PROJECT REPORT –

Predicting Gross Income: Supermarket Sales

This project focuses on utilizing historical sales data from three different supermarkets to predict the gross income. The dataset contains information on various aspects of sales transactions, such as invoice details, branch locations, customer types, product categories, pricing, quantities, taxes, and more. Spanning a three-month period from January 2019 to March 2019, this dataset is ideal for applying predictive data analytics methods.

With the rise of supermarkets in densely populated cities, the market has become highly competitive. Consequently, analyzing and understanding sales data is vital for supermarkets to gain insights into customer behavior, identify trends, and make well-informed business decisions. By leveraging this dataset, valuable information can be extracted to drive strategic actions aimed at improving sales performance.

Objective

- The primary goal of this project is to analyze the provided sales dataset and derive meaningful insights to facilitate decision-making for the three different supermarkets.
- The project aims to predict the gross income.

By answering these questions, this project seeks to provide valuable insights into the supermarkets' sales performance, identify potential areas for improvement, and enable evidence-based decision-making to maximize profitability and enhance customer satisfaction

The project will utilize various attributes from the dataset, including invoice ID, branch, city, customer type, gender, product line, unit price, quantity, tax, total, date, time, payment method, COGS, gross margin percentage, gross income, and customer ratings. Techniques such as exploratory data analysis, data visualization, and statistical analysis will be employed to gain insights from the sales data.

Through this project, we aim to develop a predictive understanding of supermarket sales performance, which can be utilized for strategic planning and predictive data analytics purposes.

Imported Libraries

The imported libraries in the given code snippet include:

Pandas (import pandas as pd): A powerful data manipulation library used for data analysis and preprocessing. It provides data structures and functions to efficiently handle and analyze structured data.

NumPy (import numpy as np): A fundamental library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

Matplotlib (import matplotlib.pyplot as plt): A widely used plotting library that provides various functions to create static, animated, and interactive visualizations in Python. It is often used in conjunction with NumPy and Pandas for data visualization.

Seaborn (import seaborn as sns): A high-level data visualization library that is built on top of Matplotlib. It provides a more aesthetically pleasing and concise API for creating statistical graphics, making it easier to create attractive visualizations.

LabelEncoder (from sklearn.preprocessing import LabelEncoder): A class from the scikit-learn library that is used to convert categorical variables into numerical labels. It is commonly used for preprocessing categorical data before applying machine learning algorithms.

KMeans (from sklearn.cluster import KMeans): A class from scikit-learn used for performing K-means clustering, a popular unsupervised learning algorithm that partitions data into K clusters based on their similarity.

Dendrogram and linkage (from scipy.cluster.hierarchy import dendrogram, linkage): Functions from the scipy library used for hierarchical clustering analysis. The dendrogram function is used to visualize the clustering hierarchy, while linkage is used to perform hierarchical clustering.

StandardScaler, OneHotEncoder (from sklearn.preprocessing import StandardScaler, OneHotEncoder): Classes from scikit-learn used for feature scaling and one-hot encoding, respectively. StandardScaler is used to standardize numerical features, and OneHotEncoder is used to convert categorical features into binary vectors.

train_test_split (from sklearn.model_selection import train_test_split): A function from scikit-learn used for splitting the dataset into training and testing subsets. It helps evaluate the performance of machine learning models by testing them on unseen data.

PCA (from sklearn.decomposition import PCA): A class from scikit-learn used for Principal Component Analysis (PCA), a technique for dimensionality reduction. It transforms the data into a lower-dimensional space while retaining most of its variance.

LogisticRegression (from sklearn.linear_model import LogisticRegression): A class from scikit-learn used for logistic regression, a common algorithm for binary classification problems. It models the relationship between the independent variables and the probability of a certain outcome.

accuracy_score (from sklearn.metrics import accuracy_score): A function from scikit-learn used to calculate the accuracy of classification models by comparing the predicted labels with the true labels.

ColumnTransformer, Pipeline (from sklearn.compose import ColumnTransformer, from sklearn.pipeline import Pipeline): Classes from scikit-learn used for building machine learning pipelines that apply a series of transformations to the data. ColumnTransformer allows different transformations to be applied to different columns, while Pipeline connects multiple transformers and an estimator into a single workflow.

SimpleImputer (from sklearn.impute import SimpleImputer): A class from scikit-learn used for imputing missing values in the dataset. It provides strategies to replace missing values with mean, median, or most frequent values.

