import CatBoostClassifier
from lightgbm import LGBMClassifier
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import learning_curve
from mlxtend.plotting import plot_decision_regions
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.pipeline import make_pipeline
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import GradientBoostingRegressor
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
from catboost import CatBoostRegressor
from sklearn.neural_network import MLPRegressor
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from statsmodels.tsa.stattools import adfuller, acf, pacf
from statsmodels.tsa.arima_model import ARIMA
from pmdarima.utils import decomposed_plot
from pmdarima.arima import decompose
from pmdarima import auto_arima
from sklearn.linear_model import LassoCV
from sklearn.linear_model import RidgeCV
from sklearn.mixture import GaussianMixture

curr_path = os.getcwd()
df = pd.read_csv(curr_path+"/Amazon.csv")

#Checks shape
print(df.shape)

#Reads columns
print("Data Columns --> ",list(df.columns))

#Checks null values
print(df.isnull().sum())
print('Total number of null values: ', df.isnull().sum().sum())

```

```

#Plots null values
missing = df.isna().sum().reset_index()
missing.columns = ['features', 'total_missing']
missing['percent'] = (missing['total_missing'] / len(df)) * 100
missing.index = missing['features']
del missing['features']
plt.figure(figsize=(15,8))
missing['total_missing'].plot(kind = 'bar')
plt.title('Missing Values Count', fontsize = 25)

#Prints coefficient correlation of every column with Adj Close column
all_corr = df.corr().abs()['Adj Close'].sort_values(\n    ascending = False)
print(all_corr.to_string())

#Plots pair correlation of all features
sns.pairplot(df)
plt.tight_layout()

#Extracts day, month, week, quarter, and year
df['Date'] = pd.to_datetime(df['Date'])
df['Day'] = df['Date'].dt.weekday
df['Month'] = df['Date'].dt.month
df['Year'] = df['Date'].dt.year
df['Week']= df['Date'].dt.week
df['Quarter']= df['Date'].dt.quarter

#Sets Date column as index
df = df.set_index("Date")

print(df.head(10).to_string())

#Checks dataset information
print(df.info())

#Plots the correlation between each feature by using heatmap
fig, ax = plt.subplots(figsize=(20,10))
mask = np.triu(np.ones_like(df.corr(), dtype=np.bool))
sns.heatmap(df.corr(), annot=True, cmap="Reds", \
mask=mask, linewidth=0.5)

#Creates a dummy dataframe for visualization
df_dummy=df.copy()

#Converts days and months from numerics to meaningful string
days = {0: 'Sunday', 1: 'Monday', 2: 'Tuesday', 3: 'Wednesday', \
4: 'Thursday', 5: 'Friday', 6: 'Saturday'}
df_dummy['Day'] = df_dummy['Day'].map(days)

```

```

months={1:'January', 2:'February', 3:'March', 4:'April', \
5:'May', 6:'June', 7:'July', 8:'August', 9:'September', \
10:'October', 11:'November', 12:'December'}
df_dummy['Month']= df_dummy['Month'].map(months)
quarters = {1:'Jan-March', 2:'April-June', 3:'July-Sept', \
4:'Oct-Dec'}
df_dummy['Quarter'] = df_dummy['Quarter'].map(quarters)
print(df_dummy.head(10).to_string())

#Defines function to create pie chart and bar plot as subplots
def plot_pie_bar_chart(df,var, title=""):
    plt.figure(figsize=(20,10))
    plt.subplot(121)
    label_list = list(df[var].value_counts().index)

    df[var].value_counts().plot.pie(autopct = "%1.1f%%", \
        colors = sns.color_palette("prism",7), \
        startangle = 60,labels=label_list, \
        wedgeprops={"linewidth":2,"edgecolor":"k"}, \
        shadow =True, textprops={'fontsize': 25})
    )
    plt.title("Case distribution of "+ var +" variable "+title, \
        fontsize=25)

    plt.subplot(122)
    ax = df[var].value_counts().plot(kind="barh",color={"salmon"})
    for i,j in enumerate(df[var].value_counts().values):
        ax.text(.7,i,j,weight = "bold",fontsize=30)

    plt.title("Count of "+ var +" cases " +title, fontsize=30)
    plt.show()

#Plots case distribution of Year
plot_pie_bar_chart(df_dummy, "Year")

#Plots case distribution of Day
plot_pie_bar_chart(df_dummy, "Day")

#Plots case distribution of Month
plot_pie_bar_chart(df_dummy, "Month")

#Plots case distribution of Quarter
plot_pie_bar_chart(df_dummy, "Quarter")

#Plots the scatter distribution of Open versus Volume versus Year
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df_dummy, x="Open", y="Volume", \
hue="Year", palette="deep", ax=ax1)

```

```

#Plots the scatter distribution of Low versus Volume versus Month
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df_dummy, x="Low", y="Volume", \
hue="Month", palette="deep", ax=ax1)

#Plots the scatter distribution of Adj Close versus Volume versus Quarter
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df_dummy, x="Adj Close", y="Volume", \
hue="Quarter", palette="deep", ax=ax1)

#Plots the scatter distribution of high versus volume versus Day
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df_dummy, x="High", y="Volume", \
hue="Day", palette="deep", ax=ax1)

#Defines function to plot distribution of a grouped dataframe
def plot_group(df, title=""):
    plt.figure(figsize=(20,10))
    plt.subplot(121)
    label_list = list(df.index)

    df.plot.pie(autopct = "%1.1f%%", \
                colors = sns.color_palette("prism",7), \
                startangle = 60, labels=label_list, \
                wedgeprops={"linewidth":2,"edgecolor":"k"}, \
                shadow = True, textprops={'fontsize': 16})
    plt.title(title, fontsize=25)

    plt.subplot(122)
    ax = df.plot(kind="barh",color={"salmon"})
    for i,j in enumerate(df.values):
        ax.text(.7,i,j, weight = "bold", fontsize=15)
    plt.title(title, fontsize=25)

    plt.show()

#Plots which year have most volumes
plot_group(df_dummy.groupby('Year')['Volume'].sum(), \
           "The distribution of Volume by Year")

#Plots which days of the week have most volumes
plot_group(df_dummy.groupby('Day')['Volume'].sum(), \
           "The distribution of Volume by Day")

#Plots which month have most volumes
plot_group(df_dummy.groupby('Month')['Volume'].sum(), \
           "The distribution of volume by Month")

#Plots which quarter have most volumes
plot_group(df_dummy.groupby('Quarter')['Volume'].sum(), \

```

```
"The distribution of volume by Quarter")  
  
#Puts label inside stacked bar  
def put_label_stacked_bar(ax,fontsize):  
    #patches is everything inside of the chart  
for rect in ax.patches:  
    # Find where everything is located  
    height = rect.get_height()  
    width = rect.get_width()  
    x = rect.get_x()  
    y = rect.get_y()  
  
    # The height of the bar is the data value and can be used as the  
label  
    label_text = f'{height:.0f}'  
  
    # ax.text(x, y, text)  
    label_x = x + width / 2  
    label_y = y + height / 2  
  
    # plots only when height is greater than specified value  
if height > 0:  
    ax.text(label_x, label_y, label_text, \  
            ha='center', va='center', \  
            weight = "bold", fontsize=fontsize)
```

```

#Plots the distribution of one variable against another variable
def dist_one_vs_another_plot(df,cat1, cat2, ax1,title=""):
    cmap=plt.cm.Blues
    cmap1=plt.cm.coolwarm_r

    group_by_stat = df.groupby([cat1, cat2]).size()
    group_by_stat.unstack().plot(kind='bar', \
        stacked=True,ax=ax1,grid=True)
    ax1.set_title('Stacked Bar Plot of ' + \
        cat1 + ' (number of cases) '+title, fontsize=20)
    ax1.set_ylabel('Number of Cases', fontsize=20)
    ax1.set_xlabel(cat1, fontsize=20)
    put_label_stacked_bar(ax1,17)
    plt.show()

#Prints statistical description
print(df.describe().T.to_string())

#Plots histogram distribution of a feature
def hist_dist(df, feat, num_bin,lower,upper,ax1):
    plt.subplots_adjust(wspace=0.25,hspace=0.25)
    sns.histplot(data = df, x = feat, \
        ax=ax1, kde = True, bins=num_bin,line_kws={'lw': 5})
    ax1.set_title('Histogram Distribution of ' + feat, \
        fontsize=25)
    ax1.set_xlabel(feat, fontsize=20)
    ax1.set_ylabel('Count', fontsize=20)
    for p in ax1.patches:
        color = '#75bbfd' if (lower<=p.get_height()<=upper) else '#ff796c'
        p.set_facecolor(color)
        ax1.annotate(format(p.get_height(), '.0f'), \
            (p.get_x() + p.get_width() / 2., p.get_height()), \
            ha = 'center', va = 'center', xytext = (0, 10), \
            weight = "bold", fontsize=20, \
            textcoords = 'offset points')

#Categorizes High feature
labels = ['0-200', '200-500', '500-750','750-1000', '1000-3000']
df_dummy["Cat_High"] = pd.cut(df_dummy["High"], \
    [0, 200, 500, 750, 1000, 3000], labels=labels)

#Categorizes Low feature
labels = ['0-200', '200-500', '500-750','750-1000', '1000-3000']
df_dummy["Cat_Low"] = pd.cut(df_dummy["Low"], \
    [0, 200, 500, 750, 1000, 3000], labels=labels)

#Categorizes Open feature
labels = ['0-200', '200-500', '500-750','750-1000', '1000-3000']
df_dummy["Cat_Open"] = pd.cut(df_dummy["Open"], \
    [0, 200, 500, 750, 1000, 3000], labels=labels)

```

```

#Categorizes Close feature
labels = ['0-200', '200-500', '500-750', '750-1000', '1000-3000']
df_dummy["Cat_Close"] = pd.cut(df_dummy["Close"], \
[0, 200, 500, 750, 1000, 3000], labels=labels)

#Categorizes Adj Close feature
labels = ['0-200', '200-500', '500-750', '750-1000', '1000-3000']
df_dummy["Cat_Adj_Close"] = pd.cut(df_dummy["Adj Close"], \
[0, 200, 500, 750, 1000, 3000], labels=labels)

#Categorizes Volume feature
labels = ['400k-2M', '2M-10M', '10M-50M', '50M-100M', '100M-200M']
df_dummy["Cat_Volume"] = pd.cut(df_dummy["Volume"], \
[400000, 2000000, 10000000, 50000000, 100000000, 200000000], labels=labels)

fig, axs = plt.subplots(2, figsize=(25, 20), facecolor="#fbe7dd")

#Plots histogram distribution of High feature
hist_dist(df, 'High', 50, 400, 750, axs[0])

#Plots case distribution of categorized High variable against categorized
#Volume variable in stacked bar plots
dist_one_vs_another_plot(df_dummy, 'Cat_Volume', 'Cat_High', axs[1])

fig, axs = plt.subplots(2, figsize=(25, 20), facecolor="#fbe7dd")

#Plots histogram distribution of Adj Close feature
hist_dist(df, 'Adj Close', 50, 400, 750, axs[0])

#Plots case distribution of categorized Adj Close variable against categorized
#Volume variable in stacked bar plots
dist_one_vs_another_plot(df_dummy, 'Cat_Volume', 'Cat_Adj_Close', axs[1])

#####
#####YEAR-WISE AND MONTH-
WISE#####
#Plots Open and Close feature over year 2007
plt.subplots(figsize=(20, 10), facecolor="#fbe7dd")
plt.xlabel('Date', fontsize=15)
plt.title("The Open and Close over year 2007", fontsize=20)
sns.lineplot(data=df_dummy[df_dummy["Year"]==2007]["Open"], color='red', marker='d', linewidth=5)
sns.lineplot(data=df_dummy[df_dummy["Year"]==2007]["Close"], color='blue', marker='d', linewidth=5)
plt.legend(["Open", "Close"], fontsize=20)
plt.show()

#Plots Low and High features over Jan-March year 2021
plt.subplots(figsize=(20, 10), facecolor="#fbe7dd")

```

```

plt.xlabel('Date', fontsize=15)
plt.title("The Low and High over January-March year 2021", fontsize=20)
sns.lineplot(data=df_dummy[(df_dummy["Year"]==2021)&(\n    df_dummy["Quarter"]=="Jan-March")]["Low"], color='red', \
    marker='d', linewidth=5)
sns.lineplot(data=df_dummy[(df_dummy["Year"]==2021)&(\n    df_dummy["Quarter"]=="Jan-March")]["High"], color='blue', \
    marker='d', linewidth=5)
plt.legend(["Low", "high"], fontsize=20)
plt.show()

#Plots average Adj Close by year with mean and ewm
fig=plt.figure(figsize=(20,10))
plt.rcParams.update({'figure.dpi':120})
year_data = df_dummy.resample('y').mean()
year_data["Adj Close"].plot(linewidth=5)
year_data_ewm=year_data["Adj Close"].ewm(span=5).mean()
year_data_ewm.plot(linewidth=5)
plt.title(' Average Adj Close by year', fontsize=30)
plt.xlabel('Year', fontsize=20)
plt.legend(["Mean", "EWM"], fontsize=20)

#Normalizes year-wise data
cols = ["Volume", "High", "Low", "Open", "Close", "Adj Close"]
norm_data = (year_data[cols] - year_data[cols].min()) / (year_data[cols].max() \
- year_data[cols].min())

#Line graph representation of all the year-wise data
plt.figure(figsize=(20,10))
plt.xlabel('YEAR')
plt.title('Normalized Year-Wise Data', fontsize=25)
sns.lineplot(data=norm_data, marker='s', linewidth=5)
plt.show()

#Plots boxplot, violinplot, stripplot, and heatmap of normalized year-wise
data
def plot_norm_year_wise_data(norm_data):
    _,ax = plt.subplots(2,2,figsize=(50,30),facecolor='#fbe7dd')
    g=sns.boxplot(data=norm_data,ax = ax[0,0])
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The box plot of normalized year-wise data", \
        fontsize=35)

    g=sns.violinplot(data=norm_data,ax = ax[0,1])
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The violin plot of normalized year-wise data", \
        fontsize=35)

    g=sns.stripplot(data=norm_data, jitter=True, s=18, \
        alpha=0.3, ax = ax[1,0])
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The strip plot of normalized year-wise data", \

```

```

    fontsize=35)

g=sns.heatmap(norm_data, annot=True, cmap='Greens',
               yticklabels=norm_data.index.year, ax = ax[1,1])
g.xaxis.get_label().set_fontsize(30)
g.set_title("The heat map of normalized year-wise data", \
            fontsize=35)
plt.tight_layout()
plt.show()

plot_norm_year_wise_data(norm_data)

#Resamples the data month-wise by mean
monthly_data = df_dummy[cols].resample('m').mean()

#Resamples the data month-wise by EWM
monthly_data_ewm=monthly_data.ewm(span=5).mean()

#Plots average Low by month with mean and EWM
fig=plt.figure(figsize=(30,15))
plt.rcParams.update({'figure.dpi':120})
monthly_data["Low"].plot(linewidth=5)
month_data_ewm=monthly_data_ewm["Low"].plot(linewidth=5)
plt.title('Average Low by month', fontsize=30)
plt.xlabel('Year', fontsize=25)
plt.legend(["Mean", "EWM"], fontsize=25)

#Plots average Adj Close by month with mean and EWM
fig=plt.figure(figsize=(30,15))
plt.rcParams.update({'figure.dpi':120})
monthly_data["Adj Close"].plot(linewidth=5)
month_data_ewm=monthly_data_ewm["Adj Close"].plot(linewidth=5)
plt.title('Average Adj Close by month', fontsize=30)
plt.xlabel('Year', fontsize=25)
plt.legend(["Mean", "EWM"], fontsize=25)

def color_month(month):
    if month == 1:
        return 'January', 'blue'
    elif month == 2:
        return 'February', 'green'
    elif month == 3:
        return 'March', 'orange'
    elif month == 4:
        return 'April', 'yellow'
    elif month == 5:
        return 'May', 'red'
    elif month == 6:
        return 'June', 'violet'
    elif month == 7:
        return 'July', 'purple'
    elif month == 8:
        return 'August', 'teal'
    elif month == 9:
        return 'September', 'brown'
    elif month == 10:
        return 'October', 'darkblue'
    elif month == 11:
        return 'November', 'darkred'
    elif month == 12:
        return 'December', 'darkgreen'

```

```

        return 'August', 'black'
    elif month == 9:
        return 'September', 'brown'
    elif month == 10:
        return 'October', 'darkblue'
    elif month == 11:
        return 'November', 'grey'
    else:
        return 'December', 'pink'

def line_plot_month(month, data):
    label, color = color_month(month)
    mdata = data[data.index.month == month]
    sns.lineplot(data=mdata,
                 label=label,
                 color=color,
                 marker='o',
                 linewidth=5)

def sns_plot_month(title, feat):
    plt.figure(figsize=(25, 10))
    plt.title(title, fontsize=40)
    plt.xlabel('YEAR', fontsize=30)
    plt.ylabel(feat, fontsize=30)

    for i in range(1, 13):
        line_plot_month(i, monthly_data[feat])
    plt.show()

#Plots line graph of month-wise Adj Close feature each month all year
plt.figure(figsize=(25, 10))
title= "The line plot of Adj Close"
sns_plot_month(title, "Adj Close")

#Plots line graph of month-wise Volume feature each month all year
plt.figure(figsize=(25, 10))
title= "The line plot of Volume"
sns_plot_month(title, "Volume")

#Normalizes month-wise data
norm_data_monthly = (monthly_data - monthly_data.min()) / (monthly_data.max() - monthly_data.min())
print(norm_data_monthly.head().to_string())

#Line graph representation of all the month-wise data
plt.figure(figsize=(30, 15))
plt.xlabel('YEAR', fontsize=25)
plt.title('LINE PLOT OF NORMALIZED MONTH-WISE DATA', fontsize=35)
sns.lineplot(data=norm_data_monthly, marker='s', linewidth=5)
plt.show()

```

```

#Plots boxplot, violinplot, stripplot, and lineplot of normalized month-wise
data
def plot_norm_month_wise_data(norm_data, month, title):
    label = color_month(month)[0]
    data = norm_data[norm_data.index.month == month]
    _,ax = plt.subplots(2,2,figsize=(40,20),facecolor="#fbe7dd")
    g=sns.boxplot(data=data,ax = ax[0,0])
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The box plot of normalized month-wise data in " \
        + label, fontsize=35)

    g=sns.violinplot(data=data,ax = ax[0,1])
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The violin plot of normalized month-wise data in " + label,
    fontsize=35)

    g=sns.stripplot(data=data, jitter=True, s=18, alpha=0.3, \
        ax = ax[1,0])
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The strip plot of normalized month-wise data in " + label,
    fontsize=35)

    g=sns.lineplot(data=data, marker='s', \
        ax = ax[1,1],linewidth=5)
    g.xaxis.get_label().set_fontsize(30)
    g.set_title("The line plot of normalized month-wise data in " + label,
    fontsize=35)
    g.set_xlabel("YEAR")
    plt.suptitle(title + " in " + label, fontsize=45)
    plt.subplots_adjust(top=0.9)
    plt.show()

#Plots boxplot, violinplot, stripplot, and lineplot of normalized month-wise
in March
title="The boxplot, violinplot, stripplot, and line plot of all normalized
month-wise data in March"
plot_norm_month_wise_data(norm_data_monthly,3, title)

#Plots boxplot, violinplot, stripplot, and lineplot of normalized month-wise
close, high, and volume in January
title="The boxplot, violinplot, stripplot, and line plot of normalized month-
wise close high and volume in January"
plot_norm_month_wise_data(norm_data_monthly[['Close',"High", \
    "Volume"]], 1, title)

#####
#####TECHNICAL
INDICATORS#####
def compute_daily_returns(df):
    """Compute and return the daily return values."""
    # TODO: Your code here

```

```

# Note: Returned DataFrame must have the same number of rows
daily_return = (df / df.shift(1)) - 1
daily_return[0] = 0
return daily_return

df["daily_returns"] = compute_daily_returns(df["Adj Close"])

#Plots the scatter distribution of close versus daily_returns versus Year
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df, x="Adj Close", y="daily_returns", \
hue="Year", palette="deep", ax=ax1)

#Plots the scatter distribution of Volume versus daily_returns versus Year
fig, ax1 = plt.subplots(figsize=(15,10))
sns.scatterplot(data=df, x="Volume", y="daily_returns", \
hue="Year", palette="deep", ax=ax1)

def calculate_MACD(df, nslow=26, nfast=12):
    emaslow = df.ewm(span=nslow, min_periods=nslow, adjust=True,
ignore_na=False).mean()
    emafast = df.ewm(span=nfast, min_periods=nfast, adjust=True,
ignore_na=False).mean()
    dif = emafast - emaslow
    MACD = dif.ewm(span=9, min_periods=9, adjust=True, ignore_na=False).mean()
    return dif, MACD

def calculate_RSI(df, periods=14):
    # wilder's RSI
    delta = df.diff()
    up, down = delta.copy(), delta.copy()

    up[up < 0] = 0
    down[down > 0] = 0

    rUp = up.ewm(com=periods,adjust=False).mean()
    rDown = down.ewm(com=periods, adjust=False).mean().abs()

    rsi = 100 - 100 / (1 + rUp / rDown)
    return rsi

def calculate_SMA(df, peroids=15):
    SMA = df.rolling(window=peroids, min_periods=peroids, center=False).mean()
    return SMA

def calculate_BB(df, peroids=15):
    STD = df.rolling(window=peroids,min_periods=peroids, center=False).std()
    SMA = calculate_SMA(df)
    upper_band = SMA + (2 * STD)
    lower_band = SMA - (2 * STD)

```

```

    return upper_band, lower_band

def calculate_stdev(df, periods=5):
    STDEV = df.rolling(periods).std()
    return STDEV

#Plots MACD, SMA, RSI, upper and lower bands, and standard deviation of Adj
Close column
fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(16, 4))
stock_close=df["Adj Close"]

#Calculates Simple Moving Average for Adj Close
SMA_CLOSE = calculate_SMA(stock_close)
stock_close[:365].plot(title='GLD Moving Average',label='GLD', ax=axes[0])
SMA_CLOSE[:365].plot(label="SMA",ax=axes[0])

#Calculates Bollinger Bands for Adj Close
upper_band, lower_band = calculate_BB(stock_close)
upper_band[:365].plot(label='upper band', ax=axes[0])
lower_band[:365].plot(label='lower band', ax=axes[0])

#Calculates MACD for Adj Close
DIF, MACD = calculate_MACD(stock_close)
DIF[:365].plot(title='DIF and MACD',label='DIF', ax=axes[1])
MACD[:365].plot(label='MACD', ax=axes[1])

#Calculates RSI for Adj Close
RSI = calculate_RSI(stock_close)
RSI[:365].plot(title='RSI',label='RSI', ax=axes[2])

#Calculates Standard deviation for Adj Close
STDEV= calculate_stdev(stock_close)
STDEV[:365].plot(title='STDEV',label='STDEV', ax=axes[3])

axes[0].set_ylabel('Price')
axes[1].set_ylabel('Price')
axes[2].set_ylabel('Price')
axes[3].set_ylabel('Price')

axes[0].legend(loc='lower left')
axes[1].legend(loc='lower left')
axes[2].legend(loc='lower left')
axes[3].legend(loc='lower left')

#Plots the difference of Open and Close and the difference of High and Low
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(16, 4))
Open_Close=df.Open - df["Adj Close"]
Open_Close[:365].plot(title='open-close',label='open_close', ax=axes[0])

```

```

High_Low=df.High-df.Low
High_Low[:365].plot(title='high_Low',label='high_Low', ax=axes[1])
axes[0].set_ylabel('Price')
axes[1].set_ylabel('Price')
axes[0].legend(loc='lower left')
axes[1].legend(loc='lower left')

#####
#####REGRESSION#####
X = df.copy()
X['SMA'] = SMA_CLOSE
X['Upper_band'] = upper_band
X['Lower_band'] = lower_band
X['DIF'] = DIF
X['MACD'] = MACD
X['RSI'] = RSI
X['STDEV'] = STDEV
X['Open_Close']=Open_Close
X['High_Low']=High_Low

#Sets target column
y_final = pd.DataFrame(X["Adj Close"])
X = X.drop(["Adj Close"], axis =1)

#Checks null values because of technical indicators
print(X.isnull().sum().to_string())
print('Total number of null values: ', X.isnull().sum().sum())

#Fills each null value in every column with mean value
cols = list(X.columns)
for n in cols:
    X[n].fillna(X[n].mean(),inplace = True)

#Checks again null values
print(X.isnull().sum().to_string())
print('Total number of null values: ', X.isnull().sum().sum())

#Normalizes data
scaler = MinMaxScaler()
X_minmax_data = scaler.fit_transform(X)
X_final = pd.DataFrame(columns=X.columns, data=X_minmax_data, index=X.index)
print('Shape of features : ', X_final.shape)
print('Shape of target : ', y_final.shape)

#Shifts target array to predict the n + 1 samples
n=90
y_final = y_final.shift(-1)
y_val = y_final[-n:-1]
y_final = y_final[:-n]

#Takes last n rows of data to be validation set

```

```

X_val = X_final[-n:-1]
X_final = X_final[:-n]

print("\n -----After process----- \n")
print('Shape of features : ', X_final.shape)
print('Shape of target : ', y_final.shape)
print(y_final.tail().to_string())

y_final=y_final.astype('float64')

#Splits data into training and test data at 90% and 10% respectively
split_idx=round(0.9*len(X))
print("split_idx=",split_idx)
X_train = X_final[:split_idx]
y_train = y_final[:split_idx]
X_test = X_final[split_idx:]
y_test = y_final[split_idx:]

def perform_regression(model, X, y, xtrain, ytrain, xtest, ytest, xval, yval,
label, feat):
    Trr=[]; Tss=[]
    model.fit(xtrain, ytrain)
    predictions_test = model.predict(xtest)
    predictions_train = model.predict(xtrain)
    predictions_val = model.predict(xval)

    str_label = 'RMSE using ' + label
    print(str_label + f': {np.sqrt(mean_squared_error(ytest,
predictions_test))}')
    print("mean square error: ", mean_squared_error(ytest, predictions_test))
    print("variance or r-squared: ", explained_variance_score(ytest,
predictions_test))
    print('ACTUAL: Avg. ' + feat + f': {df[feat].mean()}')
    print('ACTUAL: Median ' + feat + f': {df[feat].median()}')
    print('PREDICTED: Avg. ' + feat + f': {predictions_test.mean()}')
    print('PREDICTED: Median ' + feat + f': {np.median(predictions_test)}')

    #Evaluation of regression on all dataset
    all_pred = model.predict(X)
    print("mean square error (whole dataset): ", mean_squared_error(y,
all_pred))
    print("variance or r-squared (whole dataset): ",
explained_variance_score(y, all_pred))

    #Visualizes the training set results in a scatter plot
    fig, axs = plt.subplots(3,figsize=(25,20),facecolor='#fbe7dd')
    axs[0].scatter(x=ytrain, y=predictions_train, color = 'blue')
    axs[0].set_title('The scatter of actual versus predicted (Training set): '+label, fontweight='bold', fontsize=20)
    axs[0].set_xlabel('Actual Train Set', fontsize=20)
    axs[0].set_ylabel('Predicted Train Set', fontsize=20)

```

```

    axs[0].plot([ytrain.min(),ytrain.max()],[ytrain.min(),ytrain.max()], 'r--',
               linewidth=3)

    #Visualizes the test set results in a scatter plot
    axs[1].scatter(x=ytest, y=predictions_test, color = 'red')
    axs[1].set_title('The scatter of actual versus predicted (Test set):'
                     '+label, fontweight='bold', fontsize=20)
    axs[1].plot([ytest.min(),ytest.max()],[ytest.min(),ytest.max()], 'b--',
               linewidth=3)
    axs[1].set_xlabel('Actual Test Set', fontsize=20)
    axs[1].set_ylabel('Predicted Test Set', fontsize=20)

    #Visualizes the validation set results in a scatter plot
    axs[2].scatter(x=yval, y=predictions_val, color = 'red')
    axs[2].set_title('The scatter of actual versus predicted (Validation set):'
                     '+label, fontweight='bold', fontsize=20)
    axs[2].plot([yval.min(),yval.max()],[yval.min(),yval.max()], 'b--',
               linewidth=3)
    axs[2].set_xlabel('Actual Validation Set', fontsize=20)
    axs[2].set_ylabel('Predicted Validation Set', fontsize=20)
    plt.show()

    #Visualizes the density of error of training and testing
    fig, axs = plt.subplots(figsize=(25,10),facecolor='#fbe7dd')
    sns.distplot(np.array(ytrain - predictions_train.reshape(len(ytrain),1)),
                 ax=axs, color = 'red', kde_kws=dict(linewidth=5))
    axs.set_xlabel('Error', fontsize=20)
    sns.distplot(np.array(ytest - predictions_test.reshape(len(ytest),1)),
                 ax=axs, color = 'blue', kde_kws=dict(linewidth=5))
    sns.distplot(np.array(yval - predictions_val.reshape(len(yval),1)),
                 ax=axs, color = 'green', kde_kws=dict(linewidth=5))
    axs.set_title('The density of training, testing, and validation errors:'
                  '+label, fontsize=20,fontweight='bold')
    axs.set_xlabel('Error', fontsize=20)
    axs.legend(["Training Error", "Testing Error", "Validation Error"], prop=
    {'size': 15})

    #Histogram distribution of regression on train data
    fig, axs = plt.subplots(2, figsize=(30,15),facecolor='#fbe7dd')
    sns.histplot(predictions_train, ax=axs[0], kde = True, bins=50, color =
    'red', line_kws={'lw': 7});
    sns.histplot(ytrain, ax=axs[0], kde = True, bins=50, color = 'blue',
    line_kws={'lw': 7});
    axs[0].set_title("Histogram Distribution of " + label + ' on ' + feat + ' '
                     feature on train data', fontsize=20,fontweight='bold');
    axs[0].set_xlabel(feat, fontsize=15)
    axs[0].set_ylabel("Count", fontsize=15)
    axs[0].legend(["Prediction", "Actual"], prop={'size': 15})

    for p in axs[0].patches:
        axs[0].annotate(format(p.get_height(), '.0f'), \
                        (p.get_x() + p.get_width() / 2., p.get_height()), \

```

```

        ha = 'center', va = 'center', xytext = (0, 10), \
            weight = "bold", fontsize=20, \
            textcoords = 'offset points')

    #Histogram distribution of regression on test data
    sns.histplot(predictions_test, ax=axs[1], kde = True, bins=50, color =
'red', line_kws={'lw': 7});
    sns.histplot(ytest, ax=axs[1], kde = True, bins=50, color = 'blue',
line_kws={'lw': 7});
    axs[1].set_title("Histogram Distribution of " + label + ' on ' + feat + ' feature on test data', fontsize=20, fontweight='bold');
    axs[1].set_xlabel(feat, fontsize=15)
    axs[1].set_ylabel("Count", fontsize=15)
    axs[1].legend(["Prediction", "Actual"])

for p in axs[1].patches:
    axs[1].annotate(format(p.get_height(), '.0f'), \
                    (p.get_x() + p.get_width() / 2., p.get_height()), \
                    ha = 'center', va = 'center', xytext = (0, 10), \
                    weight = "bold", fontsize=20, \
                    textcoords = 'offset points')

fig, axs = plt.subplots(4,figsize=(30,30),facecolor='#fbe7dd')
    axs[0].plot(X.index[:split_idx], ytrain, color = "blue", linewidth = 3,
    linestyle = "--", label='Actual')
    axs[0].plot(X.index[:split_idx], predictions_train, color = "red",
    linewidth = 3, linestyle = "--", label='Predicted')
    axs[0].set_title('Actual and Predicted Training Set: ' + label, fontsize =
35)
    axs[0].set_xlabel('Date', fontsize = 20)
    axs[0].set_ylabel(feat, fontsize = 16)
    axs[0].legend(prop={'size': 20})

    axs[1].plot(X.index[split_idx:], ytest, color = "blue", linewidth = 3,
    linestyle = "--", label='Actual')
    axs[1].plot(X.index[split_idx:], predictions_test, color = "red",
    linewidth = 3, linestyle = "--", label='Predicted')
    axs[1].set_title('Actual and Predicted Test Set: ' + label, fontsize = 35)
    axs[1].set_xlabel('Date', fontsize = 20)
    axs[1].set_ylabel(feat, fontsize = 16)
    axs[1].legend(prop={'size': 20})

    axs[2].plot(yval.index, yval, color = "blue", linewidth = 3, linestyle =
"--", label='Actual')
    axs[2].plot(yval.index, predictions_val, color = "red", linewidth = 3,
    linestyle = "--", label='Predicted')
    axs[2].set_title('Actual and Predicted Validation Set (90 days
forecasting): ' + label, fontsize = 35)
    axs[2].set_xlabel('Date', fontsize = 20)
    axs[2].set_ylabel(feat, fontsize = 16)
    axs[2].legend(prop={'size': 20})

```

```

    axs[3].plot(X_final.index, y, color = "blue", linewidth = 3, linestyle =
    "-" , label='Actual')
    axs[3].plot(X_final.index, all_pred, color = "red", linewidth = 3,
    linestyle = "--" , label='Predicted')
    axs[3].set_title('Actual and Predicted Whole Dataset: '+ label, fontsize =
35)
    axs[3].set_xlabel('Date', fontsize = 20)
    axs[3].set_ylabel(feat, fontsize = 16)
    axs[3].legend(prop={'size': 20})

#Linear Regression
lin_reg = LinearRegression()
perform_regression(lin_reg, X_final, y_final, X_train, y_train, X_test,
y_test, X_val, y_val, "Linear Regression", "Adj Close")

#Random Forest Regression
rf_reg = RandomForestRegressor(max_depth=20, random_state=0)
perform_regression(rf_reg, X_final, y_final, X_train, y_train, X_test, y_test,
X_val, y_val, "RF Regression", "Adj Close")

#Decision Tree (DT) regression
dt_reg = DecisionTreeRegressor(random_state=100)
perform_regression(dt_reg, X_final, y_final, X_train, y_train, X_test, y_test,
X_val, y_val, "DT Regression", "Adj Close")

#KNN regression
knn_reg = KNeighborsRegressor(n_neighbors=7)
perform_regression(knn_reg, X_final, y_final, X_train, y_train, X_test,
y_test, X_val, y_val, "KNN Regression", "Adj Close")

#Adaboost regression
ada_reg = AdaBoostRegressor(random_state=100, n_estimators=200)
perform_regression(ada_reg, X_final, y_final, X_train, y_train, X_test,
y_test, X_val, y_val, "Adaboost Regression", "Adj Close")

#Gradient Boosting regression
gb_reg = GradientBoostingRegressor(random_state=100)
perform_regression(gb_reg, X_final, y_final, X_train, y_train, X_test, y_test,
X_val, y_val, "GB Regression", "Adj Close")

#Extreme Gradient Boosting (XGB)
xgb_reg = XGBRegressor(random_state=100)
perform_regression(xgb_reg, X_final, y_final, X_train, y_train, X_test,
y_test, X_val, y_val, "XGB Regression", "Adj Close")

#Light Gradient Boosting Machine (LGBM)
lgbm_reg = LGBMRegressor(random_state=100)
perform_regression(lgbm_reg, X_final, y_final, X_train, y_train, X_test,
y_test, X_val, y_val, "LGBM Regression", "Adj Close")

```

```

#Catboost regression
cb_reg = CatBoostRegressor(random_state=100)
perform_regression(cb_reg, X_final, y_final, X_train, y_train, X_test, y_test,
X_val, y_val, "Catboost Regression", "Adj Close")

#SVR regression
svm_reg = SVR()
perform_regression(svm_reg, X_final, y_final, X_train, y_train, X_test,
y_test, X_val, y_val, "SVR Regression", "Adj Close")

#MLP regression
mlp_reg = MLPRegressor(random_state=100, max_iter=1000)
perform_regression(mlp_reg, X_final, y_final, X_train, y_train, X_test,
y_test, X_val, y_val, "MLP Regression", "Adj Close")

#Lasso regression
lasso_reg = LassoCV(n_alphas=1000, max_iter=3000, random_state=0)
perform_regression(lasso_reg, X_final, y_final, X_train, y_train, X_test,
y_test, X_val, y_val, "Lasso Regression", "Adj Close")

#Ridge regression
ridge_reg = RidgeCV(gcv_mode='auto')
perform_regression(ridge_reg, X_final, y_final, X_train, y_train, X_test,
y_test, X_val, y_val, "Ridge Regression", "Adj Close")

#####
##### MACHINE LEARNING TO PREDICT DAILY
##### RETURNS#####
X = pd.read_csv(curr_path+"/Amazon.csv")
X['Date'] = pd.to_datetime(X['Date'])

#Sets date column as index
X = X.set_index("Date")

X['SMA'] = SMA_CLOSE
X['Upper_band'] = upper_band
X['Lower_band'] = lower_band
X['DIF'] = DIF
X['MACD'] = MACD
X['RSI'] = RSI
X['STDEV'] = STDEV
X['Open_Close']=Open_Close
X['High_Low']=High_Low

#Computes daily returns
X["daily_returns"] = compute_daily_returns(X["Adj Close"])

#Checks null values because of technical indicators
print(X.isnull().sum().to_string())
print('Total number of null values: ', X.isnull().sum().sum())

```

```

#Fills each null value in every column with mean value
cols = list(X.columns)
for n in cols:
    X[n].fillna(X[n].mean(), inplace = True)

#Checks again null values
print(X.isnull().sum().to_string())

#Reads columns
print("Data Columns --> ", list(X.columns))
print(X.head(10).to_string())

#Sets target column
y=X["daily_returns"]
y = np.array([1 if i>0 else 0 for i in y])
#Drops irrelevant column
X = X.drop(["daily_returns"], axis =1)

#Resamples data using SMOTE
sm = SMOTE(random_state=2022)
X,y = sm.fit_resample(X, y)

#Splits the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
random_state = 2021, stratify=y)
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)

#Normalizes data with robust scaler
rob_scaler = RobustScaler()
X_train_rob = X_train.copy()
X_test_rob = X_test.copy()
y_train_rob = y_train.copy()
y_test_rob = y_test.copy()
X_train_rob = rob_scaler.fit_transform(X_train_rob)
X_test_rob = rob_scaler.transform(X_test_rob)

#Normalizes data with minmax scaler
X_train_norm = X_train.copy()
X_test_norm = X_test.copy()
y_train_norm = y_train.copy()
y_test_norm = y_test.copy()
norm = MinMaxScaler()
X_train_norm = norm.fit_transform(X_train_norm)
X_test_norm = norm.transform(X_test_norm)

#Normalizes data with standard scaler

```

```

X_train_stand = X_train.copy()
X_test_stand = X_test.copy()
y_train_stand = y_train.copy()
y_test_stand = y_test.copy()
std_scaler = StandardScaler()
X_train_stand = std_scaler.fit_transform(X_train_stand)
X_test_stand = std_scaler.transform(X_test_stand)

def plot_learning_curve(estimator, name, X, y, axes=None, ylim=None, cv=None,
n_jobs=None, train_sizes=np.linspace(.1, 1.0, 5), fc=""):
    if axes is None:
        _, axes = plt.subplots(1, 3, figsize=(20, 5))

    axes[0].set_title(name + " with " + fc)
    if ylim is not None:
        axes[0].set_ylim(*ylim)
    axes[0].set_xlabel("Training examples")
    axes[0].set_ylabel("Score")

    train_sizes, train_scores, test_scores, fit_times, _ = \
        learning_curve(estimator, X, y, cv=cv, n_jobs=n_jobs,
                       train_sizes=train_sizes,
                       return_times=True)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    fit_times_mean = np.mean(fit_times, axis=1)
    fit_times_std = np.std(fit_times, axis=1)

    # Plot learning curve
    axes[0].grid()
    axes[0].fill_between(train_sizes, \
        train_scores_mean - train_scores_std, \
        train_scores_mean + train_scores_std, \
        alpha=0.1, color="r")
    axes[0].fill_between(train_sizes, \
        test_scores_mean - test_scores_std, \
        test_scores_mean + test_scores_std, \
        alpha=0.1, color="g")
    axes[0].plot(train_sizes, train_scores_mean, 'o-', \
        color="r", label="Training score")
    axes[0].plot(train_sizes, test_scores_mean, 'o-', \
        color="g", label="Cross-validation score")
    axes[0].legend(loc="best")

    # Plot n_samples vs fit_times
    axes[1].grid()
    axes[1].plot(train_sizes, fit_times_mean, 'o-')
    axes[1].fill_between(train_sizes, \
        fit_times_mean - fit_times_std, \

```

```

        fit_times_mean + fit_times_std, alpha=0.1)
axes[1].set_xlabel("Training examples")
axes[1].set_ylabel("fit_times")
axes[1].set_title("Scalability of the model :" + name + " with " + fc)

# Plot fit_time vs score
axes[2].grid()
axes[2].plot(fit_times_mean, test_scores_mean, 'o-')
axes[2].fill_between(fit_times_mean, \
    test_scores_mean - test_scores_std, \
    test_scores_mean + test_scores_std, alpha=0.1)
axes[2].set_xlabel("fit_times")
axes[2].set_ylabel("Score")
axes[2].set_title("Performance of the model: " + name + " with " + fc)

return plt

def plot_real_pred_val(Y_test, ypred, name, fc,ax):
    acc=accuracy_score(Y_test,ypred)
    ax.scatter(range(len(ypred)),ypred,color="blue",\
    lw=5,label="Predicted")
    ax.scatter(range(len(Y_test)), \
        Y_test,color="red",label="Actual")
    ax.set_title("Predicted Values vs True Values of " + name + " with " + fc,
    \
    fontsize=30)
    ax.set_xlabel("Accuracy: " + str(round((acc*100),3)) + "%", fontsize=20)
    ax.set_ylabel("Daily Returns", fontsize=20)
    ax.legend(fontsize=20)
    ax.grid(True, alpha=0.75, lw=1, ls='-.')
    ax.yaxis.set_ticklabels(["", "Negative Daily Returns", "", "", "", "", \
    "Positive Daily Returns"]);
    #plt.show()

def plot_cm(Y_test, ypred, name, fc,ax):
    cm = confusion_matrix(Y_test, ypred)
    sns.heatmap(cm, annot=True, linewidth=0.7, \
        linecolor='red', fmt='g', cmap="Greens", ax=ax)
    ax.set_title(name + ' Confusion Matrix ' + "with " + fc)
    ax.set_xlabel('Y predict')
    ax.set_ylabel('Y test')
    ax.xaxis.set_ticklabels(["Negative Daily Returns", "Positive Daily \
Returns"]);
    ax.yaxis.set_ticklabels(["Negative Daily Returns", "Positive Daily \
Returns"]);
    #plt.show()
    return cm

def plot_decision_boundary(model,xtest, ytest, name,fc,ax):
    #Trains model with two features
    model.fit(xtest, ytest)

```

```

    plot_decision_regions(np.array(xtest), np.array(ytest), clf=model,
legend=2,ax=ax)
    ax.set_title("Decision boundary for " + name + " (Test) " + "with " + fc)
    ax.set_xlabel('Open_Close')
    ax.set_ylabel('High_Low')
    plt.show()

#Plots ROC
def plot_roc(model,X_test, y_test, name, fc,ax):
    Y_pred_prob = model.predict_proba(X_test)
    Y_pred_prob = Y_pred_prob[:, 1]

    fpr, tpr, thresholds = roc_curve(y_test, Y_pred_prob)
    ax.plot([0,1],[0,1], color='navy', lw=5, linestyle='--')
    ax.plot(fpr,tpr, label='ANN')
    ax.set_xlabel('False Positive Rate')
    ax.set_ylabel('True Positive Rate')
    ax.set_title('ROC Curve of ' + name + " with "+fc)
    plt.grid(True)
    #plt.show()

#Chooses two features for decision boundary
X_feature = X.iloc[:, 13:14]
X_train_feat, X_test_feat, y_train_feat, y_test_feat =
train_test_split(X_feature, y, test_size = 0.3, \
random_state = 2021, stratify=y)

def train_model(model, X, y):
    model.fit(X, y)
    return model

def predict_model(model, X, proba=False):
    if ~proba:
        y_pred = model.predict(X)
    else:
        y_pred_proba = model.predict_proba(X)
        y_pred = np.argmax(y_pred_proba, axis=1)

    return y_pred

list_scores = []

def run_model(name, model, X_train, X_test, y_train, y_test, fc, proba=False):
    print(name)
    print(fc)

    model = train_model(model, X_train, y_train)
    y_pred = predict_model(model, X_test, proba)

    accuracy = accuracy_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred, average='weighted')

```

```

precision = precision_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

print('accuracy: ', accuracy)
print('recall: ', recall)
print('precision: ', precision)
print('f1: ', f1)
print(classification_report(y_test, y_pred))
fig, axes = plt.subplots(2,2, figsize=(30,20), facecolor='#fbe7dd')
plot_cm(y_test, y_pred, name, fc, axes[0, 0])
plot_real_pred_val(y_test, y_pred, name, fc, axes[0, 1])
plot_roc(model, X_test, y_test, name, fc, axes[1, 0])
plot_decision_boundary(model, X_test_feat, y_test_feat, name, fc, axes[1, 1])
plot_learning_curve(model, name, X_train, y_train, cv=3, fc=fc);
plt.show()

list_scores.append({'Model Name': name, 'Feature Scaling':fc, 'Accuracy':accuracy, 'Recall': recall, 'Precision': precision, 'F1':f1})

feature_scaling = {
    'Robust Scaler':(X_train_rob, X_test_rob, y_train_rob, y_test_rob),
    'MinMax Scaler':(X_train_norm, X_test_norm, y_train_norm, y_test_norm),
    'Standard Scaler':(X_train_stand, X_test_stand, y_train_stand,
y_test_stand),
}

model_svc = SVC(random_state=2021, probability=True)
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value
    run_model('SVC', model_svc, X_train, X_test, y_train, y_test, fc_name)

logreg = LogisticRegression(solver='lbfgs', max_iter=5000, random_state=2021)
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value
    run_model('Logistic Regression', logreg, X_train, \
X_test, y_train, y_test, fc_name, proba=True)

for fc_name, value in feature_scaling.items():
    scores_1 = []
    X_train, X_test, y_train, y_test = value

    for i in range(2,10):
        knn = KNeighborsClassifier(n_neighbors = i)
        knn.fit(X_train, y_train)

        scores_1.append(accuracy_score(y_test, \
knn.predict(X_test)))

    max_val = max(scores_1)
    max_index = np.argmax(scores_1) + 2

    knn = KNeighborsClassifier(n_neighbors = max_index)

```

```
knn.fit(X_train, y_train)

run_model(f'KNeighbors Classifier n_neighbors = {max_index}', \
          knn, X_train, X_test, y_train, y_test, \
          fc_name, proba=True)

for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value

    dt_cla = DecisionTreeClassifier()

    parameters = {'max_depth':np.arange(1,51,1), \
                  'random_state':[2021]}
    searcher = GridSearchCV(dt_cla, parameters)

    run_model('DecisionTree Classifier', searcher, X_train, \
              X_test, y_train, y_test, fc_name, proba=True)

rf = RandomForestClassifier(n_estimators=200, \
                           max_depth=50, random_state=2021)

for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value
    run_model('RandomForest Classifier', rf, X_train, \
              X_test, y_train, y_test, fc_name, proba=True)

gbt = GradientBoostingClassifier(n_estimators = 200, max_depth=20,
                                 subsample=0.8, max_features=0.2, random_state=2021)
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value
    run_model('GradientBoosting Classifier', gbt, X_train, X_test, y_train,
              y_test, fc_name, proba=True)

for fc_name, value in feature_scaling.items():
```

```

X_train, X_test, y_train, y_test = value
xgb=XGBClassifier(n_estimators = 200, max_depth=20, random_state=2021,
use_label_encoder=False, eval_metric='mlogloss')
run_model('XGBoost Classifier', xgb, X_train, X_test, y_train, y_test,
fc_name, proba=True)

mlp = MLPClassifier(random_state=2021)
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value
    run_model('MLP Classifier', mlp, X_train, X_test, y_train, y_test,
fc_name, proba=True)

lgbm = LGBMClassifier(max_depth = 20, n_estimators=500, subsample=0.8,
random_state=2021)
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value
    run_model('Lightgbm Classifier', lgbm, X_train, X_test, y_train,
y_test, fc_name, proba=True)

gm = GaussianMixture(n_components=2, random_state=100)
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value
    run_model('Gaussian Mixture Classifier', gm, X_train, X_test, y_train,
y_test, fc_name, proba=True)

extra = ExtraTreesClassifier(n_estimators=200, random_state=100)
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value
    run_model('Extra Trees Classifier', extra, X_train, X_test, y_train,
y_test, fc_name, proba=True)

```

CREATING GRAPHICAL USER INTERFACE USING

PYQT5

**CREATING
GRAPHICAL USER INTERFACE
USING
PYQT5**

DESCRIPTION
DESCRIPTION

PyQt5 is a powerful Python library used for creating graphical user interfaces (GUIs). It provides a comprehensive set of tools and functionalities to develop interactive and visually appealing desktop applications. Here are five paragraphs describing the use of PyQt5 to create Python GUIs.

1. **PyQt5 Integration:** PyQt5 allows seamless integration of Qt, a popular cross-platform application framework, with Python. It provides a Pythonic API for accessing Qt's rich set of GUI components, including buttons, labels, text fields, tables, and more. Developers can leverage the full power of Qt's extensive functionality while utilizing the simplicity and flexibility of Python programming.
2. **Drag-and-Drop Design:** PyQt5 provides a visual design tool called Qt Designer, which enables developers to create GUI layouts using a drag-and-drop interface. Qt Designer allows you to design the UI components, set their properties, and arrange them on the canvas. It generates an XML file (UI file) that can be loaded into Python code using the PyQt5 library. This separation of UI design and logic implementation promotes clean code architecture and enhances collaboration between designers and developers.
3. **Event-Driven Programming:** PyQt5 follows an event-driven programming paradigm, where user actions or system events trigger specific actions or behaviors in the application. Developers can define event handlers or callbacks to respond to events like button clicks, mouse movements, keyboard input, and more. This enables the creation of dynamic and responsive user interfaces, where the application reacts to user interactions in real-time.
4. **Customization and Theming:** PyQt5 offers extensive customization options to tailor the appearance and behavior of GUI components. It provides a rich set of style sheets that allow developers to define custom colors, fonts, sizes, and other visual properties. This enables the creation of visually consistent and branded interfaces. Additionally, PyQt5 supports theming, allowing developers to apply pre-defined or custom themes to their applications, giving them a polished and professional look.

5. Cross-Platform Compatibility: One of the key advantages of PyQt5 is its cross-platform compatibility. Qt, the underlying framework, is designed to work seamlessly on multiple platforms, including Windows, macOS, Linux, and even mobile platforms like Android and iOS. This allows developers to write GUI applications once using PyQt5 and deploy them on various operating systems without significant modifications. This cross-platform capability simplifies the development process, reduces time and effort, and expands the reach of your applications.

In conclusion, PyQt5 is a versatile and feature-rich library for creating Python GUI applications. With its integration with Qt, drag-and-drop design capabilities, event-driven programming model, customization options, and cross-platform compatibility, it provides a robust framework for building interactive and visually appealing desktop applications with ease. Whether you are developing simple utilities or complex enterprise-grade software, PyQt5 empowers you to create sophisticated and user-friendly graphical interfaces.

DESIGNING GUI

DESIGNING GUI

Step 1	Now, you will create a GUI to implement how to perform Amazon stock price forecasting using some machine learning algorithms. Open Qt Designer and choose Main Window template. Save the form as gui_amazon.ui .
Step 2	Put two Push Button widgets onto form. Set their text property as LOAD DATA and TRAIN ML MODEL . Set their objectName property as pbLoad and pbTrainML .
Step 3	Put two Table Widget onto form. Set their objectName properties as twData1 and twData2 .
Step 4	Add two Label widgets onto form. Set their text properties as Label 1 and Label 2 and set their objectName properties as label1 and label2 .
	Put three Widget from Containers panel onto form and set their ObjectName

Step 5 property as **widgetPlot1**, **widgetPlot2**, and **widgetPlot3**.

5

Step 6 Right click on the three **Widgets** and choose **Promote to** Set **Promoted class name** as **plot_class**. Click **Add** and **Promote** button. In **Object Inspector** window, you can see that **widgetPlot1**, **widgetPlot2**, and **widgetPlot3** are now an object of **plot_class** as shown in Figure 4.1.

Figure 4.1 The **widgetPlot1**, **widgetPlot2**, and **widgetPlot3** are now objects of **plot_class**

Step 7 Write the definition of **plot_class** and save it as **plot_class.py** as follows:

```
1 #plot_class.py
2 from PyQt5.QtWidgets import*
3 from matplotlib.backends.backend_qt5agg import FigureCanvas
4 from matplotlib.figure import Figure
5
6 class plot_class(QWidget):
7     def __init__(self, parent = None):
8         QWidget.__init__(self, parent)
9         self.canvas = FigureCanvas(Figure())
10
11         vertical_layout = QVBoxLayout()
12         vertical_layout.addWidget(self.canvas)
13
14         self.canvas.axis1 =
15         self.canvas.figure.add_subplot(111)
16         self.canvas.axis1 =
17         self.canvas.figure.subplots_adjust(
18                         top=0.936,
19                         bottom=0.104,
20                         left=0.047,
21                         right=0.981,
22                         hspace=0.2,
23                         wspace=0.2
24
25
26         self.canvas.figure.set_facecolor("xkcd:dark cream")
27         self.setLayout(vertical_layout)
```

Here is a step-by-step explanation of the code:

1. Import the necessary modules:

- **PyQt5.QtWidgets**: This module provides the necessary classes for creating a graphical user interface using PyQt5.
- **matplotlib.backends.backend_qt5agg.FigureCanvas**: This module provides a canvas widget that can be used to display matplotlib figures in a PyQt5 application.
- **matplotlib.figure.Figure**: This module represents a figure in matplotlib, which can contain one or more subplots.

2. Define the **plot_class** class:

The `plot_class` class is defined as a subclass of `QWidget`, which is a base class for all GUI objects in PyQt5.

3. Initialize the `plot_class`:

- The `__init__()` method is called when an instance of the class is created.
- Inside the `__init__()` method, the superclass `QWidget` is initialized using `QWidget.__init__(self, parent)`.
- Create a `FigureCanvas` object named `canvas` using an empty Figure.
- Create a `QVBoxLayout` object named `vertical_layout`.
- Add the `canvas` widget to the `vertical_layout` using `vertical_layout.addWidget(self.canvas)`.

4. Add a subplot to the `canvas` widget:

- Access the `axis1` attribute of the `canvas` object and assign it a subplot using `self.canvas.axis1 = self.canvas.figure.add_subplot(111)`.
- Adjust the subplot parameters using `self.canvas.figure.subplots_adjust()` to set the spacing and margins of the subplot.

5. Customize the appearance of the plot:

Set the `facecolor` of the figure to "xkcd:dark cream" using `self.canvas.figure.set_facecolor("xkcd:dark cream")`.

6. Set the layout of the `plot_class` widget:

Set the layout of the `plot_class` widget to the `vertical_layout` using `self.setLayout(vertical_layout)`.

These steps define a PyQt5 widget (`plot_class`) that contains a matplotlib plot displayed using a `FigureCanvas`. The widget can be added to a PyQt5 application to display the plot.

Step 8	Add two Combo Box widgets and set its <code>objectName</code> property as <code>cbData</code> and <code>cbForecasting</code> . Let them empty. You will populate them from the code.
Step 9	Add another Combo Box widget and set its <code>objectName</code> property as <code>cbClassifier</code> . Populate this widget with thirteen items as shown in Figure 4.2. Figure 4.2 Populating <code>cbClassifier</code> widget with thirteen items
Step 10	Add three Radio Button and set their <code>objectName</code> properties as <code>rbRobust</code> , <code>rbMinMax</code> , and <code>rbStandard</code> . Set their <code>text</code> properties as Robust Scaler , MinMax Scaler , and Standard Scaler .

Step Write this Python script and save it as **gui_amazon.py**:

11

```
1 #gui_amazon.py
2 from PyQt5.QtWidgets import *
3 from PyQt5.uic import loadUi
4 from matplotlib.backends.backend_qt5agg import
5 (NavigationToolbarQT as NavigationToolbar)
6 from matplotlib.colors import ListedColormap
7
8 class DemoGUI_Amazon(QMainWindow):
9     def __init__(self):
10         QMainWindow.__init__(self)
11         loadUi("gui_amazon.ui", self)
12         self.setWindowTitle(
13             "GUI Demo of Forecasting Amazon Stock Price with
14 Machine Learning")
15         self.addToolBar(NavigationToolbar(
16             self.widgetPlot1.canvas, self))
17
18 if __name__ == '__main__':
19     import sys
20     app = QApplication(sys.argv)
21     ex = DemoGUI_Amazon()
22     ex.show()
23     sys.exit(app.exec_())
```

Here is a step-by-step explanation of the code:

1. Import the necessary modules:

- **PyQt5.QtWidgets**: This module provides the necessary classes for creating a graphical user interface using PyQt5.
- **PyQt5.uic.loadUi**: This function is used to load the UI file created with Qt Designer.
- **matplotlib.backends.backend_qt5agg.NavigationToolbar2QT**: This module provides a navigation toolbar for matplotlib figures in a PyQt5 application.
- **matplotlib.colors.ListedColormap**: This module provides functionality for creating a colormap.

2. Define the **DemoGUI_Amazon** class:

The **DemoGUI_Amazon** class is defined as a subclass of **QMainWindow**, which is a main application window in PyQt5.

3. Initialize the **DemoGUI_Amazon**:

- The **__init__()** method is called when an instance of the class is created.
- Inside the **__init__()** method, the superclass **QMainWindow** is initialized using **QMainWindow.__init__(self)**.
- Load the UI file "gui_amazon.ui" using **loadUi("gui_amazon.ui", self)**. This loads the UI created with Qt Designer and sets up the

user interface.

- Set the window title using `self.setWindowTitle("GUI Demo of Forecasting Amazon Stock Price with Machine Learning")`.
- Add a navigation toolbar to the application using `self.addToolBar(NavigationToolbar(self.widgetPlot1.canvas, self))`. This adds a toolbar to the main window that allows interaction with the plot.

4. Run the application:

- The if `__name__ == '__main__'`: block ensures that the code is only executed if the script is run directly (not imported as a module).
- Create an instance of the **QApplication** class using `app = QApplication(sys.argv)`. This creates the PyQt5 application.
- Create an instance of the **DemoGUI_Amazon** class using `ex = DemoGUI_Amazon()`. This creates the main window of the application.
- Show the main window using `ex.show()`.
- Start the event loop of the application using `sys.exit(app.exec_())`. This starts the PyQt5 event loop and allows the application to respond to user interactions.

These steps define a PyQt5 application that creates a main window (**DemoGUI_Amazon**) and loads a UI file created with Qt Designer. The application adds a navigation toolbar to the main window, allowing interaction with a plot. Running the script starts the application and displays the main window.

Step

12

Run `gui_amazon.py`. You will see form's layout as shown in Figure 4.3.

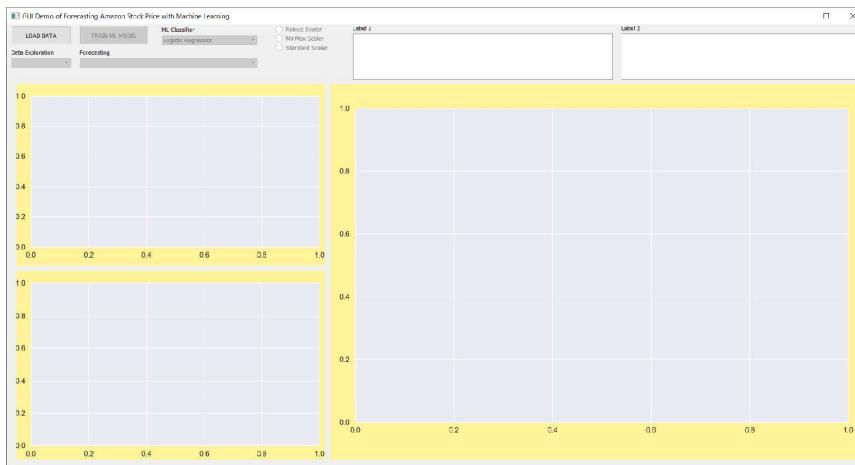


Figure 4.3 The form when it first runs

Step Import all necessary modules:

13

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib
4 import matplotlib.pyplot as plt
5 import seaborn as sns
6 import datetime as dt
7 sns.set_style('darkgrid')
8 from sklearn.preprocessing import LabelEncoder
9 import warnings
10 warnings.filterwarnings('ignore')
11 import os
12 import plotly.graph_objs as go
13 import joblib
14 import itertools
15 from sklearn.metrics import roc_auc_score,roc_curve,
16 explained_variance_score, r2_score
17 from sklearn.model_selection import cross_val_score
18 from statsmodels.tsa.seasonal import seasonal_decompose as
19 season
20 from sklearn.metrics import mean_squared_error,
21 mean_absolute_error
22 from sklearn.metrics import accuracy_score,
23 balanced_accuracy_score
24 from sklearn.model_selection import train_test_split,
25 RandomizedSearchCV, GridSearchCV,StratifiedKFold
26 from sklearn.preprocessing import StandardScaler,
27 MinMaxScaler, RobustScaler
28 from sklearn.linear_model import LogisticRegression
29 from sklearn.naive_bayes import GaussianNB
30 from sklearn.tree import DecisionTreeClassifier
31 from sklearn.svm import SVC
32 from sklearn.ensemble import RandomForestClassifier,
33 ExtraTreesClassifier
34 from sklearn.neighbors import KNeighborsClassifier
35 from sklearn.ensemble import AdaBoostClassifier,
36 GradientBoostingClassifier
37 from xgboost import XGBClassifier
38 from sklearn.neural_network import MLPClassifier
39 from sklearn.linear_model import SGDClassifier
40 from sklearn.preprocessing import StandardScaler,
41 LabelEncoder, OneHotEncoder
42 from sklearn.metrics import confusion_matrix,
43 accuracy_score, recall_score, precision_score
44 from sklearn.metrics import classification_report, f1_score,
45 plot_confusion_matrix
46 from catboost import CatBoostClassifier
47 from lightgbm import LGBMClassifier
48 from imblearn.over_sampling import SMOTE
49 from sklearn.model_selection import learning_curve
50 from mlxtend.plotting import plot_decision_regions
51 from sklearn.decomposition import PCA
52 from sklearn.linear_model import LinearRegression
53 from sklearn.ensemble import RandomForestRegressor
54 from sklearn.tree import DecisionTreeRegressor
55 from sklearn.svm import SVR
56 from sklearn.pipeline import make_pipeline
```

```

57 from sklearn.naive_bayes import GaussianNB
58 from sklearn.neighbors import KNeighborsRegressor
59 from sklearn.ensemble import AdaBoostRegressor
60 from sklearn.ensemble import GradientBoostingRegressor
61 from xgboost import XGBRegressor
62 from lightgbm import LGBMRegressor
63 from catboost import CatBoostRegressor
64 from sklearn.neural_network import MLPRegressor
65 from statsmodels.tsa.holtwinters import ExponentialSmoothing
66 from statsmodels.tsa.stattools import adfuller, acf, pacf
from statsmodels.tsa.arima_model import ARIMA
from pmdarima.utils import decomposed_plot
from pmdarima.arima import decompose
from pmdarima import auto_arima
from sklearn.linear_model import LassoCV
from sklearn.linear_model import RidgeCV
from sklearn.mixture import GaussianMixture

```

DEFINING HELPER FUNCTIONS

DEFINING HELPER FUNCTIONS

Step 1 Define **write_df_to_qtable()** and **populate_table()** methods to populate any table widget with some data:

```

1 # Takes a df and writes it to a qtable provided. df headers
2 become qtable headers
3 @staticmethod
4 def write_df_to_qtable(df,table):
5     headers = list(df)
6     table.setRowCount(df.shape[0])
7     table.setColumnCount(df.shape[1])
8     table.setHorizontalHeaderLabels(headers)
9
10    # getting data from df is computationally costly so
11    # convert it to array first
12    df_array = df.values
13    for row in range(df.shape[0]):
14        for col in range(df.shape[1]):
15            table.setItem(row, col,
16            QTableWidgetItem(str(df_array[row,col])))
17
18
19 def populate_table(self,data, table):
20     #Populates two tables
21     self.write_df_to_qtable(data,table)

    table.setAlternatingRowColors(True)
    table.setStyleSheet("alternate-background-color:
#ffb07c;background-color: #e6daa6;");

```

Here is a step-by-step explanation of each function:

1. **write_df_to_qtable(df, table)** function:
 - a. This is a static method that takes a **DataFrame** (**df**) and a **QTableWidget** (**table**) as input.
 - b. Retrieve the headers of the DataFrame using **list(df)** and store them in the **headers** variable.
 - c. Set the number of rows in the **QTableWidget** to the number of rows in the DataFrame using **table.setRowCount(df.shape[0])**.
 - d. Set the number of columns in the **QTableWidget** to the number of columns in the DataFrame using **table.setColumnCount(df.shape[1])**.
 - e. Set the horizontal header labels of the QTableWidget to the DataFrame headers using **table.setHorizontalHeaderLabels(headers)**.
 - f. Convert the DataFrame to a NumPy array using **df.values** and store it in the **df_array** variable.
 - g. Iterate over each row and column of the **DataFrame** using nested for loops.
 - h. Create a **QTableWidgetItem** using **QTableWidgetItem(str(df_array[row, col]))**.
 - i. Set the **QTableWidgetItem** as the item in the corresponding row and column of the **QTableWidget** using **table.setItem(row, col, QTableWidgetItem)**.
2. **populate_table(self, data, table)** function:
 - a. This method is a member function of a class.
 - b. It takes self (the instance of the class), data (a **DataFrame**), and table (a **QTableWidget**) as input.
 - c. Calls the **write_df_to_qtable()** method with data and table as arguments to populate the table with the **DataFrame**.
 - d. Set the **alternatingRowColors** property of the **QTableWidget** to **True** using

- table.setAlternatingRowColors(True).**
- e. Set the stylesheet of the **QTableWidget** to define the alternating background colors using **table.setStyleSheet("alternate-background-color: #ffb07c;background-color: #e6daa6;")**.

These functions are used to write the data from a **DataFrame** to a **QTableWidget** in PyQt5. The **write_df_to_qtable()** function sets up the table structure and populates it with data from the **DataFrame**, while the **populate_table()** function calls **write_df_to_qtable()** and applies additional styling to the table.

Step 2 Define **initial_state()** method to disable some widgets when form initially runs:

```

1 def initial_state(self, state):
2     self.pbTrainML.setEnabled(state)
3     self.cbData.setEnabled(state)
4     self.cbForecasting.setEnabled(state)
5     self.cbClassifier.setEnabled(state)
6     self.rbRobust.setEnabled(state)
7     self.rbMinMax.setEnabled(state)
8     self.rbStandard.setEnabled(state)

```

Here's the explanation of the **initial_state()** method:

1. **self.pbTrainML.setEnabled(state):**
 - **self.pbTrainML** refers to a widget, a QPushButton.
 - The **setEnabled()** method is a PyQt5 function that enables or disables the widget based on the value of state.
 - If state is **True**, the widget (**pbTrainML**) will be enabled, meaning it can be interacted with by the user.
 - If state is **False**, the widget will be disabled, preventing user interaction with it.
2. **self.cbData.setEnabled(state):**
 - **self.cbData** refers to a widget, a QComboBox.
 - Similar to the previous line, **setEnabled()** is used to enable or disable the widget (**cbData**) based on the value of state.

- If state is **True**, the combo box (**cbData**) will be enabled, allowing the user to select options from the dropdown list.
- If state is **False**, the combo box will be disabled, preventing any changes or selections.

3. self.cbForecasting.setEnabled(state):

- **self.cbForecasting** refers to a widget, a **QComboBox**.
- Again, **setEnabled()** is used to enable or disable the widget (**cbForecasting**) based on state.
- If state is **True**, the combo box (**cbForecasting**) will be enabled for user interaction.
- If state is **False**, the combo box will be disabled, preventing any changes or selections.

4. self.cbClassifier.setEnabled(state):

- **self.cbClassifier** refers to a widget, a **QComboBox**.
- **setEnabled()** is used to enable or disable the widget (**cbClassifier**) based on state.
- If state is **True**, the combo box (**cbClassifier**) will be enabled and the user can interact with it.
- If state is **False**, the combo box will be disabled, preventing any changes or selections.

5. self.rbRobust.setEnabled(state):

- **self.rbRobust** refers to a widget, a **QRadioButton**.
- **setEnabled()** is used to enable or disable the widget (**rbRobust**) based on state.
- If state is **True**, the radio button (**rbRobust**) will be enabled, allowing the user to select it.
- If state is **False**, the radio button will be disabled, preventing any changes or selections.

6. self.rbMinMax.setEnabled(state):

- **self.rbMinMax** refers to a widget, a **QRadioButton**.
- **setEnabled()** is used to enable or disable the widget (**rbMinMax**) based on state.
- If state is **True**, the radio button (**rbMinMax**) will be enabled and can be selected by the user.
- If state is **False**, the radio button will be disabled, preventing any changes or selections.

7. **self.rbStandard.setEnabled(state):**

- **self.rbStandard** refers to a widget, a **QRadioButton**.
- **setEnabled()** is used to enable or disable the widget (**rbStandard**) based on state.
- If state is **True**, the radio button (**rbStandard**) will be enabled and can be selected by the user.
- If state is **False**, the radio button will be disabled, preventing any changes or selections.

The purpose of the `initial_state` method is to control the enabled/disabled state of various widgets based on the value of state. By enabling or disabling these widgets, you can control their interactivity and allow or restrict user input. For example, if state is **True**, the widgets will be enabled, indicating that the user can interact with them. If state is **False**, the widgets will be disabled, meaning the user cannot interact with them.

Step 3 Define **compute_year_month_wise()** method to compute year-wise and month-wise data:

```

1 def compute_year_month_wise(self, df):
2     cols = list(df.columns)
3     cols.remove("Month")
4     cols.remove("Day")
5     cols.remove("Week")
6     cols.remove("Year")
7     cols.remove("Quarter")
8
9
10    #Resamples the data year-wise by mean
11    year_data_mean = df[cols].resample('y').mean()
12
13    #Resamples the data year-wise by ewm

```

```

13     year_data_ewm=year_data_mean.ewm(span=5).mean()
14
15     #Resamples the data month-wise by mean
16     monthly_data_mean = df[cols].resample('m').mean()
17
18
19     #Resamples the data month-wise by EWM
20     monthly_data_ewm=monthly_data_mean.ewm(span=5).mean()
21
22     return year_data_mean, year_data_ewm, monthly_data_mean,
23         monthly_data_ewm

```

Here's a step-by-step explanation of the **compute_year_month_wise(self, df)** function:

1. This method takes **self** (the instance of the class) and a **DataFrame df** as input.

2. **cols = list(df.columns)**:

Retrieves the column names of the **DataFrame df** and stores them in the **cols** list.

3. Removing specific columns from cols:

- The following columns are removed from the cols list: "Month", "Day", "Week", "Year", and "Quarter".
- This is done using the remove method of the list, which removes the specified element.

4. **year_data_mean = df[cols].resample('y').mean()**:

Creates a new **DataFrame year_data_mean** by resampling the selected columns of the original DataFrame df year-wise and computing the mean value for each year.

The 'y' argument in resample indicates that the data is resampled on a yearly basis.

5. **year_data_ewm** =

year_data_mean.ewm(span=5).mean():

- Computes an exponentially weighted moving average (EWMA) of the **year_data_mean** DataFrame using a span of 5.
- The ewm function calculates the EWMA, and mean computes the mean of the EWMA values.

6. **monthly_data_mean** =

df[cols].resample('m').mean():

- Creates a new DataFrame **monthly_data_mean** by resampling the

	<p>selected columns of the original DataFrame df month-wise and computing the mean value for each month.</p> <ul style="list-style-type: none"> The 'm' argument in resample indicates that the data is resampled on a monthly basis. <p>7. monthly_data_ewm = monthly_data_mean.ewm(span=5).mean(): Computes the EWMA of the monthly_data_mean DataFrame using a span of 5, similar to the previous step.</p> <p>8. Returns the computed data:</p> <ul style="list-style-type: none"> The function returns four values: year_data_mean, year_data_ewm, monthly_data_mean, and monthly_data_ewm. These represent the resampled data on a year-wise basis (mean and EWMA) and a month-wise basis (mean and EWMA), respectively. <p>The purpose of the compute_year_month_wise() method is to perform resampling and calculations on the input DataFrame df to obtain year-wise and month-wise data representations. It returns these computed values for further analysis or processing.</p>
Step 4	<p>Define create_new_dfs() method to extract day, month, week, quarter, and year, set Date column as index, create a dummy dataframe for visualization, compute year-wise and month-wise data, and convert days, months, and quarters from numerics to meaningful string:</p> <pre> 1 def create_new_dfs(self,df): 2 #Extracts day, month, quarter, and year 3 df['Date'] = pd.to_datetime(df['Date']) 4 df['Day'] = df['Date'].dt.weekday 5 df['Month'] = df['Date'].dt.month 6 df['Year'] = df['Date'].dt.year 7 df['Week']= df['Date'].dt.week 8 df['Quarter']= df['Date'].dt.quarter 9 10 11 #Sets Date column as index 12 df = df.set_index("Date") 13 14 15 #Creates a dummy dataframe for visualization 16 df_dummy=df.copy() </pre>

```

16
17     #Computes year-wise and month-wise data
18     self.year_data_mean, self.year_data_ewm,
19         self.monthly_data_mean, self.monthly_data_ewm =
20             self.compute_year_month_wise(df_dummy)
21
22     #Converts days, months, and quarters from numerics to
23     meaningful string
24     days = {0:'Sunday',1:'Monday',2:'Tuesday',
25 3:'Wednesday',4:'Thursday',5:'Friday',6:'Saturday'}
26     df_dummy['Day'] = df_dummy['Day'].map(days)
27     months={1:'January',2:'February',3:'March', 4:'April',
28 5:'May', 6:'June', 7:'July', 8:'August', 9:'September',
29 10:'October', 11:'November', 12:'December'}
30     df_dummy['Month']= df_dummy['Month'].map(months)
31     quarters = {1:'Jan-March', 2:'April-June', 3:'July-
Sept', 4:'Oct-Dec'}
32     df_dummy['Quarter'] = df_dummy['Quarter'].map(quarters)
33
34     return df, df_dummy

```

Here's a step-by-step explanation of the function:

1. **create_new_dfs(self, df):**

This method takes **self** (the instance of the class) and a **DataFrame df** as input.

2. Extracting date-related components:

- **df['Date'] = pd.to_datetime(df['Date'])**
converts the 'Date' column of the DataFrame df to a datetime format.
- **df['Day'] = df['Date'].dt.weekday** extracts the day of the week (0-6, where 0 is Monday and 6 is Sunday) from the 'Date' column.
- **df['Month'] = df['Date'].dt.month** extracts the month (1-12) from the 'Date' column.
- **df['Year'] = df['Date'].dt.year** extracts the year from the 'Date' column.
- **df['Week'] = df['Date'].dt.week** extracts the week number (1-53) from the 'Date' column.
- **df['Quarter'] = df['Date'].dt.quarter** extracts the quarter (1-4) from the 'Date' column.

3. Setting the 'Date' column as the index:

df = df.set_index("Date") sets the 'Date' column as the index of the **DataFrame df**.

4. Creating a dummy dataframe for visualization:

df_dummy = df.copy() creates a copy of the **DataFrame df** and assigns it to **df_dummy**.

5. Computing year-wise and month-wise data:
`self.year_data_mean, self.year_data_ewm, self.monthly_data_mean,
self.monthly_data_ewm = self.compute_year_month_wise(df_dummy)`
calls the **compute_year_month_wise()** method with the **df_dummy DataFrame** and assigns the computed results to corresponding variables.

6. Converting numeric values to meaningful strings:
- days, months, and quarters are dictionaries that map numeric values to corresponding string representations.
 - **df_dummy['Day'] = df_dummy['Day'].map(days)** maps the numeric day values to their corresponding string names.
 - **df_dummy['Month'] = df_dummy['Month'].map(months)** maps the numeric month values to their corresponding string names.
 - **df_dummy['Quarter'] = df_dummy['Quarter'].map(quarters)** maps the numeric quarter values to their corresponding string names.

7. Returning the original and dummy DataFrames:
- The function returns two DataFrames: **df** and **df_dummy**.
 - **df** represents the original DataFrame with the added date-related columns and the 'Date' column set as the index.
 - **df_dummy** is a copy of **df** with additional columns converted to meaningful string representations for visualization purposes.

The purpose of the **create_new_dfs** method is to perform data transformations and create two **DataFrames**: one for analysis (**df**) and another for visualization (**df_dummy**). The method adds date-related columns, sets the index, computes year-wise and month-wise data, and converts numeric values to meaningful strings for visualization purposes.

Step 5	Compute technical indicators: daily returns, moving average convergence-divergence (MACD), relative strength index
--------	--

(RSI), Simple Moving Average (SMA), lower and upper bands, standard deviation. Each function had been explained before.

```
1 def compute_daily_returns(self, df):
2     """Compute and return the daily return values."""
3     # TODO: Your code here
4     # Note: Returned DataFrame must have the same number of
5     # rows
6     daily_return = (df / df.shift(1)) - 1
7     daily_return[0] = 0
8     return daily_return
9
10 def calculate_SMA(self, df, peroids=15):
11     SMA = df.rolling(window=peroids, min_periods=peroids,
12                      center=False).mean()
13     return SMA
14
15 def calculate_MACD(self, df, nslow=26, nfast=12):
16     emaslow = df.ewm(span=nslow, min_periods=nslow,
17                       adjust=True, ignore_na=False).mean()
18     emafast = df.ewm(span=nfast, min_periods=nfast,
19                       adjust=True, ignore_na=False).mean()
20     dif = emafast - emaslow
21     MACD = dif.ewm(span=9, min_periods=9, adjust=True,
22                     ignore_na=False).mean()
23     return dif, MACD
24
25
26 def calculate_RSI(self, df, periods=14):
27     # wilder's RSI
28     delta = df.diff()
29     up, down = delta.copy(), delta.copy()
30
31     up[up < 0] = 0
32     down[down > 0] = 0
33
34
35     rUp = up.ewm(com=periods, adjust=False).mean()
36     rDown = down.ewm(com=periods, adjust=False).mean().abs()
37
38     rsi = 100 - 100 / (1 + rUp / rDown)
39     return rsi
40
41
42 def calculate_BB(self, df, peroids=15):
43     STD = df.rolling(window=peroids, min_periods=peroids,
44                      center=False).std()
45     SMA = self.calculate_SMA(df)
46     upper_band = SMA + (2 * STD)
47     lower_band = SMA - (2 * STD)
48     return upper_band, lower_band
49
50
51 def calculate_stdev(self, df, periods=5):
52     STDEV = df.rolling(periods).std()
53     return STDEV
54
55 def compute_technical_indicators(self, df):
    stock_close=df["Adj Close"]
```

```

56 daily_returns=self.compute_daily_returns(stock_close)
57 SMA_CLOSE = self.calculate_SMA(stock_close)
58 upper_band, lower_band = self.calculate_BB(stock_close)
59 DIF, MACD = self.calculate_MACD(stock_close)
60 RSI = self.calculate_RSI(stock_close)
61 STDEV= self.calculate_stdev(stock_close)
62 Open_Close=df.Open - df["Adj Close"]
63 High_Low=df.High-df.Low
64
65 df['daily_returns']= daily_returns
66 df['SMA']= SMA_CLOSE
67 df['Upper_band']= upper_band
68 df['Lower_band']= lower_band
69 df['DIF']= DIF
70 df['MACD']= MACD
71 df['RSI']= RSI
72 df['STDEV']= STDEV
73 df['Open_Close']=Open_Close
74 df['High_Low']=High_Low
75
76 #Checks null values because of technical indicators
77 print(df.isnull().sum().to_string())
78 print('Total number of null values: ',
79 df.isnull().sum().sum())
80
81 #Fills each null value in every column with mean value
82 cols = list(df.columns)
83 for n in cols:
84     df[n].fillna(df[n].mean(),inplace = True)
85
86 #Checks again null values
87 print(df.isnull().sum().to_string())
88 print('Total number of null values: ',
89 df.isnull().sum().sum())

```

POPULATING COMBOBOXES AND READING DATASET

POPULATING COMBOBOXES AND READING DATASET

Step 1 Define **populate_cbData()** and **populate_cbForecasting()** methods to populate **cbData** and **cbForecasting** widgets:

```

1 def populate_cbForecasting(self):
2     self.cbForecasting.addItems(["Linear Regression",
3         "Random Forest Regression"])
4     self.cbForecasting.addItems(["Decision Tree Regression",
5         "KNN Regression"])
6     self.cbForecasting.addItems(["Adaboost Regression",
7         "Gradient Boosting Regression"])
8     self.cbForecasting.addItems(["XGB Regression",
9         "LGBM Regression"])

```

```

10     self.cbForecasting.addItems(["Catboost Regression",
11         "SVR Regression"])
12     self.cbForecasting.addItems(["MLP Regression",
13         "Lasso Regression", "Ridge Regression"])
14
15     def populate_cbData(self):
16         self.cbData.addItems(["Case Distribution"])
17         self.cbData.addItems(["High", "Low", "Open", "Close"])
18         self.cbData.addItems(["Adj Close", "Volume", "Technical
19 Indicators"])
20         self.cbData.addItems(["Year-Wise", "Month-Wise"])

```

The `populate_cbForecasting()` and `populate_cbData()` methods are used to populate the items in a combo box (`QComboBox`) widget. Here's a step-by-step explanation of each method:

1. **`populate_cbForecasting(self)`** function:

- This method populates the items in the `cbForecasting` combo box.
- Adding items to `cbForecasting`:
 - a. `self.cbForecasting.addItem(["Linear Regression", "Random Forest Regression"])` adds the items "Linear Regression" and "Random Forest Regression" to the combo box.
 - b. Similarly, multiple `self.cbForecasting.addItem()` calls add additional items to the combo box, each call adding a list of items.
- The items are grouped into different calls:
 - a. The items are grouped into separate calls to `self.cbForecasting.addItem()` to organize them for better readability.
 - b. Each call adds a set of related items, such as different regression algorithms.

2. **`populate_cbData(self)`** function:

- This method populates the items in the `cbData` combo box.
- Adding items to `cbData`:
 - a. `self.cbData.addItem(["Case Distribution"])` adds the item "Case Distribution" to the combo box.

- b. Similarly, multiple calls to `self.cbData.addItems()` add additional items to the combo box, each call adding a list of items.
- The items are grouped into different calls:
 - The items are grouped into separate calls to `self.cbData.addItems()` to organize them for better readability.
 - Each call adds a set of related items, such as different data options.

The purpose of these methods is to populate the combo boxes with relevant items for the user to select from. The `populate_cbForecasting()` method adds items related to different regression algorithms, while the `populate_cbData()` method adds items related to different data options. This allows the user to choose the desired regression algorithm and data option from the respective combo boxes in the graphical user interface (GUI).

Step 2 Define `read_dataset()` method to read dataset, create two dataframes, and compute technical indicators:

```

1 def read_dataset(self, dir):
2     #Reads dataset
3     df = pd.read_csv(dir)
4
5     #Creates new Dataframes
6     self.df, self.df_dummy=self.create_new_dfs(df)
7
8     #Computes technical indicators
9     self.compute_technical_indicators(self.df)

```

Here's a step-by-step explanation of the function:

1. `read_dataset(self, dir)` function:

This method takes `self` (the instance of the class) and a directory `dir` as input.

2. Reading the dataset:

- `df = pd.read_csv(dir)` reads the dataset from the specified directory using `pd.read_csv()`.
- The dataset is stored in the **DataFrame** `df`.

3. Creating new DataFrames:

- `self.df, self.df_dummy = self.create_new_dfs(df)` calls the

- create_new_dfs()** method with the **df DataFrame** as input.
- This method creates new DataFrames **df** and **df_dummy** with additional columns and index settings based on the original df.
4. Computing technical indicators:
- self.compute_technical_indicators(self.df)** calls the **compute_technical_indicators()** method with the **df** DataFrame as input.
 - This method performs computations to derive technical indicators based on the input DataFrame **df**.

The purpose of the **read_dataset()** method is to read a dataset from a specified directory, create new DataFrames with additional columns and index settings, and compute technical indicators based on the dataset. These steps prepare the data for further analysis or visualization within the class or application.

Step 3 Define **import_dataset()** method to import datasets and populate two table widgets:

```

1  def import_dataset(self):
2      curr_path = os.getcwd()
3      dataset_dir = curr_path + "/Amazon.csv"
4
5      self.read_dataset(dataset_dir)
6      print("Dataframe has been read...")
7
8      #Populates cbData and cbForecasting
9      self.populate_cbData()
10     self.populate_cbForecasting()
11
12     #Populates tables with data
13     self.populate_table(self.df, self.twData1)
14     self.label1.setText('Data for Forecasting')
15
16     self.populate_table(self.df_dummy, self.twData2)
17     self.label2.setText('Data for Visualization')
18
19     #Turns off pbLoad
20     self.pbLoad.setEnabled(False)
21
22     #Turns on cbForecasting and cbData
23     self.cbForecasting.setEnabled(True)
24     self.cbData.setEnabled(True)

```

Here's a step-by-step explanation of the function:

1. **import_dataset(self):**

This method takes **self** (the instance of the class) as input.

2. Getting the current path and dataset directory:

- **curr_path = os.getcwd()** retrieves the current working directory using the **os.getcwd()** function.
- **dataset_dir = curr_path + "/Amazon.csv"** constructs the full path of the dataset file by appending "/Amazon.csv" to the current path.

3. Reading the dataset and printing a message:

- **self.read_dataset(dataset_dir)** calls the **read_dataset()** method with the **dataset_dir** as the directory of the dataset to be read.
- The dataset is read and processed, and the resulting DataFrames are stored in **self.df** and **self.df_dummy**.
- **print("Dataframe has been read...")** prints the message "Dataframe has been read..." to the console.

4. Populating combo boxes:

- **self.populate_cbData()** calls the **populate_cbData()** method to populate the **cbData** combo box with relevant items.
- **self.populate_cbForecasting()** calls the **populate_cbForecasting()** method to populate the **cbForecasting** combo box with relevant items.

5. Populating tables with data:

- **self.populate_table(self.df, self.twData1)** calls the **populate_table()** method to populate the first table (**twData1**) with data from the **self.df DataFrame**.
- **self.label1.setText('Data for Forecasting')** sets the text of **label1** to "Data for Forecasting".
- **self.populate_table(self.df_dummy, self.twData2)** calls the **populate_table()** method to populate the second table

	<p>(twData2) with data from the <code>self.df_dummy</code> DataFrame.</p> <ul style="list-style-type: none"> • <code>self.label2.setText('Data for Visualization')</code> sets the text of <code>label2</code> to "Data for Visualization". <p>6. Enabling and disabling GUI elements:</p> <ul style="list-style-type: none"> • <code>self.pbLoad.setEnabled(False)</code> disables the "Load" button (<code>pbLoad</code>). • <code>self.cbForecasting.setEnabled(True)</code> enables the <code>cbForecasting</code> combo box. • <code>self.cbData.setEnabled(True)</code> enables the <code>cbData</code> combo box. <p>The purpose of the <code>import_dataset()</code> method is to import the dataset, read and process it into DataFrames, populate combo boxes with relevant items, populate tables with data, and enable/disable GUI elements to control the user interface flow.</p>
Step 4	<p>Connect <code>currentIndexChanged()</code> event of <code>cbData</code> widget with <code>import_dataset()</code> method and put it inside <code>initial_state()</code> method as shown in line 9:</p> <pre> 1 def initial_state(self, state): 2 self.pbTrainML.setEnabled(state) 3 self.cbData.setEnabled(state) 4 self.cbForecasting.setEnabled(state) 5 self.cbClassifier.setEnabled(state) 6 self.rbRobust.setEnabled(state) 7 self.rbMinMax.setEnabled(state) 8 self.rbStandard.setEnabled(state) 9 self.pbLoad.clicked.connect(self.import_dataset) </pre>
Step 5	<p>Run <code>gui_amazon.py</code> and click LOAD DATA button. You will see that the two tables have been populated with data. You will see the result as shown in Figure 4.4.</p>

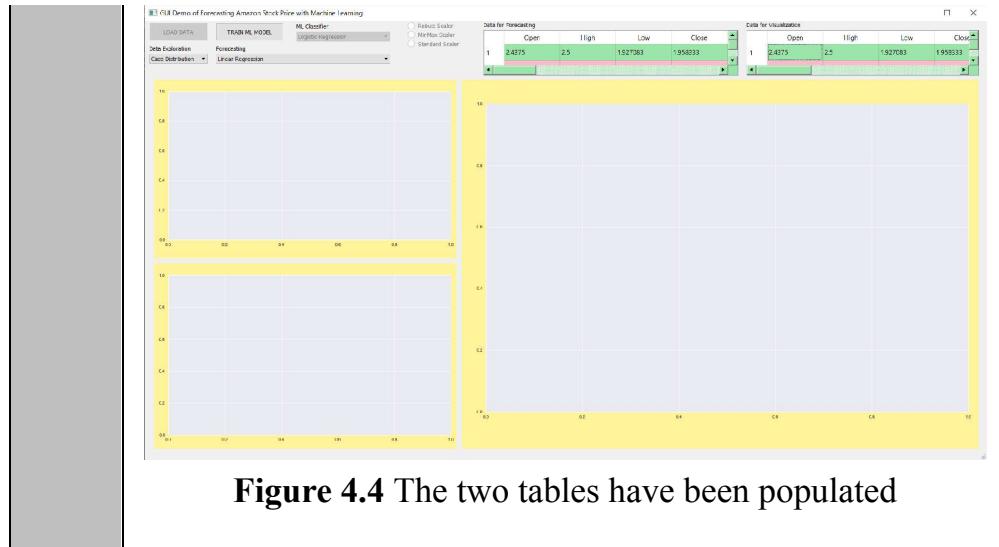


Figure 4.4 The two tables have been populated

PLOTTING CASE DISTRIBUTION

PLOTTING CASE DISTRIBUTION

Step 1 Define **plot_barchart()** and **plot_piechart()** methods to plot case distribution of a categorical feature in pie chart and plot bar:

```
1
2
3
4
5
6
7
8
9
10
11
```

```

12 #Defines function to plot case distribution of a categorical
13 feature bar plot
14 def plot_barchart(self,df,var, widget):
15     ax = df[var].value_counts().plot(kind="barh", ax =
16     widget.canvas.axis1)
17     for i,j in enumerate(df[var].value_counts().values):
18         ax.text(.7,i,j,weight = "bold", fontsize=10)
19
20     widget.canvas.axis1.set_title("Case distribution " + " of
21 " + var + " variable", fontsize=14)
22     widget.canvas.figure.tight_layout()
23     widget.canvas.draw()

#Defines function to plot case distribution of a categorical
feature in pie chart
def plot_piechart(self, df,var, widget):
    label_list = list(df[var].value_counts().index)
    df[var].value_counts().plot.pie(ax =
    widget.canvas.axis1, autopct = "%1.1f%%",
    colors = sns.color_palette("prism", 7),
    startangle = 60, labels=label_list,
    wedgeprops={"linewidth":2,"edgecolor":"k"}, 
    shadow =True, textprops={'fontsize': 10})
    widget.canvas.axis1.set_title("Case distribution " +
    " of " + var + " variable", fontsize=14)
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()

```

These are two functions defined to plot the case distribution of a categorical feature using different types of charts: a bar plot and a pie chart. Let's go through each method step by step:

plot_barchart() function:

1. The function takes four parameters: **self**, **df**, **var**, and **widget**. The **self** parameter suggests that this function is defined within a class, but without additional context, it's difficult to determine its purpose.
 - **df** is the **DataFrame** containing the data.
 - **var** is the name of the categorical feature/column in the DataFrame that we want to visualize.
 - **widget** represents the widget or canvas where the chart will be displayed.
2. The function begins by creating a horizontal bar plot using the **plot()** function from pandas. It uses **df[var].value_counts()** to get the count of each category in the specified variable and then plots it as a bar chart with **kind="barh"** (horizontal bar chart) on the **widget.canvas.axis1**.
3. The for loop is used to add text labels to each bar in the chart. It iterates over the values of **df[var].value_counts().values** (the counts of each category) and places the text at the position (0.7, i) on the chart, where i represents the index of the category.
4. The **widget.canvas.axis1.set_title()** function is used to set the title of the chart, which includes the variable name.
5. The **widget.canvas.figure.tight_layout()** function ensures that the plot fits nicely within the

- canvas.
- Finally, `widget.canvas.draw()` is called to update and display the chart.

`plot_piechart()` function:

- Like the previous function, this function takes four parameters: `self`, `df`, `var`, and `widget`.
- It starts by creating a list of labels (`label_list`) by extracting the index (categories) of the value counts of the specified variable using `df[var].value_counts().index`.
- The pie chart is created using the `plot.pie()` function from pandas. It uses the `value_counts()` of the variable `df[var]` to determine the size of each slice in the pie chart.
- Various parameters are set for the pie chart, including `ax` (the canvas where the chart will be displayed), `autopct` (the format for displaying the percentage values on the slices), `colors` (the color palette for the slices), `startangle` (the angle at which the first slice starts), `labels` (the labels for each slice), `wedgeprops` (properties for the wedge shape of the slices), `shadow` (whether to display a shadow effect), and `textprops` (properties for the text on the slices).
- The title of the chart is set using `widget.canvas.axis1.set_title()`, including the variable name.
- `widget.canvas.figure.tight_layout()` is called to ensure a proper layout.
- Finally, `widget.canvas.draw()` is used to update and display the pie chart.

Both functions provide a visual representation of the case distribution of a categorical feature using either a bar plot or a pie chart. The choice between the two methods depends on the desired visual presentation and the nature of the data.

Step 2

Define `color_month()`, `line_plot_month()`, `sns_plot_month()` methods to plot month-wise data explained before:

```

1 def color_month(self,month):
2     if month == 1:
3         return 'January','blue'
4     elif month == 2:
5         return 'February','green'
6     elif month == 3:
7         return 'March','orange'
8     elif month == 4:
9         return 'April','yellow'
10    elif month == 5:
11        return 'May','red'
12    elif month == 6:
13        return 'June','violet'
14    elif month == 7:
15        return 'July','purple'
16    elif month == 8:
17        return 'August','black'
18    elif month == 9:
19        return 'September','brown'
```

```

20     elif month == 10:
21         return 'October', 'darkblue'
22     elif month == 11:
23         return 'November', 'grey'
24     else:
25         return 'December', 'pink'
26
27
28 def line_plot_month(self, month, data, ax):
29     label, color = self.color_month(month)
30     mdata = data[data.index.month == month]
31     sns.lineplot(data=mdata, ax=ax,
32                   label=label,
33                   color=color,
34                   marker='o',
35                   linewidth=3)
36
37 def sns_plot_month(self, data, feat, ax):
38     for i in range(1,13):
39         self.line_plot_month(i, data[feat], ax)

```

Step 3

Define **choose_plot()** to read **currentText** property of **cbData** widget and act accordingly:

```

1 def choose_plot(self):
2     strCB = self.cbData.currentText()
3
4     if strCB == 'Case Distribution':
5         self.widgetPlot1.canvas.figure.clf()
6         self.widgetPlot1.canvas.axis1 =
7             self.widgetPlot1.canvas.figure.add_subplot
facecolor = '#ffd1df')
9         self.plot_barchart(self.df_dummy, "Year",
self.widgetPlot1)
10
11
12         self.widgetPlot1.canvas.axis1 =
13             self.widgetPlot1.canvas.figure.add_subplot
facecolor = '#ffd1df')
14         self.plot_piechart(self.df_dummy, "Year",
self.widgetPlot1)
15
16
17         self.widgetPlot2.canvas.figure.clf()
18         self.widgetPlot2.canvas.axis1 =
19             self.widgetPlot2.canvas.figure.add_subplot
facecolor = '#ffd1df')
20         self.plot_barchart(self.df_dummy, "Month",
self.widgetPlot2)
21
22
23         self.widgetPlot2.canvas.axis1 =
24             self.widgetPlot2.canvas.figure.add_subplot
facecolor = '#ffd1df')
25         self.plot_piechart(self.df_dummy, "Month",
self.widgetPlot2)
26
27
28         self.widgetPlot3.canvas.figure.clf()
29         self.widgetPlot3.canvas.axis1 =
30             self.widgetPlot3.canvas.figure.add_subplot
facecolor = '#ffd1df')
31         self.plot_barchart(self.df_dummy, "Day",
self.widgetPlot3)
32
33
34         self.widgetPlot3.canvas.axis1 =
35             self.widgetPlot3.canvas.figure.add_subplot
facecolor = '#ffd1df')
36         self.plot_piechart(self.df_dummy, "Day",
self.widgetPlot3)
37
38
39         self.widgetPlot3.canvas.axis1 =
40             self.widgetPlot3.canvas.figure.add_subplot
facecolor = '#ffd1df')
41         self.plot_barchart(self.df_dummy, "Quarter",
self.widgetPlot3)

```

```

        self.widgetPlot3.canvas.axis1 =
            self.widgetPlot3.canvas.figure.add_subplot
facecolor = '#ffd1df')
        self.plot_piechart(self.df_dummy, "Quarter",
self.widgetPlot3)

```

Let's break down the steps of this function:

1. **strCB = self.cbData.currentText()**: This statement retrieves the currently selected text from a **QComboBox** (**cbData**) and assigns it to the variable **strCB**.
2. **if strCB == 'Case Distribution':**: This statement checks if the value of **strCB** is equal to the string 'Case Distribution'. If it is, the following code block will be executed.
3. **self.widgetPlot1.canvas.figure.clf()**: This clears the current figure of the **widgetPlot1** canvas.
4. **self.widgetPlot1.canvas.axis1**
self.widgetPlot1.canvas.figure.add_subplot(121, facecolor = '#ffd1df'): Here, a new subplot is added to the **widgetPlot1** canvas with the subplot number (indicating a 1x2 grid of subplots, and subplot 1 is the first one). The **facecolor** parameter sets the background color of the subplot.
5. **self.plot_barchart(self.df_dummy, "Year", self.widgetPlot1)**: This line calls the **plot_barchart()** method with the **df_dummy** DataFrame, the string "Year" as the categorical variable, and the **widgetPlot1** canvas as the widget to display the bar chart.
6. **self.widgetPlot1.canvas.axis1**
self.widgetPlot1.canvas.figure.add_subplot(122, facecolor = '#ffd1df'): Similar to step 4, a second subplot is added to the **widgetPlot1** canvas, this time with the subplot number 122.
7. **self.plot_piechart(self.df_dummy, "Quarter", self.widgetPlot1)**: This line calls the **plot_piechart()** method with the same parameters as step 5, but this time it displays a pie chart in the second subplot of **widgetPlot1**.
8. The above steps (3-7) are repeated for **widgetPlot2** and **widgetPlot3**, with different categorical variables ("Month", "Day" or "Quarter") and different subplot configurations.

In summary, the **choose_plot()** function selects the chosen text from a combo box and, if it matches 'Case Distribution', it generates bar charts and pie charts for different categorical variables ("Year", "Month", "Day" or "Quarter") on three different canvas widgets (**widgetPlot1**, **widgetPlot2**, and **widgetPlot3**). The function first clears existing figures and then adds subplots to each canvas for displaying the charts. Finally, it calls the **plot_barchart()** and **plot_piechart()** methods with the appropriate parameters to generate the charts on the respective subplots.