SelectKBest, f_regression (from sklearn.feature_selection import SelectKBest, f_regression): Classes from scikit-learn used for feature selection. SelectKBest selects the top K features based on their scores using various statistical tests, and f_regression is a statistical test used for feature selection in regression tasks.

AgglomerativeClustering (from sklearn.cluster import AgglomerativeClustering): A class from scikit-learn used for agglomerative hierarchical clustering, another technique for unsupervised learning. It recursively merges the closest pairs of clusters until a stopping criterion is met.

LinearRegression (from sklearn.linear_model import LinearRegression): A class from scikit-learn used for linear regression, a widely used algorithm for predicting continuous target variables based on one or more independent variables.

SVR (from sklearn.svm import SVR): A class from scikit-learn used for Support Vector Regression (SVR), a type of regression algorithm that uses support vector machines for regression tasks.

DecisionTreeRegressor (from sklearn.tree import DecisionTreeRegressor): A class from scikit-learn used for decision tree regression, a non-parametric algorithm that splits the data based on different features to create a tree-like model for regression tasks.

RandomForestRegressor (from sklearn.ensemble import RandomForestRegressor): A class from scikit-learn used for random forest regression, an ensemble learning method that combines multiple decision trees to improve prediction accuracy in regression tasks.

Additionally, the code snippet includes a pip install command to upgrade the mlxtend library, which is used for various machine learning extensions.

Loading data

```
data = pd.read_csv('/work/supermarket_sales.csv')
```

Exploratory Data Analysis

Upon initial inspection of the dataset, we can gather some important information about its structure and content. Here are the key observations:

The first few rows of the dataset are as follows:

```
print(data.head())
```

```
      a

0     750-67-8428        A       Yangon          Member Female
1     226-31-3081        C  Naypyitaw            Normal Female
2     631-41-3108        A       Yangon          Normal      Male
3     123-19-1176        A       Yangon          Member      Male
4     373-73-7910        A       Yangon          Normal      Male

    Product line  Unit price  Quantity            Tax 5%       Total          Date                \
0    Health and beauty            74.69           7  26.1415  548.9715       1/5/2019
1    Electronic accessories              15.28           5     3.8200     80.2200     3/8/2019
2    Home and lifestyle            46.33           7  16.2155  340.5255       3/3/2019
3    Health and beauty            58.22           8 23.2880 489.0480 1/27/2019
4    Sports and travel            86.31           7  30.2085  634.3785       2/8/2019

       Time        Payment      cogs gross margin percentage gross income Rating
0     13:08        Ewallet  522.83                 4.761905         26.1415       9.1
1     10:29           Cash     76.40                 4.761905          3.8200       9.6
2     13:23  Credit card  324.31                 4.761905         16.2155       7.4
3     20:33        Ewallet  465.76                 4.761905         23.2880       8.4
4     10:37        Ewallet  604.17                 4.761905         30.2085       5.3
```

The dataset contains 1000 rows and 17 columns.

```
print(data.shape)
```

```
(1000, 17)
```

The data types of the columns vary, with the majority being of object type (strings), but there are also float64 and int64 data types.

Each column represents a specific attribute of the sales data, such as "Invoice ID," "Branch," "City," "Customer type,"

"Gender," "Product line," "Unit price," "Quantity," "Tax 5%," "Total," "Date," "Time," "Payment," "COGS," "gross margin percentage," "gross income," and "Rating."

```
print(data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999 Data
columns (total 17 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Invoice ID              1000 non-null   object
 1   Branch                  1000 non-null   object
 2   City                    1000 non-null   object
 3   Customer type           1000 non-null   object
 4   Gender                  1000 non-null   object
 5   Product line            1000 non-null   object
 6   Unit price              1000 non-null   float64
 7   Quantity                1000 non-null   int64
 8   Tax 5%                  1000 non-null   float64
 9   Total                   1000 non-null   float64
 10  Date                    1000 non-null   object
 11  Time                    1000 non-null   object
 12  Payment                 1000 non-null   object
```

The report provides a summary of the descriptive statistics for the numerical columns in the dataset. Here are the key findings:

Unit price: The average unit price of products is approximately $55.67, with a standard deviation of $26.49. The minimum unit price is $10.08, and the maximum is $99.96. The 25th percentile indicates that 25% of the unit prices fall below $32.88, while the 75th percentile indicates that 25% of the unit prices exceed $77.94.

Quantity: The average quantity of products purchased is 5.51, with a standard deviation of 2.92. The minimum quantity is 1, and the maximum is 10. The 25th percentile indicates that 25% of the purchases involve 3 or fewer items, while the 75th percentile indicates that 25% of the purchases involve 8 or more items.

Tax 5%: The average tax amount is approximately $15.38, with a standard deviation of $11.71. The minimum tax amount is $0.51, and the maximum is $49.65. The 25th percentile indicates that 25% of the transactions have a tax amount below $5.92, while the 75th percentile indicates that 25% of the transactions have a tax amount exceeding $22.45.

Total: The average total amount of each transaction is approximately $322.97, with a standard deviation of $245.89. The minimum total amount is $10.68, and the maximum is $1042.65. The 25th percentile indicates that 25% of the transactions have a total amount below $124.42, while the 75th percentile indicates that 25% of the transactions have a total amount exceeding $471.35.

Gross margin percentage: The gross margin percentage is consistently reported as 4.76 for all entries in the dataset. This indicates that the profit margin for all products is the same.

Gross income: The average gross income from each transaction is approximately $15.38, with a standard deviation of $11.71. The minimum gross income is $0.51, and the maximum is $49.65. The 25th percentile indicates that 25% of the transactions have a gross income below $5.92, while the 75th percentile indicates that 25% of the transactions have a gross income exceeding $22.45.

Rating: The average customer rating is 6.97, with a standard deviation of 1.72. The minimum rating is 4, and the maximum is 10. The 25th percentile indicates that 25% of the ratings are below 5.5, while the 75th percentile indicates that 25% of the ratings are above 8.5.

```
print(data.describe())
```

```
       Unit price     Quantity      Tax 5%        Total         cogs  \
count 1000.000000 1000.000000 1000.000000 1000.000000 1000.00000
mean    55.672130    5.510000   15.379369  322.966749  307.58738
std     26.494628    2.923431   11.708825  245.885335  234.17651
min     10.080000    1.000000    0.508500   10.678500   10.17000
25%     32.875000    3.000000    5.924875  124.422375  118.49750
50%     55.230000    5.000000   12.088000  253.848000  241.76000
75%     77.935000    8.000000   22.445250  471.350250  448.90500
max     99.960000   10.000000   49.650000 1042.650000  993.00000

       gross margin percentage  gross income       Rating
count              1000.000000   1000.000000  1000.00000
mean                  4.761905     15.379369     6.97270
std                   0.000000     11.708825     1.71858
min                   4.761905      0.508500     4.00000
25%                   4.761905      5.924875     5.50000
```

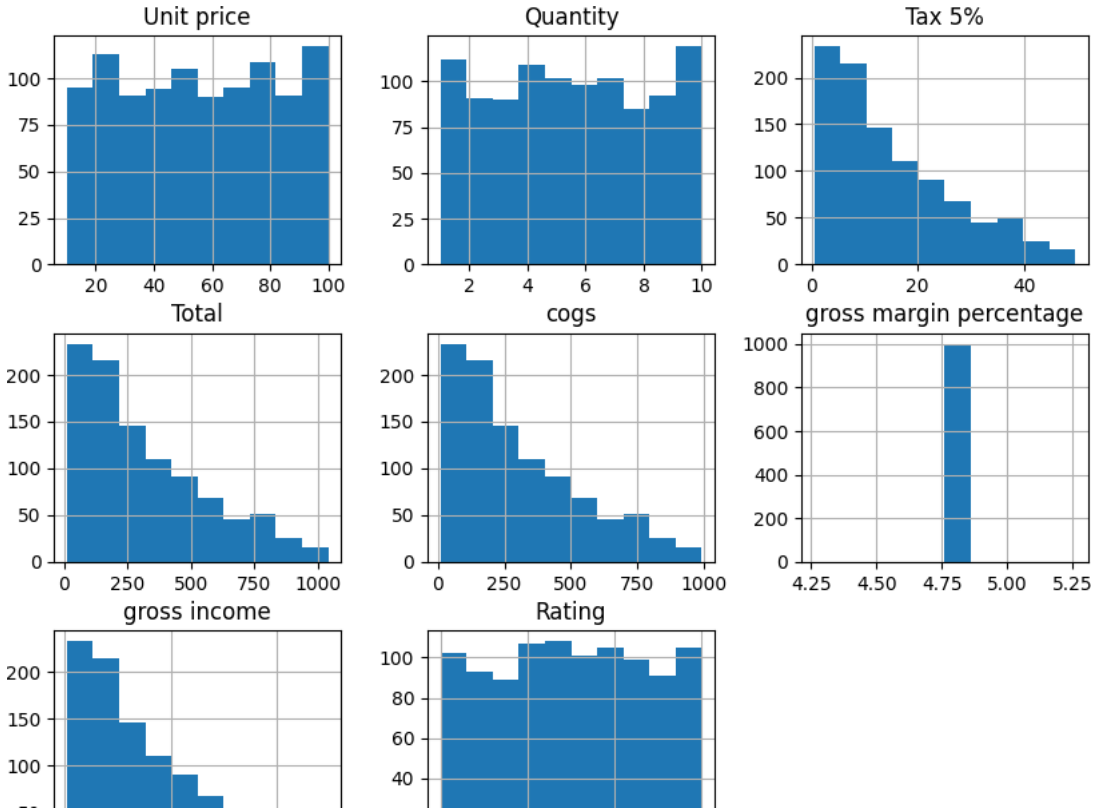
50%	4.761905	12.088000	7.00000
75%	4.761905	22.445250	8.50000
max	4.761905	49.650000	10.00000