Step 4	<p>Connect currentIndexChanged() event of cbData with choose_plot() method and put it inside initial_state() method as shown in line 10:</p> <pre> 1 def initial_state(self, state): 2 self.pbTrainML.setEnabled(state) 3 self.cbData.setEnabled(state) 4 self.cbForecasting.setEnabled(state) 5 self.cbClassifier.setEnabled(state) 6 self.rbRobust.setEnabled(state) 7 self.rbMinMax.setEnabled(state) 8 self.rbStandard.setEnabled(state) 9 self.pbLoad.clicked.connect(self.import_database) 10 self.cbData.currentIndexChanged.connect(self.choose_p </pre>
Step 5	<p>Run gui_amazon.py and click LOAD DATA and ML MODEL buttons. Then, choose Case Distribution from cbData widget. You will see the result as shown in Figure 4.5.</p>

PLOTTING FEATURES DISTRIBUTION

PLOTTING FEATURES DISTRIBUTION

Step 1	Define plot_distribution() method to plot lineplot and scatter plot of a feature in the three widgets:
	<pre> 1 def plot_distribution(self,strCB, feat1, feat2): 2 self.widgetPlot1.canvas.figure.clf() 3 self.widgetPlot1.canvas.axis1 = \ 4 self.widgetPlot1.canvas.figure.add_subplot(111, \ 5 facecolor = '#ffd1df') 6 sns.lineplot(data=self.df_dummy[strCB], color='red', \ 7 linewidth=3, ax=self.widgetPlot1.canvas.axis1) 8 self.widgetPlot1.canvas.axis1.set_title(strCB + \ 9 " feature all year", fontsize=20) 10 self.widgetPlot1.canvas.figure.tight_layout() 11 self.widgetPlot1.canvas.draw() 12 13 14 self.widgetPlot2.canvas.figure.clf() 15 self.widgetPlot2.canvas.axis1 = \ 16 self.widgetPlot2.canvas.figure.add_subplot(111, \ 17 facecolor = '#ffd1df') 18 sns.scatterplot(data=self.df_dummy, x=strCB, y=feat1, \ 19 hue="Year", palette="deep", \ 20 ax=self.widgetPlot2.canvas.axis1) 21 self.widgetPlot2.canvas.axis1.set_title(</pre>

```

22     "Scatter distribution of " + strCB + " vs " + feat1
23     +\
24         " vs Year", fontsize=20)
25     self.widgetPlot2.canvas.figure.tight_layout()
26     self.widgetPlot2.canvas.draw()
27
28
29     self.widgetPlot3.canvas.figure.clf()
30     self.widgetPlot3.canvas.axis1 = \
31         self.widgetPlot3.canvas.figure.add_subplot(221, \
32             facecolor = '#ffd1df')
33     sns.scatterplot(data=self.df_dummy, x=strCB, y=feat2, \
34         hue="Day", palette="deep", \
35         ax=self.widgetPlot3.canvas.axis1)
36     self.widgetPlot3.canvas.axis1.set_title(\n
37         "Scatter distribution of " + strCB + " vs " +feat2+\
38         " vs Day", fontsize=20)
39     self.widgetPlot3.canvas.figure.tight_layout()
40     self.widgetPlot3.canvas.draw()
41
42     self.widgetPlot3.canvas.axis1 = \
43         self.widgetPlot3.canvas.figure.add_subplot(222, \
44             facecolor = '#ffd1df')
45     self.year_data_mean[strCB].plot(linewidth=5, \
46         ax=self.widgetPlot3.canvas.axis1)
47     self.year_data_ewm[strCB].plot(linewidth=5, \
48         ax=self.widgetPlot3.canvas.axis1)
49     self.widgetPlot3.canvas.axis1.set_title(\n
50         "Year-Wise Data: Mean and EWM", fontsize=14)
51     self.widgetPlot3.canvas.axis1.set_ylabel(strCB,
52     fontsize=12)
53     self.widgetPlot3.canvas.axis1.legend(["Mean", "EWM"], \
54         fontsize=20)
55     self.widgetPlot3.canvas.figure.tight_layout()
56     self.widgetPlot3.canvas.draw()
57
58     self.widgetPlot3.canvas.axis1 = \
59         self.widgetPlot3.canvas.figure.add_subplot(223, \
60             facecolor = '#ffd1df')
61     self.monthly_data_mean[strCB].plot(linewidth=5, \
62         ax=self.widgetPlot3.canvas.axis1)
63     self.monthly_data_ewm[strCB].plot(linewidth=5, \
64         ax=self.widgetPlot3.canvas.axis1)
65     self.widgetPlot3.canvas.axis1.set_title(strCB + \
66         ": Month-Wise Data: Mean and EWM", fontsize=14)
67     self.widgetPlot3.canvas.axis1.set_ylabel(strCB,
68     fontsize=12)
69     self.widgetPlot3.canvas.axis1.legend(["Mean", "EWM"], \
70         fontsize=20)
71     self.widgetPlot3.canvas.figure.tight_layout()
72     self.widgetPlot3.canvas.draw()
73
74     self.widgetPlot3.canvas.axis1 = \
75         self.widgetPlot3.canvas.figure.add_subplot(224, \
76             facecolor = '#ffd1df')
77     self.sns_plot_month(self.monthly_data_mean, strCB, \
78         ax=self.widgetPlot3.canvas.axis1)
79     self.widgetPlot3.canvas.axis1.set_title(strCB + \
80         ": Month-Wise Data for Every Month", fontsize=14)
81     self.widgetPlot3.canvas.axis1.set_ylabel(strCB,
82     fontsize=12)
83     self.widgetPlot3.canvas.figure.tight_layout()
84     self.widgetPlot3.canvas.draw()

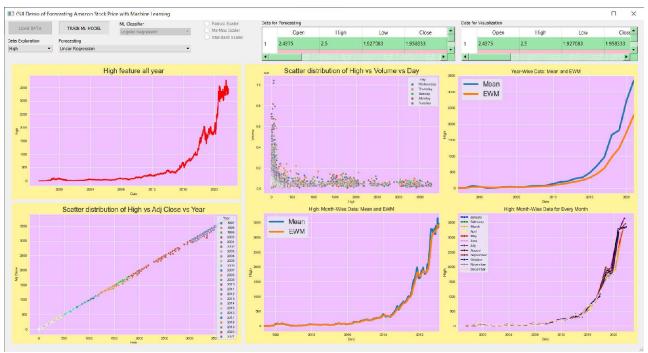
```

It is responsible for plotting various distributions based on the provided parameters. Let's break down the steps of this function:

1. **self.widgetPlot1.canvas.figure.clf()**: This line clears the current figure of the **widgetPlot1** canvas.
2. **self.widgetPlot1.canvas.axis1 = self.widgetPlot1.canvas.figure.add_subplot(111, facecolor='#ffd1df')**: A new subplot is added to the **widgetPlot1** canvas. The subplot number 111 indicates a single subplot filling the entire canvas.

3. `sns.lineplot(data=self.df_dummy[strCB], color='red', linewidth=3, ax=self.widgetPlot1.canvas.axis1)`: This line uses Seaborn's lineplot function to plot a line graph. The data parameter specifies the data to be plotted from the `df_dummy` DataFrame, with strCB as the column to be plotted. The line color is set to red, and the line width is set to 3. The line plot is drawn on `widgetPlot1.canvas.axis1`.
4. `self.widgetPlot1.canvas.axis1.set_title(strCB + " feature all year", fontsize=20)`: The title of the plot is set using the provided `strCB` parameter, indicating the feature being plotted along with "feature all year".
5. `self.widgetPlot1.canvas.figure.tight_layout()`: This function adjusts the spacing between subplots to improve the layout.
6. `self.widgetPlot1.canvas.draw()`: This line updates and displays the plot on `widgetPlot1`.
7. The above steps (1-6) are repeated for `widgetPlot2` and `widgetPlot3`, with different types of plots and different subplot configurations.
8. Several other plots and visualizations are created on `widgetPlot3`:
 - Scatter plot: `sns.scatterplot()` is used to create a scatter plot with `strCB` on the x-axis, `feat1` on the y-axis, and different colors for each year.
 - Another scatter plot: `sns.scatterplot()` is used to create a scatter plot with `strCB` on the x-axis, `feat2` on the y-axis, and different colors for each day.
 - Line plots: `self.year_data_mean[strCB].plot()` and `self.year_data_ewm[strCB].plot()` create line plots using data from `year_data_mean` and `year_data_ewm` DataFrames, respectively.
 - Line plots: `self.monthly_data_mean[strCB].plot()` and `self.monthly_data_ewm[strCB].plot()` create line plots using data from `monthly_data_mean` and `monthly_data_ewm` DataFrames, respectively.
 - Custom plot: `self.sns_plot_month()` is a custom function that plots month-wise data using the `monthly_data_mean` DataFrame and `strCB` as the column of interest.

Each plot is assigned to a specific subplot on `widgetPlot3`, and appropriate titles, labels, and legends are set. The `tight_layout()` function is called to adjust the spacing between subplots, and `widgetPlot3.canvas.draw()` is used to update and display the plot on `widgetPlot3`.

Step 2	<p>Add this code to the end of choose_plot() method:</p> <pre><code>1 if strCB == 'High': 2 self.plot_distribution(strCB, "Adj Close", "Volume")</code></pre>
Step 3	<p>Run gui_amazon.py and click LOAD DATA and TRAIN ML MODEL buttons. Then, choose High item from cbData widget. You will see the result as shown in Figure 4.6.</p> 
Step 4	<p>Add this code to the end of choose_plot() method:</p> <pre><code>1 if strCB == 'Low': 2 self.plot_distribution(strCB, "Adj Close", "Volume")</code></pre>
Step 5	<p>Run gui_amazon.py and click LOAD DATA and TRAIN ML MODEL buttons. Then, choose Low item from cbData widget. You will see the result as shown in Figure 4.7.</p> 
Step 6	<p>Add this code to the end of choose_plot() method:</p> <pre><code>1 if strCB == 'Open': 2 self.plot_distribution(strCB, "Adj Close", "Volume")</code></pre>
Step 7	<p>Run gui_amazon.py and click LOAD DATA and TRAIN</p>

ML MODEL buttons. Then, choose **Open** item from **cbData** widget. You will see the result as shown in Figure 4.8.

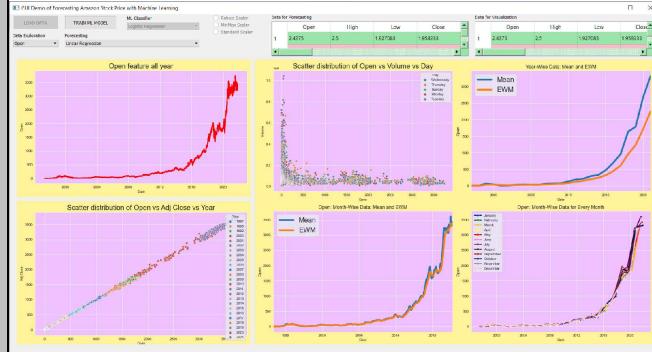


Figure 4.8 The lineplot, scatter plot, year-wise, and month-wise of **Open** feature

Step 8 Add this code to the end of **choose_plot()** method:

```
1 if strCB == 'Close':
2     self.plot_distribution(strCB, "Open", "Volume")
```

Step 9 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Close** item from **cbData** widget. You will see the result as shown in Figure 4.9.



Figure 4.9 The lineplot, scatter plot, year-wise, and month-wise of **Close** feature

Step 10 Add this code to the end of **choose_plot()** method:

```
1 if strCB == 'Adj Close':
2     self.plot_distribution(strCB, "Open", "Volume")
```

Step 11 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Adj Close** item from **cbData** widget. You will see the result as shown in Figure 4.10.



Figure 4.10 The lineplot, scatter plot, year-wise, and month-wise of Adj Close feature

PLOTTING GROUPED DISTRIBUTION

PLOTTING GROUPED DISTRIBUTION

Step 1 Define `plot_group_barchart()` and `plot_group_piechart()` methods to grouped distribution of a feature bar plot and pie chart:

```

1 #Defines function to plot grouped distribution of a
2 categorical feature bar plot
3 def plot_group_barchart(self,df, var, title, widget):
4     ax = df.plot(kind="barh",ax = widget.canvas.axis1)
5     for i,j in enumerate(df.values):
6         ax.text(.7,i,j,weight = "bold", fontsize=10)
7
8     widget.canvas.axis1.set_title(title, fontsize=14)
9     widget.canvas.axis1.set_xlabel(var)
10    widget.canvas.figure.tight_layout()
11    widget.canvas.draw()
12
13
14 #Defines function to plot case grouped distribution of a
15 categorical feature in pie chart
16 def plot_group_piechart(self, df, title, widget):
17     label_list = list(df.index)
18     df.plot.pie(ax = widget.canvas.axis1, autopct =
19     "%1.1f%%",
20     colors = sns.color_palette("prism",7),
21     startangle = 60,labels=label_list,
22     wedgeprops={"linewidth":2,"edgecolor":"k"},
23     shadow = True, textprops={'fontsize': 10})
24     widget.canvas.axis1.set_title(title, fontsize=14)
25     widget.canvas.figure.tight_layout()
26     widget.canvas.draw()

```

Let's break down the two methods step by step:

1. `plot_group_barchart(self, df, var, title, widget)`:

- This method takes in five parameters: **df** (the DataFrame containing the data), **var** (the categorical variable to be plotted), **title** (the title of the plot), and **widget** (the widget where the plot will be displayed).
- ax = df.plot(kind="barh", ax=widget.canvas.axis1)**: This line creates a horizontal bar plot using the `plot` function of the DataFrame **df**. The `kind` parameter is set to "barh" to create a horizontal bar plot, and the `ax` parameter specifies the subplot to plot on.
- for i, j in enumerate(df.values)**: This line iterates over the values of the DataFrame **df** using

enumerate to get the index i and value j at each iteration.

- **ax.text(.7, i, j, weight="bold", fontsize=10):** This line adds a text annotation to the bar plot at the coordinates (.7, i) with the value j. The text is displayed with bold font weight and fontsize of 10.
- **widget.canvas.axis1.set_title(title, fontsize=14):** Sets the title of the plot to the provided title parameter with a fontsize of 14.
- **widget.canvas.axis1.set_xlabel(var):** Sets the x-axis label of the plot to the provided var parameter.
- **widget.canvas.figure.tight_layout():** Adjusts the spacing between subplots to improve the layout.
- **widget.canvas.draw():** Updates and displays the plot on the specified widget.

2. **plot_group_piechart(self, df, title, widget):**

- This method takes in four parameters: **df** (the DataFrame containing the data), **title** (the title of the plot), and **widget** (the widget where the plot will be displayed).
- **label_list = list(df.index):** This line retrieves the index values from the DataFrame **df** and converts them into a list. These index values will be used as labels for the pie chart.
- **df.plot.pie(ax=widget.canvas.axis1, autopct="%1.1f%%", colors=sns.color_palette("prism", 7), startangle=60, labels=label_list, wedgeprops={"linewidth": 2, "edgecolor": "k"}, shadow=True, textprops={'fontsize': 10}):** This line creates a pie chart using the `plot.pie` function of the DataFrame **df**. The **ax** parameter specifies the subplot to plot on. Additional parameters such as **autopct** (percentage format), **colors** (color palette), **startangle** (starting angle of the first slice), **labels** (the labels for each slice), **wedgeprops** (properties of the wedges), **shadow** (enabling shadow effect), and **textprops** (properties of the text) are provided to customize the appearance of the pie chart.
- **widget.canvas.axis1.set_title(title, fontsize=14):** Sets the title of the plot to the provided title parameter with a fontsize of 14.
- **widget.canvas.figure.tight_layout():** Adjusts the spacing between subplots to improve the layout.
- **widget.canvas.draw():** Updates and displays the plot on the specified widget.

These methods are designed to create grouped distribution plots (bar charts and pie charts) for categorical features. The `plot_group_barchart()` function plots a horizontal bar

Step Add this code to the end of **choose_plot()** method:

2

```
1     if strCB == 'Volume':
2         self.widgetPlot1.canvas.figure.clf()
3         self.widgetPlot1.canvas.axis1 = \
4             self.widgetPlot1.canvas.figure.add_subplot(121,\n5             facecolor = '#ffd1df')
6         self.plot_group_barchart(self.df_dummy.groupby(\n7             'Year')['Volume'].sum(), "Volume", \
8             "The distribution of Volume by Year", \
9             self.widgetPlot1)
10
11        self.widgetPlot1.canvas.axis1 = \
12            self.widgetPlot1.canvas.figure.add_subplot(122,\n13            facecolor = '#ffd1df')
14        self.plot_group_piechart(self.df_dummy.groupby(\n15            'Year')['Volume'].sum(), \
16            "The distribution of Volume by Year", \
17            self.widgetPlot1)
18
19
20        self.widgetPlot2.canvas.figure.clf()
21        self.widgetPlot2.canvas.axis1 = \
22            self.widgetPlot2.canvas.figure.add_subplot(121,\n23            facecolor = '#ffd1df')
24        self.plot_group_barchart(self.df_dummy.groupby(\n25            'Quarter')['Volume'].sum(), "Volume", \
26            "The distribution of Volume by Quarter", \
27            self.widgetPlot2)
28
29
30        self.widgetPlot2.canvas.axis1 = \
31            self.widgetPlot2.canvas.figure.add_subplot(122,\n32            facecolor = '#ffd1df')
33        self.plot_group_piechart(self.df_dummy.groupby(\n34            'Quarter')['Volume'].sum(), \
35            "The distribution of Volume by Quarter", \
36            self.widgetPlot2)
37
38        self.widgetPlot3.canvas.figure.clf()
39        self.widgetPlot3.canvas.axis1 = \
40            self.widgetPlot3.canvas.figure.add_subplot(221,\n41            facecolor = '#ffd1df')
42        self.plot_group_barchart(self.df_dummy.groupby(\n43            'Day')['Volume'].sum(), "Volume", \
44            "The distribution of Volume by Days of Week", \
45            self.widgetPlot3)
46
47        self.widgetPlot3.canvas.axis1 = \
48            self.widgetPlot3.canvas.figure.add_subplot(222,\n49            facecolor = '#ffd1df')
50        self.plot_group_piechart(self.df_dummy.groupby(\n51            'Day')['Volume'].sum(), \
52            "The distribution of Volume by Days of Week", \
53            self.widgetPlot3)
54
55
56        self.widgetPlot3.canvas.axis1 = \
57            self.widgetPlot3.canvas.figure.add_subplot(223,\n58            facecolor = '#ffd1df')
59        self.plot_group_barchart(self.df_dummy.groupby(\n60            'Month')['Volume'].sum(), "Volume", \
61            "The distribution of Volume by Month", \
62            self.widgetPlot3)
63
64        self.widgetPlot3.canvas.axis1 = \
65            self.widgetPlot3.canvas.figure.add_subplot(224,\n66            facecolor = '#ffd1df')
67        self.plot_group_piechart(self.df_dummy.groupby(\n68            'Month')['Volume'].sum(), \
69            "The distribution of Volume by Month", \
70            self.widgetPlot3)
```

Here's a detailed step-by-step explanation of the code:

1. Check if strCB is equal to 'Volume':

if strCB == 'Volume':

2. Clear the first plot (**widgetPlot1**) and create a subplot within the first widget (**widgetPlot1.canvas.axis1**):

- **self.widgetPlot1.canvas.figure.clf()**
- **self.widgetPlot1.canvas.axis1 = self.widgetPlot1.canvas.figure.add_subplot(121, facecolor='#ffd1df')**

3. Call the **plot_group_barchart()** function to create a grouped bar chart for the 'Volume' feature by year:

```
self.plot_group_barchart(self.df_dummy.groupby('Year')  
['Volume'].sum(), "Volume", "The distribution of Volume  
by Year", self.widgetPlot1)
```

In the **plot_group_barchart()** function:

- Create a horizontal bar plot (**df.plot(kind="barh")**) using the grouped data (**self.df_dummy.groupby('Year') ['Volume'].sum()**) and the specified widget (**widget.canvas.axis1**).
- Iterate over the values in the bar plot and add the values as text annotations to the bars.
- Set the title of the plot to indicate the feature and grouping variable.

4. Create another subplot in the first widget for a grouped pie chart of the 'Volume' feature by year:

```
self.widgetPlot1.canvas.axis1 =  
self.widgetPlot1.canvas.figure.add_subplot(122,  
facecolor='#ffd1df')
```

5. Call the **plot_group_piechart()** function to create a grouped pie chart for the 'Volume' feature by year:

```
self.plot_group_piechart(self.df_dummy.groupby('Year')  
['Volume'].sum(), "The distribution of Volume by Year",  
self.widgetPlot1)
```

In the **plot_group_piechart** function:

- Create a pie chart (**df.plot.pie()**) using the grouped data (**self.df_dummy.groupby('Year') ['Volume'].sum()**) and the specified widget (**widget.canvas.axis1**).
- Customize the pie chart appearance, such as colors, start angle, and wedge properties.
- Set the title of the plot to indicate the feature and grouping variable.

6. Repeat the same steps (2-5) for the second and third widgets (**widgetPlot2** and **widgetPlot3**) with different grouping variables (Quarter, Day, and Month) instead of Year.

For **widgetPlot2**:

- Group the data by Quarter and calculate the sum of Volume for each Quarter.
- Create a grouped bar chart and a grouped pie chart for the 'Volume' feature by Quarter.

For **widgetPlot3**:

- Group the data by Day and calculate the sum of Volume for each day of the week.
- Create a grouped bar chart and a grouped pie chart for the 'Volume' feature by day of the week.

- Group the data by Month and calculate the sum of Volume for each month.
- Create a grouped bar chart and a grouped pie chart for the 'Volume' feature by month.

In summary, this code segment generates a set of grouped distribution plots for the 'Volume' feature, displaying its distribution across different time periods (year, quarter, day of the week, and month). The plots include both grouped bar charts and grouped pie charts, providing insights into the distribution patterns of the 'Volume' feature within each grouping category.

Step 3 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Volume** item from **cbData** widget. You will see the result as shown in Figure 4.11.



Figure 4.11 The distribution of Volume by Year, Month, and Quarter

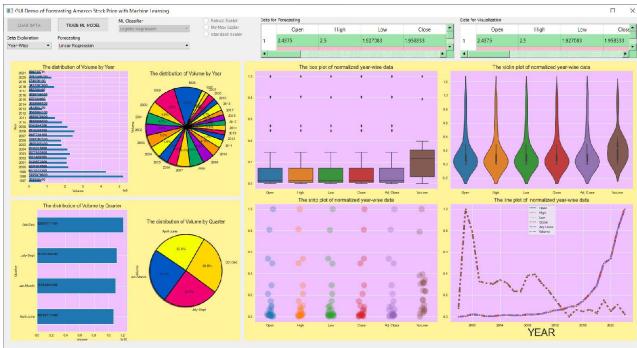
PLOTTING MONTH-WISE AND YEAR-WISE DISTRIBUTION

Step 1 Define **plot_norm_wise_data()** method to show boxplot, violinplot, stripplot, and lineplot of normalized year-wise and month-wise data as explained before:

```

1 #Plots boxplot, violinplot, stripplot, and heatmap of
2 normalized year-wise data
3 def plot_norm_wise_data(self, norm_data, ax1, ax2, ax3, ax4,
4 title):
5     g=sns.boxplot(data=norm_data,ax = ax1)
6     g.xaxis.get_label().set_fontsize(10)
7     g.set_title("The box plot of " + title, fontsize=15)
8
9
10    g=sns.violinplot(data=norm_data,ax = ax2)
11    g.xaxis.get_label().set_fontsize(10)
12    g.set_title("The violin plot of " + title, fontsize=15)
13
14    g=sns.stripplot(data=norm_data, jitter=True, s=18, \
15                      alpha=0.3, ax = ax3)
16    g.xaxis.get_label().set_fontsize(10)
17    g.set_title("The strip plot of " + title, fontsize=15)
18
19    g=sns.lineplot(data=norm_data, marker='s', \
20                  ax = ax4,linewidth=5)
21    g.xaxis.get_label().set_fontsize(30)
22    g.set_title("The line plot of " + title, fontsize=15)

```

	<pre>g.set_xlabel("YEAR")</pre>
Step 2	<p>Add this code to the end of choose_plot() method:</p> <pre> 1 if strCB == 'Year-Wise': 2 self.widgetPlot3.canvas.figure.clf() 3 self.widgetPlot3.canvas.axis1 = \ 4 self.widgetPlot3.canvas.figure.add_subplot(221,\n 5 facecolor = '#ffd1df') 6 self.widgetPlot3.canvas.axis2 = \ 7 self.widgetPlot3.canvas.figure.add_subplot(222,\n 8 facecolor = '#ffd1df') 9 self.widgetPlot3.canvas.axis3 = \ 10 self.widgetPlot3.canvas.figure.add_subplot(223,\n 11 facecolor = '#ffd1df') 12 self.widgetPlot3.canvas.axis4 = \ 13 self.widgetPlot3.canvas.figure.add_subplot(224,\n 14 facecolor = '#ffd1df') 15 norm_data = (self.year_data_mean - \ 16 self.year_data_mean.min()) / \ 17 (self.year_data_mean.max() - \ 18 self.year_data_mean.min()) 19 self.plot_norm_wise_data(norm_data, \ 20 self.widgetPlot3.canvas.axis1, \ 21 self.widgetPlot3.canvas.axis2, \ 22 self.widgetPlot3.canvas.axis3, \ 23 self.widgetPlot3.canvas.axis4, \ 24 "normalized year-wise data") 25 self.widgetPlot3.canvas.figure.tight_layout() self.widgetPlot3.canvas.draw() </pre>  <p>Figure 4.12 Showing the normalized year-wise data</p> <p>Here's a step-by-step explanation of the code:</p> <ol style="list-style-type: none"> 1. The code first checks if the value of strCB is equal to 'Year-Wise'. 2. If strCB is 'Year-Wise', the code proceeds with the following steps: 3. The third plot (widgetPlot3) is cleared to remove any previous content. 4. Four subplots (widgetPlot3.canvas.axis1, widgetPlot3.canvas.axis2, widgetPlot3.canvas.axis3, widgetPlot3.canvas.axis4) are created within the third widget (widgetPlot3) to display the visualizations. 5. The data is normalized by subtracting the minimum value from each value and dividing it by the range (maximum value minus minimum value). This step ensures that the data is scaled between 0 and 1.

	<p>6. The normalized data is then plotted using the plot_norm_wise_data() function, which takes the normalized data and the four subplots as input. This function is responsible for visualizing the normalized year-wise data in different formats.</p> <p>7. Finally, the layout of the third widget (widgetPlot3.canvas.figure) is adjusted to ensure that the visualizations are properly arranged and displayed.</p> <p>8. The resulting plots are drawn and displayed in the widgetPlot3 widget for the user to view.</p>
Step 3	Run gui_amazon.py and click LOAD DATA and TRAIN ML MODEL buttons. Then, choose Year-Wise item from cbData widget. You will see the result as shown in Figure 4.12.
Step 4	<p>Add this code to the end of choose_plot() method:</p> <pre> 1 if strCB == 'Month-Wise': 2 self.widgetPlot3.canvas.figure.clf() 3 self.widgetPlot3.canvas.axis1 = \ 4 self.widgetPlot3.canvas.figure.add_subplot(221, \ 5 facecolor = '#ffd1df') 6 self.widgetPlot3.canvas.axis2 = \ 7 self.widgetPlot3.canvas.figure.add_subplot(222, \ 8 facecolor = '#ffd1df') 9 self.widgetPlot3.canvas.axis3 = \ 10 self.widgetPlot3.canvas.figure.add_subplot(223, \ 11 facecolor = '#ffd1df') 12 self.widgetPlot3.canvas.axis4 = \ 13 self.widgetPlot3.canvas.figure.add_subplot(224, \ 14 facecolor = '#ffd1df') 15 norm_data = (self.monthly_data_mean - \ 16 self.monthly_data_mean.min()) / \ 17 (self.monthly_data_mean.max() - \ 18 self.monthly_data_mean.min()) 19 self.plot_norm_wise_data(norm_data, \ 20 self.widgetPlot3.canvas.axis1, \ 21 self.widgetPlot3.canvas.axis2, \ 22 self.widgetPlot3.canvas.axis3, \ 23 self.widgetPlot3.canvas.axis4, \ 24 "normalized month-wise data") 25 self.widgetPlot3.canvas.figure.tight_layout() 26 self.widgetPlot3.canvas.draw() </pre>

Step-by-step explanation of the code:

1. If the variable **strCB** is equal to the string 'Month-Wise', the following steps are executed:
2. **self.widgetPlot3.canvas.figure.clf()**: This line clears the current figure of the canvas associated with **widgetPlot3**.
3. **self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(221, facecolor='#ffd1df')**: This line adds a subplot (**axis**) with the index 221 (which corresponds to the top-left position in a 2x2 grid) to the figure. The **facecolor** parameter sets the background color of the subplot to '#ffd1df'.
4. **self.widgetPlot3.canvas.axis2 = self.widgetPlot3.canvas.figure.add_subplot(222, facecolor='#ffd1df')**: This line adds a subplot with the

	<p>index 222 (top-right position) to the figure, also setting the background color.</p> <p>5. <code>self.widgetPlot3.canvas.axis3 = self.widgetPlot3.canvas.figure.add_subplot(223, facecolor='#ffd1df');</code>: This line adds a subplot with the index 223 (bottom-left position) to the figure, with the specified background color.</p> <p>6. <code>self.widgetPlot3.canvas.axis4 = self.widgetPlot3.canvas.figure.add_subplot(224, facecolor='#ffd1df');</code>: This line adds a subplot with the index 224 (bottom-right position) to the figure, using the specified background color.</p> <p>7. <code>norm_data = (self.monthly_data_mean - self.monthly_data_mean.min()) / (self.monthly_data_mean.max() - self.monthly_data_mean.min());</code>: This line calculates normalized data by subtracting the minimum value of <code>monthly_data_mean</code> from each element and then dividing it by the range (max-min) of <code>monthly_data_mean</code>.</p> <p>8. <code>self.plot_norm_wise_data(norm_data, self.widgetPlot3.canvas.axis1, self.widgetPlot3.canvas.axis2, self.widgetPlot3.canvas.axis3, self.widgetPlot3.canvas.axis4, "normalized month-wise data");</code>: This line calls a function <code>plot_norm_wise_data()</code> and passes the <code>norm_data</code> as well as the four subplot axes (<code>axis1</code>, <code>axis2</code>, <code>axis3</code>, <code>axis4</code>) and a string "normalized month-wise data" as arguments. This function presumably plots the normalized data on the given subplot axes.</p> <p>9. <code>self.widgetPlot3.canvas.figure.tight_layout();</code>: This line adjusts the spacing between subplots to make them fit within the figure area more neatly.</p> <p>10. <code>self.widgetPlot3.canvas.draw();</code>: This line redraws the canvas associated with <code>widgetPlot3</code> to reflect the changes made, displaying the updated figure with the normalized data plotted on the subplots.</p>
Step 5	Run <code>gui_amazon.py</code> and click LOAD DATA and TRAIN ML MODEL buttons. Then, choose Month-Wise item from <code>cbData</code> widget. You will see the result as shown in Figure 4.13.

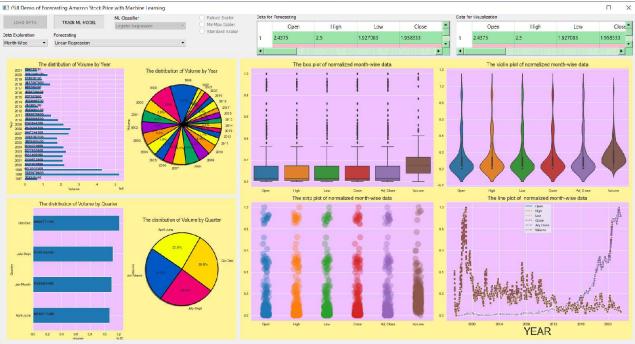


Figure 4.13 Showing the normalized month-wise data

PLOTTING TECHNICAL INDICATORS

PLOTTING TECHNICAL INDICATORS

Step 1 Define `plot_technical_indicators()` method to plot MACD, SMA, RSI, upper and lower bands, standard deviation, and daily returns of Adj Close column as explained before:

```

1 #Plots MACD, SMA, RSI, upper and lower bands, standard
2 deviation, and daily returns of Adj Close column
3 def plot_technical_indicators(self,df,ax1,ax2,ax3,ax4,ax5):
4     stock_close=df["Adj Close"]
5     SMA_CLOSE=df['SMA']
6     stock_close[:365].plot(title='GLD Moving Average',
7         label='GLD', linewidth=3, ax=ax1)
8     SMA_CLOSE[:365].plot(label="SMA", linewidth=3, ax=ax1)
9
10    upper_band=df['Upper_band']
11    lower_band=df['Lower_band']
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37

```

```

upper_band[:365].plot(title="Upper-Lower Band", \
    label='upper band', linewidth=3, ax=ax1)
lower_band[:365].plot(label='lower band', linewidth=3,
ax=ax1)

DIF=df['DIF']
MACD=df['MACD']
DIF[:365].plot(title='DIF and MACD',label='DIF', \
    linewidth=3, ax=ax2)
MACD[:365].plot(label='MACD', linewidth=3, ax=ax2)

RSI=df['RSI']
RSI[:365].plot(title='RSI',label='RSI', linewidth=3, ax=ax3)

STDEV=df['STDEV']
STDEV[:365].plot(title='STDEV',label='STDEV', \
    linewidth=3, ax=ax4)

Daily_Return=df['daily_returns']
Daily_Return[:365].plot(title="Daily Returns", \
    label='Daily Returns', linewidth=3, ax=ax5)

ax1.set_ylabel('Price')
ax2.set_ylabel('Price')
ax3.set_ylabel('Price')
ax4.set_ylabel('Price')
ax5.set_ylabel('Price')

```

Step 2

Add this code to the end of **choose_plot()** method:

```

1   if strCB == 'Technical Indicators':
2       self.widgetPlot3.canvas.figure.clf()
3       self.widgetPlot3.canvas.axis1 = \
4           self.widgetPlot3.canvas.figure.add_subplot
5           facecolor = '#ffd1df')
6       self.widgetPlot3.canvas.axis2 = \
7           self.widgetPlot3.canvas.figure.add_subplot
8           facecolor = '#ffd1df')
9       self.widgetPlot3.canvas.axis3 = \
10          self.widgetPlot3.canvas.figure.add_subplot
11          facecolor = '#ffd1df')

12      self.widgetPlot1.canvas.figure.clf()
13      self.widgetPlot1.canvas.axis4 = \
14          self.widgetPlot1.canvas.figure.add_subplot
15          facecolor = '#ffd1df')

16      self.widgetPlot2.canvas.figure.clf()
17      self.widgetPlot2.canvas.axis5 = \
18          self.widgetPlot2.canvas.figure.add_subplot
19          facecolor = '#ffd1df')

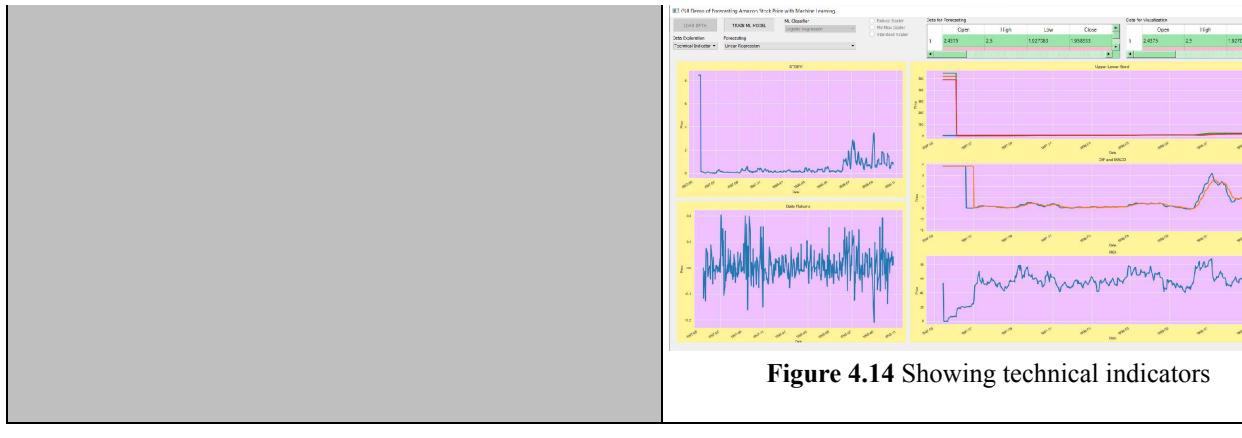
20      self.plot_technical_indicators(self.df,\ \
21          self.widgetPlot3.canvas.axis1,\ \
22          self.widgetPlot3.canvas.axis2,\ \
23          self.widgetPlot3.canvas.axis3,\ \
24          self.widgetPlot1.canvas.axis4,\ \
25          self.widgetPlot2.canvas.axis5)
26      self.widgetPlot3.canvas.figure.tight_layout()
27      self.widgetPlot2.canvas.figure.tight_layout()
28      self.widgetPlot1.canvas.figure.tight_layout()
29      self.widgetPlot3.canvas.draw()
30      self.widgetPlot2.canvas.draw()
31      self.widgetPlot1.canvas.draw()
32
33
34

```

Step-by-step explanation of the code is as follows:

1. If the variable **strCB** is equal to the 'Technical Indicators', the following step executed:
2. Clear the current figure of the canvas associated with **widgetPlot3**.

	<p>3. Add a subplot (axis) with the index 311 (corresponds to the top position in a 3x1 grid) to the figure associated with widgetPlot3, with the specified background color.</p> <p>4. Add a subplot with the index 312 (1 position) to the figure associated with widgetPlot3, with the specified background color.</p> <p>5. Add a subplot with the index 313 (the position) to the figure associated with widgetPlot3, using the specified background color.</p> <p>6. Clear the current figure of the canvas associated with widgetPlot1.</p> <p>7. Add a subplot with the index 111 to the figure associated with widgetPlot1, setting the background color.</p> <p>8. Clear the current figure of the canvas associated with widgetPlot2.</p> <p>9. Add a subplot with the index 111 to the figure associated with widgetPlot2, using the specified background color.</p> <p>10. Call a function plot_technical_indicators and pass the dataframe df, as well as the five subplot axes (axis1, axis2, axis3, axis4, axis5) as arguments. This function presumably plots the technical indicators on the provided subplot.</p> <p>11. Adjust the spacing between subplots in the figure associated with widgetPlot3 to make them fit within the figure area more neatly.</p> <p>12. Adjust the spacing between subplots in the figure associated with widgetPlot2 to make them fit within the figure area more neatly.</p> <p>13. Adjust the spacing between subplots in the figure associated with widgetPlot1 to make them fit within the figure area more neatly.</p> <p>14. Redraw the canvas associated with widgetPlot3 to reflect the changes made, displaying the updated figure with the technical indicators plotted on the subplots.</p> <p>15. Redraw the canvas associated with widgetPlot2 to reflect the changes made, displaying the updated figure.</p> <p>16. Redraw the canvas associated with widgetPlot1 to reflect the changes made, displaying the updated figure.</p>
Step 3	Run gui_amazon.py and click LOAD DATA and ML MODEL buttons. Then, choose Technical Indicators item from cbData widget. You will see the result as in Figure 4.14.



CREATING GUI FOR REGRESSION-BASED FORECASTING

CREATING GUI FOR REGRESSION-BASED FORECASTING

SPLITTING DATA FOR REGRESSION

SPLITTING DATA FOR REGRESSION

Step Define `split_data_regression()` method to split data for regression:

```
1 def split_data_regression(self,X):
2     #Sets target column
3     y_final = pd.DataFrame(X["Adj Close"])
4     X = X.drop(["Adj Close"], axis =1)
5
6     #Normalizes data
7     scaler = MinMaxScaler()
8     X_minmax_data = scaler.fit_transform(X)
9     X_final = pd.DataFrame(columns=X.columns, \
10                             data=X_minmax_data, index=X.index)
11    print('Shape of features : ', X_final.shape)
12    print('Shape of target : ', y_final.shape)
13
14
15    #Shifts target array to predict the n + 1 samples
16    n=90
17    y_final = y_final.shift(-1)
18    y_val = y_final[-n:-1]
19    y_final = y_final[:-n]
20
21
22    #Takes last n rows of data to be validation set
23    X_val = X_final[-n:-1]
24    X_final = X_final[:-n]
25
26    print("\n -----After process----- \n")
27    print('Shape of features : ', X_final.shape)
28    print('Shape of target : ', y_final.shape)
29    print(y_final.tail().to_string())
30
31    y_final=y_final.astype('float64')
32
33
34    #Splits data into training and test data at 90% and 10%
35    #respectively
36    self.split_idx=round(0.9*len(X))
37    print("split_idx=",self.split_idx)
```

```

38     X_train = X_final[:self.split_idx]
39     y_train = y_final[:self.split_idx]
X_test = X_final[self.split_idx:]
y_test = y_final[self.split_idx:]

return X_final, y_final, X_train, y_train, X_test,
y_test, X_val, y_val

```

It splits the data into training, testing, and validation sets for regression tasks. Let's break down the steps:

1. Set the target column by creating a new DataFrame `y_final` containing the values from the "Adj Close" column of the input DataFrame `X`.
2. Remove the "Adj Close" column from the input DataFrame `X` using the drop method with `axis=1` (column axis).
3. Normalize the remaining features in `X` using the `MinMaxScaler`. This scales the values between 0 and 1.
4. Create a new DataFrame `X_final` with the normalized data, maintaining the same column names and index as `X`.
5. Print the shapes of the features (`X_final`) and the target (`y_final`) DataFrames.
6. Set the shift size `n` to 90. This will be used to shift the target array to predict the next `n+1` samples.
7. Shift the `y_final` DataFrame upwards by one position using the `shift` method. This shifts the target values, effectively aligning them with the corresponding features for prediction.
8. Assign the last `n-1` rows of the shifted `y_final` DataFrame to `y_val`. This will be used as the validation target set.
9. Remove the last `n-1` rows from `y_final`, leaving the remaining shifted target values for training.
10. Assign the last `n-1` rows of the `X_final` DataFrame to `X_val`. This will be used as the validation feature set.
11. Remove the last `n-1` rows from `X_final`, leaving the remaining feature values for training.
12. Print the shapes of the features (`X_final`) and the target (`y_final`) DataFrames after the above process, along with the last 5 rows of the target DataFrame (`y_final.tail()`).
13. Convert the `y_final` DataFrame to the `float64` data type.
14. Calculate the split index as 90% of the length of `X_final` (the training data).
15. Assign the first `split_idx` rows of `X_final` to `X_train`, representing the training feature set.
16. Assign the first `split_idx` rows of `y_final` to `y_train`, representing the training target set.
17. Assign the remaining rows of `X_final` (starting from `split_idx`) to `X_test`, representing the testing feature set.
18. Assign the remaining rows of `y_final` (starting from `split_idx`) to `y_test`, representing the testing target set.
19. Return the following variables: `X_final`, `y_final`, `X_train`, `y_train`, `X_test`, `y_test`, `X_val`, `y_val`.

The function essentially prepares the data for regression modeling by setting the target column, normalizing the features, shifting the

	target values, and splitting the data into training, testing, and validation sets.
Step 2	Modify read_dataset() method to invoke split_data_regression() method as shown in line 9-11:
Step 3	<pre> 1 def read_dataset(self, dir): 2 #Reads dataset 3 df = pd.read_csv(dir) 4 5 #Creates new Dataframes 6 self.df, self.df_dummy=self.create_new_dfs(df) 7 8 #Splits data for regression 9 self.X_final, self.y_final, self.X_train, self.y_train, 10 \ self.X_test, self.y_test, self.X_val, \ 11 self.y_val=self.split_data_regression(self.df) 12 13 #Computes technical indicators 14 self.compute_technical_indicators(self.df) </pre> <p>Connect currentIndexChanged() event of cbForecasting widget with choose_forecasting() method and put it inside initial_state() method as shown in line 11:</p> <pre> 1 def initial_state(self, state): 2 self.pbTrainML.setEnabled(state) 3 self.cbData.setEnabled(state) 4 self.cbForecasting.setEnabled(state) 5 self.cbClassifier.setEnabled(state) 6 self.rbRobust.setEnabled(state) 7 self.rbMinMax.setEnabled(state) 8 self.rbStandard.setEnabled(state) 9 self.pbLoad.clicked.connect(self.import_dataset) 10 self.cbData.currentIndexChanged.connect(self.choose_plot) 11 12 self.cbForecasting.currentIndexChanged.connect(self.choose_forecasting)) </pre>

FORECASTING USING LINEAR REGRESSION

FORECASTING USING LINEAR REGRESSION

Step 1	Define two helper functions named clear_create_canvas() and redraw_canvas() .
	<pre> 1 def clear_create_canvas(self): 2 self.widgetPlot1.canvas.figure.clf() 3 self.widgetPlot1.canvas.axis1 = 4 self.widgetPlot1.canvas.figure.add_subplot(111,facecolor = 5 '#efc0fe') 6 7 self.widgetPlot2.canvas.figure.clf() 8 self.widgetPlot2.canvas.axis2 = 9 self.widgetPlot2.canvas.figure.add_subplot(111,facecolor = 10 '#efc0fe') 11 12 self.widgetPlot3.canvas.figure.clf() 13 self.widgetPlot3.canvas.axis3 = 14 self.widgetPlot3.canvas.figure.add_subplot(421,facecolor = 15 '#efc0fe') 16 self.widgetPlot3.canvas.axis4 = 17 self.widgetPlot3.canvas.figure.add_subplot(422,facecolor = </pre>

```

18 '#efc0fe')
19     self.widgetPlot3.canvas.axis5 =
20 self.widgetPlot3.canvas.figure.add_subplot(423,facecolor =
21 '#efc0fe')
22     self.widgetPlot3.canvas.axis6 =
23 self.widgetPlot3.canvas.figure.add_subplot(424,facecolor =
24 '#efc0fe')
25     self.widgetPlot3.canvas.axis7 =
26 self.widgetPlot3.canvas.figure.add_subplot(425,facecolor =
27 '#efc0fe')
28     self.widgetPlot3.canvas.axis8 =
29 self.widgetPlot3.canvas.figure.add_subplot(426,facecolor =
30 '#efc0fe')
31     self.widgetPlot3.canvas.axis9 =
32 self.widgetPlot3.canvas.figure.add_subplot(427,facecolor =
33 '#efc0fe')
34     self.widgetPlot3.canvas.axis10 =
35 self.widgetPlot3.canvas.figure.add_subplot(428,facecolor =
 '#efc0fe')

    def redraw_canvas(self):
        self.widgetPlot3.canvas.figure.tight_layout()
        self.widgetPlot2.canvas.figure.tight_layout()
        self.widgetPlot1.canvas.figure.tight_layout()
        self.widgetPlot3.canvas.draw()
        self.widgetPlot2.canvas.draw()
        self.widgetPlot1.canvas.draw()

```

Let's explain each method in detail:

clear_create_canvas() function:

1. This method is responsible for clearing and creating the canvas for plotting.
2. The current figure of **widgetPlot1** is cleared using **self.widgetPlot1.canvas.figure.clf()**.
3. A new subplot is added to the cleared figure of **widgetPlot1** using **self.widgetPlot1.canvas.figure.add_subplot(111, facecolor='#efc0fe')**. The subplot is assigned to **self.widgetPlot1.canvas.axis1**.
4. Similar to **widgetPlot1**, the current figure of **widgetPlot2** is cleared using **self.widgetPlot2.canvas.figure.clf()**.
5. A new subplot is added to the cleared figure of **widgetPlot2** using **self.widgetPlot2.canvas.figure.add_subplot(111, facecolor='#efc0fe')**. The subplot is assigned to **self.widgetPlot2.canvas.axis2**.
6. The current figure of **widgetPlot3** is cleared using **self.widgetPlot3.canvas.figure.clf()**.
7. Multiple subplots are added to the cleared figure of **widgetPlot3** using **self.widgetPlot3.canvas.figure.add_subplot()** with different indexes and background colors. There are 10 subplots added to **widgetPlot3**, numbered from 421 to 428. Each subplot is assigned to a corresponding **self.widgetPlot3.canvas.axisX**, where X ranges from 3 to 10.

redraw_canvas() function:

1. This method is responsible for updating and redrawing the canvas after the plotting is completed.

	<p>2. The tight_layout() method is called on the figure associated with widgetPlot3 to adjust the subplot spacing and ensure they fit within the figure area more neatly.</p> <p>3. Similarly, the tight_layout() method is called on the figure associated with widgetPlot2 to adjust the subplot spacing.</p> <p>4. The tight_layout() method is called on the figure associated with widgetPlot1 to adjust the subplot spacing.</p> <p>5. The canvas associated with widgetPlot3 is redrawn using self.widgetPlot3.canvas.draw(), reflecting the changes made and displaying the updated figure with the subplots.</p> <p>6. Similarly, the canvas associated with widgetPlot2 is redrawn using self.widgetPlot2.canvas.draw().</p> <p>7. The canvas associated with widgetPlot1 is redrawn using self.widgetPlot1.canvas.draw().</p> <p>8. In summary, these methods provide a convenient way to clear and create the canvas for plotting, and then redraw the canvas after the plotting is complete. This ensures that the figures and subplots are properly updated and displayed in the application's GUI. The background color of the subplots is set to '#efc0fe' for all three canvas widgets (widgetPlot1, widgetPlot2, widgetPlot3).</p>
Step 2	<p>Define choose_forecasting() to read currentText property of cbForecasting widget and act accordingly:</p> <pre> 1 def choose_forecasting(self): 2 strCB = self.cbForecasting.currentText() 3 4 if strCB == "Linear Regression": 5 #Clears and creates canvas 6 self.clear_create_canvas() 7 8 lin_reg = LinearRegression() 9 self.perform_regression(lin_reg, self.X_final, 10 self.y_final, \ 11 self.X_train, self.y_train, self.X_test, 12 self.y_test, \ 13 self.X_val, self.y_val, "Linear Regression", 14 "Adj Close", \ 15 self.widgetPlot1.canvas.axis1, 16 self.widgetPlot2.canvas.axis2, \ 17 self.widgetPlot3.canvas.axis3, \ 18 self.widgetPlot3.canvas.axis4, 19 self.widgetPlot3.canvas.axis5, \ 20 self.widgetPlot3.canvas.axis6, 21 self.widgetPlot3.canvas.axis7, \ 22 self.widgetPlot3.canvas.axis8, 23 self.widgetPlot3.canvas.axis9, 24 self.widgetPlot3.canvas.axis10) 25 26 27 #Redraws canvas 28 self.redraw_canvas() </pre> <p>It is responsible for handling the selection of a forecasting method. Here is an explanation of what the method does:</p> <ol style="list-style-type: none"> 1. Retrieve the selected forecasting method:

2. The method retrieves the selected forecasting method from a combo box widget and assigns it to the variable **strCB** using **self.cbForecasting.currentText()**. The variable **strCB** will hold the selected method as a string.
3. Perform Linear Regression:
- If the selected forecasting method is "Linear Regression", the following steps are executed:
 - The method **clear_create_canvas(self)** is called to clear and create the canvas for plotting.
 - An instance of the **LinearRegression** model is created and assigned to the variable **lin_reg**.
 - The method **perform_regression()** is called with the necessary parameters, including the **lin_reg** model, input features (**self.X_final**), target variable (**self.y_final**), training, test, and validation datasets (**self.X_train**, **self.y_train**, **self.X_test**, **self.y_test**, **self.X_val**, **self.y_val**), label ("Linear Regression"), feature ("Adj Close"), and the respective plot axes (**self.widgetPlot1.canvas.axis1**, **self.widgetPlot2.canvas.axis2**, **self.widgetPlot3.canvas.axis3**, **self.widgetPlot3.canvas.axis4**, **self.widgetPlot3.canvas.axis5**, **self.widgetPlot3.canvas.axis6**, **self.widgetPlot3.canvas.axis7**, **self.widgetPlot3.canvas.axis8**, **self.widgetPlot3.canvas.axis9**, **self.widgetPlot3.canvas.axis10**).
 - The **perform_regression()** method performs the regression analysis and generates the necessary visualizations and evaluation metrics.
 - The method **redraw_canvas(self)** is called to redraw the canvas and display the generated plots.