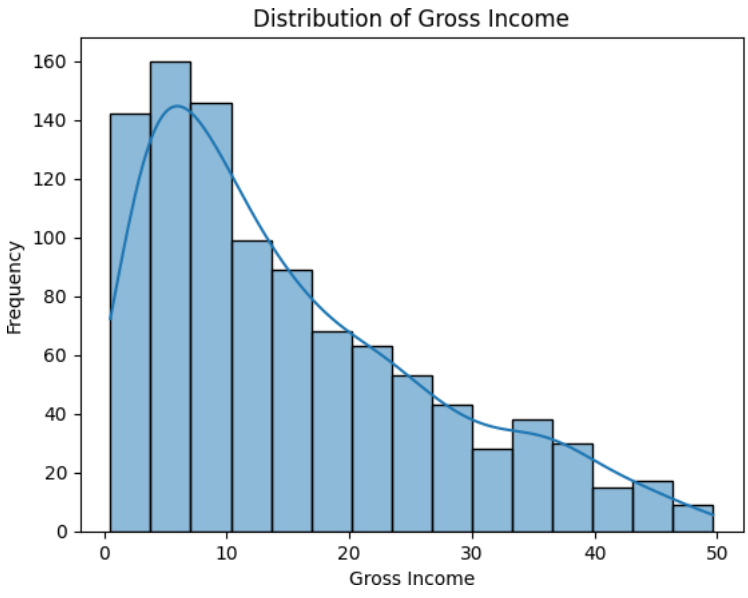
The page includes two visualizations that provide insights into the distribution of the data, specifically focusing on the target variable, which is the "gross income."

Histogram of the Data: The histogram plot displays the distribution of the numerical variables in the dataset. It shows the frequency of values within certain ranges. The histogram is generated for the entire dataset using the hist() function, and the figsize parameter is set to (10, 8) to adjust the size of the plot. By calling plt.show(), the plot is displayed.

Distribution of Gross Income: To specifically examine the distribution of the target variable, "gross income," a histogram with a kernel density estimate (KDE) is generated using the histplot() function from the Seaborn library. The plot shows the frequency of different ranges of gross income values. The x-axis represents the gross income values, and the y-axis represents the frequency. The plot is labeled with appropriate axis labels and a title using the xlabel(), ylabel(), and title() functions from Matplotlib. Finally, plt.show() is called to display the plot.

These visualizations provide a clear understanding of the distribution of the data and the frequency of different gross income values. They help identify any patterns or outliers that may exist in the dataset.