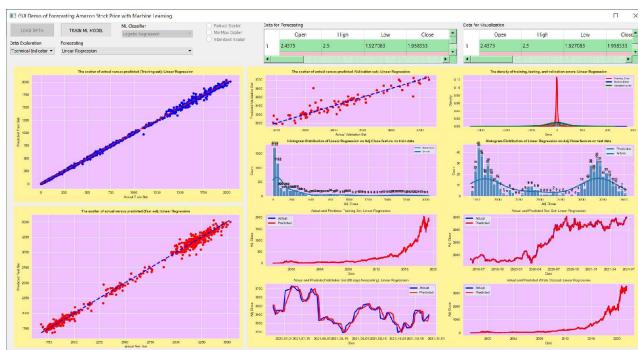


Figure 5.1 Showing forecasting results using Linear Regression

By selecting "Linear Regression" as the forecasting method, the code executes the necessary steps to perform linear regression, including clearing the canvas, creating the regression model, analyzing and evaluating the model's performance, and displaying the visualizations.

Step 3 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Linear Regression** item from **cbForecasting** widget. You will see the result as shown in Figure 5.1.

FORECASTING USING RANDOM REGRESSION

FORECASTING USING RANDOM REGRESSION

Step 1 Add this code to the end of **choose_forecasting()** method:

```

1      if strCB == "Random Forest Regression":
2          #Clears and creates canvas
3          self.clear_create_canvas()
4
5          rf_reg = RandomForestRegressor()
6          self.perform_regression(rf_reg, self.X_final,
7          self.y_final, \
8              self.X_train, self.y_train, self.X_test,
9              self.y_test, \
10             self.X_val, self.y_val, "RF Regression",
11             "Adj Close", \
12             self.widgetPlot1.canvas.axis1,
13             self.widgetPlot2.canvas.axis2, \
14             self.widgetPlot3.canvas.axis3, \
15             self.widgetPlot3.canvas.axis4,
16             self.widgetPlot3.canvas.axis5, \
17             self.widgetPlot3.canvas.axis6,
self.widgetPlot3.canvas.axis7, \
self.widgetPlot3.canvas.axis8,
self.widgetPlot3.canvas.axis9,
self.widgetPlot3.canvas.axis10)
18
19          #Redraws canvas
20          self.redraw_canvas()

```

It handles the case when the selected forecasting method is "Random Forest Regression". Here's an explanation of what this part does:

1. Perform Random Forest Regression:
 - If the selected forecasting method is "Random Forest Regression", the following steps are executed:
 - The method **clear_create_canvas(self)** is called to clear and create the canvas for plotting.
 - An instance of the **RandomForestRegressor** model is created and assigned to the variable **rf_reg**.

- The method **perform_regression()** is called with the necessary parameters, including the **rf_reg** model, input features (**self.X_final**), target variable (**self.y_final**), training, test, and validation datasets (**self.X_train**, **self.y_train**, **self.X_test**, **self.y_test**, **self.X_val**, **self.y_val**), label ("RF Regression"), feature ("Adj Close"), and the respective plot axes (**self.widgetPlot1.canvas.axis1**, **self.widgetPlot2.canvas.axis2**, **self.widgetPlot3.canvas.axis3**, **self.widgetPlot3.canvas.axis4**, **self.widgetPlot3.canvas.axis5**, **self.widgetPlot3.canvas.axis6**, **self.widgetPlot3.canvas.axis7**, **self.widgetPlot3.canvas.axis8**, **self.widgetPlot3.canvas.axis9**, **self.widgetPlot3.canvas.axis10**).
- The **perform_regression()** method performs the regression analysis using the random forest regression model and generates the necessary visualizations and evaluation metrics.
- The method **redraw_canvas(self)** is called to redraw the canvas and display the generated plots.

By selecting "Random Forest Regression" as the forecasting method, the code executes the necessary steps to perform random forest regression, including clearing the canvas, creating the regression model, analyzing and evaluating the model's performance, and displaying the visualizations.

Step 2 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Random Forest Regression** item from **cbForecasting** widget. You will see the result as shown in Figure 5.2.

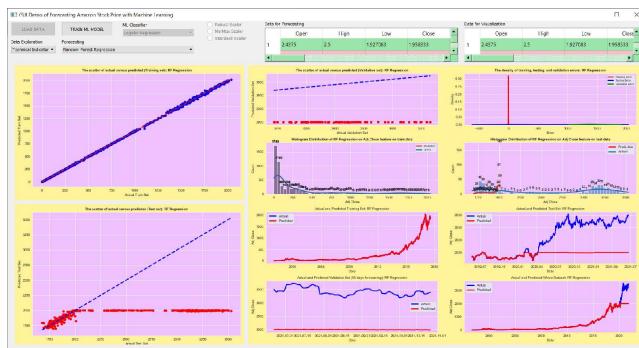


Figure 5.2 Showing forecasting results using Random Forest Regression

FORECASTING USING DECISION TREE REGRESSION

FORECASTING USING DECISION TREE REGRESSION

Step 1 Add this code to the end of **choose_forecasting()** method:

```
1     if strCB == "Decision Tree Regression":  
2         #Clears and creates canvas  
3         self.clear_create_canvas()  
4  
5         dt_reg = DecisionTreeRegressor(random_state=100)  
6         self.perform_regression(dt_reg, self.X_final,  
7         self.y_final, \  
8         self.X_train, self.y_train, self.X_test,  
9         self.y_test, \  
10        self.X_val, self.y_val, "DT Regression",  
11        "Adj Close", \  
12        self.widgetPlot1.canvas.axis1,  
13        self.widgetPlot2.canvas.axis2, \  
14        self.widgetPlot3.canvas.axis3,\  
15        self.widgetPlot3.canvas.axis4,  
16        self.widgetPlot3.canvas.axis5, \  
17        self.widgetPlot3.canvas.axis6,  
self.widgetPlot3.canvas.axis7, \  
self.widgetPlot3.canvas.axis8,  
self.widgetPlot3.canvas.axis9, \  
self.widgetPlot3.canvas.axis10)  
  
        #Redraws canvas  
        self.redraw_canvas()
```

It handles the case when the selected forecasting method is "Decision Tree Regression". Here's a breakdown of the code:

1. Clear and Create Canvas:

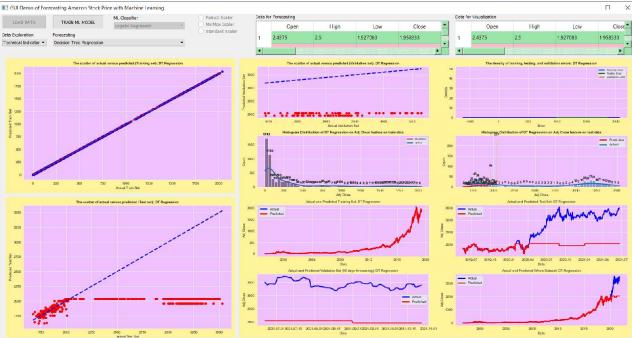
The **clear_create_canvas(self)** function is called to clear the canvas and prepare it for new plots.

2. Decision Tree Regression Model:

- An instance of the **DecisionTreeRegressor** model is created and assigned to the variable **dt_reg**.
- The **random_state** parameter is set to 100 for reproducibility.

3. Perform Regression and Visualization:

- The **perform_regression()** method is called with the following parameters:
 - **dt_reg**: The decision tree regression model instance.
 - Input features (**self.X_final**) and target variable (**self.y_final**).
 - Training, test, and validation datasets: **self.X_train**, **self.y_train**, **self.X_test**, **self.y_test**, **self.X_val**, **self.y_val**.
 - Label: "DT Regression".
 - Feature: "Adj Close".
 - Plot axes for visualizations: **self.widgetPlot1.canvas.axis1**, **self.widgetPlot2.canvas.axis2**, **self.widgetPlot3.canvas.axis3**, **self.widgetPlot3.canvas.axis4**, **self.widgetPlot3.canvas.axis5**,

	<pre>self.widgetPlot3.canvas.axis6, self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, self.widgetPlot3.canvas.axis10.</pre> <p>4. Adjust Plot Layout: The tight_layout() method is called on each of the three canvas figures (self.widgetPlot3.canvas.figure, self.widgetPlot2.canvas.figure, self.widgetPlot1.canvas.figure) to ensure proper spacing and layout of the plots.</p> <p>5. Redraw Canvas: The draw() method is called on each of the canvas figures (self.widgetPlot3.canvas, self.widgetPlot2.canvas, self.widgetPlot1.canvas) to update and display the generated plots on the canvas.</p> <p>In summary, this code segment handles the "Decision Tree Regression" case by creating the model, performing regression analysis, generating visualizations, adjusting the plot layout, and redrawing the canvas to display the updated plots.</p>
Step 2	<p>Run gui_amazon.py and click LOAD DATA and TRAIN ML MODEL buttons. Then, choose Decision Tree Regression item from cbForecasting widget. You will see the result as shown in Figure 5.3.</p>  <p>Figure 5.3 Showing forecasting results using Decision Tree Regression</p>

FORECASTING USING K-NEAREST NEIGHBORS REGRESSION

FORECASTING USING K-NEAREST NEIGHBORS REGRESSION

Step 1	Add this code to the end of choose_forecasting() method:
	<pre>1 if strCB == "KNN Regression": 2 #Clears and creates canvas 3 self.clear_create_canvas() 4 5 knn_reg = KNeighborsRegressor(n_neighbors=7) 6 self.perform_regression(knn_reg, self.X_final, 7 self.y_final, \ 8 self.X_train, self.y_train, self.X_test, 9 self.y_test, \</pre>

```

10         self.X_val, self.y_val, "KNN Regression",
11     "Adj Close", \
12         self.widgetPlot1.canvas.axis1,
13 self.widgetPlot2.canvas.axis2, \
14         self.widgetPlot3.canvas.axis3,\ 
15         self.widgetPlot3.canvas.axis4,
16 self.widgetPlot3.canvas.axis5, \
17         self.widgetPlot3.canvas.axis6,\ 
18         self.widgetPlot3.canvas.axis7,
self.widgetPlot3.canvas.axis8, \
19         self.widgetPlot3.canvas.axis9,\ 
20         self.widgetPlot3.canvas.axis10)

#Redraws canvas
self.redraw_canvas()

```

It handles the case when the selected forecasting method is "KNN Regression". Here's an explanation of the code:

1. Clear and Create Canvas:

The **clear_create_canvas(self)** function is called to clear the canvas and prepare it for new plots.

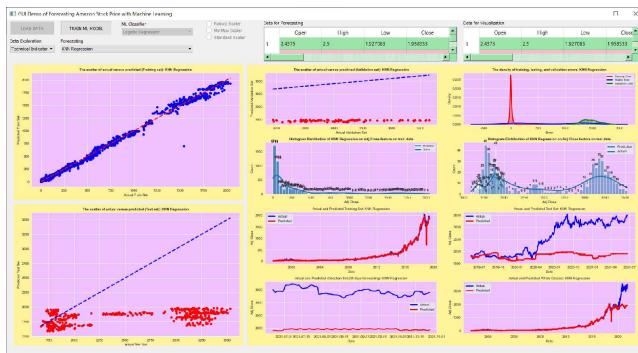


Figure 5.4 Showing forecasting results using KNN Regression

2. KNN Regression Model:

- An instance of the **KNeighborsRegressor** model is created and assigned to the variable **knn_reg**.
- The **n_neighbors** parameter is set to 7, specifying the number of neighbors to consider for regression.

3. Perform Regression and Visualization:

- The **perform_regression()** method is called with the following parameters:
 - **knn_reg**: The KNN regression model instance.
 - Input features (**self.X_final**) and target variable (**self.y_final**).
 - Training, test, and validation datasets: **self.X_train**, **self.y_train**, **self.X_test**, **self.y_test**, **self.X_val**, **self.y_val**.
 - Label: "KNN Regression".
 - Feature: "Adj Close".
 - Plot axes for visualizations: **self.widgetPlot1.canvas.axis1**, **self.widgetPlot2.canvas.axis2**, **self.widgetPlot3.canvas.axis3**, **self.widgetPlot3.canvas.axis4**,

	<pre>self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, self.widgetPlot3.canvas.axis10.</pre> <p>4. Redraw Canvas:</p> <p>The redraw_canvas(self) function is called to redraw the canvas, displaying the updated plots.</p> <p>In summary, this code segment handles the "KNN Regression" case by creating the model, performing regression analysis, generating visualizations, and redrawing the canvas to display the updated plots.</p>
Step 2	Run gui_amazon.py and click LOAD DATA and TRAIN ML MODEL buttons. Then, choose KNN Regression item from cbForecasting widget. You will see the result as shown in Figure 5.4.

FORECASTING USING ADABOOST REGRESSION

FORECASTING USING ADABOOST REGRESSION

Step 1 Add this code to the end of **choose_forecasting()** method:

```
1      if strCB == "Adaboost Regression":
2          #Clears and creates canvas
3          self.clear_create_canvas()
4
5          ada_reg = AdaBoostRegressor(random_state=100,
6          n_estimators=200)
7          self.perform_regression(ada_reg, self.X_final,
8          self.y_final, \
9              self.X_train, self.y_train, self.X_test,
10             self.y_test, \
11             self.X_val, self.y_val, "Adaboost
12 Regression", "Adj Close",\
13             self.widgetPlot1.canvas.axis1,
14             self.widgetPlot2.canvas.axis2, \
15             self.widgetPlot3.canvas.axis3,\ \
16             self.widgetPlot3.canvas.axis4,
17             self.widgetPlot3.canvas.axis5, \
18             self.widgetPlot3.canvas.axis6,\ \
19             self.widgetPlot3.canvas.axis7,
self.widgetPlot3.canvas.axis8, \
20             self.widgetPlot3.canvas.axis9,\ \
21             self.widgetPlot3.canvas.axis10)
22
23
24      #Redraws canvas
25      self.redraw_canvas()
```

It handles the case when the selected forecasting method is "Adaboost Regression". Here's an explanation of the code:

1. Clear and Create Canvas:

The **clear_create_canvas(self)** function is called to clear the canvas and prepare it for new plots.

2. Adaboost Regression Model:

- An instance of the **AdaBoostRegressor** model is created and assigned to the variable

ada_reg

- The **random_state** parameter is set to 100, providing a random seed for reproducibility.
- The **n_estimators** parameter is set to 200, specifying the number of boosting stages to perform.

3. Perform Regression and Visualization:

The **perform_regression()** method is called with the following parameters:

- **ada_reg**: The Adaboost regression model instance.
- Input features (`self.X_final`) and target variable (`self.y_final`).
- Training, test, and validation datasets: `self.X_train`, `self.y_train`, `self.X_test`, `self.y_test`, `self.X_val`, `self.y_val`.
- Label: "Adaboost Regression".
- Feature: "Adj Close".
- Plot axes for visualizations: `self.widgetPlot1.canvas.axis1`, `self.widgetPlot2.canvas.axis2`, `self.widgetPlot3.canvas.axis3`, `self.widgetPlot3.canvas.axis4`, `self.widgetPlot3.canvas.axis5`, `self.widgetPlot3.canvas.axis6`, `self.widgetPlot3.canvas.axis7`, `self.widgetPlot3.canvas.axis8`, `self.widgetPlot3.canvas.axis9`, `self.widgetPlot3.canvas.axis10`.

4. Redraw Canvas:

The **redraw_canvas(self)** function is called to redraw the canvas, displaying the updated plots.

In summary, this code segment handles the "Adaboost Regression" case by creating the model, performing regression analysis using Adaboost, generating visualizations, and redrawing the canvas to display the updated plots.

Step 2 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Adaboost Regression** item from **cbForecasting** widget. You will see the result as shown in Figure 5.5.

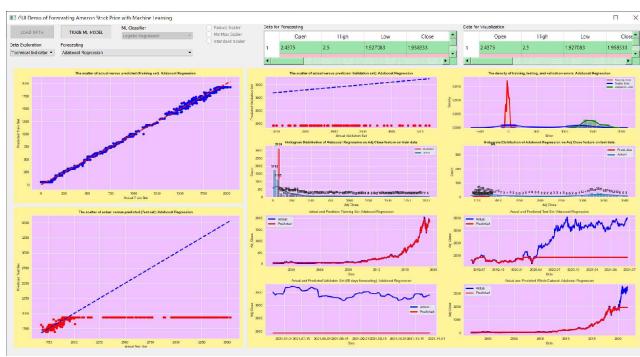


Figure 5.5 Showing forecasting results using Adaboost Regression

FORECASTING USING GRADIENT BOOSTING REGRESSION

FORECASTING USING GRADIENT BOOSTING REGRESSION

Step 1 Add this code to the end of **choose_forecasting()** method:

```

1      if strCB == "Gradient Boosting Regression":
2          #Clears and creates canvas
3          self.clear_create_canvas()
4
5          gb_reg =
6          GradientBoostingRegressor(random_state=100)
7          self.perform_regression(gb_reg, self.X_final,
8          self.y_final, \
9              self.X_train, self.y_train, self.X_test,
10             self.y_test, \
11                 self.X_val, self.y_val, "GB Regression",
12                 "Adj Close", \
13                     self.widgetPlot1.canvas.axis1,
14                     self.widgetPlot2.canvas.axis2, \
15                         self.widgetPlot3.canvas.axis3, \
16                             self.widgetPlot3.canvas.axis4,
17                             self.widgetPlot3.canvas.axis5, \
18                                 self.widgetPlot3.canvas.axis6, \
19                                     self.widgetPlot3.canvas.axis7,
self.widgetPlot3.canvas.axis8, \
self.widgetPlot3.canvas.axis9, \
self.widgetPlot3.canvas.axis10)
20
21
#Redraws canvas
22 self.redraw_canvas()

```

It handles the case when the selected forecasting method is "Gradient Boosting Regression". Here's an explanation of the code:

1. Clear and Create Canvas:

The **clear_create_canvas(self)** function is called to clear the canvas and prepare it for new plots.

2. Gradient Boosting Regression Model:

- An instance of the **GradientBoostingRegressor** model is created and assigned to the variable **gb_reg**.
- The **random_state** parameter is set to 100, providing a random seed for reproducibility.

3. Perform Regression and Visualization:

The **perform_regression()** method is called with the following parameters:

- **gb_reg**: The Gradient Boosting regression model instance.
- Input features (**self.X_final**) and target variable (**self.y_final**).
- Training, test, and validation datasets: **self.X_train**, **self.y_train**, **self.X_test**, **self.y_test**, **self.X_val**, **self.y_val**.
- Label: "GB Regression".
- Feature: "Adj Close".
- Plot axes for visualizations: **self.widgetPlot1.canvas.axis1**,

	<pre>self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, self.widgetPlot3.canvas.axis10.</pre> <p>4. Redraw Canvas:</p> <p>The redraw_canvas(self) function is called to redraw the canvas, displaying the updated plots.</p> <p>In summary, this code segment handles the "Gradient Boosting Regression" case by creating the model, performing regression analysis using Gradient Boosting, generating visualizations, and redrawing the canvas to display the updated plots.</p>
Step 2	<p>Run gui_amazon.py and click LOAD DATA and TRAIN ML MODEL buttons. Then, choose Gradient Boosting Regression item from cbForecasting widget. You will see the result as shown in Figure 5.6.</p> <p>Figure 5.6 Showing forecasting results using Gradient Boosting Regression</p>

FORECASTING USING EXTREME GRADIENT BOOSTING REGRESSION

FORECASTING USING EXTREME GRADIENT BOOSTING REGRESSION

Step 1	Add this code to the end of choose_forecasting() method:
	<pre>1 if strCB == "XGB Regression": 2 #Clears and creates canvas 3 self.clear_create_canvas() 4 5 xgb_reg = XGBRegressor(random_state=100) 6 self.perform_regression(xgb_reg, self.X_final, 7 self.y_final, \ 8 self.X_train, self.y_train, self.X_test, 9 self.X_val, self.y_val, "XGB Regression", 10 "Adj Close",\ 11 self.widgetPlot1.canvas.axis1, 12 self.widgetPlot2.canvas.axis2, \ 13 self.widgetPlot3.canvas.axis3,\</pre>

```

16         self.widgetPlot3.canvas.axis4,
17     self.widgetPlot3.canvas.axis5, \
18         self.widgetPlot3.canvas.axis6, \
self.widgetPlot3.canvas.axis7, \
        self.widgetPlot3.canvas.axis8, \
        self.widgetPlot3.canvas.axis9, \
        self.widgetPlot3.canvas.axis10)

#Redraws canvas
self.redraw_canvas()

```

Here's a step-by-step explanation of the code:

1. The code checks if the value of the variable strCB is equal to "XGB Regression".
2. If the condition is true, the **clear_create_canvas(self)** function is called. This function clears and creates canvas objects for plotting.
3. An instance of the **XGBRegressor** class is created with the random_state parameter set to 100.
4. The **perform_regression()** function is called with the following arguments:
 - xgb_reg: The XGBRegressor model instance.
 - self.X_final: The input features for the entire dataset.
 - self.y_final: The target variable for the entire dataset.
 - self.X_train: The input features for the training set.
 - self.y_train: The target variable for the training set.
 - self.X_test: The input features for the test set.
 - self.y_test: The target variable for the test set.
 - self.X_val: The input features for the validation set.
 - self.y_val: The target variable for the validation set.
 - "XGB Regression": A label for the regression model.
 - "Adj Close": The feature label.
 - self.widgetPlot1.canvas.axis1 to self.widgetPlot3.canvas.axis10: Axes objects for plotting the results.
5. The **perform_regression()** function performs the regression analysis using the given model and data, prints various evaluation metrics and predictions, and creates scatter plots, histograms, and line plots to visualize the results.

After the regression analysis is performed and plotted, the **redraw_canvas(self)** function is called. This function adjusts the layout of the canvas objects and redraws them.

Step 2 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **XGB Regression** item from **cbForecasting** widget. You will see the result as shown in Figure 5.7.



Figure 5.7 Showing forecasting results using XGB Regression

FORECASTING USING LIGHT GRADIENT BOOSTING MACHINE REGRESSION

FORECASTING USING LIGHT GRADIENT BOOSTING MACHINE REGRESSION

Step 1 Add this code to the end of **choose_forecasting()** method:

```

1  if strCB == "LGBM Regression":
2      #Clears and creates canvas
3      self.clear_create_canvas()
4
5      lgbm_reg = LGBMRegressor(random_state=100)
6      self.perform_regression(lgbm_reg, self.X_final,
7      self.y_final, \
8          self.X_train, self.y_train, \
9          self.X_test, self.y_test, self.X_val,
10         self.y_val, \
11         "LGBM Regression", "Adj Close", \
12         self.widgetPlot1.canvas.axis1,
13         self.widgetPlot2.canvas.axis2, \
14         self.widgetPlot3.canvas.axis3, \
15         self.widgetPlot3.canvas.axis4,
16         self.widgetPlot3.canvas.axis5, \
17         self.widgetPlot3.canvas.axis6, \
18         self.widgetPlot3.canvas.axis7, \
19         self.widgetPlot3.canvas.axis8, \
20         self.widgetPlot3.canvas.axis9, \
21         self.widgetPlot3.canvas.axis10)
22
23
24      #Redraws canvas
25      self.redraw_canvas()

```

This code block is similar to the previous one, but it checks if the value of the variable **strCB** is equal to "LGBM Regression". If the condition is true, it performs regression analysis using the LightGBM (**LGBMRegressor**) model. Here's a breakdown of the steps:

1. The code checks if the value of the variable **strCB** is equal to "LGBM Regression".
2. If the condition is true, the **clear_create_canvas(self)** function is called. This function clears and creates canvas objects for plotting.
3. An instance of the **LGBMRegressor** class is created with the `random_state` parameter set to 100.
4. The **perform_regression()** function is called with the following arguments:

- `lgbm_reg`: The `LGBMRegressor` model instance.
- `self.X_final`: The input features for the entire dataset.
- `self.y_final`: The target variable for the entire dataset.
- `self.X_train`: The input features for the training set.
- `self.y_train`: The target variable for the training set.
- `self.X_test`: The input features for the test set.
- `self.y_test`: The target variable for the test set.
- `self.X_val`: The input features for the validation set.
- `self.y_val`: The target variable for the validation set.
- "LGBM Regression": A label for the regression model.
- "Adj Close": The feature label.
- `self.widgetPlot1.canvas.axis1` to `self.widgetPlot3.canvas.axis10`: Axes objects for plotting the results.

5. The `perform_regression()` function performs the regression analysis using the given model and data, prints various evaluation metrics and predictions, and creates scatter plots, histograms, and line plots to visualize the results.

After the regression analysis is performed and plotted, the `redraw_canvas(self)` function is called. This function adjusts the layout of the canvas objects and redraws them.

Step 2 Run `gui_amazon.py` and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **LGBM Regression** item from **cbForecasting** widget. You will see the result as shown in Figure 5.8.

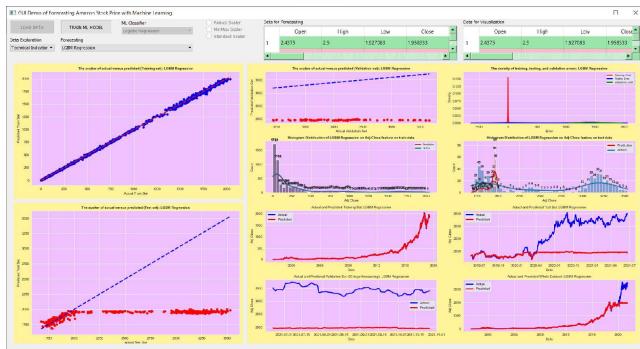


Figure 5.8 Showing forecasting results using LGBM Regression

FORECASTING USING CATBOOST REGRESSION

Step 1 Add this code to the end of **choose_forecasting()** method:

```

1      if strCB == "Catboost Regression":
2          #Clears and creates canvas
3          self.clear_create_canvas()
4
5          cb_reg = CatBoostRegressor(random_state=100)
6          self.perform_regression(cb_reg, self.X_final,
7          self.y_final, \
8              self.X_train, self.y_train, \
9              self.X_test, self.y_test, self.X_val,
10             self.y_val, \
11             "Catboost Regression", "Adj Close", \
12             self.widgetPlot1.canvas.axis1,
13             self.widgetPlot2.canvas.axis2, \
14             self.widgetPlot3.canvas.axis3, \
15             self.widgetPlot3.canvas.axis4,
16             self.widgetPlot3.canvas.axis5, \
17             self.widgetPlot3.canvas.axis6, \
18             self.widgetPlot3.canvas.axis7,
19             self.widgetPlot3.canvas.axis8, \
20             self.widgetPlot3.canvas.axis9, \
21             self.widgetPlot3.canvas.axis10)
22
23
24      #Redraws canvas
25      self.redraw_canvas()

```

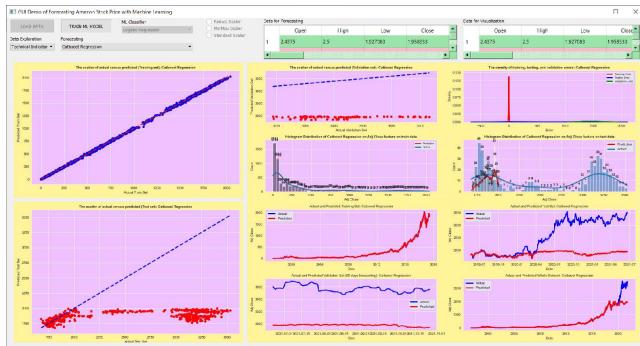


Figure 5.9 Showing forecasting results using Catboost Regression

This code block is another conditional statement that checks if the value of the variable **strCB** is equal to "Catboost Regression". If the condition is true, it performs regression analysis using the CatBoost (**CatBoostRegressor**) model. Here's a breakdown of the steps:

1. The code checks if the value of the variable **strCB** is equal to "Catboost Regression".
2. If the condition is true, the **clear_create_canvas(self)** function is called. This function clears and creates canvas objects for plotting.
3. An instance of the **CatBoostRegressor** class is created with the `random_state` parameter set to 100.
4. The **perform_regression()** function is called with the following arguments:

	<ul style="list-style-type: none"> • cb_reg: The CatBoostRegressor model instance. • self.X_final: The input features for the entire dataset. • self.y_final: The target variable for the entire dataset. • self.X_train: The input features for the training set. • self.y_train: The target variable for the training set. • self.X_test: The input features for the test set. • self.y_test: The target variable for the test set. • self.X_val: The input features for the validation set. • self.y_val: The target variable for the validation set. • "Catboost Regression": A label for the regression model. • "Adj Close": The feature label. • self.widgetPlot1.canvas.axis1 to self.widgetPlot3.canvas.axis10: Axes objects for plotting the results. <p>5. The perform_regression() function performs the regression analysis using the given model and data, prints various evaluation metrics and predictions, and creates scatter plots, histograms, and line plots to visualize the results.</p> <p>After the regression analysis is performed and plotted, the redraw_canvas(self) function is called. This function adjusts the layout of the canvas objects and redraws them.</p>
Step 2	Run gui_amazon.py and click LOAD DATA and TRAIN ML MODEL buttons. Then, choose Catboost Regression item from cbForecasting widget. You will see the result as shown in Figure 5.9.

FORECASTING USING SUPPORT VECTOR REGRESSION

FORECASTING USING SUPPORT VECTOR REGRESSION

Step 1	Add this code to the end of choose_forecasting() method:
	<pre> 1 if strCB == "SVR Regression": 2 #Clears and creates canvas 3 self.clear_create_canvas() 4 5 svm_reg = SVR() 6 self.perform_regression(svm_reg, self.X_final, 7 self.y_final, \ 8 self.X_train, self.y_train, \ 9 self.X_test, self.y_test, self.X_val, 10 self.y_val, \ 11 "SVR Regression", "Adj Close",\ 12 self.widgetPlot1.canvas.axis1, 13 self.widgetPlot2.canvas.axis2, \ 14 self.widgetPlot3.canvas.axis3,\</pre>

```

15         self.widgetPlot3.canvas.axis4,
16     self.widgetPlot3.canvas.axis5, \
17         self.widgetPlot3.canvas.axis6, \
18         self.widgetPlot3.canvas.axis7,
19     self.widgetPlot3.canvas.axis8, \
20         self.widgetPlot3.canvas.axis9, \
21         self.widgetPlot3.canvas.axis10)

#Redraws canvas
self.redraw_canvas()

```

This code block is another conditional statement that checks if the value of the variable `strCB` is equal to "SVR Regression". If the condition is true, it performs regression analysis using the Support Vector Regression (SVR) model. Here's a breakdown of the steps:

1. The code checks if the value of the variable **`strCB`** is equal to "SVR Regression".
2. If the condition is true, the **`clear_create_canvas(self)`** function is called. This function clears and creates canvas objects for plotting.
3. An instance of the SVR class is created without specifying any parameters, using the default settings.
4. The **`perform_regression()`** function is called with the following arguments:
 - `svm_reg`: The SVR model instance.
 - `self.X_final`: The input features for the entire dataset.
 - `self.y_final`: The target variable for the entire dataset.
 - `self.X_train`: The input features for the training set.
 - `self.y_train`: The target variable for the training set.
 - `self.X_test`: The input features for the test set.
 - `self.y_test`: The target variable for the test set.
 - `self.X_val`: The input features for the validation set.
 - `self.y_val`: The target variable for the validation set.
 - "SVR Regression": A label for the regression model.
 - "Adj Close": The feature label.
 - `self.widgetPlot1.canvas.axis1` to `self.widgetPlot3.canvas.axis10`: Axes objects for plotting the results.
5. The **`perform_regression()`** function performs the regression analysis using the given model and data, prints various evaluation metrics and predictions, and creates scatter plots, histograms, and line plots to visualize the results.

After the regression analysis is performed and plotted, the **`redraw_canvas(self)`** function is called. This function adjusts the layout of the canvas objects and redraws them.

Step 2	<p>Run gui_amazon.py and click LOAD DATA and TRAIN ML MODEL buttons. Then, choose SVR Regression item from cbForecasting widget. You will see the result as shown in Figure 5.10.</p> <p>Figure 5.10 Showing forecasting results using SVR Regression</p>
--------	---

FORECASTING USING MULTI LAYER PERCEPTRON REGRESSION

FORECASTING USING MULTI LAYER PERCEPTRON REGRESSION

Step 1	<p>Add this code to the end of choose_forecasting() method:</p> <pre> 1 if strCB == "MLP Regression": 2 #Clears and creates canvas 3 self.clear_create_canvas() 4 5 mlp_reg = MLPRegressor(random_state=100, 6 max_iter=1000) 7 self.perform_regression(mlp_reg, self.X_final, 8 self.y_final, \ 9 self.X_train, self.y_train, \ 10 self.X_test, self.y_test, self.X_val, 11 self.y_val, \ 12 "MLP Regression", "Adj Close",\ 13 self.widgetPlot1.canvas.axis1, 14 self.widgetPlot2.canvas.axis2, \ 15 self.widgetPlot3.canvas.axis3,\ 16 self.widgetPlot3.canvas.axis4,\ 17 self.widgetPlot3.canvas.axis5,\ 18 self.widgetPlot3.canvas.axis6,\ 19 self.widgetPlot3.canvas.axis7,\ self.widgetPlot3.canvas.axis8,\ self.widgetPlot3.canvas.axis9,\ self.widgetPlot3.canvas.axis10) 20 21 22 #Redraws canvas 23 self.redraw_canvas() </pre>
--------	--

Let's break down the code block in more detail:

1. The code checks if the value of the variable **strCB** is equal to the string "**MLP Regression**".
2. If the condition is true, the **clear_create_canvas(self)** function is called. This function is responsible for clearing and creating canvas objects used for plotting.

	<p>3. An instance of the MLPRegressor class is created and assigned to the variable mlp_reg. The MLPRegressor is a class from the scikit-learn library and represents the Multi-Layer Perceptron (MLP) Regression model. It takes two arguments:</p> <ul style="list-style-type: none"> • random_state=100: This sets the random seed to 100, ensuring reproducibility of the results. • max_iter=1000: This sets the maximum number of iterations for the training process of the MLPRegressor model. <p>4. The perform_regression() function is called with several arguments:</p> <ul style="list-style-type: none"> • mlp_reg: The MLPRegressor instance created in the previous step. • self.X_final, self.y_final: The input features and target variable for the entire dataset. • self.X_train, self.y_train: The input features and target variable for the training set. • self.X_test, self.y_test: The input features and target variable for the test set. • self.X_val, self.y_val: The input features and target variable for the validation set. • "MLP Regression": A label for the regression model. • "Adj Close": The label for the feature being predicted. • self.widgetPlot1.canvas.axis1 to self.widgetPlot3.canvas.axis10: These are axes objects associated with different plots, used for visualizing the results of the regression analysis. <p>5. The perform_regression() function is responsible for executing the regression analysis using the given model and data. It performs training, prediction, and evaluation steps, and creates various plots to visualize the results. It likely includes calculations of evaluation metrics such as mean squared error (MSE), mean absolute error (MAE), and R-squared value.</p> <p>Once the regression analysis is completed, the redraw_canvas(self) function is called. This function adjusts the layout of the canvas objects and redraws them, ensuring that the updated plots are displayed correctly.</p>
Step 2	Run gui_amazon.py and click LOAD DATA and TRAIN ML MODEL buttons. Then, choose MLP Regression item from cbForecasting widget. You will see the result as shown in Figure 5.11.

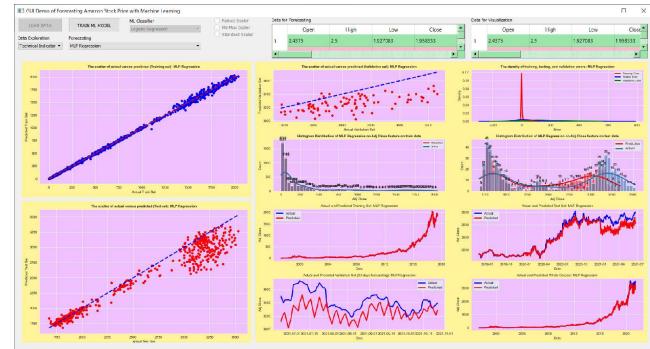


Figure 5.11 Showing forecasting results using MLP Regression

FORECASTING USING LASSO REGRESSION

FORECASTING USING LASSO REGRESSION

Step 1 Add this code to the end of **choose_forecasting()** method:

```

1      if strCB == "Lasso Regression":
2          #Clears and creates canvas
3          self.clear_create_canvas()
4
5          lasso_reg = LassoCV(n_alphas=1000,
6          max_iter=3000, random_state=0)
7          self.perform_regression(lasso_reg, self.X_final,
8          self.y_final,\n
9              self.X_train, self.y_train,\n
10             self.X_test, self.y_test, self.X_val,\n
11             self.y_val,\n
12             "Lasso Regression", "Adj Close",\n
13             self.widgetPlot1.canvas.axis1,\n
14             self.widgetPlot2.canvas.axis2,\n
15             self.widgetPlot3.canvas.axis3,\n
16             self.widgetPlot3.canvas.axis4,\n
17             self.widgetPlot3.canvas.axis5,\n
18             self.widgetPlot3.canvas.axis6,\n
19             self.widgetPlot3.canvas.axis7,\n
self.widgetPlot3.canvas.axis8,\n
self.widgetPlot3.canvas.axis9,\n
self.widgetPlot3.canvas.axis10)\n
\n
#Redraws canvas
self.redraw_canvas()

```

Let's break down the code block in more detail:

1. The code checks if the value of the variable `strCB` is equal to the string "Lasso Regression".
2. If the condition is true, the `clear_create_canvas(self)` function is called. This function clears and creates canvas objects used for plotting.
3. An instance of the `LassoCV` class is created and assigned to the variable `lasso_reg`. The `LassoCV` is a class from the scikit-learn library and represents the Lasso Regression model with cross-validation. It takes the following arguments:

	<ul style="list-style-type: none"> • n_alphas=1000: The number of alphas, or regularization strengths, to test during cross-validation. • max_iter=3000: The maximum number of iterations for the Lasso regression algorithm. • random_state=0: The random seed used for reproducibility of results. <p>4. The perform_regression() function is called with several arguments:</p> <ul style="list-style-type: none"> • lasso_reg: The LassoCV instance created in the previous step. • self.X_final, self.y_final: The input features and target variable for the entire dataset. • self.X_train, self.y_train: The input features and target variable for the training set. • self.X_test, self.y_test: The input features and target variable for the test set. • self.X_val, self.y_val: The input features and target variable for the validation set. • "Lasso Regression": A label for the regression model. • "Adj Close": The label for the feature being predicted. • self.widgetPlot1.canvas.axis1 to self.widgetPlot3.canvas.axis10: These are axes objects associated with different plots, used for visualizing the results of the regression analysis. <p>5. The perform_regression() function is responsible for executing the regression analysis using the given model and data. It performs training, prediction, and evaluation steps, and creates various plots to visualize the results. It likely includes calculations of evaluation metrics such as mean squared error (MSE), mean absolute error (MAE), and R-squared value.</p> <p>Once the regression analysis is completed, the redraw_canvas(self) function is called. This function adjusts the layout of the canvas objects and redraws them, ensuring that the updated plots are displayed correctly.</p>
Step 2	Run gui_amazon.py and click LOAD DATA and TRAIN ML MODEL buttons. Then, choose Lasso Regression item from cbForecasting widget. You will see the result as shown in Figure 5.12.

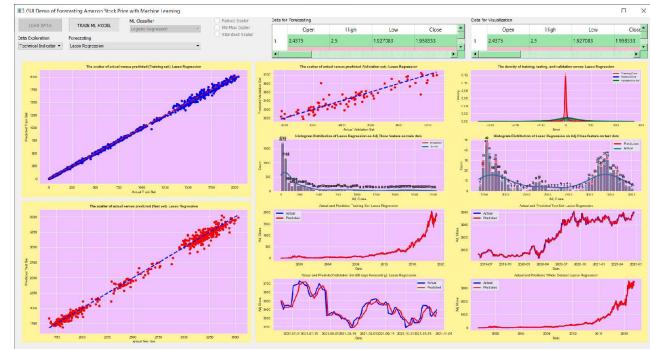


Figure 5.12 Showing forecasting results using Lasso Regression

FORECASTING USING RIDGE REGRESSION

FORECASTING USING RIDGE REGRESSION

Step 1 Add this code to the end of **choose_forecasting()** method:

```

1     if strCB == "Ridge Regression":
2         #Clears and creates canvas
3         self.clear_create_canvas()
4
5         ridge_reg = RidgeCV(gcv_mode='auto')
6         self.perform_regression(ridge_reg, self.X_final,
7         self.y_final, \
8             self.X_train, self.y_train, \
9             self.X_test, self.y_test, self.X_val,
10            self.y_val, \
11            "Ridge Regression", "Adj Close",\
12            self.widgetPlot1.canvas.axis1,
13            self.widgetPlot2.canvas.axis2, \
14            self.widgetPlot3.canvas.axis3,\ 
15            self.widgetPlot3.canvas.axis4,
16            self.widgetPlot3.canvas.axis5, \
17            self.widgetPlot3.canvas.axis6,\ 
18            self.widgetPlot3.canvas.axis7,
19            self.widgetPlot3.canvas.axis8, \
20            self.widgetPlot3.canvas.axis9,\ 
21            self.widgetPlot3.canvas.axis10)
22
23         #Redraws canvas
24         self.redraw_canvas()

```

Let's break down the code block in more detail:

1. The code checks if the value of the variable **strCB** is equal to the string "Ridge Regression".
2. If the condition is true, the **clear_create_canvas(self)** function is called. This function clears and creates canvas objects used for plotting.
3. An instance of the RidgeCV class is created and assigned to the variable **ridge_reg**. The RidgeCV is a class from the scikit-learn library and represents the Ridge Regression model with cross-validation. It takes the following argument:

gcv_mode='auto': The mode for generalized cross-validation. Setting it to 'auto' lets the model

automatically choose the best mode based on the problem size and other factors.

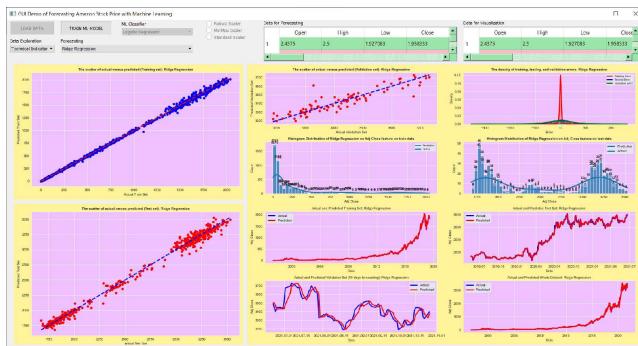


Figure 5.13 Showing forecasting results using Ridge Regression

4. The `perform_regression()` function is called with several arguments:

- ridge_reg: The RidgeCV instance created in the previous step.
 - self.X_final, self.y_final: The input features and target variable for the entire dataset.
 - self.X_train, self.y_train: The input features and target variable for the training set.
 - self.X_test, self.y_test: The input features and target variable for the test set.
 - self.X_val, self.y_val: The input features and target variable for the validation set.
 - "Ridge Regression": A label for the regression model.
 - "Adj Close": The label for the feature being predicted.
 - self.widgetPlot1.canvas.axis1 to self.widgetPlot3.canvas.axis10: These are axes objects associated with different plots, used for visualizing the results of the regression analysis.

The **perform_regression()** function is responsible for executing the regression analysis using the given model and data. It performs training, prediction, and evaluation steps, and creates various plots to visualize the results. It likely includes calculations of evaluation metrics such as mean squared error (MSE), mean absolute error (MAE), and R-squared value.

- Once the regression analysis is completed, the **redraw_canvas(self)** function is called. This function adjusts the layout of the canvas objects and redraws them, ensuring that the updated plots are displayed correctly.

Step 2 Run `gui_amazon.py` and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Ridge Regression** item.

from **cbForecasting** widget. You will see the result as shown in Figure 5.13.

**CREATING GUI
FOR
DAILY RETURN PREDICTION**

**CREATING GUI
FOR
DAILY RETURN PREDICTION**

IMPLEMENTING FEATURE SCALING AND SPLITTING DATA

IMPLEMENTING FEATURE SCALING AND SPLITTING DATA

Step 1	<p>Modify import_dataset() method to enable pbTrainML widget as shown in line 27:</p> <pre>1 def import_dataset(self): 2 curr_path = os.getcwd() 3 dataset_dir = curr_path + "/Amazon.csv" 4 5 self.read_dataset(dataset_dir) 6 print("Dataframe has been read...") 7 8 #Populates cbData and cbForecasting 9 self.populate_cbData() 10 self.populate_cbForecasting() 11 12 #Populates tables with data 13 self.populate_table(self.df, self.twData1) 14 self.label1.setText('Data for Forecasting') 15 16 self.populate_table(self.df_dummy, self.twData2) 17 self.label2.setText('Data for Visualization') 18 19 #Turns off pbLoad 20 self.pbLoad.setEnabled(False) 21 22 #Turns on cbForecasting and cbData 23 self.cbForecasting.setEnabled(True) 24 self.cbData.setEnabled(True) 25 26 #Turns on pbTrainML widget 27 self.pbTrainML.setEnabled(True)</pre>
Step 2	<p>Define split_data() to split dataset into train and test data with Robust Scaler, normalized, and standardized feature scaling. Here's a detailed step-by-step explanation of the split_data() function:</p> <ol style="list-style-type: none">1. Extract input and output variables: The function starts by calling the extract_data() method to extract the input features (X) and target variable (y) from the DataFrame

`self.df`. This step assumes that the `extract_data()` method is defined elsewhere in the code and returns `X` and `y`.

2. Resample data using SMOTE: To address class imbalance, the function uses the Synthetic Minority Over-sampling Technique (SMOTE). SMOTE generates synthetic samples for the minority class by interpolating between neighboring samples. This resampling technique helps balance the classes in the dataset and improve the performance of machine learning models. The SMOTE object is created with a specific random state for reproducibility. Then, the `fit_resample()` method is called on the input features `X` and target variable `y` to generate resampled data, which is assigned back to `X` and `y`.
3. Split data into training and testing sets: The resampled data (`X` and `y`) is split into separate training and testing sets using the `train_test_split()` function from the scikit-learn library. This function randomly divides the data into two subsets based on the specified test size. The test size determines the proportion of the data allocated for testing, while the remaining data is used for training. The function also shuffles the data before splitting (`shuffle=True`) and ensures that the class distribution is preserved in the splits by stratifying them based on the target variable `y`. The split data is assigned to `X_train`, `X_test`, `y_train`, and `y_test` variables.
4. Normalize data with robust scaler: The training and testing data are normalized using the Robust Scaler. The **RobustScaler** is an object from scikit-learn that scales features using statistics that are robust to outliers. First, a **RobustScaler** object is created. Then, copies of the training and testing data (`X_train` and `X_test`) as well as the corresponding target variables (`y_train` and `y_test`) are created. These copies are used to preserve the original data for future reference. The `fit_transform` method is called on `X_train` using the `rob_scaler` object to compute the scaling parameters based on the training data and transform the training data accordingly. The `transform` method is then used on `X_test` to transform the testing data using the same scaling parameters. The normalized training data is assigned to `self.X_train_rob`, and the normalized testing data is assigned to `self.X_test_rob`.
5. Save preprocessed data: The preprocessed data, including `self.X_train_rob`, `self.X_test_rob`, `self.y_train_rob`, and `self.y_test_rob`, is saved into separate pickle files using the `joblib.dump` function. Pickle files are a binary format used to store Python objects persistently, allowing the data to be loaded and used later without recomputing or preprocessing it again. The data is saved with meaningful file names, such as '`X_train_rob.pkl`', '`X_test_rob.pkl`',

'y_train_rob.pkl', and 'y_test_rob.pkl', for easy identification and retrieval.

6. Repeat normalization with different scalers: The data normalization process is repeated with two additional scalers: MinMax Scaler and Standard Scaler. Similar to the previous step, copies of the training and testing data are created to preserve the original data. The training data is then normalized using the corresponding scaler, and the testing data is transformed using the same scaling parameters. The normalized training data and transformed testing data are assigned to different variables (self.X_train_norm, self.X_test_norm, self.X_train_stand, self.X_test_stand). These preprocessed data are also saved into separate pickle files using joblib.dump.

By performing these steps, the **split_data()** function ensures that the data is properly resampled, split into training and testing sets, and normalized using different scaling techniques. The preprocessed data is then saved into pickle files for future use in model training and evaluation.

```
1  def split_data(self):
2      #Extracts input and output variables
3      X,y=self.extract_data(self.df)
4
5      #Resamples data using SMOTE
6      sm = SMOTE(random_state=42)
7      X,y = sm.fit_resample(X, y.ravel())
8
9      #Splits the data into training and testing
10     X_train, X_test, y_train, y_test =
11         train_test_split(X, \
12             y, test_size = 0.2, random_state = 2021,
13             shuffle=True, \
14             stratify=y)
15
16      #Normalizes data with robust scaler
17      rob_scaler = RobustScaler()
18      X_train_rob = X_train.copy()
19      X_test_rob = X_test.copy()
20      self.y_train_rob = y_train.copy()
21      self.y_test_rob = y_test.copy()
22      self.X_train_rob =
23          rob_scaler.fit_transform(X_train_rob)
24      self.X_test_rob = rob_scaler.transform(X_test_rob)
25
26      #Saves into pkl files
27      joblib.dump(self.X_train_rob, 'X_train_rob.pkl')
28      joblib.dump(self.X_test_rob, 'X_test_rob.pkl')
29      joblib.dump(self.y_train_rob, 'y_train_rob.pkl')
30      joblib.dump(self.y_test_rob, 'y_test_rob.pkl')
31
32      #Normalizes data with MinMax Scaler
33      X_train_norm = X_train.copy()
34      X_test_norm = X_test.copy()
35      self.y_train_norm = y_train.copy()
36      self.y_test_norm = y_test.copy()
37
38      norm = MinMaxScaler()
39      self.X_train_norm = norm.fit_transform(X_train_norm)
40      self.X_test_norm = norm.transform(X_test_norm)
41
42
43      #Saves into pkl files
44      joblib.dump(self.X_train_norm, 'X_train_norm.pkl')
45      joblib.dump(self.X_test_norm, 'X_test_norm.pkl')
46      joblib.dump(self.y_train_norm, 'y_train_norm.pkl')
```

```

48         joblib.dump(self.y_test_norm, 'y_test_norm.pkl')
49
50     #Normalizes data with Standard Scaler
51     X_train_stand = X_train.copy()
52     X_test_stand = X_test.copy()
53     self.y_train_stand = y_train.copy()
54     self.y_test_stand = y_test.copy()
55     scaler = StandardScaler()
56     self.X_train_stand =
57     scaler.fit_transform(X_train_stand)
58     self.X_test_stand = scaler.transform(X_test_stand)
59
60     #Saves into pkl files
61     joblib.dump(self.X_train_stand, 'X_train_stand.pkl')
62     joblib.dump(self.X_test_stand, 'X_test_stand.pkl')
63     joblib.dump(self.y_train_stand, 'y_train_stand.pkl')
64     joblib.dump(self.y_test_stand, 'y_test_stand.pkl')

```

Step 3 Define **split_data_ML()** method to execute splitting dataset into train and test data:

```

1  def split_data_ML(self):
2      #Loads or creates robust scaled, minmax scaled, and
3      #standard scaled train and test sets
4      if path.isfile('X_train_rob.pkl'):
5          self.X_train_rob, self.X_test_rob, self.y_train_rob,
6          \
7              self.y_test_rob = self.load_rob_files()
8          self.X_train_norm, self.X_test_norm,
9          self.y_train_norm, \
10             self.y_test_norm = self.load_norm_files()
11          self.X_train_stand, self.X_test_stand, \
12              self.y_train_stand, self.y_test_stand = \
13                  self.load_stand_files()
14
15      else:
16          self.split_data()
17
18
19      #Prints each shape
20      print('X train ROB: ', self.X_train_rob.shape)
21      print('X test ROB: ', self.X_test_rob.shape)
22      print('Y train ROB: ', self.y_train_rob.shape)
23      print('Y test ROB: ', self.y_test_rob.shape)
24
25      #Prints each shape
26      print('X train NORM: ', self.X_train_norm.shape)
27      print('X test NORM: ', self.X_test_norm.shape)
28      print('Y train NORM: ', self.y_train_norm.shape)
29      print('Y test NORM: ', self.y_test_norm.shape)
30
31      #Prints each shape
32      print('X train STAND: ', self.X_train_stand.shape)
33      print('Y train STAND: ', self.y_train_stand.shape)
34      print('X test STAND: ', self.X_test_stand.shape)
35      print('Y test STAND: ', self.y_test_stand.shape)
36
37
38  def load_rob_files(self):
39      X_train_rob = joblib.load('X_train_rob.pkl')
40      X_test_rob = joblib.load('X_test_rob.pkl')
41      y_train_rob = joblib.load('y_train_rob.pkl')
42      y_test_rob = joblib.load('y_test_rob.pkl')
43      return X_train_rob, X_test_rob, y_train_rob, y_test_rob
44
45
46  def load_norm_files(self):
47      X_train_norm = joblib.load('X_train_norm.pkl')
48      X_test_norm = joblib.load('X_test_norm.pkl')
49      y_train_norm = joblib.load('y_train_norm.pkl')
50      y_test_norm = joblib.load('y_test_norm.pkl')
51      return X_train_norm, X_test_norm, y_train_norm,
52      y_test_norm
53
54  def load_stand_files(self):
55      X_train_stand = joblib.load('X_train_stand.pkl')

```

```

X_test_stand = joblib.load('X_test_stand.pkl')
y_train_stand = joblib.load('y_train_stand.pkl')
y_test_stand = joblib.load('y_test_stand.pkl')
return X_train_stand, X_test_stand, y_train_stand,
y_test_stand

```

Let's break down each method step by step:

1. **split_data_ML(self)** method:

- a. Checks if the pickle files containing the preprocessed data exist (X_train_rob.pkl, etc.) by using the path.isfile function from the os module. If the files exist, it means that the data has been previously split and saved, so the method proceeds to load the data using the load_rob_files(), load_norm_files(), and load_stand_files() methods.
- b. If the pickle files don't exist, it means that the data needs to be split and preprocessed. In this case, it calls the **split_data()** method to perform the data splitting and normalization.
- c. Prints the shape of each set of data (train and test) for the robust scaled, minmax scaled, and standard scaled versions. This provides information about the number of samples and features in each set.

2. **load_rob_files(self)** method:

- a. Loads the robust scaled training data (X_train_rob) from the 'X_train_rob.pkl' file using the joblib.load() function.
- b. Loads the robust scaled testing data (X_test_rob) from the 'X_test_rob.pkl' file.
- c. Loads the robust scaled training target variable (y_train_rob) from the 'y_train_rob.pkl' file.
- d. Loads the robust scaled testing target variable (y_test_rob) from the 'y_test_rob.pkl' file.
- e. Returns the loaded data (X_train_rob, X_test_rob, y_train_rob, y_test_rob) as a tuple.

3. **load_norm_files(self)** method:

- a. Loads the minmax scaled training data (X_train_norm) from the 'X_train_norm.pkl' file using the joblib.load() function.
- b. Loads the minmax scaled testing data (X_test_norm) from the 'X_test_norm.pkl' file.
- c. Loads the minmax scaled training target variable (y_train_norm) from the 'y_train_norm.pkl' file.
- d. Loads the minmax scaled testing target variable (y_test_norm) from the 'y_test_norm.pkl' file.
- e. Returns the loaded data (X_train_norm, X_test_norm, y_train_norm, y_test_norm) as a tuple.

4. **load_stand_files(self)** method:

- a. Loads the standard scaled training data (X_train_stand) from the 'X_train_stand.pkl' file using the joblib.load() function.
- b. Loads the standard scaled testing data (X_test_stand) from the 'X_test_stand.pkl' file.

	<p>c. Loads the standard scaled training target variable (<code>y_train_stand</code>) from the '<code>y_train_stand.pkl</code>' file. d. Loads the standard scaled testing target variable (<code>y_test_stand</code>) from the '<code>y_test_stand.pkl</code>' file. e. Returns the loaded data (<code>X_train_stand</code>, <code>X_test_stand</code>, <code>y_train_stand</code>, <code>y_test_stand</code>) as a tuple.</p> <p>These methods are designed to handle the loading of preprocessed data from pickle files. If the files exist, they load the data, and if the files don't exist, they assume that the data needs to be split and preprocessed using the <code>split_data()</code> method. The loaded data is then returned for further use, such as model training and evaluation.</p>
Step 4	<p>Define <code>train_model_ML()</code> method to invoke <code>split_data_ML()</code> method.</p> <pre> 1 def train_model_ML(self): 2 self.split_data_ML() 3 4 #Turns on two widgets 5 self.cbData.setEnabled(True) 6 self.cbClassifier.setEnabled(True) 7 8 #Turns off pbTrainML 9 self.pbTrainML.setEnabled(False) 10 11 #Turns on three radio buttons 12 self.rbRobust.setEnabled(True) 13 self.rbRobust.setChecked(True) 14 self.rbMinMax.setEnabled(True) 15 self.rbStandard.setEnabled(True) </pre> <p>Let's go through the <code>train_model_ML(self)</code> method step by step:</p> <ol style="list-style-type: none"> 1. <code>self.split_data_ML()</code>: This method is called to split and load the preprocessed data. It performs the necessary data splitting and loads the data into the appropriate variables. 2. Turning on widgets and buttons: <ol style="list-style-type: none"> a. <code>self.cbData.setEnabled(True)</code>: This line enables the <code>cbData</code> widget, presumably a dropdown or combobox, allowing the user to select data options. b. <code>self.cbClassifier.setEnabled(True)</code>: This line enables the <code>cbClassifier</code> widget, presumably another dropdown or combobox, allowing the user to select classifier options. c. <code>self.pbTrainML.setEnabled(False)</code>: This line disables the <code>pbTrainML</code> button, presumably a button used for triggering the model training process. d. <code>self.rbRobust.setEnabled(True)</code>: This line enables the <code>rbRobust</code> radio button, which is related to the robust scaling option. e. <code>self.rbRobust.setChecked(True)</code>: This line sets the <code>rbRobust</code> radio button as checked by default. f. <code>self.rbMinMax.setEnabled(True)</code>: This line enables the <code>rbMinMax</code> radio button, which is related to the minmax scaling option. g. <code>self.rbStandard.setEnabled(True)</code>: This line enables the <code>rbStandard</code> radio button, which is related to the standard scaling option.

	These lines of code manipulate the state and behavior of various widgets and buttons in the user interface, allowing the user to interact with the application and choose options for data and classifier selection.
Step 5	Connect <code>clicked()</code> event of <code>pbTrainML</code> widget with <code>train_model_ML()</code> and put it inside <code>initial_state()</code> method as shown in line 12:
	<pre> 1 def initial_state(self, state): 2 self.pbTrainML.setEnabled(state) 3 self.cbData.setEnabled(state) 4 self.cbForecasting.setEnabled(state) 5 self.cbClassifier.setEnabled(state) 6 self.rbRobust.setEnabled(state) 7 self.rbMinMax.setEnabled(state) 8 self.rbStandard.setEnabled(state) 9 self.pbLoad.clicked.connect(self.import_dataset) 10 self.cbData.currentIndexChanged.connect(self.choose_plot) 11 12 self.cbForecasting.currentIndexChanged.connect(self.choose_forecasting) 13 14 self.pbTrainML.clicked.connect(self.train_model_ML) </pre>

DEFINING HELPER FUNCTIONS

DEFINING HELPER FUNCTIONS

Step 1	Define <code>plot_real_pred_val()</code> method to plot true values and predicted values and <code>plot_cm()</code> method to calculate and plot confusion matrix:
	<pre> 1 def plot_real_pred_val(self, Y_pred, Y_test, widget, title): 2 #Calculate Metrics 3 acc=accuracy_score(Y_test,Y_pred) 4 5 #Output plot 6 widget.canvas.figure.clf() 7 widget.canvas.axis1 = \ 8 widget.canvas.figure.add_subplot(111, \ 9 facecolor='steelblue') 10 widget.canvas.axis1.scatter(range(len(Y_pred)), \ 11 Y_pred,color="yellow",lw=5,label="Predictions") 12 widget.canvas.axis1.scatter(range(len(Y_test)), \ 13 Y_test,color="red",label="Actual") 14 widget.canvas.axis1.set_title(15 "Prediction Values vs Real Values of " + title, \ 16 fontsize=10) 17 widget.canvas.axis1.set_xlabel("Accuracy: " + \ 18 str(round((acc*100),3)) + "%") 19 widget.canvas.axis1.legend() 20 widget.canvas.axis1.grid(True, alpha=0.75, lw=1, 21 ls='-.') 22 widget.canvas.axis1.yaxis.set_ticklabels([""], \ 23 </pre>

```

24         "Negative Daily Returns", "", "", "", "", \
25         "Positive Daily Returns"]);
26     widget.canvas.draw()
27
28 def plot_cm(self, Y_pred, Y_test, widget, title):
29     cm=confusion_matrix(Y_test,Y_pred)
30     widget.canvas.figure.clf()
31     widget.canvas.axis1 =
32     widget.canvas.figure.add_subplot(111)
33     class_label = ["Negative Daily Returns", "Positive Daily
34 Returns"]
35     df_cm = pd.DataFrame(cm, \
36             index=class_label,columns=class_label)
37     sns.heatmap(df_cm, ax=widget.canvas.axis1, \
38             annot=True, cmap='plasma',linelwidths=2,fmt='d')
39     widget.canvas.axis1.set_title("Confusion Matrix of " + \
40             title, fontsize=10)
41     widget.canvas.axis1.set_xlabel("Predicted")
42     widget.canvas.axis1.set_ylabel("True")
43     widget.canvas.axis1.xaxis.set_ticklabels(\
44         ["Negative Daily Returns", "Positive Daily
45 Returns"]);
46     widget.canvas.axis1.yaxis.set_ticklabels(\
47         ["Negative Daily Returns", "Positive Daily
48 Returns"]);
48     widget.canvas.draw()

```

Here's a step-by-step explanation of each method without explicit code:

1. **plot_real_pred_val(self, Y_pred, Y_test, widget, title):** This method is responsible for plotting the predicted values versus the real values of a certain target variable.
 - a. Calculate Metrics: The method calculates the accuracy score by comparing the predicted values (Y_{pred}) with the actual values (Y_{test}).
 - b. Output Plot: The method creates a scatter plot to visualize the predicted values and the actual values. The predicted values are represented by yellow points, while the actual values are represented by red points. The plot also includes a title indicating the target variable, the accuracy score as the x-axis label, and a legend to differentiate between the predicted and actual values.
2. **plot_cm(self, Y_pred, Y_test, widget, title):** This method is used to plot the confusion matrix for evaluating the performance of a classification model.
 - a. Create Confusion Matrix: The method constructs a confusion matrix by comparing the predicted values (Y_{pred}) with the actual values (Y_{test}).
 - b. Output Plot: The method generates a heatmap visualization of the confusion matrix using a color map. The heatmap is annotated with the count of each class's predictions. The plot includes a title indicating the target variable, and the x-axis and y-axis are labeled as "Predicted" and "True," respectively, representing the class labels.

Step 2 Define **plot_learning_curve()** to plot learning curve of any machine learning model:

```
1 def plot_learning_curve(self, estimator, title, X, y, widget,
2     ylim=None, cv=None, n_jobs=None, train_sizes=np.linspace(.1,
3     1.0, 5)):
4     widget.canvas.axis1.set_title(title)
5     if ylim is not None:
6         widget.canvas.axis1.set_ylim(*ylim)
7     widget.canvas.axis1.set_xlabel("Training examples")
8     widget.canvas.axis1.set_ylabel("Score")
9
10    train_sizes, train_scores, test_scores, fit_times, _ = \
11        learning_curve(estimator, X, y, cv=cv,
12        n_jobs=n_jobs,
13        train_sizes=train_sizes,
14        return_times=True)
15    train_scores_mean = np.mean(train_scores, axis=1)
16    train_scores_std = np.std(train_scores, axis=1)
17    test_scores_mean = np.mean(test_scores, axis=1)
18    test_scores_std = np.std(test_scores, axis=1)
19
20    # Plot learning curve
21    widget.canvas.axis1.grid()
22    widget.canvas.axis1.fill_between(train_sizes, \
23        train_scores_mean - train_scores_std,\ \
24        train_scores_mean + train_scores_std,
25        alpha=0.1,color="r")
26    widget.canvas.axis1.fill_between(train_sizes, \
27        test_scores_mean - test_scores_std,\ \
28        test_scores_mean + test_scores_std, alpha=0.1,
29        color="g")
30    widget.canvas.axis1.plot(train_sizes, train_scores_mean,
31        'o-', color="r", label="Training score")
32    widget.canvas.axis1.plot(train_sizes, test_scores_mean,
33        'o-', color="g", label="Cross-validation score")
34    widget.canvas.axis1.legend(loc="best")
```

Here's an explanation of the **plot_learning_curve()** method:

1. This method is used to plot the learning curve of an estimator (machine learning model) to evaluate its performance on training and cross-validation sets.
2. Set Plot Title and Labels: The method sets the title of the plot based on the provided title parameter. It also sets the x-axis label as "Training examples" and the y-axis label as "Score". Optionally, the y-axis limits (ylim) can be specified.
3. Calculate Learning Curve Data: The method uses the `learning_curve` function to calculate the training and cross-validation scores for different training set sizes (train_sizes). It passes the provided estimator, input features (X), and target variable (y). Additional parameters such as cross-validation (cv), number of parallel jobs (n_jobs), and return of fit times are also passed.
4. Compute Mean and Standard Deviation: The method calculates the mean and standard deviation of the training scores, cross-validation scores, and optionally the fit times across different training set sizes.
5. Plot Learning Curve: The method plots the learning curve using the obtained data. It creates a grid, fills the area between the mean scores minus/plus one

	<p>standard deviation with a transparent color (red for training scores, green for cross-validation scores), and plots the mean training and cross-validation scores as lines with markers.</p> <p>6. Add Legend: The method adds a legend to the plot, positioning it at the best location determined by loc="best". The legend differentiates between the training score and cross-validation score.</p> <p>The resulting plot provides insights into the model's performance by visualizing how the training and cross-validation scores change as the number of training examples increases.</p>
--	--

Step 3 Define `plot_scalability_curve()` to plot the scalability of the model and `plot_performance_curve()` method to plot performance of the model on a widget:

```

1 def plot_scalability_curve(self, estimator, title, X, y,
2     widget, ylim=None, cv=None, n_jobs=None,
3     train_sizes=np.linspace(.1, 1.0, 5)):
4     widget.canvas.axis1.set_title(title, \
5         fontweight ="bold", fontsize=15)
6     if ylim is not None:
7         widget.canvas.axis1.set_ylim(*ylim)
8         widget.canvas.axis1.set_xlabel("Training examples")
9         widget.canvas.axis1.set_ylabel("Score")
10
11    train_sizes, train_scores, test_scores, fit_times, _ = \
12        learning_curve(estimator, X, y, cv=cv,
13        n_jobs=n_jobs,
14            train_sizes=train_sizes,
15            return_times=True)
16    fit_times_mean = np.mean(fit_times, axis=1)
17    fit_times_std = np.std(fit_times, axis=1)
18
19    # Plot n_samples vs fit_times
20    widget.canvas.axis1.grid()
21    widget.canvas.axis1.plot(train_sizes, fit_times_mean,
22        'o-')
23    widget.canvas.axis1.fill_between(train_sizes, \
24        fit_times_mean - fit_times_std,\
25        fit_times_mean + fit_times_std, alpha=0.1)
26    widget.canvas.axis1.set_xlabel("Training examples")
27    widget.canvas.axis1.set_ylabel("fit_times")
28
29
30 def plot_performance_curve(self, estimator, title, X, y, \
31     widget, ylim=None, cv=None, n_jobs=None, \
32     train_sizes=np.linspace(.1, 1.0, 5)):
33
34     widget.canvas.axis1.set_title(title, \
35         fontweight ="bold", fontsize=15)
36     if ylim is not None:
37         widget.canvas.axis1.set_ylim(*ylim)
38         widget.canvas.axis1.set_xlabel("Training examples")
39         widget.canvas.axis1.set_ylabel("Score")
40
41
42    train_sizes, train_scores, test_scores, fit_times, _ = \
43        learning_curve(estimator, X, y, cv=cv,
44        n_jobs=n_jobs,
45            train_sizes=train_sizes,
46            return_times=True)
47    test_scores_mean = np.mean(test_scores, axis=1)
48    test_scores_std = np.std(test_scores, axis=1)
49    fit_times_mean = np.mean(fit_times, axis=1)
50
51
52    # Plot n_samples vs fit_times
53    widget.canvas.axis1.grid()
54    widget.canvas.axis1.plot(fit_times_mean, \

```

```
|      test_scores_mean, 'o-')
|      widget.canvas.axis1.fill_between(fit_times_mean, \
|
```

```

    test_scores_mean - test_scores_std,\n
    test_scores_mean + test_scores_std, alpha=0.1)\n
    widget.canvas.axis1.set_xlabel("fit_times")\n
    widget.canvas.axis1.set_ylabel("Score")

```

Let's go through each method step by step:

1. **plot_scalability_curve()** method:

- The method takes the following arguments: self (referring to the current instance of the class), estimator (a machine learning estimator object), title (a string representing the title of the plot), X (input features), y (target variable), widget (a widget object representing the plot), ylim (optional y-axis limits for the plot), cv (optional cross-validation splitting strategy), n_jobs (optional number of parallel jobs to run), and train_sizes (an array specifying the proportion of the dataset to use for training).
- The method sets the title of the plot on the axis1 of the widget.canvas object with the provided title, using a bold font and font size of 15.
- If ylim is provided, the y-axis limits of axis1 are set accordingly.
- The x-axis label of axis1 is set as "Training examples", and the y-axis label is set as "Score".
- The learning_curve function is called with the provided arguments to obtain the learning curve results. The learning_curve() function returns the train_sizes, train_scores, test_scores, fit_times, and _.
- The mean and standard deviation of fit_times are calculated along the rows using np.mean and np.std functions, respectively, and stored in fit_times_mean and fit_times_std variables.
- The grid is enabled on axis1 of the widget.canvas object.
- The train_sizes and fit_times_mean are plotted as a line plot on axis1 using the plot function.
- The area between fit_times_mean - fit_times_std and fit_times_mean + fit_times_std is filled with a transparent color using the fill_between() function to indicate the variability of fit times.
- Finally, the x-axis label is set as "Training examples" and the y-axis label is set as "fit_times".

2. **plot_performance_curve()** method:

- The method has the same structure as `plot_scalability_curve()` with a few differences in the plot and the labels.
- Similar to `plot_scalability_curve()`, the title, y-axis limits (if provided), x-axis label, and y-axis label are set on axis1 of the widget.canvas object.
- The `learning_curve()` function is called to obtain the learning curve results.
- The mean and standard deviation of test_scores are calculated along the rows and stored in test_scores_mean and test_scores_std variables.
- The mean of fit_times is calculated along the rows and stored in the fit_times_mean variable.
- The grid is enabled on axis1 of the widget.canvas object.
- The fit_times_mean and test_scores_mean are plotted as a line plot on axis1.
- The area between test_scores_mean - test_scores_std and test_scores_mean + test_scores_std is filled with a transparent color using the fill_between function to indicate the variability of test scores.
- Finally, the x-axis label is set as "fit_times" and the y-axis label is set as "Score".

In summary, both methods create a plot using the provided estimator, input features, and target variable. The `learning_curve()` function is used to obtain the learning curve results, and the plots show the relationship between training examples (or fit times) and the score (or performance) of the estimator. The shaded areas represent the variability in the results.

Step 4

Define `plot_decision()` method to plot decision bound of two chose features:

```

1 def plot_decision(self, df, cla, feat1, feat2, widget,
2 title=""):
3     X,y=self.extract_data(df)
4
5     #Plots decision boundary of two features
6     X_feature = np.array(X.iloc[:, 13:14])
7     X_train_feature, X_test_feature, y_train_feature,
8     y_test_feature= train_test_split(X_feature, y,
9     test_size=0.3, random_state = 42)
10    cla.fit(X_train_feature, y_train_feature)
11
12    plot_decision_regions(X_test_feature, \
13        y_test_feature.ravel(), clf=cla, legend=2, \
14        ax=widget.canvas.axis1)
15    widget.canvas.axis1.set_title(title, fontweight
16    ="bold", \
17        fontsize=15)
18    widget.canvas.axis1.set_xlabel(feat1)
19    widget.canvas.axis1.set_ylabel(feat2)
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()

```

It plots the decision boundary of two features using the classifier. Here is a step-by-step breakdown of the method:

1. The method takes the following argument (referring to the current instance of the class): df (a pandas DataFrame containing the data), cla (the classifier object), feat1 (the name of the first feature), feat2 (the name of the second feature), axis1 (a widget object representing the plot area) and an optional title argument for the plot title.
2. The method calls the extract_data() function on df to separate the features (X) and the target variable (y) from the DataFrame.
3. The method selects the data for the two specified features (feat1 and feat2) from the X DataFrame X and stores it in X_feature.
4. It splits the data into training and testing sets using train_test_split, with 70% of the data used for training (train_size=0.3) and a random state of 42.
5. The classifier (cla) is fitted using the training data (X_train_feature and y_train_feature).
6. The plot_decision_regions() function is called to plot the decision regions for the test data (X_test_feature and y_test_feature.ravel()) using the trained classifier (cla). The resulting decision plot is displayed on the axis1 object of the widget.canvas object.
7. The method sets the title of the plot on axis1 using the provided title, with a bold font at a size of 15.
8. The x-axis label of axis1 is set to the name of the first feature (feat1), and the y-axis label is set to the name of the second feature (feat2).
9. The tight_layout() function is called on the widget.canvas.figure to ensure that the plot is properly adjusted.
10. Finally, the draw function is called on the widget.canvas to update and display the plot.

In summary, the **plot_decision()** method extracts the specified features and target variable from the DataFrame, fits the classifier using the training data, and plots the decision boundary for the two features using the trained classifier on a specified widget canvas.

Step 5

Define **run_model()** method to calculate accuracy and precision. It also invokes six methods to plot confusion matrix, true values versus predicted values distribution learning curve, and decision boundaries:

```
1 def train_model(self, model, X, y):  
2     model.fit(X, y)  
3     return model  
4  
5 def predict_model(self, model, X, proba=False):  
6     if ~proba:  
7         return model.predict(X)  
8     else:  
9         return model.predict_proba(X)
```

```

7         y_pred = model.predict(X)
8
9     else:
10        y_pred_proba = model.predict_proba(X)
11        y_pred = np.argmax(y_pred_proba, axis=1)
12
13    return y_pred
14
15 def run_model(self, name, scaling, model, X_train, X_t
16 y_train, y_test, train=True, proba=True):
17     if train == True:
18         model = self.train_model(model, X_train, y_trai
19         y_pred = self.predict_model(model, X_test, proba)
20
21     accuracy = accuracy_score(y_test, y_pred)
22     recall = recall_score(y_test, y_pred,
23 average='weighted')
24     precision = precision_score(y_test, y_pred, \
25         average='weighted')
26     f1 = f1_score(y_test, y_pred, average='weighted')
27
28     print('accuracy: ', accuracy)
29     print('recall: ', recall)
30     print('precision: ', precision)
31     print('f1: ', f1)
32     print(classification_report(y_test, y_pred))
33
34     self.widgetPlot1.canvas.figure.clf()
35     self.widgetPlot1.canvas.axis1 = \
36         self.widgetPlot1.canvas.figure.add_subplot(111
37         facecolor = '#fbe7dd')
38     self.plot_cm(y_pred, y_test, self.widgetPlot1, \
39         name + " -- " + scaling)
40     self.widgetPlot1.canvas.figure.tight_layout()
41     self.widgetPlot1.canvas.draw()
42
43
44     self.widgetPlot2.canvas.figure.clf()
45     self.widgetPlot2.canvas.axis1 = \
46         self.widgetPlot2.canvas.figure.add_subplot(111
47         facecolor = '#fbe7dd')
48     self.plot_real_pred_val(y_pred, y_test, \
49         self.widgetPlot2, name + " -- " + scaling)
50     self.widgetPlot1.canvas.figure.tight_layout()
51     self.widgetPlot1.canvas.draw()
52
53
54     self.widgetPlot3.canvas.figure.clf()
55     self.widgetPlot3.canvas.axis1 = \
56         self.widgetPlot3.canvas.figure.add_subplot(221
57         facecolor = '#fbe7dd')
58     self.plot_decision(self.df_rfm, model, 'Open_Close
59         'High_Low', self.widgetPlot3, \
60         title="The decision boundaries of " + \
61         name + " -- " + scaling)
62
63     self.widgetPlot3.canvas.axis1 = \
64         self.widgetPlot3.canvas.figure.add_subplot(222
65         facecolor = '#fbe7dd')
66     self.plot_learning_curve(model, \
67         'Learning Curve' + " -- " + scaling, X_train,
68         y_train, self.widgetPlot3)
69     self.widgetPlot3.canvas.figure.tight_layout()
70
71     self.widgetPlot3.canvas.axis1 = \
72         self.widgetPlot3.canvas.figure.add_subplot(223
73         facecolor = '#fbe7dd')
74     self.plot_scalability_curve(model, 'Scalability of
75         name + " -- " + scaling, X_train, y_train, \
76         self.widgetPlot3)
77     self.widgetPlot3.canvas.figure.tight_layout()
78
79     self.widgetPlot3.canvas.axis1 = \
80         self.widgetPlot3.canvas.figure.add_subplot(224
81         facecolor = '#fbe7dd')
82     self.plot_performance_curve(model, \
83         'Performance of ' + name + " -- " + scaling, \
84         X_train, y_train, self.widgetPlot3)
85     self.widgetPlot3.canvas.figure.tight_layout()
86
87     self.widgetPlot3.canvas.draw()

```

Let's go through each method and explain their functionality:

1. **train_model(self, model, X, y):** This method trains a given model using the provided feature data X and target variable y. It calls the fit method of the model and passes X and y to train the model. Finally, it returns the trained model.
2. **predict_model(self, model, X, proba=False):** This method makes predictions using a trained model on the given feature data X. If proba is set to False, it uses the predict method of the model to obtain the predicted labels (y_pred). If proba is set to True, it uses the predict_proba method of the model to obtain the predicted probabilities for each class, and then selects the class with the highest probability using np.argmax. The method returns the predicted labels (y_pred).
3. **run_model(self, name, scaling, model, X_train, X_test, y_train, y_test, train=True, proba=True):** This method runs the entire machine learning pipeline, including training, prediction, evaluation, and plotting.
 - a. If train is set to True, it trains the model using the train_model method with the training data (X_train and y_train).
 - b. It calls the predict_model() method to obtain the predicted labels (y_pred) using the trained model and the test data (X_test).
 - c. Various evaluation metrics such as accuracy, recall, precision, and F1 score are calculated using functions like accuracy_score, recall_score, precision_score, and f1_score, comparing the predicted labels (y_pred) with the true labels (y_test).
 - d. The evaluation metrics and a classification report are printed to the console.
 - e. Plots and visualizations are created using different methods such as plot_learning_curve(), plot_real_pred_val(), plot_decision_boundary(), plot_scalability_curve(), and plot_performance_curve(). These methods generate plots based on the predicted labels (y_pred), true labels (y_test), and the trained model.
 - f. The generated plots are displayed in respective widgetPlot objects.

Each of these methods contributes to different aspects of the machine learning model pipeline, including training, prediction, evaluation, and visualization.

PREDICTING DAILY RETURN USING LOGISTIC REGRESSION

PREDICTING DAILY RETURN USING LOGISTIC REGRESSION

Step 1 Define `build_train_lr()` method to build and train Logistic Regression (LR) classifier using three feature scaling: **Robust Scaler**, **MinMax Scaler**, and **Standard Scaler**:

```
1  def build_train_lr(self):
2      try:
3          if os.path.isfile('logregRob.pkl'):
4              # Loads model
5              self.logregRob =
6              joblib.load('logregRob.pkl')
7              self.logregNorm =
8              joblib.load('logregNorm.pkl')
9              self.logregStand =
10             joblib.load('logregStand.pkl')
11
12             if self.rbRobust.isChecked():
13                 self.run_model('Logistic Regression',
14 'Robust', \
15                     self.logregRob, self.X_train_rob, \
16                     self.X_test_rob, self.y_train_rob, \
17                     self.y_test_rob)
18             if self.rbMinMax.isChecked():
19                 self.run_model('Logistic Regression',
20 'MinMax', \
21                     self.logregNorm, self.X_train_norm, \
22                     self.X_test_norm, self.y_train_norm, \
23                     self.y_test_norm)
24
25             if self.rbStandard.isChecked():
26                 self.run_model('Logistic Regression',
27 'Standard', \
28                     self.logregStand, self.X_train_stand, \
29                     self.X_test_stand, self.y_train_stand, \
30                     self.y_test_stand)
31
32         else:
33             raise FileNotFoundError("Model files not
34 found.")
35
36
37     except Exception as e:
38         print("An error occurred:", e)
39         # Builds and trains Logistic Regression
40         self.logregRob =
41 LogisticRegression(solver='lbfgs', \
42                         max_iter=2000, random_state=2021)
43         self.logregNorm =
44 LogisticRegression(solver='lbfgs', \
45                         max_iter=2000, random_state=2021)
46         self.logregStand =
47 LogisticRegression(solver='lbfgs', \
48                         max_iter=2000, random_state=2021)
49
50
51         if self.rbRobust.isChecked():
52             self.run_model('Logistic Regression',
53 'Robust', \
54                     self.logregRob, self.X_train_rob, \
55                     self.X_test_rob, self.y_train_rob, \
56                     self.y_test_rob)
57         if self.rbMinMax.isChecked():
58             self.run_model('Logistic Regression',
59 'MinMax', \
60                     self.logregNorm, self.X_train_norm, \
61                     self.X_test_norm, self.y_train_norm,
62                     self.y_test_norm)
63
64         if self.rbStandard.isChecked():
65             self.run_model('Logistic Regression',
66 'Standard', \
67                     self.logregStand, self.X_train_stand, \
68                     self.X_test_stand, self.y_train_stand, \
69                     self.y_test_stand)
70
71     try:
72         # Saves model
73         joblib.dump(self.logregRob, 'logregRob.pkl')
```

```

        joblib.dump(self.logregNorm,
'logregNorm.pkl')
        joblib.dump(self.logregStand,
'logregStand.pkl')
    except Exception as e:
        print("An error occurred while saving the
model:", e)

```

Here's a step-by-step explanation of the **build_train_lr()** method:

1. The method **build_train_lr()** is defined with the **self** parameter, indicating that it is a member method of a class.
2. The code is wrapped in a try block to catch any exceptions that may occur during the execution.
3. Within the try block, the code checks if the model files 'logregRob.pkl', 'logregNorm.pkl', and 'logregStand.pkl' exist using the `os.path.isfile` function.
4. If the model files exist, the code proceeds to load the models using `joblib.load` and assigns them to the corresponding variables `self.logregRob`, `self.logregNorm`, and `self.logregStand`.
5. Depending on the selected options (`self.rbRobust.isChecked()`, `self.rbMinMax.isChecked()`, `self.rbStandard.isChecked()`), the code calls the `run_model` method with the appropriate parameters to perform model evaluation and visualization.
6. If the model files do not exist, the code raises a `FileNotFoundException` exception with the message "Model files not found."
7. If an exception occurs in the try block, it is caught by the `except` block, and the error message is printed using `print("An error occurred:", e)`.
8. Inside the `except` block, new instances of `LogisticRegression` are created and assigned to `self.logregRob`, `self.logregNorm`, and `self.logregStand`.
9. Similar to step 5, the code calls the `run_model` method to perform model evaluation and visualization for the newly created models.
10. After the model evaluation and visualization, the code attempts to save the models using `joblib.dump` and the corresponding file names ('logregRob.pkl', 'logregNorm.pkl', 'logregStand.pkl').
11. If an exception occurs during the model saving process, it is caught by another `except` block, and the error message is printed using `print("An error occurred while saving the
model:", e)`.

That's the step-by-step explanation of the **build_train_lr()** method. It checks if the model files exist, loads them if available, or builds new models if the files are not found. Then it performs model evaluation, visualization, and saves the models if necessary.

Step 2 Define **choose_ML_model()** method to read the selected item in **cbClassifier** widget:

```

1 def choose_ML_model(self):
2     strCB = self.cbClassifier.currentText()

```

```

3
4     if strCB == 'Logistic Regression':
5         self.build_train_lr()

```

Step 3 Connect **currentIndexChanged()** event of **cbClassifier** widget with **choose_ML_model()** method and put it inside **initial_state()** method as shown in line 13:

```

1 def initial_state(self, state):
2     self.pbTrainML.setEnabled(state)
3     self.cbData.setEnabled(state)
4     self.cbForecasting.setEnabled(state)
5     self.cbClassifier.setEnabled(state)
6     self.rbRobust.setEnabled(state)
7     self.rbMinMax.setEnabled(state)
8     self.rbStandard.setEnabled(state)
9     self.pbLoad.clicked.connect(self.import_dataset)
10    self.cbData.currentIndexChanged.connect(self.choose_plot)
11
12 self.cbForecasting.currentIndexChanged.connect(self.choose_forecasting)
13
14 self.pbTrainML.clicked.connect(self.train_model_ML)
15
16 self.cbClassifier.currentIndexChanged.connect(self.choose_ML_model)

```

Step 4 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Robust Scaler** radio button. Then, choose **Logistic Regression** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.1.

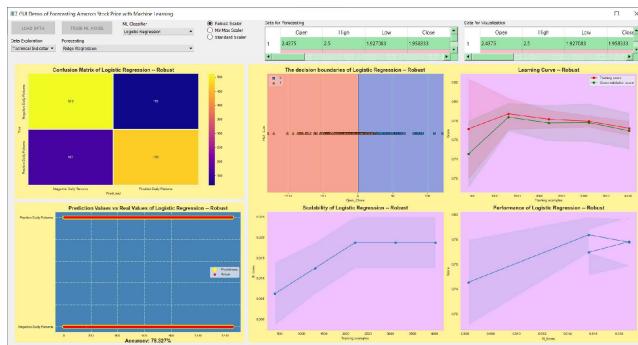


Figure 6.1 The result using LR model with robust scaler



Figure 6.2 The result using LR model with minmax scaler



Figure 6.3 The result using LR model with standard scaler

Click on **MinMax Scaler** radio button. Then, choose **Logistic Regression** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.2.

Click on **Standard Scaler** radio button. Then, choose **Logistic Regression** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.3.

PREDICTING DAILY RETURN USING SUPPORT VECTOR MACHINE

PREDICTING DAILY RETURN USING SUPPORT VECTOR MACHINE

Step 1 Define **build_train_svm()** method to build and train Support Vector Machine (SVM) classifier using three feature scaling: Robust Scaler, MinMax Scaler, and Standard Scaler:

```

1  def build_train_svm(self):
2      try:
3          if path.isfile('SVMRob.pkl'):
4              # Loads model
5              self.SVMRob = joblib.load('SVMRob.pkl')
6              self.SVMNorm = joblib.load('SVMNorm.pkl')
7              self.SVMStand = joblib.load('SVMStand.pkl')
8
9
10         if self.rbRobust.isChecked():
11             self.run_model('SVM', 'Robust',
12 self.SVMRob, \
13 self.X_train_rob, self.X_test_rob,
14 self.y_train_rob, \
15 self.y_test_rob)
16
17         if self.rbMinMax.isChecked():
18             self.run_model('SVM', 'MinMax',
19 self.SVMNorm, \
20             self.X_train_norm, self.X_test_norm,
21             self.y_train_norm, self.y_test_norm)
22
23         if self.rbStandard.isChecked():
24             self.run_model('SVM', 'Standard',
25 self.SVMStand, \
26 self.X_train_stand, \
27 self.X_test_stand, \
28 self.y_train_stand, \
29 self.y_test_stand)
30 else:
31     # Builds and trains SVM model
32
33
34

```

```

35             self.SVMRob = SVC(random_state=2021,
36 probability=True)
37             self.SVMNorm = SVC(random_state=2021,
38 probability=True)
39             self.SVMStand = SVC(random_state=2021,
40 probability=True)
41
42         if self.rbRobust.isChecked():
43             self.run_model('SVM', 'Robust',
44 self.SVMRob, \
45             self.X_train_rob, self.X_test_rob,
46 self.y_train_rob, \
47             self.y_test_rob)
48
49         if self.rbMinMax.isChecked():
50             self.run_model('SVM', 'MinMax',
51 self.SVMNorm, \
52             self.X_train_norm, self.X_test_norm,
53 self.y_train_norm, \
54             self.y_test_norm)
55
56         if self.rbStandard.isChecked():
57             self.run_model('SVM', 'Standard',
58 self.SVMStand, \
59             self.X_train_stand,
60 self.X_test_stand, \
61             self.y_train_stand,
62 self.y_test_stand)
63
64         # Saves model
65         joblib.dump(self.SVMRob, 'SVMRob.pkl')
66         joblib.dump(self.SVMNorm, 'SVMNorm.pkl')
67         joblib.dump(self.SVMStand, 'SVMStand.pkl')
68     except Exception as e:
69         print("An error occurred: ", str(e))

```

Here's a step-by-step explanation of the code:

1. The **build_train_svm()** method is defined within a class. This method is responsible for building and training SVM models.
2. The code is wrapped in a try block, indicating that any exceptions that occur within the block will be caught and handled.
3. Inside the try block, the code checks if the files 'SVMRob.pkl', 'SVMNorm.pkl', and 'SVMStand.pkl' exist using the path.isfile function. This checks if the model files have been saved previously.
4. If the model files exist, the code loads the SVM models from the files using joblib.load. The loaded models are assigned to the corresponding variables: self.SVMRob, self.SVMNorm, and self.SVMStand.
5. The code then checks the status of three radio buttons: self.rbRobust, self.rbMinMax, and self.rbStandard. If any of these radio buttons are checked, it calls the run_model method with the appropriate arguments, passing the corresponding SVM model and data variables.
6. If the model files don't exist, the code creates new SVM models using SVC and assigns them to self.SVMRob, self.SVMNorm, and self.SVMStand.
7. Similar to the previous step, the code checks the status of the radio buttons and calls the run_model

	<p>method with the appropriate arguments for each checked radio button.</p> <p>8. After building and training the SVM models, the code saves the models to files using joblib.dump. The models are saved as 'SVMRob.pkl', 'SVMNorm.pkl', and 'SVMStand.pkl'.</p> <p>9. If any exceptions occur within the try block, the code jumps to the except block. The exception is caught and assigned to the variable e. The code then prints an error message indicating that an error occurred, along with the details of the exception.</p> <p>Overall, this code tries to load pre-existing SVM models from files and, if they don't exist, builds new SVM models and saves them to files. It also runs the run_model() method for each selected radio button, passing the appropriate SVM model and data variables. Any exceptions that occur during this process are caught and printed as error messages.</p>
Step 2	<p>Add this code to the end of choose_ML_model() method:</p> <pre> 1 if strCB == 'Support Vector Machine': 2 self.build_train_svm() </pre>
Step 3	<p>Run gui_amazon.py and click LOAD DATA and TRAIN ML MODEL buttons. Click on Robust Scaler radio button. Then, choose Support Vector Machine item from cbClassifier widget. Then, you will see the result as shown in Figure 6.4.</p>

Figure 6.4 The result using SVM model with robust scaler

Click on **MinMax Scaler** radio button. Then, choose **Support Vector Machine** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.5.

Click on **Standard Scaler** radio button. Then, choose **Support Vector Machine** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.6.

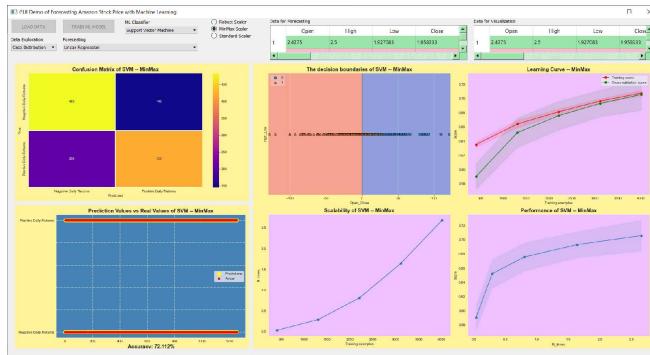


Figure 6.5 The result using SVM model with minmax scaler

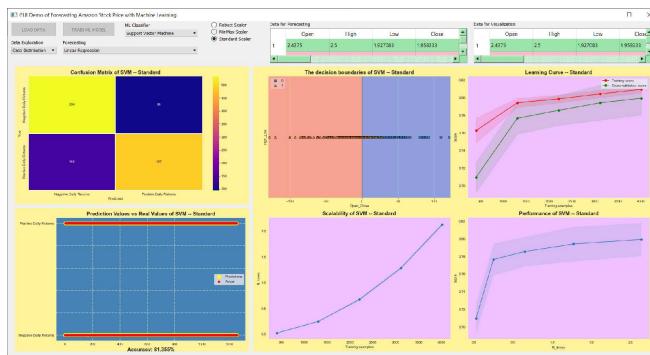


Figure 6.6 The result using SVM model with standard scaler

PREDICTING DAILY RETURN USING DECISION TREE

PREDICTING DAILY RETURN USING DECISION TREE

Step 1 Define **build_train_dt()** method to build and train Decision Tree (DT) classifier using three feature scaling: Robust Scaler, MinMax Scaler, and Standard Scaler:

```

1  def build_train_dt(self):
2      try:
3          if path.isfile('DTRob.pkl'):
4              #Loads model
5              self.DTRob = joblib.load('DTRob.pkl')
6              self.DTNorm = joblib.load('DTNorm.pkl')
7              self.DTStand = joblib.load('DTStand.pkl')
8
9          if self.rbRobust.isChecked():
10              self.run_model('DT', 'Robust',
11              self.DTRob, \
12                  self.X_train_rob, self.X_test_rob,
13                  self.y_train_rob, \
14                  self.y_test_rob)
15          if self.rbMinMax.isChecked():
16              self.run_model('DT', 'MinMax',
17              self.DTNorm, \
18                  self.X_train_norm, self.X_test_norm,
19                  self.y_train_norm, \
20                  self.y_test_norm)
21
22          if self.rbStandard.isChecked():
23              self.run_model('DT', 'Standard',
24              self.DTStand, \
25
26

```

```

27         self.X_train_stand,
28         self.X_test_stand, \
29             self.y_train_stand,
30             self.y_test_stand)
31
32     else:
33         #Builds and trains Decision Tree
34         dt_cla = DecisionTreeClassifier()
35         parameters = {
36             'max_depth':np.arange(1,20,1), 'random_state':[2021]}
37         self.DTRob = GridSearchCV(dt_cla,
38         parameters)
39         self.DTNorm = GridSearchCV(dt_cla,
40         parameters)
41         self.DTStand = GridSearchCV(dt_cla,
42         parameters)
43
44         if self.rbRobust.isChecked():
45             self.run_model('DT', 'Robust',
46             self.DTRob, \
47                 self.X_train_rob, self.X_test_rob,
48             self.y_train_rob, \
49                 self.y_test_rob)
50         if self.rbMinMax.isChecked():
51             self.run_model('DT', 'MinMax',
52             self.DTNorm, \
53                 self.X_train_norm, self.X_test_norm,
self.y_train_norm, \
self.y_test_norm)
54
55         if self.rbStandard.isChecked():
56             self.run_model('DT', 'Standard',
self.DTStand, \
self.X_test_stand, \
self.y_train_stand,
self.y_test_stand)
57
58         #Saves model
59         joblib.dump(self.DTRob, 'DTRob.pkl')
60         joblib.dump(self.DTNorm, 'DTNorm.pkl')
61         joblib.dump(self.DTStand, 'DTStand.pkl')
62     except Exception as e:
63         print("An error occurred: ", str(e))

```

Here's a step-by-step explanation of the code:

1. The **build_train_dt()** method is defined within a class. This method is responsible for building and training Decision Tree models.
2. The code is wrapped in a try block, indicating that any exceptions that occur within the block will be caught and handled.
3. Inside the try block, the code checks if the files 'DTRob.pkl', 'DTNorm.pkl', and 'DTStand.pkl' exist using the `path.isfile` function. This checks if the Decision Tree model files have been saved previously.
4. If the model files exist, the code loads the Decision Tree models from the files using `joblib.load`. The loaded models are assigned to the corresponding variables: `self.DTRob`, `self.DTNorm`, and `self.DTStand`.
5. The code then checks the status of three radio buttons: `self.rbRobust`, `self.rbMinMax`, and `self.rbStandard`. If any of these radio buttons are checked, it calls the `run_model()` method with the

	<p>appropriate arguments, passing the corresponding Decision Tree model and data variables.</p> <ol style="list-style-type: none"> 6. If the model files don't exist, the code creates new Decision Tree models using <code>DecisionTreeClassifier</code>. It also defines a dictionary <code>parameters</code> with hyperparameter values for <code>max_depth</code> and <code>random_state</code> that will be used in the grid search. 7. Grid search is performed using <code>GridSearchCV</code> to find the best hyperparameters for each Decision Tree model. The <code>DecisionTreeClassifier</code> object <code>dt_cla</code> and the <code>parameters</code> dictionary are passed to <code>GridSearchCV</code>. The resulting <code>GridSearchCV</code> objects are assigned to <code>self.DTRob</code>, <code>self.DTNorm</code>, and <code>self.DTStand</code>. 8. Similar to the previous step, the code checks the status of the radio buttons and calls the <code>run_model()</code> method with the appropriate arguments for each checked radio button. 9. After building and training the Decision Tree models, the code saves the models to files using <code>joblib.dump</code>. The models are saved as '<code>DTRob.pkl</code>', '<code>DTNorm.pkl</code>', and '<code>DTStand.pkl</code>'. 10. If any exceptions occur within the try block, the code jumps to the except block. The exception is caught and assigned to the variable <code>e</code>. The code then prints an error message indicating that an error occurred, along with the details of the exception. <p>Overall, this code follows a similar structure as the previous ones, but it is specific to building and training Decision Tree models. It tries to load pre-existing Decision Tree models from files and, if they don't exist, builds new Decision Tree models using grid search to find the best hyperparameters. It also runs the <code>run_model</code> method for each selected radio button, passing the appropriate Decision Tree model and data variables. Any exceptions that occur during this process are caught and printed as error messages.</p>
Step 2	Add this code to the end of <code>choose_ML_model()</code> method:
Step 3	<p>Run <code>gui_amazon.py</code> and click LOAD DATA and TRAIN ML MODEL buttons. Click on Robust Scaler radio button. Then, choose Decission Tree item from <code>cbClassifier</code> widget. Then, you will see the result as shown in Figure 6.7.</p> <p>Click on MinMax Scaler radio button. Then, choose Decission Tree item from <code>cbClassifier</code> widget. Then, you will see the result as shown in Figure 6.8.</p> <p>Click on Standard Scaler radio button. Then, choose Decission Tree item from <code>cbClassifier</code> widget. Then, you will</p> <pre> 1 if strCB == 'Decision Tree': 2 self.build_train_dt() </pre>

see the result as shown in Figure 6.9.

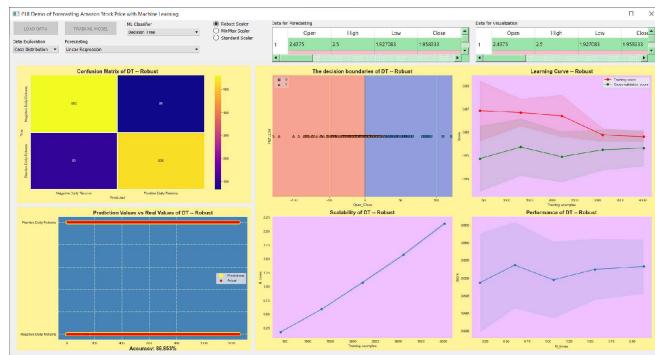


Figure 6.7 The result using DT model with robust scaler

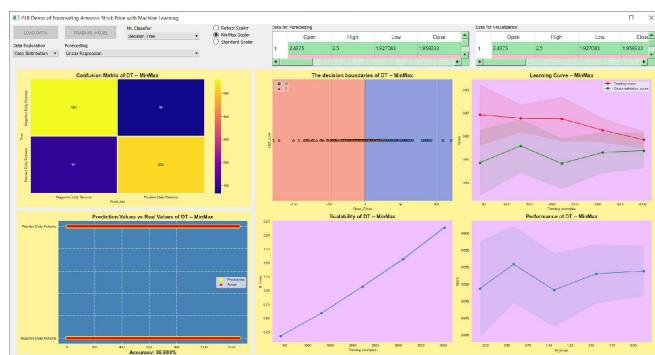


Figure 6.8 The result using DT model with minmax scaler

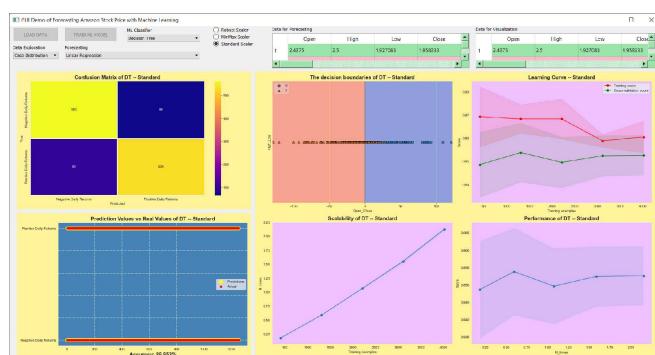


Figure 6.9 The result using DT model with standard scaler

PREDICTING DAILY RETURN USING K-NEAREST NEIGHBORS

PREDICTING DAILY RETURN USING K-NEAREST NEIGHBORS

Step 1 Define **build_train_knn()** method to build and train K-Nearest Neighbor (KNN) classifier using three feature scaling: Robust Scaler, MinMax Scaler, and Standard Scaler:

```

1  def build_train_knn(self):
2      try:
3          if path.isfile('KNNRob.pk1'):

```

```

4             #Loads model
5             self.KNNRob = joblib.load('KNNRob.pkl')
6             self.KNNNorm = joblib.load('KNNNorm.pkl')
7             self.KNStand = joblib.load('KNStand.pkl')
8
9             if self.rbRobust.isChecked():
10                 self.run_model('KNN', 'Robust',
11 self.KNNRob, self.X_train_rob, self.X_test_rob,
12 self.y_train_rob, self.y_test_rob)
13                 if self.rbMinMax.isChecked():
14                     self.run_model('KNN', 'MinMax',
15 self.KNNNorm, self.X_train_norm, self.X_test_norm,
16 self.y_train_norm, self.y_test_norm)
17
18             if self.rbStandard.isChecked():
19                 self.run_model('KNN', 'Standard',
20 self.KNStand, self.X_train_stand, self.X_test_stand,
21 self.y_train_stand, self.y_test_stand)
22
23         else:
24             #Builds and trains K-Nearest Neighbor
25             self.KNNRob =
26 KNeighborsClassifier(n_neighbors = 10)
27             self.KNNNorm =
28 KNeighborsClassifier(n_neighbors = 10)
29             self.KNStand =
30 KNeighborsClassifier(n_neighbors = 10)
31
32             if self.rbRobust.isChecked():
33                 self.run_model('KNN', 'Robust',
34 self.KNNRob, self.X_train_rob, self.X_test_rob,
35 self.y_train_rob, self.y_test_rob)
36                 if self.rbMinMax.isChecked():
37                     self.run_model('KNN', 'MinMax',
38 self.KNNNorm, self.X_train_norm, self.X_test_norm,
39 self.y_train_norm, self.y_test_norm)
40
41             if self.rbStandard.isChecked():
42                 self.run_model('KNN', 'Standard',
43 self.KNStand, self.X_train_stand, self.X_test_stand,
44 self.y_train_stand, self.y_test_stand)
45
46             #Saves model
47             joblib.dump(self.KNNRob, 'KNNRob.pkl')
48             joblib.dump(self.KNNNorm, 'KNNNorm.pkl')
49             joblib.dump(self.KNStand, 'KNStand.pkl')
50
51     except Exception as e:
52         print("An error occurred: ", str(e))

```

Here's a step-by-step explanation of the code:

1. The `build_train_knn()` method is defined within a class. This method is responsible for building and training KNN models.
2. The code is wrapped in a try block, indicating that any exceptions that occur within the block will be caught and handled.
3. Inside the try block, the code checks if the files '`KNNRob.pkl`', '`KNNNorm.pkl`', and '`KNStand.pkl`' exist using the `path.isfile` function. This checks if the KNN model files have been saved previously.
4. If the model files exist, the code loads the KNN models from the files using `joblib.load`. The loaded models are assigned to the corresponding variables: `self.KNNRob`, `self.KNNNorm`, and `self.KNStand`.
5. The code then checks the status of three radio buttons: `self.rbRobust`, `self.rbMinMax`, and `self.rbStandard`. If any of these radio buttons are

- checked, it calls the run_model method with the appropriate arguments, passing the corresponding KNN model and data variables.
6. If the model files don't exist, the code creates new KNN models using KNeighborsClassifier with n_neighbors set to 10. It creates three instances of KNeighborsClassifier and assigns them to self.KNNRob, self.KNNNorm, and self.KNNStand.
 7. Similar to the previous step, the code checks the status of the radio buttons and calls the run_model() method with the appropriate arguments for each checked radio button.
 8. After building and training the KNN models, the code saves the models to files using joblib.dump. The models are saved as 'KNNRob.pkl', 'KNNNorm.pkl', and 'KNNStand.pkl'.
 9. If any exceptions occur within the try block, the code jumps to the except block. The exception is caught and assigned to the variable e. The code then prints an error message indicating that an error occurred, along with the details of the exception.

In summary, this code follows a similar structure as the previous examples. It checks if the KNN model files exist and loads them if they do. If the files don't exist, it creates new KNN models with a specified number of neighbors. It then runs the run_model method for each selected radio button, passing the appropriate KNN model and data variables. Any exceptions that occur during this process are caught and printed as error messages.

Step 2 Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'K-Nearest Neighbor':
2     self.build_train_knn()
```

Step 3 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Robust Scaler** radio button. Then, choose **K-Nearest Neighbor** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.10.

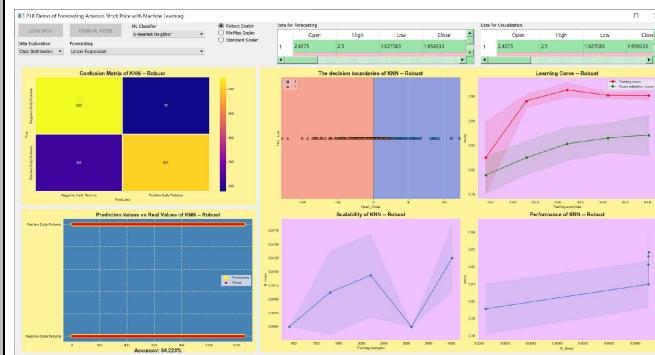


Figure 6.10 The result using KNN model with robust scaler

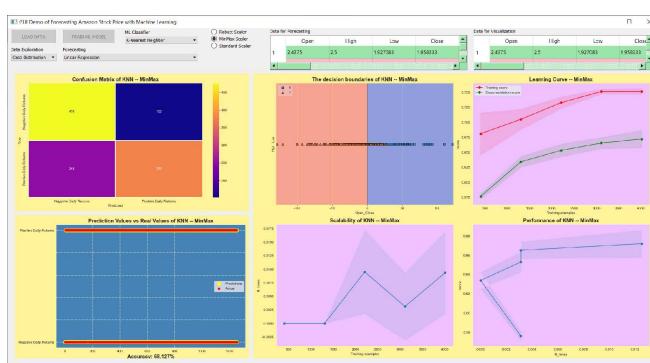


Figure 6.11 The result using KNN model with minmax scaler

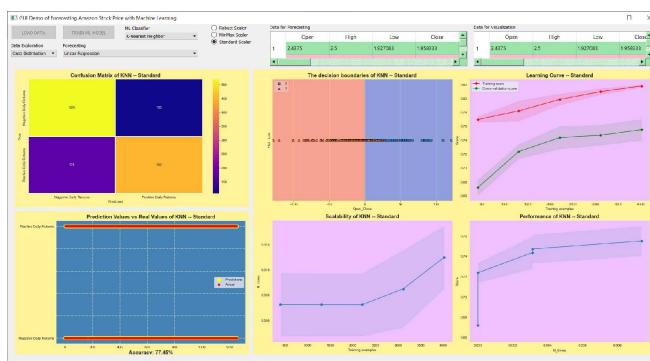


Figure 6.12 The result using KNN model with standard scaler

Click on **MinMax Scaler** radio button. Then, choose **K-Nearest Neighbor** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.11.

Click on **Standard Scaler** radio button. Then, choose **K-Nearest Neighbor** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.12.

PREDICTING DAILY RETURN USING RANDOM FOREST

PREDICTING DAILY RETURN USING RANDOM FOREST

Step 1 Define **build_train_rf()** method to build and train Random Forest (RF) classifier using three feature scaling: Robust Scaler, MinMax Scaler, and Standard Scaler:

```

1  def build_train_rf(self):
2      try:
3          if path.isfile('RFRob.pkl'):
4              #Loads model
5              self.RFRob = joblib.load('RFRob.pkl')
6              self.RFNorm = joblib.load('RFNorm.pkl')
7              self.RFStand = joblib.load('RFStand.pkl')
8
9          if self.rbRobust.isChecked():
10              self.run_model('RF', 'Robust',
11              self.RFRob, self.X_train_rob, self.X_test_rob,
12              self.y_train_rob, self.y_test_rob)

```

```
13             if self.rbMinMax.isChecked():
14                 self.run_model('RF', 'MinMax',
15 self.RFNorm, self.X_train_norm, self.X_test_norm,
16 self.y_train_norm, self.y_test_norm)
17
18             if self.rbStandard.isChecked():
19                 self.run_model('RF', 'Standard',
20 self.RFStand, self.X_train_stand, self.X_test_stand,
21 self.y_train_stand, self.y_test_stand)
22
23         else:
24             #Builds and trains Random Forest
25             self.RFRob =
26 RandomForestClassifier(n_estimators=200, max_depth=20,
27 random_state=2021)
28             self.RFNorm =
29 RandomForestClassifier(n_estimators=200, max_depth=20,
30 random_state=2021)
31             self.RFStand =
32 RandomForestClassifier(n_estimators=200, max_depth=20,
33 random_state=2021)
34
35             if self.rbRobust.isChecked():
36                 self.run_model('RF', 'Robust',
37 self.RFRob, self.X_train_rob, self.X_test_rob,
38 self.y_train_rob, self.y_test_rob)
39                 if self.rbMinMax.isChecked():
40                     self.run_model('RF', 'MinMax',
41 self.RFNorm, self.X_train_norm, self.X_test_norm,
42 self.y_train_norm, self.y_test_norm)
43
44                 if self.rbStandard.isChecked():
45                     self.run_model('RF', 'Standard',
46 self.RFStand, self.X_train_stand, self.X_test_stand,
47 self.y_train_stand, self.y_test_stand)
48
49         #Saves model
50         joblib.dump(self.RFRob, 'RFRob.pkl')
```

```

        joblib.dump(self.RFNorm, 'RFNorm.pkl')
        joblib.dump(self.RFStand, 'RFStand.pkl')
    except Exception as e:
        print("An error occurred: ", str(e))

```

Here's a step-by-step explanation of the code:

1. The `build_train_rf()` method is defined within a class. This method is responsible for building and training Random Forest models.
2. The code is wrapped in a try block, indicating that any exceptions that occur within the block will be caught and handled.
3. Inside the try block, the code checks if the files '`RFRob.pkl`', '`RFNorm.pkl`', and '`RFStand.pkl`' exist using the `path.isfile` function. This checks if the Random Forest model files have been saved previously.
4. If the model files exist, the code loads the Random Forest models from the files using `joblib.load`. The loaded models are assigned to the corresponding variables: `self.RFRob`, `self.RFNorm`, and `self.RFStand`.
5. The code then checks the status of three radio buttons: `self.rbRobust`, `self.rbMinMax`, and `self.rbStandard`. If any of these radio buttons are checked, it calls the `run_model` method with the appropriate arguments, passing the corresponding Random Forest model and data variables.
6. If the model files don't exist, the code creates new Random Forest models using `RandomForestClassifier` with specified parameters: `n_estimators=200`, `max_depth=20`, and `random_state=2021`. It creates three instances of `RandomForestClassifier` and assigns them to `self.RFRob`, `self.RFNorm`, and `self.RFStand`.
7. Similar to the previous step, the code checks the status of the radio buttons and calls the `run_model()` method with the appropriate arguments for each checked radio button.
8. After building and training the Random Forest models, the code saves the models to files using `joblib.dump`. The models are saved as '`RFRob.pkl`', '`RFNorm.pkl`', and '`RFStand.pkl`'.
9. If any exceptions occur within the try block, the code jumps to the except block. The exception is caught and assigned to the variable `e`. The code then prints an error message indicating that an error occurred, along with the details of the exception.

In summary, this code follows a similar structure as the previous examples. It checks if the Random Forest model files exist and loads them if they do. If the files don't exist, it creates new Random Forest models with specified parameters. It then runs the `run_model` method for each

selected radio button, passing the appropriate Random Forest model and data variables. Any exceptions that occur during this process are caught and printed as error messages.

Step 2

Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'Random Forest':
2     self.build_train_rf()
```

Step 3

Run **gui_amazon.py** and click **LOAD DATA** and **THE ML MODEL** buttons. Click on **Robust Scaler** radio button. Then, choose **Random Forest** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.13.

Click on **MinMax Scaler** radio button. Then, choose **Random Forest** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.14.

Click on **Standard Scaler** radio button. Then, choose **Random Forest** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.15.

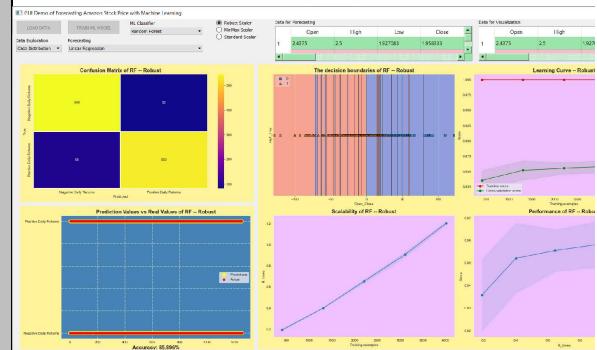


Figure 6.13 The result using RF model with robust scaling.

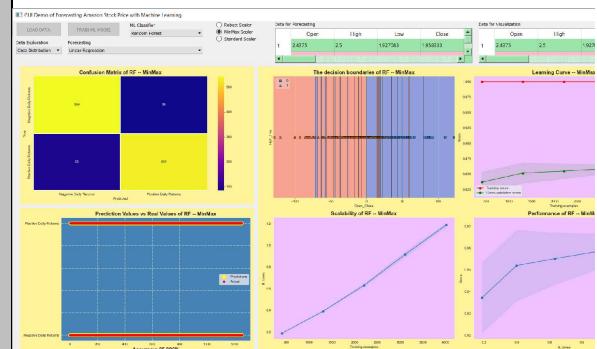


Figure 6.14 The result using RF model with minmax scaling.

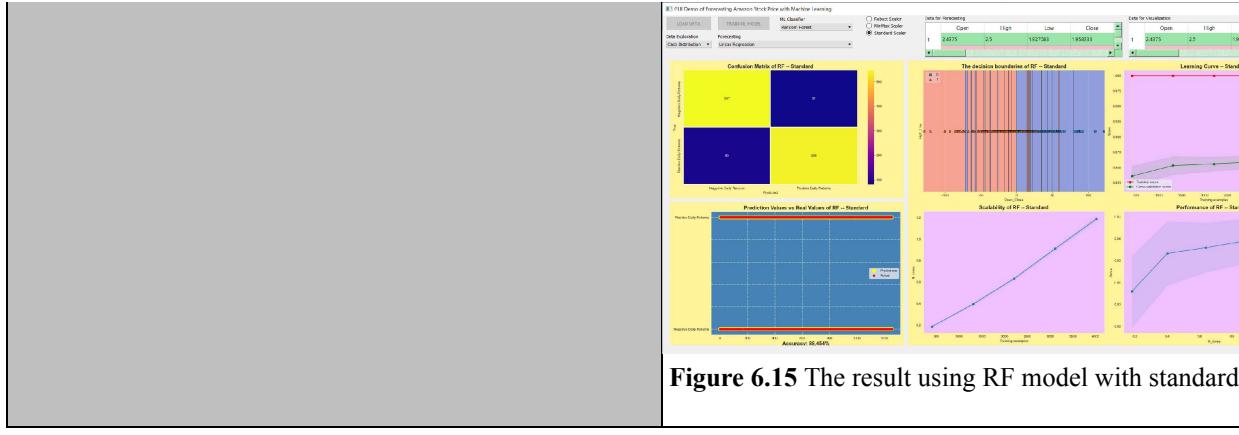


Figure 6.15 The result using RF model with standard

PREDICTING DAILY RETURN USING GRADIENT BOOSTING

PREDICTING DAILY RETURN USING GRADIENT BOOSTING

Step 1 Define **build_train_gb()** method to build and train Gradient Boosting (GB) classifier using three feature scaling: Robust Scaler, MinMax Scaler, and Standard Scaler:

```

1  def build_train_gb(self):
2      try:
3          if path.isfile('GBRob.pkl'):
4              #Loads model
5              self.GBRob = joblib.load('GBRob.pkl')
6              self.GBNorm = joblib.load('GBNorm.pkl')
7              self.GBStand = joblib.load('GBStand.pkl')
8
9          if self.rbRobust.isChecked():
10              self.run_model('Gradient Boosting',
11 'Robust', self.GBRob, self.X_train_rob, self.X_test_rob,
12 self.y_train_rob, self.y_test_rob)
13              if self.rbMinMax.isChecked():
14                  self.run_model('Gradient Boosting',
15 'MinMax', self.GBNorm, self.X_train_norm, self.X_test_norm,
16 self.y_train_norm, self.y_test_norm)
17
18          if self.rbStandard.isChecked():
19              self.run_model('Gradient Boosting',
20 'Standard', self.GBStand, self.X_train_stand,
21 self.X_test_stand, self.y_train_stand, self.y_test_stand)
22
23      else:
24          #Builds and trains Gradient Boosting
25          self.GBRob =
26 GradientBoostingClassifier(n_estimators = 50, max_depth=10,
27 subsample=0.8, max_features=0.2, random_state=2021)
28          self.GBNorm =
29 GradientBoostingClassifier(n_estimators = 50, max_depth=20,
30 subsample=0.8, max_features=0.2, random_state=2021)
31          self.GBStand =
32 GradientBoostingClassifier(n_estimators = 50, max_depth=20,
33 subsample=0.8, max_features=0.2, random_state=2021)
34
35          if self.rbRobust.isChecked():
36              self.run_model('Gradient Boosting',
37 'Robust', self.GBRob, self.X_train_rob, self.X_test_rob,
38 self.y_train_rob, self.y_test_rob)
39              if self.rbMinMax.isChecked():
40                  self.run_model('Gradient Boosting',
41 'MinMax', self.GBNorm, self.X_train_norm, self.X_test_norm,
42 self.y_train_norm, self.y_test_norm)
43
44          if self.rbStandard.isChecked():
45

```

```

47         self.run_model('Gradient Boosting',
'standard', self.GBStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

        #Saves model
        joblib.dump(self.GBRob, 'GBRob.pkl')
        joblib.dump(self.GBNorm, 'GBNorm.pkl')
        joblib.dump(self.GBStand, 'GBStand.pkl')
    except Exception as e:
        print("An error occurred: ", str(e))

```

Here's a step-by-step explanation of the code:

1. The `build_train_gb()` method is defined within a class. This method is responsible for building and training Gradient Boosting models.
2. The code is wrapped in a try block, indicating that any exceptions that occur within the block will be caught and handled.
3. Inside the try block, the code checks if the files '`GBRob.pkl`', '`GBNorm.pkl`', and '`GBStand.pkl`' exist using the `path.isfile` function. This checks if the Gradient Boosting model files have been saved previously.
4. If the model files exist, the code loads the Gradient Boosting models from the files using `joblib.load`. The loaded models are assigned to the corresponding variables: `self.GBRob`, `self.GBNorm`, and `self.GBStand`.
5. The code then checks the status of three radio buttons: `self.rbRobust`, `self.rbMinMax`, and `self.rbStandard`. If any of these radio buttons are checked, it calls the `run_model` method with the appropriate arguments, passing the corresponding Gradient Boosting model and data variables.
6. If the model files don't exist, the code creates new Gradient Boosting models using `GradientBoostingClassifier` with specified parameters: `n_estimators`, `max_depth`, `subsample`, `max_features`, and `random_state`. It creates three instances of `GradientBoostingClassifier` and assigns them to `self.GBRob`, `self.GBNorm`, and `self.GBStand`.
7. Similar to the previous step, the code checks the status of the radio buttons and calls the `run_model()` method with the appropriate arguments for each checked radio button.
8. After building and training the Gradient Boosting models, the code saves the models to files using `joblib.dump`. The models are saved as '`GBRob.pkl`', '`GBNorm.pkl`', and '`GBStand.pkl`'.
9. If any exceptions occur within the try block, the code jumps to the `except` block. The exception is caught and assigned to the variable `e`. The code then prints an error message indicating that an error occurred, along with the details of the exception.

In summary, this code follows a similar structure as the previous examples. It checks if the Gradient Boosting model files exist and loads them if they do. If the files don't exist, it creates new Gradient Boosting models with specified parameters. It then runs the `run_model` method for each selected radio button, passing the appropriate Gradient Boosting model and data variables. Any exceptions that occur during this process are caught and printed as error messages.

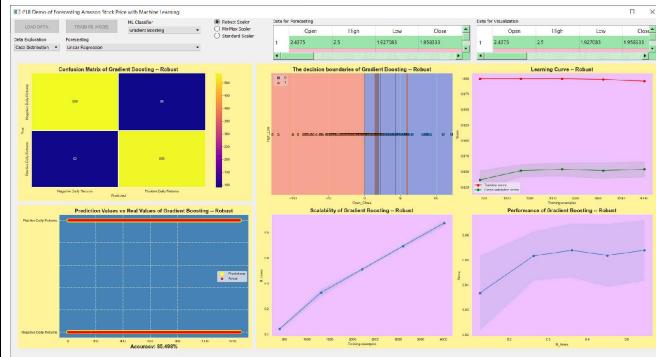


Figure 6.16 The result using GB model with robust scaler

Step 2 Add this code to the end of `choose_ML_model()` method:

```
1 if strCB == 'Gradient Boosting':
2     self.build_train_gb()
```

Step 3 Run `gui_amazon.py` and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Robust Scaler** radio button. Then, choose **Gradient Boosting** item from `cbClassifier` widget. Then, you will see the result as shown in Figure 6.16.

Click on **MinMax Scaler** radio button. Then, choose **Gradient Boosting** item from `cbClassifier` widget. Then, you will see the result as shown in Figure 6.17.

Click on **Standard Scaler** radio button. Then, choose **Gradient Boosting** item from `cbClassifier` widget. Then, you will see the result as shown in Figure 6.18.

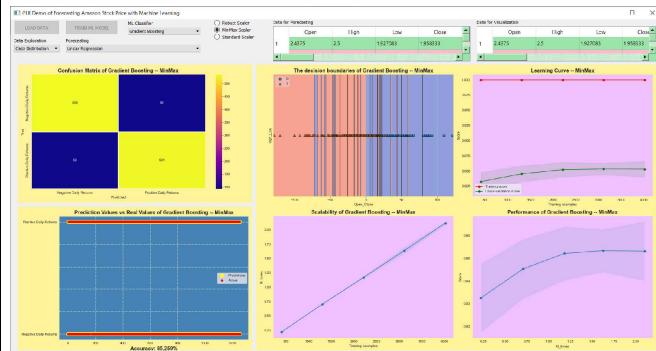


Figure 6.17 The result using GB model with minmax scaler

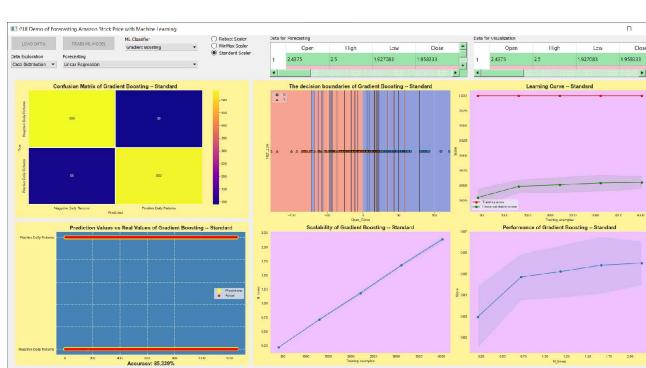


Figure 6.18 The result using GB model with standard scaler

PREDICTING DAILY RETURN USING NAÏVE BAYES

Step 1 Define `build_train_nb()` method to build and train Naïve Bayes (NB) classifier using three feature scaling: Robust Scaler, MinMax Scaler, and Standard Scaler:

```
1 def build_train_nb(self):
2     try:
3         if path.isfile('NBRob.pkl'):
4             #Loads model
5             self.NBRob = joblib.load('NBRob.pkl')
6             self.NBNorm = joblib.load('NBNorm.pkl')
7             self.NBStand = joblib.load('NBStand.pkl')
8
9         if self.rbRobust.isChecked():
10             self.run_model('Naive Bayes', 'Robust',
11 self.NBRob, self.X_train_rob, self.X_test_rob,
12 self.y_train_rob, self.y_test_rob)
13         if self.rbMinMax.isChecked():
14             self.run_model('Naive Bayes', 'MinMax',
15 self.NBNorm, self.X_train_norm, self.X_test_norm,
16 self.y_train_norm, self.y_test_norm)
17
18         if self.rbStandard.isChecked():
19             self.run_model('Naive Bayes',
20 'Standard', self.NBStand, self.X_train_stand,
21 self.X_test_stand, self.y_train_stand, self.y_test_stand)
22
23     else:
24         #Builds and trains Naive Bayes
25         self.NBRob = GaussianNB()
26         self.NBNorm = GaussianNB()
27         self.NBStand = GaussianNB()
28
29         if self.rbRobust.isChecked():
30             self.run_model('Naive Bayes', 'Robust',
31 self.NBRob, self.X_train_rob, self.X_test_rob,
32 self.y_train_rob, self.y_test_rob)
33         if self.rbMinMax.isChecked():
34             self.run_model('Naive Bayes', 'MinMax',
35 self.NBNorm, self.X_train_norm, self.X_test_norm,
36 self.y_train_norm, self.y_test_norm)
37
38         if self.rbStandard.isChecked():
39             self.run_model('Naive Bayes',
40 'Standard', self.NBStand, self.X_train_stand,
41 self.X_test_stand, self.y_train_stand, self.y_test_stand)
42
43
44         #Saves model
45         joblib.dump(self.NBRob, 'NBRob.pkl')
46         joblib.dump(self.NBNorm, 'NBNorm.pkl')
47         joblib.dump(self.NBStand, 'NBStand.pkl')
48 except Exception as e:
```

```
print("An error occurred: ", str(e))
```

Here's a step-by-step explanation of the code:

1. The **build_train_nb()** method is defined within a class. This method is responsible for building and training Naive Bayes models.
2. The code is wrapped in a try block, indicating that any exceptions that occur within the block will be caught and handled.
3. Inside the try block, the code checks if the files 'NBRob.pkl', 'NBNorm.pkl', and 'NBStand.pkl' exist using the path.isfile function. This checks if the Naive Bayes model files have been saved previously.
4. If the model files exist, the code loads the Naive Bayes models from the files using joblib.load. The loaded models are assigned to the corresponding variables: self.NBRob, self.NBNorm, and self.NBStand.
5. The code then checks the status of three radio buttons: self.rbRobust, self.rbMinMax, and self.rbStandard. If any of these radio buttons are checked, it calls the **run_model()** method with the appropriate arguments, passing the corresponding Naive Bayes model and data variables.
6. If the model files don't exist, the code creates new Naive Bayes models using GaussianNB. It creates three instances of GaussianNB and assigns them to self.NBRob, self.NBNorm, and self.NBStand.
7. Similar to the previous step, the code checks the status of the radio buttons and calls the **run_model()** method with the appropriate arguments for each checked radio button.
8. After building and training the Naive Bayes models, the code saves the models to files using joblib.dump. The models are saved as 'NBRob.pkl', 'NBNorm.pkl', and 'NBStand.pkl'.
9. If any exceptions occur within the try block, the code jumps to the except block. The exception is caught and assigned to the variable e. The code then prints an error message indicating that an error occurred, along with the details of the exception.

In summary, this code follows a similar structure as the previous examples. It checks if the Naive Bayes model files exist and loads them if they do. If the files don't exist, it creates new Naive Bayes models using GaussianNB. It then runs the **run_model()** method for each selected radio button, passing the appropriate Naive Bayes model and data variables. Any exceptions that occur during this process are caught and printed as error messages.

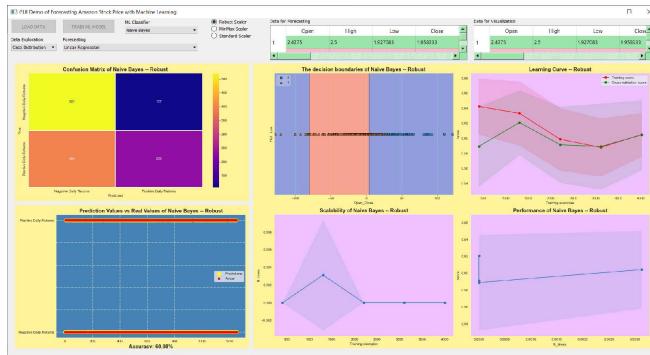


Figure 6.19 The result using NB model with robust scaler

Step 2 Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'Naive Bayes':
2     self.build_train_nb()
```

Step 3 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Robust Scaler** radio button. Then, choose **Naïve Bayes** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.19.

Click on **MinMax Scaler** radio button. Then, choose **Naïve Bayes** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.20.

Click on **Standard Scaler** radio button. Then, choose **Naïve Bayes** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.21.

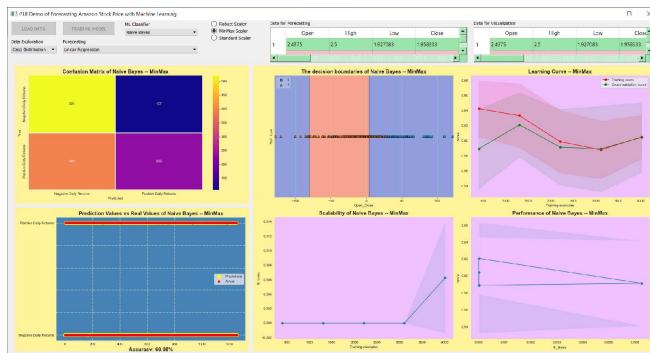


Figure 6.20 The result using NB model with minmax scaler

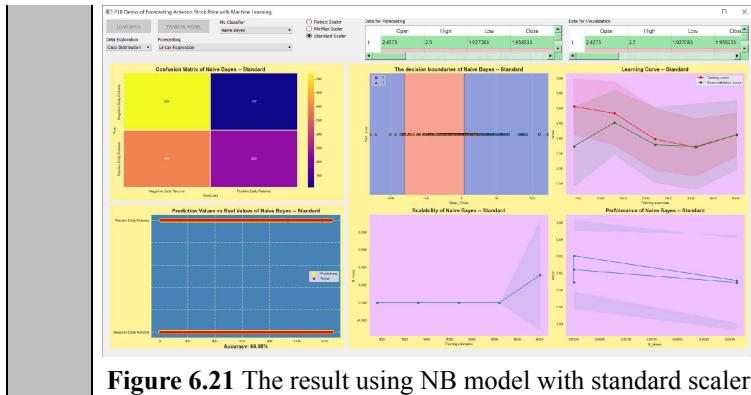


Figure 6.21 The result using NB model with standard scaler

PREDICTING DAILY RETURN USING ADABOOST

PREDICTING DAILY RETURN USING ADABOOST

Step 1 Define **build_train_ada()** method to build and train Adaboost classifier using three feature scaling: Robust Scaler, MinMax Scaler, and Standard Scaler:

```

1  def build_train_ada(self):
2      try:
3          if path.isfile('ADARob.pkl'):
4              #Loads model
5              self.ADARob = joblib.load('ADARob.pkl')
6              self.ADNorm = joblib.load('ADANorm.pkl')
7              self.ADAStand = joblib.load('ADAStand.pkl')
8
9                  if self.rbRobust.isChecked():
10                      self.run_model('Adaboost', 'Robust',
11 self.ADARob, self.X_train_rob, self.X_test_rob,
12 self.y_train_rob, self.y_test_rob)
13                  if self.rbMinMax.isChecked():
14                      self.run_model('Adaboost', 'MinMax',
15 self.ADNorm, self.X_train_norm, self.X_test_norm,
16 self.y_train_norm, self.y_test_norm)
17
18                  if self.rbStandard.isChecked():
19                      self.run_model('Adaboost', 'Standard',
20 self.ADAStand, self.X_train_stand, self.X_test_stand,
21 self.y_train_stand, self.y_test_stand)
22
23
24          else:
25              #Builds and trains Adaboost
26              self.ADARob =
27 AdaBoostClassifier(n_estimators = 50, learning_rate=0.01)
28              self.ADNorm =
29 AdaBoostClassifier(n_estimators = 50, learning_rate=0.01)
30              self.ADAStand =
31 AdaBoostClassifier(n_estimators = 50, learning_rate=0.01)
32
33                  if self.rbRobust.isChecked():
34                      self.run_model('Adaboost', 'Robust',
35 self.ADARob, self.X_train_rob, self.X_test_rob,
36 self.y_train_rob, self.y_test_rob)
37                  if self.rbMinMax.isChecked():
38                      self.run_model('Adaboost', 'MinMax',
39 self.ADNorm, self.X_train_norm, self.X_test_norm,
40 self.y_train_norm, self.y_test_norm)
41
42                  if self.rbStandard.isChecked():
43                      self.run_model('Adaboost', 'Standard',
44 self.ADAStand, self.X_train_stand, self.X_test_stand,
45 self.y_train_stand, self.y_test_stand)
46
47
#Saves model
joblib.dump(self.ADARob, 'ADARob.pkl')
joblib.dump(self.ADNorm, 'ADANorm.pkl')
```

```
        joblib.dump(self.ADARob, 'ADARob.pkl')
    except Exception as e:
        print("An error occurred: ", str(e))
```

Here's a step-by-step explanation of the code:

1. The **build_train_ada()** method is defined within a class. This method is responsible for building and training Adaboost models.
2. Similar to the previous methods, the code is wrapped in a try block to catch any exceptions that may occur.
3. Inside the try block, the code checks if the files 'ADARob.pkl', 'ADANorm.pkl', and 'ADAStand.pkl' exist using the `path.isfile` function. This checks if the Adaboost model files have been saved previously.
4. If the model files exist, the code loads the Adaboost models from the files using `joblib.load`. The loaded models are assigned to the corresponding variables: `self.ADARob`, `self.ADANorm`, and `self ADAStand`.
5. The code then checks the status of three radio buttons: `self.rbRobust`, `self.rbMinMax`, and `self.rbStandard`. If any of these radio buttons are checked, it calls the `run_model()` method with the appropriate arguments, passing the corresponding Adaboost model and data variables.
6. If the model files don't exist, the code creates new Adaboost models using `AdaBoostClassifier`. It creates three instances of `AdaBoostClassifier` and assigns them to `self.ADARob`, `self.ADANorm`, and `self ADAStand`.
7. Similar to the previous step, the code checks the status of the radio buttons and calls the `run_model()` method with the appropriate arguments for each checked radio button.
8. After building and training the Adaboost models, the code saves the models to files using `joblib.dump`. The models are saved as 'ADARob.pkl', 'ADANorm.pkl', and 'ADAStand.pkl'.
9. If any exceptions occur within the try block, the code jumps to the except block. The exception is caught and assigned to the variable `e`. The code then prints an error message indicating that an error occurred, along with the details of the exception.

In summary, this code follows the same structure as the previous examples. It checks if the Adaboost model files exist and loads them if they do. If the files don't exist, it creates new Adaboost models using `AdaBoostClassifier`. It then runs the `run_model()` method for each selected radio button, passing the appropriate Adaboost model and data variables. Any exceptions that occur during this process are caught and printed as error messages.

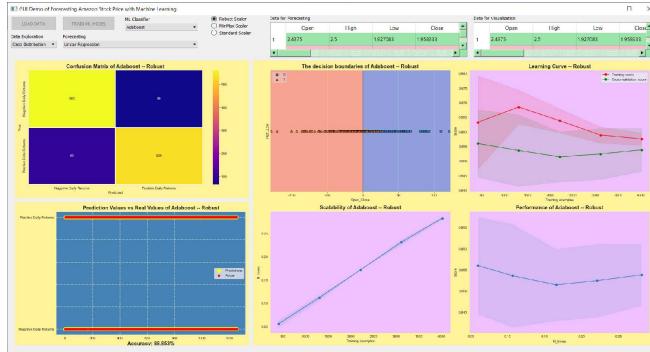


Figure 6.22 The result using Adaboost model with robust scaler

Step 2 Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'Adaboost':
2     self.build_train_ada()
```

Step 3 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Robust Scaler** radio button. Then, choose **Adaboost** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.22.

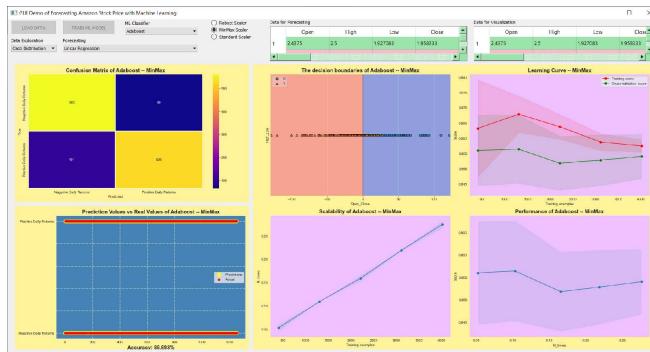


Figure 6.23 The result using Adaboost model with minmax scaler

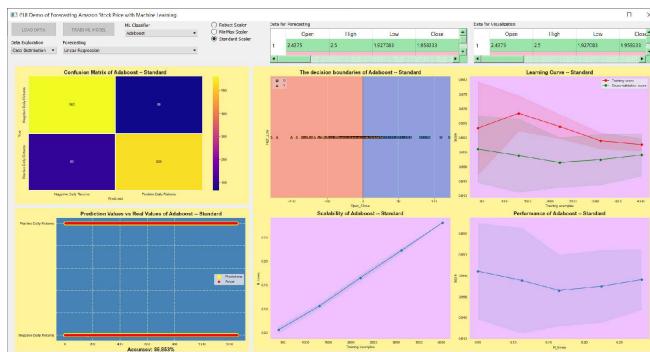


Figure 6.24 The result using Adaboost model with standard scaler

Click on **MinMax Scaler** radio button. Then, choose **Adaboost** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.23.

Click on **Standard Scaler** radio button. Then, choose **Adaboost** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.24.

PREDICTING DAILY RETURN USING EXTREME GRADIENT BOOSTING

PREDICTING DAILY RETURN USING EXTREME GRADIENT BOOSTING

Step 1 Define **build_train_xgb()** method to build and train XGB classifier using three feature scaling: Robust Scaler, MinMax Scaler, and Standard Scaler:

```

1  def build_train_xgb(self):
2      try:
3          if path.isfile('XGBRob.pkl'):
4              #Loads model
5              self.XGBRob = joblib.load('XGBRob.pkl')
6              self.XGBNorm = joblib.load('XGBNorm.pkl')
7              self.XGBStand = joblib.load('XGBStand.pkl')
8
9          if self.rbRobust.isChecked():
10             self.run_model('XGB', 'Robust',
11 self.XGBRob, self.X_train_rob, self.X_test_rob,
12 self.y_train_rob, self.y_test_rob)
13         if self.rbMinMax.isChecked():
14             self.run_model('XGB', 'MinMax',
15 self.XGBNorm, self.X_train_norm, self.X_test_norm,
16 self.y_train_norm, self.y_test_norm)
17
18         if self.rbStandard.isChecked():
19             self.run_model('XGB', 'Standard',
20 self.XGBStand, self.X_train_stand, self.X_test_stand,
21 self.y_train_stand, self.y_test_stand)
22
23     else:
24         #Builds and trains XGB classifier
25         self.XGBRob = XGBClassifier(n_estimators =
26 50, max_depth=10, random_state=2021,
27 use_label_encoder=False, eval_metric='mlogloss')
28         self.XGBNorm = XGBClassifier(n_estimators =
29 50, max_depth=10, random_state=2021,
30 use_label_encoder=False, eval_metric='mlogloss')
31         self.XGBStand = XGBClassifier(n_estimators =
32 50, max_depth=10, random_state=2021,
33 use_label_encoder=False, eval_metric='mlogloss')
34
35         if self.rbRobust.isChecked():
36             self.run_model('XGB', 'Robust',
37 self.XGBRob, self.X_train_rob, self.X_test_rob,
38 self.y_train_rob, self.y_test_rob)
39         if self.rbMinMax.isChecked():
40             self.run_model('XGB', 'MinMax',
41 self.XGBNorm, self.X_train_norm, self.X_test_norm,
42 self.y_train_norm, self.y_test_norm)
43
44         if self.rbStandard.isChecked():
45             self.run_model('XGB', 'Standard',
46 self.XGBStand, self.X_train_stand, self.X_test_stand,
47 self.y_train_stand, self.y_test_stand)
48
49     #Saves model
50     joblib.dump(self.XGBRob, 'XGBRob.pkl')
51     joblib.dump(self.XGBNorm, 'XGBNorm.pkl')
```

```

        joblib.dump(self.XGBStand, 'XGBStand.pkl')
    except Exception as e:
        print("An error occurred: ", str(e))

```

Here's a step-by-step explanation of the code:

1. The **build_train_xgb()** method is defined within a class. This method is responsible for building and training **XGBoost** models.
2. The code is wrapped in a try block to catch any exceptions that may occur during the process.
3. Inside the try block, the code checks if the files 'XGBRob.pkl', 'XGBNorm.pkl', and 'XGBStand.pkl' exist using the path.isfile function. This checks if the XGBoost model files have been saved previously.
4. If the model files exist, the code loads the XGBoost models from the files using joblib.load. The loaded models are assigned to the corresponding variables: self.XGBRob, self.XGBNorm, and self.XGBStand.
5. The code then checks the status of three radio buttons: self.rbRobust, self.rbMinMax, and self.rbStandard. If any of these radio buttons are checked, it calls the run_model() method with the appropriate arguments, passing the corresponding XGBoost model and data variables.
6. If the model files don't exist, the code creates new XGBoost models using XGBClassifier. It creates three instances of XGBClassifier and assigns them to self.XGBRob, self.XGBNorm, and self.XGBStand.
7. Similar to the previous step, the code checks the status of the radio buttons and calls the run_model() method with the appropriate arguments for each checked radio button.
8. After building and training the XGBoost models, the code saves the models to files using joblib.dump. The models are saved as 'XGBRob.pkl', 'XGBNorm.pkl', and 'XGBStand.pkl'.
9. If any exceptions occur within the try block, the code jumps to the except block. The exception is caught and assigned to the variable e. The code then prints an error message indicating that an error occurred, along with the details of the exception.

In summary, this code follows a similar structure to the previous examples. It checks if the XGBoost model files exist and loads them if they do. If the files don't exist, it creates new XGBoost models using **XGBClassifier**. It then runs the **run_model()** method for each selected radio button, passing the appropriate XGBoost model and data variables. Any exceptions that occur during this process are caught and printed as error messages.

Step 2	Add this code to the end of choose_ML_model() method:
--------	--

```

1 if strCB == 'XGB Classifier':
2     self.build_train_xgb()

```

Step 3 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Robust Scaler** radio button. Then, choose **XGB Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.25.

Click on **MinMax Scaler** radio button. Then, choose **XGB Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.26.

Click on **Standard Scaler** radio button. Then, choose **XGB Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.27.

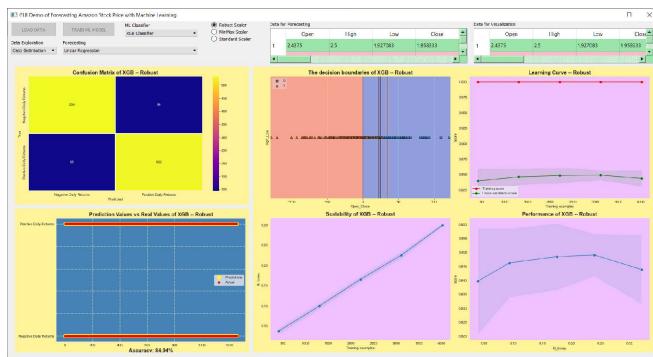


Figure 6.25 The result using XGB model with robust scaler

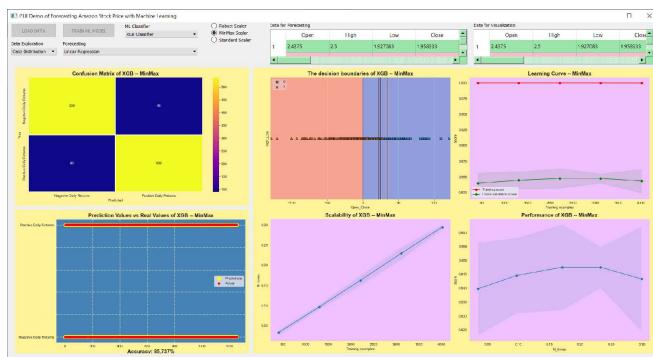


Figure 6.26 The result using XGB model with minmax scaler

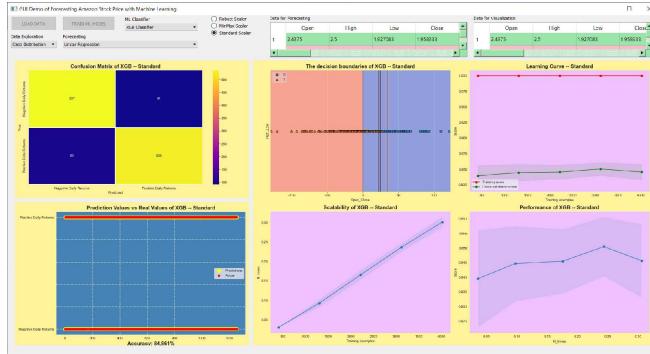


Figure 6.27 The result using XGB model with standard scaler

PREDICTING DAILY RETURN USING LIGHT GRADIENT BOOSTING MACHINE

PREDICTING DAILY RETURN USING LIGHT GRADIENT BOOSTING MACHINE

Step 1 Define **build_train_lgbm()** method to build and train LGBM classifier using three feature scaling: Robust Scaler, MinMax Scaler, and Standard Scaler:

```

1  def build_train_lgbm(self):
2      try:
3          if path.isfile('LGBMRob.pkl'):
4              #Loads model
5              self.LGBMRob = joblib.load('LGBMRob.pkl')
6              self.LGBMNorm = joblib.load('LGBMNorm.pkl')
7              self.LGBMStand =
8              joblib.load('LGBMStand.pkl')
9
10         if self.rbRobust.isChecked():
11             self.run_model('LGBM', 'Robust',
12 self.LGBMRob, self.X_train_rob, self.X_test_rob,
13 self.y_train_rob, self.y_test_rob)
14         if self.rbMinMax.isChecked():
15             self.run_model('LGBM', 'MinMax',
16 self.LGBMNorm, self.X_train_norm, self.X_test_norm,
17 self.y_train_norm, self.y_test_norm)
18
19         if self.rbStandard.isChecked():
20             self.run_model('LGBM', 'Standard',
21 self.LGBMStand, self.X_train_stand, self.X_test_stand,
22 self.y_train_stand, self.y_test_stand)
23
24     else:
25         #Builds and trains LGBM classifier
26         self.LGBMRob = LGBMClassifier(max_depth =
27 10, n_estimators=50, subsample=0.8, random_state=2021)
28         self.LGBMNorm = LGBMClassifier(max_depth =
29 10, n_estimators=50, subsample=0.8, random_state=2021)
30         self.LGBMStand = LGBMClassifier(max_depth =
31 10, n_estimators=50, subsample=0.8, random_state=2021)
32         if self.rbRobust.isChecked():
33             self.run_model('LGBM', 'Robust',
34 self.LGBMRob, self.X_train_rob, self.X_test_rob,
35 self.y_train_rob, self.y_test_rob)
36         if self.rbMinMax.isChecked():
37             self.run_model('LGBM', 'MinMax',
38 self.LGBMNorm, self.X_train_norm, self.X_test_norm,
39 self.y_train_norm, self.y_test_norm)
40
41         if self.rbStandard.isChecked():
42             self.run_model('LGBM', 'Standard',
43 self.LGBMStand, self.X_train_stand, self.X_test_stand,
44 self.y_train_stand, self.y_test_stand)
45

```

```

    #Saves model
    joblib.dump(self.LGBMRob, 'LGBMRob.pkl')
    joblib.dump(self.LGBMNorm, 'LGBMNorm.pkl')
    joblib.dump(self.LGBMStand, 'LGBMStand.pkl')
except Exception as e:
    print("An error occurred: ", str(e))

```

Here's an explanation of how the code works:

1. The **build_train_xgb()** method is defined within a class. This method is responsible for building and training XGBoost models.
2. The code is wrapped in a try block to catch any exceptions that may occur during the process.
3. Inside the try block, the code checks if the files 'XGBRob.pkl', 'XGBNorm.pkl', and 'XGBStand.pkl' exist using the path.isfile function. This checks if the XGBoost model files have been saved previously.
4. If the model files exist, the code loads the XGBoost models from the files using joblib.load. The loaded models are assigned to the corresponding variables: self.XGBRob, self.XGBNorm, and self.XGBStand.
5. The code then checks the status of three radio buttons: self.rbRobust, self.rbMinMax, and self.rbStandard. If any of these radio buttons are checked, it calls the **run_model()** method with the appropriate arguments, passing the corresponding XGBoost model and data variables.
6. If the model files don't exist, the code creates new XGBoost models using XGBClassifier. It creates three instances of XGBClassifier and assigns them to self.XGBRob, self.XGBNorm, and self.XGBStand.
7. Similar to the previous step, the code checks the status of the radio buttons and calls the run_model() method with the appropriate arguments for each checked radio button.
8. After building and training the XGBoost models, the code saves the models to files using joblib.dump. The models are saved as 'XGBRob.pkl', 'XGBNorm.pkl', and 'XGBStand.pkl'.
9. If any exceptions occur within the try block, the code jumps to the except block. The exception is caught and assigned to the variable e. The code then prints an error message indicating that an error occurred, along with the details of the exception.

In summary, this code follows a similar structure to the previous examples. It checks if the XGBoost model files exist and loads them if they do. If the files don't exist, it creates new XGBoost models using XGBClassifier. It then runs the run_model() method for each selected radio button, passing the appropriate XGBoost model and data variables. Any

exceptions that occur during this process are caught and printed as error messages.

Step 2 Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'LGBM Classifier':
2     self.build_train_lgbm()
```

Step 3 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Robust Scaler** radio button. Then, choose **LGBM Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.28.

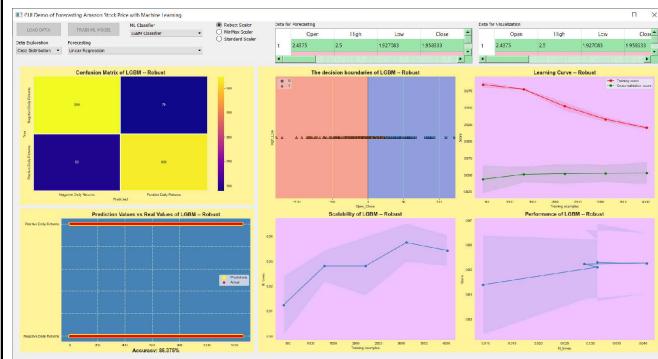


Figure 6.28 The result using LGBM model with robust scaler

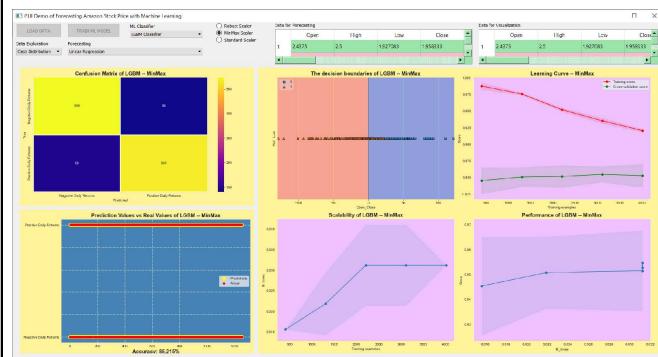


Figure 6.29 The result using LGBM model with minmax scaler

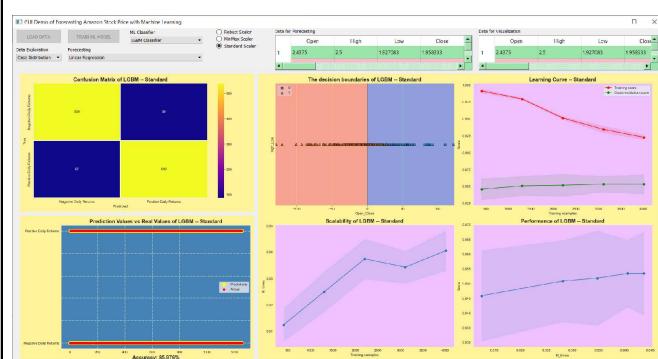


Figure 6.30 The result using LGBM model with standard scaler

Click on **MinMax Scaler** radio button. Then, choose **LGBM Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.29.

Click on **Standard Scaler** radio button. Then, choose **LGBM Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.30.

PREDICTING DAILY RETURN USING MULTI LAYER PERCEPTRON

PREDICTING DAILY RETURN USING MULTI LAYER PERCEPTRON

Step 1 Define **build_train_mlp()** method to build and train MLP classifier using three feature scaling: Robust Scaler, MinMax Scaler, and Standard Scaler:

```

1  def build_train_mlp(self):
2      try:
3          if path.isfile('MLPRob.pkl'):
4              #Loads model
5              self.MLPRob = joblib.load('MLPRob.pkl')
6              self.MLPNorm = joblib.load('MLPNorm.pkl')
7              self.MLPStand = joblib.load('MLPStand.pkl')
8
9          if self.rbRobust.isChecked():
10              self.run_model('MLP', 'Robust',
11              self.MLPRob, self.X_train_rob, self.X_test_rob,
12              self.y_train_rob, self.y_test_rob)
13              if self.rbMinMax.isChecked():
14                  self.run_model('MLP', 'MinMax',
15                  self.MLPNorm, self.X_train_norm, self.X_test_norm,
16                  self.y_train_norm, self.y_test_norm)
17
18          if self.rbStandard.isChecked():
19              self.run_model('MLP', 'Standard',
20              self.MLPStand, self.X_train_stand, self.X_test_stand,
21              self.y_train_stand, self.y_test_stand)
22
23      else:
24          #Builds and trains MLP classifier
25          self.MLPRob =
26          MLPClassifier(random_state=2021)
27          self.MLPNorm =
28          MLPClassifier(random_state=2021)
29          self.MLPStand =
30          MLPClassifier(random_state=2021)
31
32          if self.rbRobust.isChecked():
33              self.run_model('MLP', 'Robust',
34              self.MLPRob, self.X_train_rob, self.X_test_rob,
35              self.y_train_rob, self.y_test_rob)
36              if self.rbMinMax.isChecked():
37                  self.run_model('MLP', 'MinMax',
38                  self.MLPNorm, self.X_train_norm, self.X_test_norm,
39                  self.y_train_norm, self.y_test_norm)
40
41          if self.rbStandard.isChecked():
42              self.run_model('MLP', 'Standard',
43              self.MLPStand, self.X_train_stand, self.X_test_stand,
44              self.y_train_stand, self.y_test_stand)
45
46      #Saves model
47      joblib.dump(self.MLPRob, 'MLPRob.pkl')
```

```

        joblib.dump(self.MLPNorm, 'MLPNorm.pkl')
        joblib.dump(self.MLPStand, 'MLPStand.pkl')
    except Exception as e:
        print("An error occurred: ", str(e))

```

Here's an explanation of how the code works:

1. The **build_train_mlp()** method is defined within a class. This method is responsible for building and training MLP classifiers.
2. Similar to the previous methods, the code is wrapped in a try block to catch any exceptions that may occur during the process.
3. Inside the try block, the code checks if the files 'MLPRob.pkl', 'MLPNorm.pkl', and 'MLPStand.pkl' exist using the path.isfile function. This checks if the MLP model files have been saved previously.
4. If the model files exist, the code loads the MLP models from the files using joblib.load. The loaded models are assigned to the corresponding variables: self.MLPRob, self.MLPNorm, and self.MLPStand.
5. The code then checks the status of three radio buttons: self.rbRobust, self.rbMinMax, and self.rbStandard. If any of these radio buttons are checked, it calls the run_model() method with the appropriate arguments, passing the corresponding MLP model and data variables.
6. If the model files don't exist, the code creates new MLP classifiers using MLPClassifier. It creates three instances of MLPClassifier and assigns them to self.MLPRob, self.MLPNorm, and self.MLPStand.
7. Similar to the previous step, the code checks the status of the radio buttons and calls the run_model() method with the appropriate arguments for each checked radio button.
8. After building and training the MLP classifiers, the code saves the models to files using joblib.dump. The models are saved as 'MLPRob.pkl', 'MLPNorm.pkl', and 'MLPStand.pkl'.
9. If any exceptions occur within the try block, the code jumps to the except block. The exception is caught and assigned to the variable e. The code then prints an error message indicating that an error occurred, along with the details of the exception.

In summary, this code follows a similar structure to the previous examples. It checks if the MLP model files exist and loads them if they do. If the files don't exist, it creates new MLP classifiers using MLPClassifier. It then runs the run_model method for each selected radio button, passing the appropriate MLP model and data variables. Any exceptions that occur during this process are caught and printed as error messages.

Step 2	Add this code to the end of choose_ML_model() method:
--------	--

```

1 if strCB == 'MLP Classifier':
2     self.build_train_mlp()

```

- Step 3 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Robust Scaler** radio button. Then, choose **MLP Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.31.

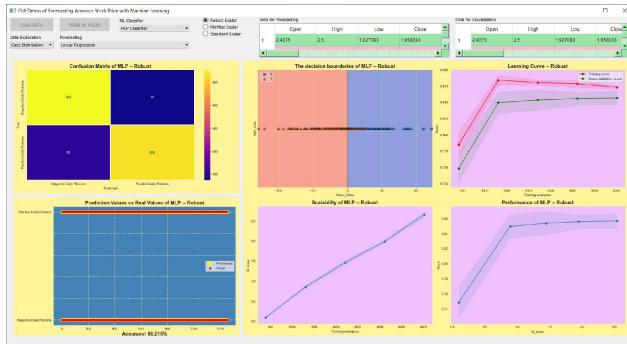


Figure 6.31 The result using MLP model with robust scaler

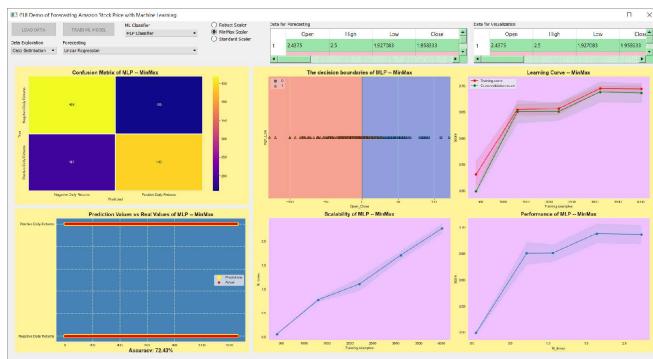


Figure 6.32 The result using MLP model with minmax scaler

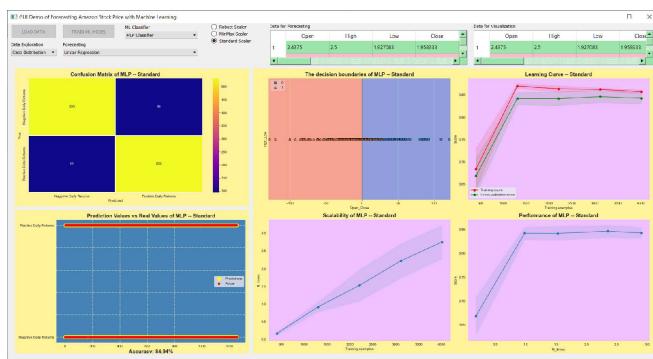


Figure 6.33 The result using MLP model with standard scaler

Click on **MinMax Scaler** radio button. Then, choose **MLP Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.32.

Click on **Standard Scaler** radio button. Then, choose **MLP Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.33.

PREDICTING DAILY RETURN USING EXTRA TREES

Step 1 Define **build_train_extra()** method to build and train Extra Trees classifier using three feature scaling: Robust Scaler, MinMax Scaler, and Standard Scaler:

```

1  def build_train_extra(self):
2      try:
3          if path.isfile('ExtraRob.pkl'):
4              #Loads model
5              self.ExtraRob = joblib.load('ExtraRob.pkl')
6              self.ExtraNorm =
7              joblib.load('ExtraNorm.pkl')
8              self.ExtraStand =
9              joblib.load('ExtraStand.pkl')
10
11         if self.rbRobust.isChecked():
12             self.run_model('Extra Trees', 'Robust',
13             self.ExtraRob, self.X_train_rob, self.X_test_rob,
14             self.y_train_rob, self.y_test_rob)
15         if self.rbMinMax.isChecked():
16             self.run_model('Extra Trees', 'MinMax',
17             self.ExtraNorm, self.X_train_norm, self.X_test_norm,
18             self.y_train_norm, self.y_test_norm)
19
20         if self.rbStandard.isChecked():
21             self.run_model('Extra Trees',
22             'Standard', self.ExtraStand, self.X_train_stand,
23             self.X_test_stand, self.y_train_stand, self.y_test_stand)
24
25     else:
26         #Builds and trains Gaussian Mixture
27 classifier
28
29         self.ExtraRob =
30 ExtraTreesClassifier(n_estimators=200, random_state=100)
31         self.ExtraNorm =
32 ExtraTreesClassifier(n_estimators=200, random_state=100)
33         self.ExtraStand =
34 ExtraTreesClassifier(n_estimators=200, random_state=100)
35
36         if self.rbRobust.isChecked():
37             self.run_model('Extra Trees', 'Robust',
38             self.ExtraRob, self.X_train_rob, self.X_test_rob,
39             self.y_train_rob, self.y_test_rob)
40         if self.rbMinMax.isChecked():
41             self.run_model('Extra Trees', 'MinMax',
42             self.ExtraNorm, self.X_train_norm, self.X_test_norm,
43             self.y_train_norm, self.y_test_norm)
44
45         if self.rbStandard.isChecked():
46             self.run_model('Extra Trees',
47             'Standard', self.ExtraStand, self.X_train_stand,
             self.X_test_stand, self.y_train_stand, self.y_test_stand)
48
49         #Saves model
50         joblib.dump(self.ExtraRob, 'ExtraRob.pkl')
51         joblib.dump(self.ExtraNorm, 'ExtraNorm.pkl')
52         joblib.dump(self.ExtraStand,
53             'ExtraStand.pkl')
54     except Exception as e:
55         print("An error occurred: ", str(e))

```

Here's an explanation of how the code works:

1. The **build_train_extra()** method is defined within a class. This method is responsible for building and training Extra Trees classifiers.
2. Similar to the previous methods, the code is wrapped in a try block to catch any exceptions that may occur during the process.
3. Inside the try block, the code checks if the files 'ExtraRob.pkl', 'ExtraNorm.pkl', and 'ExtraStand.pkl' exist using the path.isfile function. This checks if the Extra Trees model files have been saved previously.
4. If the model files exist, the code loads the Extra Trees models from the files using joblib.load. The loaded models are assigned to the corresponding variables: self.ExtraRob, self.ExtraNorm, and self.ExtraStand.
5. The code then checks the status of three radio buttons: self.rbRobust, self.rbMinMax, and self.rbStandard. If any of these radio buttons are checked, it calls the run_model method with the appropriate arguments, passing the corresponding Extra Trees model and data variables.
6. If the model files don't exist, the code creates new Extra Trees classifiers using ExtraTreesClassifier. It creates three instances of ExtraTreesClassifier and assigns them to self.ExtraRob, self.ExtraNorm, and self.ExtraStand.
7. Similar to the previous step, the code checks the status of the radio buttons and calls the run_model() method with the appropriate arguments for each checked radio button.
8. After building and training the Extra Trees classifiers, the code saves the models to files using joblib.dump. The models are saved as 'ExtraRob.pkl', 'ExtraNorm.pkl', and 'ExtraStand.pkl'.
9. If any exceptions occur within the try block, the code jumps to the except block. The exception is caught and assigned to the variable e. The code then prints an error message indicating that an error occurred, along with the details of the exception.

In summary, this code follows a similar structure to the previous examples. It checks if the Extra Trees model files exist and loads them if they do. If the files don't exist, it creates new Extra Trees classifiers using ExtraTreesClassifier. It then runs the run_model() method for each selected radio button, passing the appropriate Extra Trees model and data variables. Any exceptions that occur during this process are caught and printed as error messages.

Step 2 Add this code to the end of **choose_ML_model()** method:

```

1     if strCB == 'Extra Trees Classifier':
2         self.build_train_extra()

```

- Step 3 Run **gui_amazon.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Robust Scaler** radio button. Then, choose **Extra Trees Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.34.

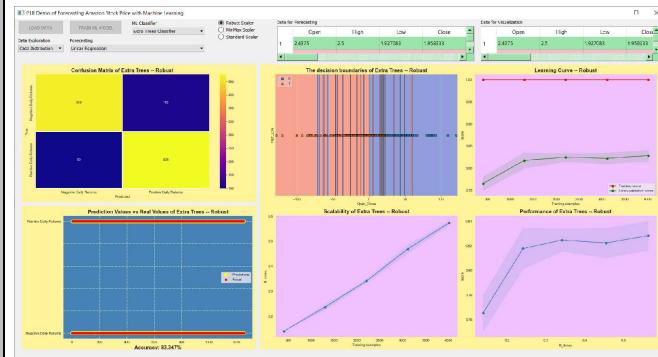


Figure 6.34 The result using Extra Trees model with robust scaler

Click on **MinMax Scaler** radio button. Then, choose **Extra Trees Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.35.

Click on **Standard Scaler** radio button. Then, choose **Extra Trees Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 6.36.

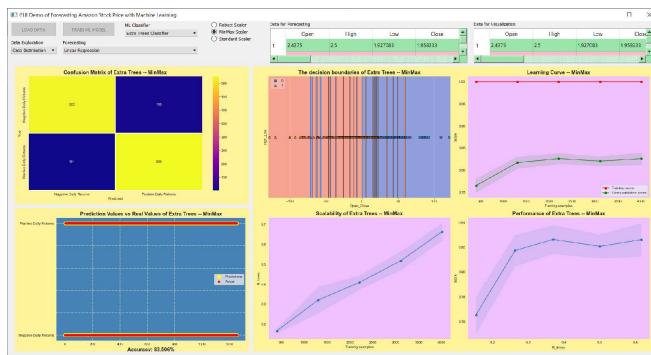


Figure 6.35 The result using Extra Trees model with minmax scaler

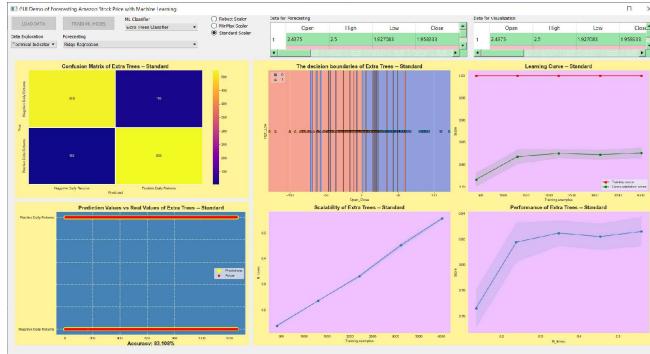


Figure 6.36 The result using Extra Trees model with standard scaler

This is the final version of `gui_amazon.py`:

```
#gui_amazon.py
from PyQt5.QtWidgets import *
from PyQt5.uic import loadUi
from matplotlib.colors import ListedColormap

import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import datetime as dt
sns.set_style('darkgrid')
from sklearn.preprocessing import LabelEncoder
import warnings
warnings.filterwarnings('ignore')
import os
from os import path
import joblib
import itertools
from sklearn.metrics import roc_auc_score,roc_curve,
explained_variance_score, r2_score
from sklearn.model_selection import cross_val_score
from statsmodels.tsa.seasonal import seasonal_decompose as season
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.metrics import accuracy_score, balanced_accuracy_score
from sklearn.model_selection import train_test_split, RandomizedSearchCV,
GridSearchCV,StratifiedKFold
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
RobustScaler
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
```

```

from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler, \
    LabelEncoder, OneHotEncoder
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score, precision_score
from sklearn.metrics import classification_report, f1_score, plot_confusion_matrix
from catboost import CatBoostClassifier
from lightgbm import LGBMClassifier
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import learning_curve
from mlxtend.plotting import plot_decision_regions
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.pipeline import make_pipeline
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import GradientBoostingRegressor
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
from catboost import CatBoostRegressor
from sklearn.neural_network import MLPRegressor
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from statsmodels.tsa.ststools import adfuller, acf, pacf
from statsmodels.tsa.arima_model import ARIMA
from pmdarima.utils import decomposed_plot
from pmdarima.arima import decompose
from pmdarima import auto_arima
from sklearn.linear_model import LassoCV
from sklearn.linear_model import RidgeCV
from sklearn.mixture import GaussianMixture

class DemoGUI_Amazon(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)
        loadUi("gui_amazon.ui", self)
        self.setWindowTitle(\n            "GUI Demo of Forecasting Amazon Stock Price with Machine Learning")
        self.initial_state(False)

    def initial_state(self, state):
        self.pbTrainML.setEnabled(state)
        self.cbData.setEnabled(state)
        self.cbForecasting.setEnabled(state)
        self.cbClassifier.setEnabled(state)
        self.rbRobust.setEnabled(state)
        self.rbMinMax.setEnabled(state)
        self.rbStandard.setEnabled(state)
        self.pbLoad.clicked.connect(self.import_dataset)
        self.cbData.currentIndexChanged.connect(self.choose_plot)
        self.cbForecasting.currentIndexChanged.connect(self.choose_forecasting)
        self.pbTrainML.clicked.connect(self.train_model_ML)
        self.cbClassifier.currentIndexChanged.connect(self.choose_ML_model)

    # Takes a df and writes it to a qtable provided. df headers become qtable headers
    @staticmethod
    def write_df_to_qtable(df, table):
        headers = list(df)
        table.setRowCount(df.shape[0])
        table.setColumnCount(df.shape[1])
        table.setHorizontalHeaderLabels(headers)

        # getting data from df is computationally costly so convert it to array first
        df_array = df.values
        for row in range(df.shape[0]):
            for col in range(df.shape[1]):
                table.setItem(row, col, QTableWidgetItem(str(df_array[row,col])))

    def populate_table(self, data, table):
        #Populates two tables

```

```

        self.write_df_to_qtable(data,table)

        table.setAlternatingRowColors(True)
        table.setStyleSheet("alternate-background-color: #ffbacd;background-color: #9be5aa;");

    def create_new_dfs(self,df):
        #Extracts day, month, week, quarter, and year
        df['Date'] = pd.to_datetime(df['Date'])
        df['Day'] = df['Date'].dt.weekday
        df['Month'] = df['Date'].dt.month
        df['Year'] = df['Date'].dt.year
        df['Week']= df['Date'].dt.week
        df['Quarter']= df['Date'].dt.quarter

        #Sets Date column as index
        df = df.set_index("Date")

        #Creates a dummy dataframe for visualization
        df_dummy=df.copy()

        #Computes year-wise and month-wise data
        self.year_data_mean, self.year_data_ewm, self.monthly_data_mean, self.monthly_data_ewm = self.compute_year_month_wi

        #Converts days, months, and quarters from numerics to meaningful string
        days = {0:'Sunday',1:'Monday',2:'Tuesday',3:'Wednesday',4:'Thursday',5: 'Friday',6:'Saturday'}
        df_dummy['Day'] = df_dummy['Day'].map(days)
        months=
{1: 'January',2: 'February',3: 'March',4: 'April',5: 'May',6: 'June',7: 'July',8: 'August',9: 'September',10: 'October',11: 'November'
df_dummy['Month']= df_dummy['Month'].map(months)
quarters = {1: 'Jan-March', 2: 'April-June',3: 'July-Sept',4: 'Oct-Dec'}
df_dummy['Quarter'] = df_dummy['Quarter'].map(quarters)

        return df, df_dummy

    def compute_daily_returns(self, df):
        """Compute and return the daily return values."""
        # TODO: Your code here
        # Note: Returned DataFrame must have the same number of rows
        daily_return = (df / df.shift(1)) - 1
        daily_return[0] = 0
        return daily_return

    def calculate_SMA(self, df, peroids=15):
        SMA = df.rolling(window=peroids, min_periods=peroids, center=False).mean()
        return SMA

    def calculate_MACD(self, df, nslow=26, nfast=12):
        emaslow = df.ewm(span=nslow, min_periods=nslow, adjust=True, ignore_na=False).mean()
        emafast = df.ewm(span=nfast, min_periods=nfast, adjust=True, ignore_na=False).mean()
        dif = emafast - emaslow
        MACD = dif.ewm(span=9, min_periods=9, adjust=True, ignore_na=False).mean()
        return dif, MACD

    def calculate_RSI(self, df, periods=14):
        # wilder's RSI
        delta = df.diff()
        up, down = delta.copy(), delta.copy()

        up[up < 0] = 0
        down[down > 0] = 0

        rUp = up.ewm(com=periods,adjust=False).mean()
        rDown = down.ewm(com=periods, adjust=False).mean().abs()

        rsi = 100 - 100 / (1 + rUp / rDown)
        return rsi

    def calculate_BB(self, df, peroids=15):
        STD = df.rolling(window=peroids,min_periods=peroids, center=False).std()
        SMA = self.calculate_SMA(df)
        upper_band = SMA + (2 * STD)
        lower_band = SMA - (2 * STD)
        return upper_band, lower_band

    def calculate_stdev(self, df,periods=5):

```

```

STDEV = df.rolling(periods).std()
return STDEV

def compute_technical_indicators(self,df):
    stock_close=df["Adj Close"]
    daily_returns=self.compute_daily_returns(stock_close)
    SMA_CLOSE = self.calculate_SMA(stock_close)
    upper_band, lower_band = self.calculate_BB(stock_close)
    DIF, MACD = self.calculate_MACD(stock_close)
    RSI = self.calculate_RSI(stock_close)
    STDEV= self.calculate_stdev(stock_close)
    Open_Close=df.Open - df["Adj Close"]
    High_Low=df.High-df.Low

    df['daily_returns'] = daily_returns
    df['SMA'] = SMA_CLOSE
    df['Upper_band'] = upper_band
    df['Lower_band'] = lower_band
    df['DIF'] = DIF
    df['MACD'] = MACD
    df['RSI'] = RSI
    df['STDEV'] = STDEV
    df['Open_Close']=Open_Close
    df['High_Low']=High_Low

#Checks null values because of technical indicators
print(df.isnull().sum().to_string())
print('Total number of null values: ', df.isnull().sum().sum())

#Fills each null value in every column with mean value
cols = list(df.columns)
for n in cols:
    df[n].fillna(df[n].mean(),inplace = True)

#Checks again null values
print(df.isnull().sum().to_string())
print('Total number of null values: ', df.isnull().sum().sum())

def compute_year_month_wise(self, df):
    cols = list(df.columns)
    cols.remove("Month")
    cols.remove("Day")
    cols.remove("Week")
    cols.remove("Year")
    cols.remove("Quarter")
    #Resamples the data year-wise by mean
    year_data_mean = df[cols].resample('y').mean()

    #Resamples the data year-wise by ewm
    year_data_ewm=year_data_mean.ewm(span=5).mean()

    #Resamples the data month-wise by mean
    monthly_data_mean = df[cols].resample('m').mean()

    #Resamples the data month-wise by EWM
    monthly_data_ewm=monthly_data_mean.ewm(span=5).mean()

    return year_data_mean, year_data_ewm, monthly_data_mean, monthly_data_ewm

def read_dataset(self, dir):
    #Reads dataset
    df = pd.read_csv(dir)

    #Creates new Dataframes
    self.df, self.df_dummy=self.create_new_dfs(df)

    #Splits data for regression
    self.X_final, self.y_final, self.X_train, self.y_train, \
        self.X_test, self.y_test, self.X_val, self.y_val=self.split_data_regression(self.df)

    #Computes technical indicators
    self.compute_technical_indicators(self.df)

def import_dataset(self):
    curr_path = os.getcwd()
    dataset_dir = curr_path + "/Amazon.csv"

    self.read_dataset(dataset_dir)

```

```

print("Dataframe has been read...")

#Populates cbData and cbForecasting
self.populate_cbData()
self.populate_cbForecasting()

#Populates tables with data
self.populate_table(self.df, self.twData1)
self.label1.setText('Data for Forecasting')

self.populate_table(self.df_dummy, self.twData2)
self.label2.setText('Data for Visualization')

#Turns off pbLoad
self.pbLoad.setEnabled(False)

#Turns on cbForecasting and cbData
self.cbForecasting.setEnabled(True)
self.cbData.setEnabled(True)

#Turns on pbTrainML widget
self.pbTrainML.setEnabled(True)

def populate_cbForecasting(self):
    self.cbForecasting.addItems(["Linear Regression", "Random Forest Regression"])
    self.cbForecasting.addItems(["Decision Tree Regression", "KNN Regression"])
    self.cbForecasting.addItems(["Adaboost Regression", "Gradient Boosting Regression"])
    self.cbForecasting.addItems(["XGB Regression", "LGBM Regression"])
    self.cbForecasting.addItems(["Catboost Regression", "SVR Regression"])
    self.cbForecasting.addItems(["MLP Regression", "Lasso Regression", "Ridge Regression"])

def populate_cbData(self):
    self.cbData.addItems(["Case Distribution"])
    self.cbData.addItems(["High", "Low", "Open", "Close"])
    self.cbData.addItems(["Adj Close", "Volume", "Technical Indicators"])
    self.cbData.addItems(["Year-Wise", "Month-Wise"])

#Defines function to plot case distribution of a categorical feature bar plot
def plot_barchart(self, df, var, widget):
    ax = df[var].value_counts().plot(kind="barh", ax = widget.canvas.axis1)
    for i,j in enumerate(df[var].value_counts().values):
        ax.text(.7,i,j,weight = "bold", fontsize=10)

    widget.canvas.axis1.set_title("Case distribution "+ " of " + var +" variable", fontsize=14)
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()

#Defines function to plot case distribution of a categorical feature in pie chart
def plot_piechart(self, df, var, widget):
    label_list = list(df[var].value_counts().index)
    df[var].value_counts().plot.pie(ax = widget.canvas.axis1, autopct = "%1.1f%%", \
        colors = sns.color_palette("prism",7), \
        startangle = 60, labels=label_list, \
        wedgeprops={"linewidth":2,"edgecolor":"k"}, \
        shadow = True, textprops={'fontsize': 10})
    widget.canvas.axis1.set_title("Case distribution "+ " of " + var +" variable", fontsize=14)
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()

def color_month(self,month):
    if month == 1:
        return 'January','blue'
    elif month == 2:
        return 'February','green'
    elif month == 3:
        return 'March','orange'
    elif month == 4:
        return 'April','yellow'
    elif month == 5:
        return 'May','red'
    elif month == 6:
        return 'June','violet'
    elif month == 7:
        return 'July','purple'
    elif month == 8:
        return 'August','black'
    elif month == 9:

```

```

        return 'September','brown'
    elif month == 10:
        return 'October','darkblue'
    elif month == 11:
        return 'November','grey'
    else:
        return 'December','pink'

def line_plot_month(self,month, data,ax):
    label, color = self.color_month(month)
    mdata = data[data.index.month == month]
    sns.lineplot(data=mdata, ax=ax,
                 label=label,
                 color=color,
                 marker='o',
                 linewidth=3)

def sns_plot_month(self, data, feat,ax):
    for i in range(1,13):
        self.line_plot_month(i,data[feat],ax)

def choose_plot(self):
    strCB = self.cbData.currentText()

    if strCB == 'Case Distribution':
        self.widgetPlot1.canvas.figure.clf()
        self.widgetPlot1.canvas.axis1 = self.widgetPlot1.canvas.figure.add_subplot(121,facecolor = '#efc0fe')
        self.plot_barchart(self.df_dummy, "Year", self.widgetPlot1)

        self.widgetPlot1.canvas.axis1 = self.widgetPlot1.canvas.figure.add_subplot(122,facecolor = '#efc0fe')
        self.plot_piechart(self.df_dummy, "Year", self.widgetPlot1)

        self.widgetPlot2.canvas.figure.clf()
        self.widgetPlot2.canvas.axis1 = self.widgetPlot2.canvas.figure.add_subplot(121,facecolor = '#efc0fe')
        self.plot_barchart(self.df_dummy, "Month", self.widgetPlot2)

        self.widgetPlot2.canvas.axis1 = self.widgetPlot2.canvas.figure.add_subplot(122,facecolor = '#efc0fe')
        self.plot_piechart(self.df_dummy, "Month", self.widgetPlot2)

        self.widgetPlot3.canvas.figure.clf()
        self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(221,facecolor = '#efc0fe')
        self.plot_barchart(self.df_dummy, "Day", self.widgetPlot3)

        self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(222,facecolor = '#efc0fe')
        self.plot_piechart(self.df_dummy, "Day", self.widgetPlot3)

        self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(223,facecolor = '#efc0fe')
        self.plot_barchart(self.df_dummy, "Quarter", self.widgetPlot3)

        self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(224,facecolor = '#efc0fe')
        self.plot_piechart(self.df_dummy, "Quarter", self.widgetPlot3)

    if strCB == 'High':
        self.plot_distribution(strCB, "Adj Close", "Volume")

    if strCB == 'Low':
        self.plot_distribution(strCB, "Adj Close", "Volume")

    if strCB == 'Open':
        self.plot_distribution(strCB, "Adj Close", "Volume")

    if strCB == 'Close':
        self.plot_distribution(strCB, "Open", "Volume")

    if strCB == 'Adj Close':
        self.plot_distribution(strCB, "Open", "Volume")

    if strCB == 'Volume':
        self.widgetPlot1.canvas.figure.clf()
        self.widgetPlot1.canvas.axis1 = self.widgetPlot1.canvas.figure.add_subplot(121,facecolor = '#efc0fe')
        self.plot_group_barchart(self.df_dummy.groupby('Year')['Volume'].sum(), "Volume", "The distribution of Volume t
self.widgetPlot1

        self.widgetPlot1.canvas.axis1 = self.widgetPlot1.canvas.figure.add_subplot(122,facecolor = '#efc0fe')

```

```

        self.plot_group_piechart(self.df_dummy.groupby('Year')['Volume'].sum(), "The distribution of Volume by Year",
self.widgetPlot1)

        self.widgetPlot2.canvas.figure.clf()
        self.widgetPlot2.canvas.axis1 = self.widgetPlot2.canvas.figure.add_subplot(121,facecolor = '#efc0fe')
        self.plot_group_barchart(self.df_dummy.groupby('Quarter')['Volume'].sum(), "Volume", "The distribution of Volum
self.widgetPlot2

        self.widgetPlot2.canvas.axis1 = self.widgetPlot2.canvas.figure.add_subplot(122,facecolor = '#efc0fe')
        self.plot_group_piechart(self.df_dummy.groupby('Quarter')['Volume'].sum(), "The distribution of Volume by Quart
self.widgetPlot2

        self.widgetPlot3.canvas.figure.clf()
        self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(221,facecolor = '#efc0fe')
        self.plot_group_barchart(self.df_dummy.groupby('Day')['Volume'].sum(), "Volume", "The distribution of Volume by
self.widgetPlot3

        self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(222,facecolor = '#efc0fe')
        self.plot_group_piechart(self.df_dummy.groupby('Day')['Volume'].sum(), "The distribution of Volume by Days of W
self.widgetPlot3

        self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(223,facecolor = '#efc0fe')
        self.plot_group_barchart(self.df_dummy.groupby('Month')['Volume'].sum(), "Volume", "The distribution of Volume
self.widgetPlot3

        self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(224,facecolor = '#efc0fe')
        self.plot_group_piechart(self.df_dummy.groupby('Month')['Volume'].sum(), "The distribution of Volume by Month",
self.widgetPlot3)

    if strCB == 'Year-Wise':
        self.widgetPlot3.canvas.figure.clf()
        self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(221,facecolor = '#efc0fe')
        self.widgetPlot3.canvas.axis2 = self.widgetPlot3.canvas.figure.add_subplot(222,facecolor = '#efc0fe')
        self.widgetPlot3.canvas.axis3 = self.widgetPlot3.canvas.figure.add_subplot(223,facecolor = '#efc0fe')
        self.widgetPlot3.canvas.axis4 = self.widgetPlot3.canvas.figure.add_subplot(224,facecolor = '#efc0fe')
        norm_data = (self.year_data_mean - self.year_data_mean.min()) / (self.year_data_mean.max() - self.year_data_me
        self.plot_norm_wise_data(norm_data, self.widgetPlot3.canvas.axis1, \
            self.widgetPlot3.canvas.axis2, self.widgetPlot3.canvas.axis3, self.widgetPlot3.canvas.axis4, "normalized ye
        self.widgetPlot3.canvas.figure.tight_layout()
        self.widgetPlot3.canvas.draw()

    if strCB == 'Month-Wise':
        self.widgetPlot3.canvas.figure.clf()
        self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(221,facecolor = '#efc0fe')
        self.widgetPlot3.canvas.axis2 = self.widgetPlot3.canvas.figure.add_subplot(222,facecolor = '#efc0fe')
        self.widgetPlot3.canvas.axis3 = self.widgetPlot3.canvas.figure.add_subplot(223,facecolor = '#efc0fe')
        self.widgetPlot3.canvas.axis4 = self.widgetPlot3.canvas.figure.add_subplot(224,facecolor = '#efc0fe')
        norm_data = (self.monthly_data_mean - self.monthly_data_mean.min()) / (self.monthly_data_mean.max() - self.m
        self.plot_norm_wise_data(norm_data, self.widgetPlot3.canvas.axis1, \
            self.widgetPlot3.canvas.axis2, self.widgetPlot3.canvas.axis3, self.widgetPlot3.canvas.axis4, "normalized mor
        self.widgetPlot3.canvas.figure.tight_layout()
        self.widgetPlot3.canvas.draw()

    if strCB == 'Technical Indicators':
        self.widgetPlot3.canvas.figure.clf()
        self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(311,facecolor = '#efc0fe')
        self.widgetPlot3.canvas.axis2 = self.widgetPlot3.canvas.figure.add_subplot(312,facecolor = '#efc0fe')
        self.widgetPlot3.canvas.axis3 = self.widgetPlot3.canvas.figure.add_subplot(313,facecolor = '#efc0fe')

        self.widgetPlot1.canvas.figure.clf()
        self.widgetPlot1.canvas.axis4 = self.widgetPlot1.canvas.figure.add_subplot(111,facecolor = '#efc0fe')

        self.widgetPlot2.canvas.figure.clf()
        self.widgetPlot2.canvas.axis5 = self.widgetPlot2.canvas.figure.add_subplot(111,facecolor = '#efc0fe')

        self.plot_technical_indicators(self.df,self.widgetPlot3.canvas.axis1,self.widgetPlot3.canvas.axis2,\n            self.wi
        self.widgetPlot3.canvas.axis3,self.widgetPlot1.canvas.axis4,self.widgetPlot2.canvas.axis5)
        self.widgetPlot3.canvas.figure.tight_layout()
        self.widgetPlot2.canvas.figure.tight_layout()
        self.widgetPlot1.canvas.figure.tight_layout()
        self.widgetPlot3.canvas.draw()
        self.widgetPlot2.canvas.draw()
        self.widgetPlot1.canvas.draw()

def plot_distribution(self,strCB, feat1, feat2):

```

```

    self.widgetPlot1.canvas.figure.clf()
    self.widgetPlot1.canvas.axis1 = self.widgetPlot1.canvas.figure.add_subplot(111, facecolor = '#efc0fe')
    sns.lineplot(data=self.df_dummy[strCB], color='red', linewidth=3, ax=self.widgetPlot1.canvas.axis1)
    self.widgetPlot1.canvas.axis1.set_title(strCB + " feature all year", fontsize=20)
    self.widgetPlot1.canvas.figure.tight_layout()
    self.widgetPlot1.canvas.draw()

    self.widgetPlot2.canvas.figure.clf()
    self.widgetPlot2.canvas.axis1 = self.widgetPlot2.canvas.figure.add_subplot(111, facecolor = '#efc0fe')
    sns.scatterplot(data=self.df_dummy, x=strCB, y=feat1, hue="Year", palette="deep", ax=self.widgetPlot2.canvas.axis1)
    self.widgetPlot2.canvas.axis1.set_title("Scatter distribution of " + strCB + " vs " + feat1 + " vs Year", fontsize=20)
    self.widgetPlot2.canvas.figure.tight_layout()
    self.widgetPlot2.canvas.draw()

    self.widgetPlot3.canvas.figure.clf()
    self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(221, facecolor = '#efc0fe')
    sns.scatterplot(data=self.df_dummy, x=strCB, y=feat2, hue="Day", palette="deep", ax=self.widgetPlot3.canvas.axis1)
    self.widgetPlot3.canvas.axis1.set_title("Scatter distribution of " + strCB + " vs " + feat2 + " vs Day", fontsize=20)
    self.widgetPlot3.canvas.figure.tight_layout()
    self.widgetPlot3.canvas.draw()

    self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(222, facecolor = '#efc0fe')
    self.year_data_mean[strCB].plot(linewidth=5, ax=self.widgetPlot3.canvas.axis1)
    self.year_data_ewm[strCB].plot(linewidth=5, ax=self.widgetPlot3.canvas.axis1)
    self.widgetPlot3.canvas.axis1.set_title("Year-Wise Data: Mean and EWM", fontsize=14)
    self.widgetPlot3.canvas.axis1.set_ylabel(strCB, fontsize=12)
    self.widgetPlot3.canvas.axis1.legend(["Mean", "EWM"], fontsize=20)
    self.widgetPlot3.canvas.figure.tight_layout()
    self.widgetPlot3.canvas.draw()

    self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(223, facecolor = '#efc0fe')
    self.monthly_data_mean[strCB].plot(linewidth=5, ax=self.widgetPlot3.canvas.axis1)
    self.monthly_data_ewm[strCB].plot(linewidth=5, ax=self.widgetPlot3.canvas.axis1)
    self.widgetPlot3.canvas.axis1.set_title(strCB + ": Month-Wise Data: Mean and EWM", fontsize=14)
    self.widgetPlot3.canvas.axis1.set_ylabel(strCB, fontsize=12)
    self.widgetPlot3.canvas.axis1.legend(["Mean", "EWM"], fontsize=20)
    self.widgetPlot3.canvas.figure.tight_layout()
    self.widgetPlot3.canvas.draw()

    self.widgetPlot3.canvas.axis1 = self.widgetPlot3.canvas.figure.add_subplot(224, facecolor = '#efc0fe')
    self.sns_plot_month(self.monthly_data_mean, strCB, ax=self.widgetPlot3.canvas.axis1)
    self.widgetPlot3.canvas.axis1.set_title(strCB + ": Month-Wise Data for Every Month", fontsize=14)
    self.widgetPlot3.canvas.axis1.set_ylabel(strCB, fontsize=12)
    self.widgetPlot3.canvas.figure.tight_layout()
    self.widgetPlot3.canvas.draw()

#Defines function to plot grouped distribution of a categorical feature bar plot
def plot_group_barchart(self, df, var, title, widget):
    ax = df.plot(kind="barh", ax=widget.canvas.axis1)
    for i,j in enumerate(df.values):
        ax.text(.7,i,j, weight = "bold", fontsize=10)

    widget.canvas.axis1.set_title(title, fontsize=14)
    widget.canvas.axis1.set_xlabel(var)
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()

#Defines function to plot case grouped distribution of a categorical feature in pie chart
def plot_group_piechart(self, df, title, widget):
    label_list = list(df.index)
    df.plot.pie(ax=widget.canvas.axis1, autopct = "%1.1f%%", \
                colors = sns.color_palette("prism",7), \
                startangle = 60, labels=label_list, \
                wedgeprops={"linewidth":2, "edgecolor":"k"}, \
                shadow =True, textprops={'fontsize': 10})
    widget.canvas.axis1.set_title(title, fontsize=14)
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()

#Plots boxplot, violinplot, stripplot, and heatmap of normalized year-wise data
def plot_norm_wise_data(self, norm_data, ax1, ax2, ax3, ax4, title):
    g=sns.boxplot(data=norm_data,ax = ax1)
    g.xaxis.get_label().set_fontsize(10)
    g.set_title("The box plot of " + title, fontsize=15)

    g=sns.violinplot(data=norm_data,ax = ax2)
    g.xaxis.get_label().set_fontsize(10)

```

```

g.set_title("The violin plot of " + title, fontsize=15)
g=sns.stripplot(data=norm_data, jitter=True, s=18, alpha=0.3, ax = ax3)
g.xaxis.get_label().set_fontsize(10)
g.set_title("The strip plot of " + title, fontsize=15)

g=sns.lineplot(data=norm_data, marker='s', ax = ax4,linewidth=5)
g.xaxis.get_label().set_fontsize(30)
g.set_title("The line plot of " + title, fontsize=15)
g.set_xlabel("YEAR")

#Plots MACD, SMA, RSI, upper and lower bands, standard deviation, and daily returns of Adj Close column
def plot_technical_indicators(self,df,ax1,ax2,ax3,ax4,ax5):
    stock_close=df['Adj Close']
    SMA_CLOSE=df['SMA']
    stock_close[:365].plot(title='GLD Moving Average',label='GLD', linewidth=3, ax=ax1)
    SMA_CLOSE[:365].plot(label='SMA', linewidth=3, ax=ax1)

    upper_band=df['Upper_band']
    lower_band=df['Lower_band']
    upper_band[:365].plot(title="Upper-Lower Band", label='upper band', linewidth=3, ax=ax1)
    lower_band[:365].plot(label='lower band', linewidth=3, ax=ax1)

    DIF=df['DIF']
    MACD=df['MACD']
    DIF[:365].plot(title='DIF and MACD',label='DIF', linewidth=3, ax=ax2)
    MACD[:365].plot(label='MACD', linewidth=3, ax=ax2)

    RSI=df['RSI']
    RSI[:365].plot(title='RSI',label='RSI', linewidth=3, ax=ax3)

    STDEV=df['STDEV']
    STDEV[:365].plot(title='STDEV',label='STDEV', linewidth=3, ax=ax4)

    Daily_Return=df['daily_returns']
    Daily_Return[:365].plot(title="Daily Returns", label='Daily Returns', linewidth=3, ax=ax5)

    ax1.set_ylabel('Price')
    ax2.set_ylabel('Price')
    ax3.set_ylabel('Price')
    ax4.set_ylabel('Price')
    ax5.set_ylabel('Price')

def split_data_regression(self,X):
    #Sets target column
    y_final = pd.DataFrame(X["Adj Close"])
    X = X.drop(["Adj Close"], axis =1)

    #Normalizes data
    scaler = MinMaxScaler()
    X_minmax_data = scaler.fit_transform(X)
    X_final = pd.DataFrame(columns=X.columns, data=X_minmax_data, index=X.index)
    print('Shape of features : ', X_final.shape)
    print('Shape of target : ', y_final.shape)

    #Shifts target array to predict the n + 1 samples
    n=90
    y_final = y_final.shift(-1)
    y_val = y_final[-n:-1]
    y_final = y_final[:-n]

    #Takes last n rows of data to be validation set
    X_val = X_final[-n:-1]
    X_final = X_final[:-n]

    print("\n -----After process----- \n")
    print('Shape of features : ', X_final.shape)
    print('Shape of target : ', y_final.shape)
    print(y_final.tail().to_string())

    y_final=y_final.astype('float64')

    #Splits data into training and test data at 90% and 10% respectively
    self.split_idx=round(0.9*len(X))
    print("split_idx=",self.split_idx)

```

```

X_train = X_final[:self.split_idx]
y_train = y_final[:self.split_idx]
X_test = X_final[self.split_idx:]
y_test = y_final[self.split_idx:]

return X_final, y_final, X_train, y_train, X_test, y_test, X_val, y_val

def perform_regression(self, model, X, y, xtrain, ytrain, xtest, ytest, xval, yval, label,
feat, ax0, ax1, ax2, ax3, ax4, ax5, ax6, ax7, ax8, ax9):
    model.fit(xtrain, ytrain)
    predictions_test = model.predict(xtest)
    predictions_train = model.predict(xtrain)
    predictions_val = model.predict(xval)

    str_label = 'RMSE using ' + label
    print(str_label + f': {np.sqrt(mean_squared_error(ytest, predictions_test))}')
    print("mean square error: ", mean_squared_error(ytest, predictions_test))
    print("variance or r-squared: ", explained_variance_score(ytest, predictions_test))
    print('PREDICTED: Avg. ' + feat + f': {predictions_test.mean()')
    print('PREDICTED: Median ' + feat + f': {np.median(predictions_test)})')

#Evaluation of regression on all dataset
all_pred = model.predict(X)
print("mean square error (whole dataset): ", mean_squared_error(y, all_pred))
print("variance or r-squared (whole dataset): ", explained_variance_score(y, all_pred))

#Visualizes the training set results in a scatter plot
ax0.scatter(x=xtrain, y=predictions_train, color = 'blue')
ax0.set_title('The scatter of actual versus predicted (Training set): '+label, fontweight='bold', fontsize=10)
ax0.set_xlabel('Actual Train Set', fontsize=10)
ax0.set_ylabel('Predicted Train Set', fontsize=10)
ax0.plot([ytrain.min(),ytrain.max()], [ytrain.min(),ytrain.max()], 'r--', linewidth=3)

#Visualizes the test set results in a scatter plot
ax1.scatter(x=ytest, y=predictions_test, color = 'red')
ax1.set_title('The scatter of actual versus predicted (Test set): '+label, fontweight='bold', fontsize=10)
ax1.plot([ytest.min(),ytest.max()], [ytest.min(),ytest.max()], 'b--', linewidth=3)
ax1.set_xlabel('Actual Test Set', fontsize=10)
ax1.set_ylabel('Predicted Test Set', fontsize=10)

#Visualizes the validation set results in a scatter plot
ax2.scatter(x=yval, y=predictions_val, color = 'red')
ax2.set_title('The scatter of actual versus predicted (Validation set): '+label, fontweight='bold', fontsize=10)
ax2.plot([yval.min(),yval.max()], [yval.min(),yval.max()], 'b--', linewidth=3)
ax2.set_xlabel('Actual Validation Set', fontsize=10)
ax2.set_ylabel('Predicted Validation Set', fontsize=10)

#Visualizes the density of error of training and testing
sns.distplot(np.array(ytrain - predictions_train.reshape(len(ytrain),1)), ax=ax3, color = 'red', kde_kws=dict(linewidth=3))
ax3.set_xlabel('Error', fontsize=10)
sns.distplot(np.array(ytest - predictions_test.reshape(len(ytest),1)), ax=ax3, color = 'blue', kde_kws=dict(linewidth=3))
sns.distplot(np.array(yval - predictions_val.reshape(len(yval),1)), ax=ax3, color = 'green', kde_kws=dict(linewidth=3))
ax3.set_title('The density of training, testing, and validation errors: '+label, fontsize=10, fontweight='bold')
ax3.set_xlabel('Error', fontsize=10)
ax3.legend(["Training Error", "Testing Error", "Validation Error"], prop={'size': 8})

#Histogram distribution of regression on train data
sns.histplot(predictions_train, ax=ax4, kde = True, bins=50, color = 'red', line_kws={'lw': 3});
sns.histplot(ytrain, ax=ax4, kde = True, bins=50, color = 'blue', line_kws={'lw': 3});
ax4.set_title("Histogram Distribution of " + label + ' on ' + feat + ' feature on train data', fontsize=10, fontweight=bold)
ax4.set_xlabel(feat, fontsize=10)
ax4.set_ylabel("Count", fontsize=10)
ax4.legend(["Prediction", "Actual"], prop={'size': 8})

for p in ax4.patches:
    ax4.annotate(format(p.get_height(), '.0f'), \
        (p.get_x() + p.get_width() / 2., p.get_height()), \
        ha = 'center', va = 'center', xytext = (0, 10), \
        weight = "bold", fontsize=10, \
        textcoords = 'offset points')

#Histogram distribution of regression on test data
sns.histplot(predictions_test, ax=ax5, kde = True, bins=50, color = 'red', line_kws={'lw': 3});
sns.histplot(ytest, ax=ax5, kde = True, bins=50, color = 'blue', line_kws={'lw': 3});
ax5.set_title("Histogram Distribution of " + label + ' on ' + feat + ' feature on test data', fontsize=10, fontweight=bold)
ax5.set_xlabel(feat, fontsize=10)
ax5.set_ylabel("Count", fontsize=10)

```

```

ax5.legend(["Prediction", "Actual"])

for p in ax5.patches:
    ax5.annotate(format(p.get_height(), '.0f'), \
        (p.get_x() + p.get_width() / 2., p.get_height()), \
        ha = 'center', va = 'center', xytext = (0, 10), \
        weight = "bold", fontsize=10, \
        textcoords = 'offset points')

ax6.plot(X.index[:self.split_idx], ytrain, color = "blue", linewidth = 3, linestyle = "-", label='Actual')
ax6.plot(X.index[:self.split_idx], predictions_train, color = "red", linewidth = 3, linestyle = "--", label='Predicted')
ax6.set_title('Actual and Predicted Training Set: '+ label, fontsize = 10)
ax6.set_xlabel('Date', fontsize = 10)
ax6.set_ylabel(feat, fontsize = 10)
ax6.legend(prop={'size': 10})

ax7.plot(X.index[self.split_idx:], ytest, color = "blue", linewidth = 3, linestyle = "-", label='Actual')
ax7.plot(X.index[self.split_idx:], predictions_test, color = "red", linewidth = 3, linestyle = "--", label='Predicted')
ax7.set_title('Actual and Predicted Test Set: '+ label, fontsize = 10)
ax7.set_xlabel('Date', fontsize = 10)
ax7.set_ylabel(feat, fontsize = 10)
ax7.legend(prop={'size': 10})

ax8.plot(yval.index, yval, color = "blue", linewidth = 3, linestyle = "-", label='Actual')
ax8.plot(yval.index, predictions_val, color = "red", linewidth = 3, linestyle = "--", label='Predicted')
ax8.set_title('Actual and Predicted Validation Set (90 days forecasting): '+ label, fontsize = 10)
ax8.set_xlabel('Date', fontsize = 10)
ax8.set_ylabel(feat, fontsize = 10)
ax8.legend(prop={'size': 10})

ax9.plot(X.index, y, color = "blue", linewidth = 3, linestyle = "-", label='Actual')
ax9.plot(X.index, all_pred, color = "red", linewidth = 3, linestyle = "--", label='Predicted')
ax9.set_title('Actual and Predicted Whole Dataset: '+ label, fontsize = 10)
ax9.set_xlabel('Date', fontsize = 10)
ax9.set_ylabel(feat, fontsize = 10)
ax9.legend(prop={'size': 10})

def clear_create_canvas(self):
    self.widgetPlot1.canvas.figure.clf()
    self.widgetPlot1.canvas.axis1 = self.widgetPlot1.canvas.figure.add_subplot(111, facecolor = '#efc0fe')

    self.widgetPlot2.canvas.figure.clf()
    self.widgetPlot2.canvas.axis2 = self.widgetPlot2.canvas.figure.add_subplot(111, facecolor = '#efc0fe')

    self.widgetPlot3.canvas.figure.clf()
    self.widgetPlot3.canvas.axis3 = self.widgetPlot3.canvas.figure.add_subplot(421, facecolor = '#efc0fe')
    self.widgetPlot3.canvas.axis4 = self.widgetPlot3.canvas.figure.add_subplot(422, facecolor = '#efc0fe')
    self.widgetPlot3.canvas.axis5 = self.widgetPlot3.canvas.figure.add_subplot(423, facecolor = '#efc0fe')
    self.widgetPlot3.canvas.axis6 = self.widgetPlot3.canvas.figure.add_subplot(424, facecolor = '#efc0fe')
    self.widgetPlot3.canvas.axis7 = self.widgetPlot3.canvas.figure.add_subplot(425, facecolor = '#efc0fe')
    self.widgetPlot3.canvas.axis8 = self.widgetPlot3.canvas.figure.add_subplot(426, facecolor = '#efc0fe')
    self.widgetPlot3.canvas.axis9 = self.widgetPlot3.canvas.figure.add_subplot(427, facecolor = '#efc0fe')
    self.widgetPlot3.canvas.axis10 = self.widgetPlot3.canvas.figure.add_subplot(428, facecolor = '#efc0fe')

def redraw_canvas(self):
    self.widgetPlot3.canvas.figure.tight_layout()
    self.widgetPlot2.canvas.figure.tight_layout()
    self.widgetPlot1.canvas.figure.tight_layout()
    self.widgetPlot3.canvas.draw()
    self.widgetPlot2.canvas.draw()
    self.widgetPlot1.canvas.draw()

def choose_forecasting(self):
    strCB = self.cbForecasting.currentText()

    if strCB == "Linear Regression":
        #Clears and creates canvas
        self.clear_create_canvas()

        lin_reg = LinearRegression()
        self.perform_regression(lin_reg, self.X_final, self.y_final, self.X_train, self.y_train, \
            self.X_test, self.y_test, self.X_val, self.y_val, "Linear Regression", "Adj Close", \
            self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \
            self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \
            self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \
            self.widgetPlot3.canvas.axis10)

```

```

#Redraws canvas
self.redraw_canvas()

if strCB == "Random Forest Regression":
    #Clears and creates canvas
    self.clear_create_canvas()

    rf_reg = RandomForestRegressor()
    self.perform_regression(rf_reg, self.X_final, self.y_final, self.X_train, self.y_train, \
        self.X_test, self.y_test, self.X_val, self.y_val, "RF Regression", "Adj Close", \
        self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \
        self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \
        self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \
        self.widgetPlot3.canvas.axis10)

    #Redraws canvas
    self.redraw_canvas()

if strCB == "Decision Tree Regression":
    #Clears and creates canvas
    self.clear_create_canvas()

    dt_reg = DecisionTreeRegressor(random_state=100)
    self.perform_regression(dt_reg, self.X_final, self.y_final, self.X_train, self.y_train, \
        self.X_test, self.y_test, self.X_val, self.y_val, "DT Regression", "Adj Close", \
        self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \
        self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \
        self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \
        self.widgetPlot3.canvas.axis10)

    #Redraws canvas
    self.redraw_canvas()

if strCB == "KNN Regression":
    #Clears and creates canvas
    self.clear_create_canvas()

    knn_reg = KNeighborsRegressor(n_neighbors=7)
    self.perform_regression(knn_reg, self.X_final, self.y_final, self.X_train, self.y_train, \
        self.X_test, self.y_test, self.X_val, self.y_val, "KNN Regression", "Adj Close", \
        self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \
        self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \
        self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \
        self.widgetPlot3.canvas.axis10)

    #Redraws canvas
    self.redraw_canvas()

if strCB == "Adaboost Regression":
    #Clears and creates canvas
    self.clear_create_canvas()

    ada_reg = AdaBoostRegressor(random_state=100, n_estimators=200)
    self.perform_regression(ada_reg, self.X_final, self.y_final, self.X_train, self.y_train, \
        self.X_test, self.y_test, self.X_val, self.y_val, "Adaboost Regression", "Adj Close", \
        self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \
        self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \
        self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \
        self.widgetPlot3.canvas.axis10)

    #Redraws canvas
    self.redraw_canvas()

if strCB == "Gradient Boosting Regression":
    #Clears and creates canvas
    self.clear_create_canvas()

    gb_reg = GradientBoostingRegressor(random_state=100)
    self.perform_regression(gb_reg, self.X_final, self.y_final, self.X_train, self.y_train, \
        self.X_test, self.y_test, self.X_val, self.y_val, "GB Regression", "Adj Close", \
        self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \
        self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \
        self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \
        self.widgetPlot3.canvas.axis10)

    #Redraws canvas

```

```

        self.redraw_canvas()

    if strCB == "XGB Regression":
        #Clears and creates canvas
        self.clear_create_canvas()

        xgb_reg = XGBRegressor(random_state=100)
        self.perform_regression(xgb_reg, self.X_final, self.y_final, self.X_train, self.y_train, \
            self.X_test, self.y_test, self.X_val, self.y_val, "XGB Regression", "Adj Close", \
            self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \
            self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \
            self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \
            self.widgetPlot3.canvas.axis10)

        #Redraws canvas
        self.redraw_canvas()

    if strCB == "LGBM Regression":
        #Clears and creates canvas
        self.clear_create_canvas()

        lgbm_reg = LGBMRegressor(random_state=100)
        self.perform_regression(lgbm_reg, self.X_final, self.y_final, self.X_train, self.y_train, \
            self.X_test, self.y_test, self.X_val, self.y_val, "LGBM Regression", "Adj Close", \
            self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \
            self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \
            self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \
            self.widgetPlot3.canvas.axis10)

        #Redraws canvas
        self.redraw_canvas()

    if strCB == "Catboost Regression":
        #Clears and creates canvas
        self.clear_create_canvas()

        cb_reg = CatBoostRegressor(random_state=100)
        self.perform_regression(cb_reg, self.X_final, self.y_final, self.X_train, self.y_train, \
            self.X_test, self.y_test, self.X_val, self.y_val, "Catboost Regression", "Adj Close", \
            self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \
            self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \
            self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \
            self.widgetPlot3.canvas.axis10)

        #Redraws canvas
        self.redraw_canvas()

    if strCB == "SVR Regression":
        #Clears and creates canvas
        self.clear_create_canvas()

        svm_reg = SVR()
        self.perform_regression(svm_reg, self.X_final, self.y_final, self.X_train, self.y_train, \
            self.X_test, self.y_test, self.X_val, self.y_val, "SVR Regression", "Adj Close", \
            self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \
            self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \
            self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \
            self.widgetPlot3.canvas.axis10)

        #Redraws canvas
        self.redraw_canvas()

    if strCB == "MLP Regression":
        #Clears and creates canvas
        self.clear_create_canvas()

        mlp_reg = MLPRegressor(random_state=100, max_iter=1000)
        self.perform_regression(mlp_reg, self.X_final, self.y_final, self.X_train, self.y_train, \
            self.X_test, self.y_test, self.X_val, self.y_val, "MLP Regression", "Adj Close", \
            self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \
            self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \
            self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \
            self.widgetPlot3.canvas.axis10)

        #Redraws canvas
        self.redraw_canvas()

```

```
if strCB == "Lasso Regression":  
    #Clears and creates canvas  
    self.clear_create_canvas()  
  
    lasso_reg = LassoCV(n_alphas=1000, max_iter=3000, random_state=0)  
    self.perform_regression(lasso_reg, self.X_final, self.y_final, self.X_train, self.y_train, \  
        self.X_test, self.y_test, self.X_val, self.y_val, "Lasso Regression", "Adj Close", \  
        self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2, self.widgetPlot3.canvas.axis3, \  
        self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5, self.widgetPlot3.canvas.axis6, \  
        self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8, self.widgetPlot3.canvas.axis9, \  
        self.widgetPlot3.canvas.axis10)
```

```

#Redraws canvas
self.redraw_canvas()

if strCB == "Ridge Regression":
    #Clears and creates canvas
    self.clear_create_canvas()

    ridge_reg = RidgeCV(gcv_mode='auto')
    self.perform_regression(ridge_reg, self.X_final, self.y_final, self.X_train,
self.y_train, \
                           self.X_test, self.y_test, self.X_val, self.y_val, "Ridge Regression", "Adj
Close",\
                           self.widgetPlot1.canvas.axis1, self.widgetPlot2.canvas.axis2,
self.widgetPlot3.canvas.axis3,\ \
                           self.widgetPlot3.canvas.axis4, self.widgetPlot3.canvas.axis5,
self.widgetPlot3.canvas.axis6,\ \
                           self.widgetPlot3.canvas.axis7, self.widgetPlot3.canvas.axis8,
self.widgetPlot3.canvas.axis9,\ \
                           self.widgetPlot3.canvas.axis10)

    #Redraws canvas
    self.redraw_canvas()

def extract_data(self, X):
    #Extracts output and input variables
    y=X["daily_returns"]
    y = np.array([1 if i>0 else 0 for i in y])

    #Drops irrelevant column
    X = X.drop(["daily_returns", "Day", "Week", "Month", "Year", "Quarter"], axis =1)

    return X, y

def split_data(self):
    #Extracts input and output variables
    X,y=self.extract_data(self.df)

    #Resamples data using SMOTE
    sm = SMOTE(random_state=42)
    X,y = sm.fit_resample(X, y.ravel())

    #Splits the data into training and testing
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
random_state = 2021, shuffle=True, stratify=y)

    #Normalizes data with robust scaler
    rob_scaler = RobustScaler()
    X_train_rob = X_train.copy()
    X_test_rob = X_test.copy()
    self.y_train_rob = y_train.copy()
    self.y_test_rob = y_test.copy()
    self.X_train_rob = rob_scaler.fit_transform(X_train_rob)
    self.X_test_rob = rob_scaler.transform(X_test_rob)

    #Saves into pkl files
    joblib.dump(self.X_train_rob, 'X_train_rob.pkl')
    joblib.dump(self.X_test_rob, 'X_test_rob.pkl')
    joblib.dump(self.y_train_rob, 'y_train_rob.pkl')

```

```

joblib.dump(self.y_test_rob, 'y_test_rob.pkl')

#Normalizes data with MinMax Scaler
X_train_norm = X_train.copy()
X_test_norm = X_test.copy()
self.y_train_norm = y_train.copy()
self.y_test_norm = y_test.copy()

norm = MinMaxScaler()
self.X_train_norm = norm.fit_transform(X_train_norm)
self.X_test_norm = norm.transform(X_test_norm)

#Saves into pkl files
joblib.dump(self.X_train_norm, 'X_train_norm.pkl')
joblib.dump(self.X_test_norm, 'X_test_norm.pkl')
joblib.dump(self.y_train_norm, 'y_train_norm.pkl')
joblib.dump(self.y_test_norm, 'y_test_norm.pkl')

#Normalizes data with Standard Scaler
X_train_stand = X_train.copy()
X_test_stand = X_test.copy()
self.y_train_stand = y_train.copy()
self.y_test_stand = y_test.copy()
scaler = StandardScaler()
self.X_train_stand = scaler.fit_transform(X_train_stand)
self.X_test_stand = scaler.transform(X_test_stand)

#Saves into pkl files
joblib.dump(self.X_train_stand, 'X_train_stand.pkl')
joblib.dump(self.X_test_stand, 'X_test_stand.pkl')
joblib.dump(self.y_train_stand, 'y_train_stand.pkl')
joblib.dump(self.y_test_stand, 'y_test_stand.pkl')

def load_rob_files(self):
    X_train_rob = joblib.load('X_train_rob.pkl')
    X_test_rob = joblib.load('X_test_rob.pkl')
    y_train_rob = joblib.load('y_train_rob.pkl')
    y_test_rob = joblib.load('y_test_rob.pkl')

    return X_train_rob, X_test_rob, y_train_rob, y_test_rob

def load_norm_files(self):
    X_train_norm = joblib.load('X_train_norm.pkl')
    X_test_norm = joblib.load('X_test_norm.pkl')
    y_train_norm = joblib.load('y_train_norm.pkl')
    y_test_norm = joblib.load('y_test_norm.pkl')

    return X_train_norm, X_test_norm, y_train_norm, y_test_norm

def load_stand_files(self):
    X_train_stand = joblib.load('X_train_stand.pkl')
    X_test_stand = joblib.load('X_test_stand.pkl')
    y_train_stand = joblib.load('y_train_stand.pkl')
    y_test_stand = joblib.load('y_test_stand.pkl')

    return X_train_stand, X_test_stand, y_train_stand, y_test_stand

def split_data_ML(self):
    #Loads or creates raw, normalized, and standardized train and test sets
    if path.isfile('X_train_rob.pkl'):

```

```

        self.X_train_rob, self.X_test_rob, self.y_train_rob, self.y_test_rob =
self.load_rob_files()
        self.X_train_norm, self.X_test_norm, self.y_train_norm, self.y_test_norm =
self.load_norm_files()
        self.X_train_stand, self.X_test_stand, self.y_train_stand, self.y_test_stand =
self.load_stand_files()

    else:
        self.split_data()

#Prints each shape
print('X train ROB: ', self.X_train_rob.shape)
print('X test ROB: ', self.X_test_rob.shape)
print('Y train ROB: ', self.y_train_rob.shape)
print('Y test ROB: ', self.y_test_rob.shape)

#Prints each shape
print('X train NORM: ', self.X_train_norm.shape)
print('X test NORM: ', self.X_test_norm.shape)
print('Y train NORM: ', self.y_train_norm.shape)
print('Y test NORM: ', self.y_test_norm.shape)

#Prints each shape
print('X train STAND: ', self.X_train_stand.shape)
print('Y train STAND: ', self.y_train_stand.shape)
print('X test STAND: ', self.X_test_stand.shape)
print('Y test STAND: ', self.y_test_stand.shape)

def train_model_ML(self):
    self.split_data_ML()

    #Turns on two widgets
    self.cbData.setEnabled(True)
    self.cbClassifier.setEnabled(True)

    #Turns off pbTrainML
    self.pbTrainML.setEnabled(False)

    #Turns on three radio buttons
    self.rbRobust.setEnabled(True)
    self.rbRobust.setChecked(True)
    self.rbMinMax.setEnabled(True)
    self.rbStandard.setEnabled(True)

def plot_real_pred_val(self, Y_pred, Y_test, widget, title):
    #Calculate Metrics
    acc=accuracy_score(Y_test,Y_pred)

    #Output plot
    widget.canvas.figure.clf()
    widget.canvas.axis1 = widget.canvas.figure.add_subplot(111,facecolor='steelblue')

    widget.canvas.axis1.scatter(range(len(Y_pred)),Y_pred,color="yellow",lw=5,label="Predictions")
    widget.canvas.axis1.scatter(range(len(Y_test)), Y_test,color="red",label="Actual")
    widget.canvas.axis1.set_title("Prediction Values vs Real Values of " + title,
fontweight ="bold",fontsize=15)
    widget.canvas.axis1.set_xlabel("Accuracy: " + str(round((acc*100),3)) + "%",fontweight =
"bold",fontsize=15)
    widget.canvas.axis1.legend()
    widget.canvas.axis1.grid(True, alpha=0.75, lw=1, ls='-.')

```

```

        widget.canvas.axis1.yaxis.set_ticklabels(["", "Negative Daily Returns", "", "", "", "", "", "Positive Daily Returns"]);
        widget.canvas.figure.tight_layout()
        widget.canvas.draw()

    def plot_cm(self, Y_pred, Y_test, widget, title):
        cm=confusion_matrix(Y_test,Y_pred)
        widget.canvas.figure.clf()
        widget.canvas.axis1 = widget.canvas.figure.add_subplot(111)
        class_label =["Negative Daily Returns", "Positive Daily Returns"]
        df_cm = pd.DataFrame(cm, index=class_label,columns=class_label)
        sns.heatmap(df_cm, ax=widget.canvas.axis1, annot=True,
cmap='plasma',linewidhts=2,fmt='d')
        widget.canvas.axis1.set_title("Confusion Matrix of " + title, fontweight
="bold",fontsize=15)
        widget.canvas.axis1.set_xlabel("Predicted")
        widget.canvas.axis1.set_ylabel("True")
        widget.canvas.axis1.xaxis.set_ticklabels(["Negative Daily Returns", "Positive Daily
Returns"]);
        widget.canvas.axis1.yaxis.set_ticklabels(["Negative Daily Returns", "Positive Daily
Returns"]);
        widget.canvas.draw()

    def plot_learning_curve(self,estimator, title, X, y, widget, ylim=None, cv=None,
                           n_jobs=None, train_sizes=np.linspace(.1, 1.0, 5)):
        widget.canvas.axis1.set_title(title, fontweight ="bold",fontsize=15)
        if ylim is not None:
            widget.canvas.axis1.set_ylim(*ylim)
        widget.canvas.axis1.set_xlabel("Training examples")
        widget.canvas.axis1.set_ylabel("Score")

        train_sizes, train_scores, test_scores, fit_times, _ = \
            learning_curve(estimator, X, y, cv=cv, n_jobs=n_jobs,
                           train_sizes=train_sizes,
                           return_times=True)
        train_scores_mean = np.mean(train_scores, axis=1)
        train_scores_std = np.std(train_scores, axis=1)
        test_scores_mean = np.mean(test_scores, axis=1)
        test_scores_std = np.std(test_scores, axis=1)

        # Plot learning curve
        widget.canvas.axis1.grid()
        widget.canvas.axis1.fill_between(train_sizes, train_scores_mean - train_scores_std,
                                         train_scores_mean + train_scores_std, alpha=0.1,
                                         color="r")
        widget.canvas.axis1.fill_between(train_sizes, test_scores_mean - test_scores_std,
                                         test_scores_mean + test_scores_std, alpha=0.1,
                                         color="g")
        widget.canvas.axis1.plot(train_sizes, train_scores_mean, 'o-', color="r",
                               label="Training score")
        widget.canvas.axis1.plot(train_sizes, test_scores_mean, 'o-', color="g",
                               label="Cross-validation score")
        widget.canvas.axis1.legend(loc="best")

    def plot_scalability_curve(self,estimator, title, X, y, widget, ylim=None, cv=None,
                               n_jobs=None, train_sizes=np.linspace(.1, 1.0, 5)):
        widget.canvas.axis1.set_title(title, fontweight ="bold",fontsize=15)
        if ylim is not None:
            widget.canvas.axis1.set_ylim(*ylim)
        widget.canvas.axis1.set_xlabel("Training examples")
        widget.canvas.axis1.set_ylabel("Score")

```

```

train_sizes, train_scores, test_scores, fit_times, _ = \
    learning_curve(estimator, X, y, cv=cv, n_jobs=n_jobs,
                   train_sizes=train_sizes,
                   return_times=True)
fit_times_mean = np.mean(fit_times, axis=1)
fit_times_std = np.std(fit_times, axis=1)

# Plot n_samples vs fit_times
widget.canvas.axis1.grid()
widget.canvas.axis1.plot(train_sizes, fit_times_mean, 'o-')
widget.canvas.axis1.fill_between(train_sizes, fit_times_mean - fit_times_std,
                                fit_times_mean + fit_times_std, alpha=0.1)
widget.canvas.axis1.set_xlabel("Training examples")
widget.canvas.axis1.set_ylabel("fit_times")

def plot_performance_curve(self, estimator, title, X, y, widget, ylim=None, cv=None,
                           n_jobs=None, train_sizes=np.linspace(.1, 1.0, 5)):
    widget.canvas.axis1.set_title(title, fontweight ="bold", fontsize=15)
    if ylim is not None:
        widget.canvas.axis1.set_ylim(*ylim)
    widget.canvas.axis1.set_xlabel("Training examples")
    widget.canvas.axis1.set_ylabel("Score")

    train_sizes, train_scores, test_scores, fit_times, _ = \
        learning_curve(estimator, X, y, cv=cv, n_jobs=n_jobs,
                       train_sizes=train_sizes,
                       return_times=True)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    fit_times_mean = np.mean(fit_times, axis=1)

    # Plot n_samples vs fit_times
    widget.canvas.axis1.grid()
    widget.canvas.axis1.plot(fit_times_mean, test_scores_mean, 'o-')
    widget.canvas.axis1.fill_between(fit_times_mean, test_scores_mean - test_scores_std,
                                    test_scores_mean + test_scores_std, alpha=0.1)
    widget.canvas.axis1.set_xlabel("fit_times")
    widget.canvas.axis1.set_ylabel("Score")

def plot_decision(self, df, cla, feat1, feat2, widget, title=""):
    X, y = self.extract_data(df)

    #Plots decision boundary of two features
    X_feature = np.array(X.iloc[:, 13:14])
    X_train_feature, X_test_feature, y_train_feature, y_test_feature =
    train_test_split(X_feature, y, test_size=0.3, random_state = 42)
    cla.fit(X_train_feature, y_train_feature)

    plot_decision_regions(X_test_feature, y_test_feature.ravel(), clf=cla, legend=2,
    ax=widget.canvas.axis1)
    widget.canvas.axis1.set_title(title, fontweight ="bold", fontsize=15)
    widget.canvas.axis1.set_xlabel(feat1)
    widget.canvas.axis1.set_ylabel(feat2)
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()

def train_model(self, model, X, y):
    model.fit(X, y)
    return model

```

```

def predict_model(self, model, X, proba=False):
    if ~proba:
        y_pred = model.predict(X)
    else:
        y_pred_proba = model.predict_proba(X)
        y_pred = np.argmax(y_pred_proba, axis=1)

    return y_pred

def run_model(self, name, scaling, model, X_train, X_test, y_train, y_test, train=True,
proba=True):
    if train == True:
        model = self.train_model(model, X_train, y_train)
    y_pred = self.predict_model(model, X_test, proba)

    accuracy = accuracy_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred, average='weighted')
    precision = precision_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    print('accuracy: ', accuracy)
    print('recall: ', recall)
    print('precision: ', precision)
    print('f1: ', f1)
    print(classification_report(y_test, y_pred))

    self.widgetPlot1.canvas.figure.clf()
    self.widgetPlot1.canvas.axis1 =
self.widgetPlot1.canvas.figure.add_subplot(111,facecolor = '#efc0fe')
    self.plot_cm(y_pred, y_test, self.widgetPlot1, name + " -- " + scaling)
    self.widgetPlot1.canvas.figure.tight_layout()
    self.widgetPlot1.canvas.draw()

    self.widgetPlot2.canvas.figure.clf()
    self.widgetPlot2.canvas.axis1 =
self.widgetPlot2.canvas.figure.add_subplot(111,facecolor = '#efc0fe')
    self.plot_real_pred_val(y_pred, y_test, self.widgetPlot2, name + " -- " + scaling)
    self.widgetPlot2.canvas.figure.tight_layout()
    self.widgetPlot2.canvas.draw()

    self.widgetPlot3.canvas.figure.clf()
    self.widgetPlot3.canvas.axis1 =
self.widgetPlot3.canvas.figure.add_subplot(221,facecolor = '#efc0fe')
    self.plot_decision(self.df, model, 'Open_Close', 'High_Low', self.widgetPlot3,
title="The decision boundaries of " + name + " -- " + scaling)

    self.widgetPlot3.canvas.axis1 =
self.widgetPlot3.canvas.figure.add_subplot(222,facecolor = '#efc0fe')
    self.plot_learning_curve(model, 'Learning Curve' + " -- " + scaling, X_train, y_train,
self.widgetPlot3)

    self.widgetPlot3.canvas.axis1 =
self.widgetPlot3.canvas.figure.add_subplot(223,facecolor = '#efc0fe')
    self.plot_scalability_curve(model, 'Scalability of ' + name + " -- " + scaling,
X_train, y_train, self.widgetPlot3)

    self.widgetPlot3.canvas.axis1 =
self.widgetPlot3.canvas.figure.add_subplot(224,facecolor = '#efc0fe')
    self.plot_performance_curve(model, 'Performance of ' + name + " -- " + scaling,
X_train, y_train, self.widgetPlot3)

```

```

        self.widgetPlot3.canvas.figure.tight_layout()
        self.widgetPlot3.canvas.draw()

    def build_train_lr(self):
        try:
            if os.path.isfile('logregRob.pkl'):
                # Loads model
                self.logregRob = joblib.load('logregRob.pkl')
                self.logregNorm = joblib.load('logregNorm.pkl')
                self.logregStand = joblib.load('logregStand.pkl')

                if self.rbRobust.isChecked():
                    self.run_model('Logistic Regression', 'Robust', \
                        self.logregRob, self.X_train_rob, \
                        self.X_test_rob, self.y_train_rob, \
                        self.y_test_rob)
                if self.rbMinMax.isChecked():
                    self.run_model('Logistic Regression', 'MinMax', \
                        self.logregNorm, self.X_train_norm, \
                        self.X_test_norm, self.y_train_norm, \
                        self.y_test_norm)

                if self.rbStandard.isChecked():
                    self.run_model('Logistic Regression', 'Standard', \
                        self.logregStand, self.X_train_stand, \
                        self.X_test_stand, self.y_train_stand, \
                        self.y_test_stand)

            else:
                raise FileNotFoundError("Model files not found.")

        except Exception as e:
            print("An error occurred:", e)
            # Builds and trains Logistic Regression
            self.logregRob = LogisticRegression(solver='lbfgs', \
                max_iter=2000, random_state=2021)
            self.logregNorm = LogisticRegression(solver='lbfgs', \
                max_iter=2000, random_state=2021)
            self.logregStand = LogisticRegression(solver='lbfgs', \
                max_iter=2000, random_state=2021)

            if self.rbRobust.isChecked():
                self.run_model('Logistic Regression', 'Robust', \
                    self.logregRob, self.X_train_rob, \
                    self.X_test_rob, self.y_train_rob, \
                    self.y_test_rob)
            if self.rbMinMax.isChecked():
                self.run_model('Logistic Regression', 'MinMax', \
                    self.logregNorm, self.X_train_norm, \
                    self.X_test_norm, self.y_train_norm, self.y_test_norm)

            if self.rbStandard.isChecked():
                self.run_model('Logistic Regression', 'Standard', \
                    self.logregStand, self.X_train_stand, \
                    self.X_test_stand, self.y_train_stand, \
                    self.y_test_stand)

        try:
            # Saves model
            joblib.dump(self.logregRob, 'logregRob.pkl')

```

```

        joblib.dump(self.logregNorm, 'logregNorm.pkl')
        joblib.dump(self.logregStand, 'logregStand.pkl')
    except Exception as e:
        print("An error occurred while saving the model:", e)

def build_train_svm(self):
    try:
        if path.isfile('SVMRob.pkl'):
            # Loads model
            self.SVMRob = joblib.load('SVMRob.pkl')
            self.SVMNorm = joblib.load('SVMNorm.pkl')
            self.SVMStand = joblib.load('SVMStand.pkl')

            if self.rbRobust.isChecked():
                self.run_model('SVM', 'Robust', self.SVMRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)

            if self.rbMinMax.isChecked():
                self.run_model('SVM', 'MinMax', self.SVMNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

            if self.rbStandard.isChecked():
                self.run_model('SVM', 'Standard', self.SVMStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)
            else:
                # Builds and trains SVM model
                self.SVMRob = SVC(random_state=2021, probability=True)
                self.SVMNorm = SVC(random_state=2021, probability=True)
                self.SVMStand = SVC(random_state=2021, probability=True)

            if self.rbRobust.isChecked():
                self.run_model('SVM', 'Robust', self.SVMRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)

            if self.rbMinMax.isChecked():
                self.run_model('SVM', 'MinMax', self.SVMNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

            if self.rbStandard.isChecked():
                self.run_model('SVM', 'Standard', self.SVMStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

        # Saves model
        joblib.dump(self.SVMRob, 'SVMRob.pkl')
        joblib.dump(self.SVMNorm, 'SVMNorm.pkl')
        joblib.dump(self.SVMStand, 'SVMStand.pkl')
    except Exception as e:
        print("An error occurred: ", str(e))

def build_train_knn(self):
    try:
        if path.isfile('KNNRob.pkl'):
            # Loads model
            self.KNNRob = joblib.load('KNNRob.pkl')
            self.KNNNorm = joblib.load('KNNNorm.pkl')
            self.KNNStand = joblib.load('KNNStand.pkl')

            if self.rbRobust.isChecked():

```

```

        self.run_model('KNN', 'Robust', self.KNNRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
        if self.rbMinMax.isChecked():
            self.run_model('KNN', 'MinMax', self.KNNNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

        if self.rbStandard.isChecked():
            self.run_model('KNN', 'Standard', self.KNNStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

    else:
        #Builds and trains K-Nearest Neighbor
        self.KNNRob = KNeighborsClassifier(n_neighbors = 10)
        self.KNNNorm = KNeighborsClassifier(n_neighbors = 10)
        self.KNNStand = KNeighborsClassifier(n_neighbors = 10)

        if self.rbRobust.isChecked():
            self.run_model('KNN', 'Robust', self.KNNRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
            if self.rbMinMax.isChecked():
                self.run_model('KNN', 'MinMax', self.KNNNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

        if self.rbStandard.isChecked():
            self.run_model('KNN', 'Standard', self.KNNStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

        #Saves model
        joblib.dump(self.KNNRob, 'KNNRob.pkl')
        joblib.dump(self.KNNNorm, 'KNNNorm.pkl')
        joblib.dump(self.KNNStand, 'KNNStand.pkl')
except Exception as e:
    print("An error occurred: ", str(e))

def build_train_dt(self):
    try:
        if path.isfile('DTRob.pkl'):
            #Loads model
            self.DTRob = joblib.load('DTRob.pkl')
            self.DTNorm = joblib.load('DTNorm.pkl')
            self.DTStand = joblib.load('DTStand.pkl')

            if self.rbRobust.isChecked():
                self.run_model('DT', 'Robust', self.DTRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
                if self.rbMinMax.isChecked():
                    self.run_model('DT', 'MinMax', self.DTNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

            if self.rbStandard.isChecked():
                self.run_model('DT', 'Standard', self.DTStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

        else:
            #Builds and trains Decision Tree
            dt_cla = DecisionTreeClassifier()
            parameters = {'max_depth':np.arange(1,20,1), 'random_state':[2021]}
            self.DTRob = GridSearchCV(dt_cla, parameters)
            self.DTNorm = GridSearchCV(dt_cla, parameters)
            self.DTStand = GridSearchCV(dt_cla, parameters)

```

```

        if self.rbRobust.isChecked():
            self.run_model('DT', 'Robust', self.DTRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
        if self.rbMinMax.isChecked():
            self.run_model('DT', 'MinMax', self.DTNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

        if self.rbStandard.isChecked():
            self.run_model('DT', 'Standard', self.DTStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

        #Saves model
        joblib.dump(self.DTRob, 'DTRob.pkl')
        joblib.dump(self.DTNorm, 'DTNorm.pkl')
        joblib.dump(self.DTStand, 'DTStand.pkl')
    except Exception as e:
        print("An error occurred: ", str(e))

def build_train_rf(self):
    try:
        if path.isfile('RFRob.pkl'):
            #Loads model
            self.RFRob = joblib.load('RFRob.pkl')
            self.RFNorm = joblib.load('RFNorm.pkl')
            self.RFStand = joblib.load('RFStand.pkl')

            if self.rbRobust.isChecked():
                self.run_model('RF', 'Robust', self.RFRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
            if self.rbMinMax.isChecked():
                self.run_model('RF', 'MinMax', self.RFNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

            if self.rbStandard.isChecked():
                self.run_model('RF', 'Standard', self.RFStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

        else:
            #Builds and trains Random Forest
            self.RFRob = RandomForestClassifier(n_estimators=200, max_depth=20,
random_state=2021)
            self.RFNorm = RandomForestClassifier(n_estimators=200, max_depth=20,
random_state=2021)
            self.RFStand = RandomForestClassifier(n_estimators=200, max_depth=20,
random_state=2021)

            if self.rbRobust.isChecked():
                self.run_model('RF', 'Robust', self.RFRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
            if self.rbMinMax.isChecked():
                self.run_model('RF', 'MinMax', self.RFNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

            if self.rbStandard.isChecked():
                self.run_model('RF', 'Standard', self.RFStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

        #Saves model
        joblib.dump(self.RFRob, 'RFRob.pkl')
    
```

```

        joblib.dump(self.RFNorm, 'RFNorm.pkl')
        joblib.dump(self.RFStand, 'RFStand.pkl')
    except Exception as e:
        print("An error occurred: ", str(e))

    def build_train_gb(self):
        try:
            if path.isfile('GBRob.pkl'):
                #Loads model
                self.GBRob = joblib.load('GBRob.pkl')
                self.GBNorm = joblib.load('GBNorm.pkl')
                self.GBStand = joblib.load('GBStand.pkl')

                if self.rbRobust.isChecked():
                    self.run_model('Gradient Boosting', 'Robust', self.GBRob,
self.X_train_rob, self.X_test_rob, self.y_train_rob, self.y_test_rob)
                if self.rbMinMax.isChecked():
                    self.run_model('Gradient Boosting', 'MinMax', self.GBNorm,
self.X_train_norm, self.X_test_norm, self.y_train_norm, self.y_test_norm)

                if self.rbStandard.isChecked():
                    self.run_model('Gradient Boosting', 'Standard', self.GBStand,
self.X_train_stand, self.X_test_stand, self.y_train_stand, self.y_test_stand)

            else:
                #Builds and trains Gradient Boosting
                self.GBRob = GradientBoostingClassifier(n_estimators = 50, max_depth=10,
subsample=0.8, max_features=0.2, random_state=2021)
                self.GBNorm = GradientBoostingClassifier(n_estimators = 50, max_depth=20,
subsample=0.8, max_features=0.2, random_state=2021)
                self.GBStand = GradientBoostingClassifier(n_estimators = 50, max_depth=20,
subsample=0.8, max_features=0.2, random_state=2021)

                if self.rbRobust.isChecked():
                    self.run_model('Gradient Boosting', 'Robust', self.GBRob,
self.X_train_rob, self.X_test_rob, self.y_train_rob, self.y_test_rob)
                if self.rbMinMax.isChecked():
                    self.run_model('Gradient Boosting', 'MinMax', self.GBNorm,
self.X_train_norm, self.X_test_norm, self.y_train_norm, self.y_test_norm)

                if self.rbStandard.isChecked():
                    self.run_model('Gradient Boosting', 'Standard', self.GBStand,
self.X_train_stand, self.X_test_stand, self.y_train_stand, self.y_test_stand)

            #Saves model
            joblib.dump(self.GBRob, 'GBRob.pkl')
            joblib.dump(self.GBNorm, 'GBNorm.pkl')
            joblib.dump(self.GBStand, 'GBStand.pkl')
        except Exception as e:
            print("An error occurred: ", str(e))

    def build_train_nb(self):
        try:
            if path.isfile('NBRob.pkl'):
                #Loads model
                self.NBRob = joblib.load('NBRob.pkl')
                self.NBNorm = joblib.load('NBNorm.pkl')
                self.NBStand = joblib.load('NBStand.pkl')

                if self.rbRobust.isChecked():

```

```

            self.run_model('Naive Bayes', 'Robust', self.NBRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
            if self.rbMinMax.isChecked():
                self.run_model('Naive Bayes', 'MinMax', self.NBNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

        if self.rbStandard.isChecked():
            self.run_model('Naive Bayes', 'Standard', self.NBStand,
self.X_train_stand, self.X_test_stand, self.y_train_stand, self.y_test_stand)

    else:
        #Builds and trains Naive Bayes
        self.NBRob = GaussianNB()
        self.NBNorm = GaussianNB()
        self.NBStand = GaussianNB()

        if self.rbRobust.isChecked():
            self.run_model('Naive Bayes', 'Robust', self.NBRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
            if self.rbMinMax.isChecked():
                self.run_model('Naive Bayes', 'MinMax', self.NBNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

        if self.rbStandard.isChecked():
            self.run_model('Naive Bayes', 'Standard', self.NBStand,
self.X_train_stand, self.X_test_stand, self.y_train_stand, self.y_test_stand)

        #Saves model
        joblib.dump(self.NBRob, 'NBRob.pkl')
        joblib.dump(self.NBNorm, 'NBNorm.pkl')
        joblib.dump(self.NBStand, 'NBStand.pkl')
except Exception as e:
    print("An error occurred: ", str(e))

def build_train_ada(self):
    try:
        if path.isfile('ADARob.pkl'):
            #Loads model
            self.ADARob = joblib.load('ADARob.pkl')
            self.ADANorm = joblib.load('ADANorm.pkl')
            self.ADAStand = joblib.load('ADAStand.pkl')

            if self.rbRobust.isChecked():
                self.run_model('Adaboost', 'Robust', self.ADARob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
                if self.rbMinMax.isChecked():
                    self.run_model('Adaboost', 'MinMax', self.ADANorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

            if self.rbStandard.isChecked():
                self.run_model('Adaboost', 'Standard', self.ADAStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

        else:
            #Builds and trains Adaboost
            self.ADARob = AdaBoostClassifier(n_estimators = 50, learning_rate=0.01)
            self.ADANorm = AdaBoostClassifier(n_estimators = 50, learning_rate=0.01)
            self.ADAStand = AdaBoostClassifier(n_estimators = 50, learning_rate=0.01)

        if self.rbRobust.isChecked():

```

```

            self.run_model('Adaboost', 'Robust', self.ADARob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
            if self.rbMinMax.isChecked():
                self.run_model('Adaboost', 'MinMax', self.ADANorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

        if self.rbStandard.isChecked():
            self.run_model('Adaboost', 'Standard', self.ADASTand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

        #Saves model
        joblib.dump(self.ADARob, 'ADARob.pkl')
        joblib.dump(self.ADANorm, 'ADANorm.pkl')
        joblib.dump(self.ADASTand, 'ADASTand.pkl')
    except Exception as e:
        print("An error occurred: ", str(e))

def build_train_xgb(self):
    try:
        if path.isfile('XGBRob.pkl'):
            #Loads model
            self.XGBRob = joblib.load('XGBRob.pkl')
            self.XGBNorm = joblib.load('XGBNorm.pkl')
            self.XGBStand = joblib.load('XGBStand.pkl')

            if self.rbRobust.isChecked():
                self.run_model('XGB', 'Robust', self.XGBRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
                if self.rbMinMax.isChecked():
                    self.run_model('XGB', 'MinMax', self.XGBNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

            if self.rbStandard.isChecked():
                self.run_model('XGB', 'Standard', self.XGBStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

        else:
            #Builds and trains XGB classifier
            self.XGBRob = XGBClassifier(n_estimators = 50, max_depth=10,
random_state=2021, use_label_encoder=False, eval_metric='mlogloss')
            self.XGBNorm = XGBClassifier(n_estimators = 50, max_depth=10,
random_state=2021, use_label_encoder=False, eval_metric='mlogloss')
            self.XGBStand = XGBClassifier(n_estimators = 50, max_depth=10,
random_state=2021, use_label_encoder=False, eval_metric='mlogloss')

            if self.rbRobust.isChecked():
                self.run_model('XGB', 'Robust', self.XGBRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
                if self.rbMinMax.isChecked():
                    self.run_model('XGB', 'MinMax', self.XGBNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

            if self.rbStandard.isChecked():
                self.run_model('XGB', 'Standard', self.XGBStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

        #Saves model
        joblib.dump(self.XGBRob, 'XGBRob.pkl')
        joblib.dump(self.XGBNorm, 'XGBNorm.pkl')
        joblib.dump(self.XGBStand, 'XGBStand.pkl')
    
```

```

        except Exception as e:
            print("An error occurred: ", str(e))

    def build_train_lgbm(self):
        try:
            if path.isfile('LGBMRob.pkl'):
                #Loads model
                self.LGBMRob = joblib.load('LGBMRob.pkl')
                self.LGBMNorm = joblib.load('LGBMNorm.pkl')
                self.LGBMStand = joblib.load('LGBMStand.pkl')

                if self.rbRobust.isChecked():
                    self.run_model('LGBM', 'Robust', self.LGBMRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
                if self.rbMinMax.isChecked():
                    self.run_model('LGBM', 'MinMax', self.LGBMNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

                if self.rbStandard.isChecked():
                    self.run_model('LGBM', 'Standard', self.LGBMStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

            else:
                #Builds and trains LGBM classifier
                self.LGBMRob = LGBMClassifier(max_depth = 10, n_estimators=50, subsample=0.8,
random_state=2021)
                self.LGBMNorm = LGBMClassifier(max_depth = 10, n_estimators=50, subsample=0.8,
random_state=2021)
                self.LGBMStand = LGBMClassifier(max_depth = 10, n_estimators=50,
subsample=0.8, random_state=2021)

                if self.rbRobust.isChecked():
                    self.run_model('LGBM', 'Robust', self.LGBMRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
                if self.rbMinMax.isChecked():
                    self.run_model('LGBM', 'MinMax', self.LGBMNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

                if self.rbStandard.isChecked():
                    self.run_model('LGBM', 'Standard', self.LGBMStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

            #Saves model
            joblib.dump(self.LGBMRob, 'LGBMRob.pkl')
            joblib.dump(self.LGBMNorm, 'LGBMNorm.pkl')
            joblib.dump(self.LGBMStand, 'LGBMStand.pkl')
        except Exception as e:
            print("An error occurred: ", str(e))

    def build_train_mlp(self):
        try:
            if path.isfile('MLPRob.pkl'):
                #Loads model
                self.MLPRob = joblib.load('MLPRob.pkl')
                self.MLPNorm = joblib.load('MLPNorm.pkl')
                self.MLPStand = joblib.load('MLPStand.pkl')

                if self.rbRobust.isChecked():
                    self.run_model('MLP', 'Robust', self.MLPRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
                if self.rbMinMax.isChecked():


```

```

        self.run_model('MLP', 'MinMax', self.MLPNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

    if self.rbStandard.isChecked():
        self.run_model('MLP', 'Standard', self.MLPStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

else:
    #Builds and trains MLP classifier
    self.MLPRob = MLPClassifier(random_state=2021)
    self.MLPNorm = MLPClassifier(random_state=2021)
    self.MLPStand = MLPClassifier(random_state=2021)

    if self.rbRobust.isChecked():
        self.run_model('MLP', 'Robust', self.MLPRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
        if self.rbMinMax.isChecked():
            self.run_model('MLP', 'MinMax', self.MLPNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

    if self.rbStandard.isChecked():
        self.run_model('MLP', 'Standard', self.MLPStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

    #Saves model
    joblib.dump(self.MLPRob, 'MLPRob.pkl')
    joblib.dump(self.MLPNorm, 'MLPNorm.pkl')
    joblib.dump(self.MLPStand, 'MLPStand.pkl')
except Exception as e:
    print("An error occurred: ", str(e))

def build_train_gaussian(self):
    if path.isfile('GMMRob.pkl'):
        #Loads model
        self.GMMRob = joblib.load('GMMRob.pkl')
        self.GMNNorm = joblib.load('GMNNorm.pkl')
        self.GMMStand = joblib.load('GMMStand.pkl')

        if self.rbRobust.isChecked():
            self.run_model('GMM', 'Robust', self.GMMRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
            if self.rbMinMax.isChecked():
                self.run_model('GMM', 'MinMax', self.GMNNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

        if self.rbStandard.isChecked():
            self.run_model('GMM', 'Standard', self.GMMStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

    else:
        #Builds and trains Gaussian Mixture classifier
        self.GMMRob = GaussianMixture(n_components=2, random_state=100)
        self.GMNNorm = GaussianMixture(n_components=2, random_state=100)
        self.GMMStand = GaussianMixture(n_components=2, random_state=100)

        if self.rbRobust.isChecked():
            self.run_model('GMM', 'Robust', self.GMMRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
            if self.rbMinMax.isChecked():

```

```

        self.run_model('GMM', 'MinMax', self.GMMNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

    if self.rbStandard.isChecked():
        self.run_model('GMM', 'Standard', self.GMMStand, self.X_train_stand,
self.X_test_stand, self.y_train_stand, self.y_test_stand)

    #Saves model
    joblib.dump(self.GMMRob, 'GMMRob.pkl')
    joblib.dump(self.GMMNorm, 'GMMNorm.pkl')
    joblib.dump(self.GMMStand, 'GMMStand.pkl')

def build_train_extra(self):
    try:
        if path.isfile('ExtraRob.pkl'):
            #Loads model
            self.ExtraRob = joblib.load('ExtraRob.pkl')
            self.ExtraNorm = joblib.load('ExtraNorm.pkl')
            self.ExtraStand = joblib.load('ExtraStand.pkl')

            if self.rbRobust.isChecked():
                self.run_model('Extra Trees', 'Robust', self.ExtraRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
            if self.rbMinMax.isChecked():
                self.run_model('Extra Trees', 'MinMax', self.ExtraNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

            if self.rbStandard.isChecked():
                self.run_model('Extra Trees', 'Standard', self.ExtraStand,
self.X_train_stand, self.X_test_stand, self.y_train_stand, self.y_test_stand)

        else:
            #Builds and trains Gaussian Mixture classifier
            self.ExtraRob = ExtraTreesClassifier(n_estimators=200, random_state=100)
            self.ExtraNorm = ExtraTreesClassifier(n_estimators=200, random_state=100)
            self.ExtraStand = ExtraTreesClassifier(n_estimators=200, random_state=100)

            if self.rbRobust.isChecked():
                self.run_model('Extra Trees', 'Robust', self.ExtraRob, self.X_train_rob,
self.X_test_rob, self.y_train_rob, self.y_test_rob)
            if self.rbMinMax.isChecked():
                self.run_model('Extra Trees', 'MinMax', self.ExtraNorm, self.X_train_norm,
self.X_test_norm, self.y_train_norm, self.y_test_norm)

            if self.rbStandard.isChecked():
                self.run_model('Extra Trees', 'Standard', self.ExtraStand,
self.X_train_stand, self.X_test_stand, self.y_train_stand, self.y_test_stand)

    except Exception as e:
        print("An error occurred: ", str(e))

def choose_DL_model(self):
    strCB = self.cbClassifier.currentText()

    if strCB == 'Logistic Regression':
        self.build_train_lr()

```

```
if strCB == 'Support Vector Machine':
    self.build_train_svm()

if strCB == 'K-Nearest Neighbor':
    self.build_train_knn()

if strCB == 'Decision Tree':
    self.build_train_dt()

if strCB == 'Random Forest':
    self.build_train_rf()

if strCB == 'Gradient Boosting':
    self.build_train_gb()

if strCB == 'Naive Bayes':
    self.build_train_nb()

if strCB == 'Adaboost':
    self.build_train_ada()

if strCB == 'XGB Classifier':
    self.build_train_xgb()

if strCB == 'LGBM Classifier':
    self.build_train_lgbm()

if strCB == 'MLP Classifier':
    self.build_train_mlp()

if strCB == 'Gaussian Mixture Classifier':
    self.build_train_gaussian()

if strCB == 'Extra Trees Classifier':
    self.build_train_extra()

if __name__ == '__main__':
    import sys
    app = QApplication(sys.argv)
    ex = DemoGUI_Amazon()
    ex.show()
    sys.exit(app.exec_())
```