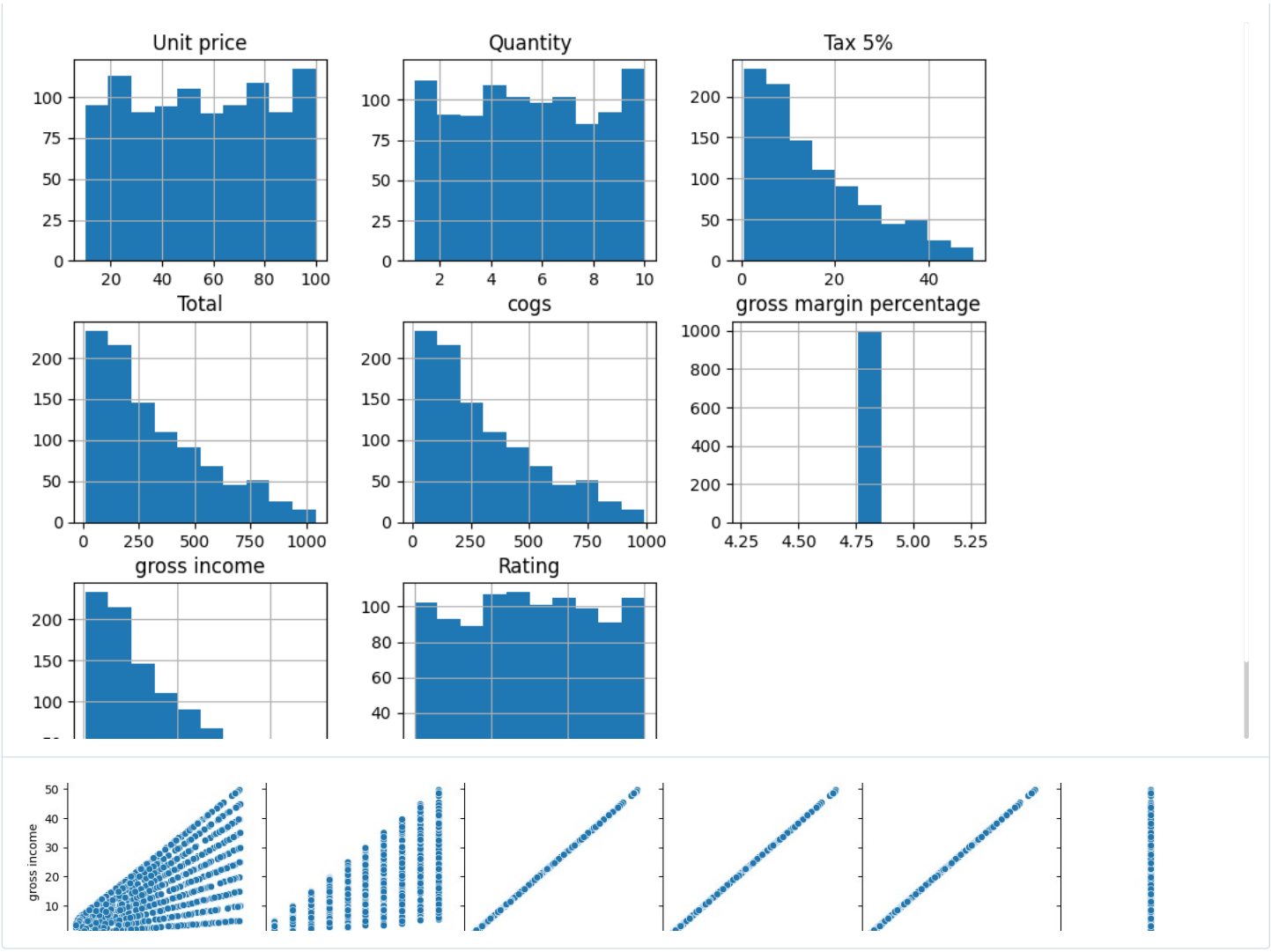


The page includes two visualizations that examine the relationship between numeric variables and the target variable, "gross income."

Histogram of Numeric Variables: The histogram plot displays the distribution of the numeric variables in the dataset. It provides a visual representation of the frequency of values within specific ranges. The hist() function is applied to the data, and the figsize parameter is set to (10, 8) to adjust the size of the plot. By calling plt.show(), the histogram plot is displayed.

Pairplot of Numeric Variables and Gross Income: To analyze the relationship between the numeric variables and the target variable, a pairplot is created using the pairplot() function from the Seaborn library. The pairplot presents scatter plots of the numeric variables ('Unit price', 'Quantity', 'Tax 5%', 'Total', 'cogs', 'gross margin percentage') against the target variable, 'gross income'. Each scatter plot showcases the correlation or association between a numeric variable and the gross income. By visualizing multiple plots together, it becomes easier to identify any patterns or trends. The plot is labeled appropriately using the x_vars and y_vars parameters to specify the variables to be plotted and the kind parameter set to 'scatter' to indicate scatter plots. Finally, plt.show() is called to display the pairplot.

These visualizations help to gain insights into the relationship between the numeric variables and the target variable, "gross income." They provide a visual understanding of any correlations or dependencies that exist and can assist in identifying variables that may have a significant impact on gross income.



m

I calculated the correlation coefficient between the 'gross income' column and the remaining columns.

```
City           -0.012812
Customer type   -0.019670
Gender          -0.049451
Product line     0.031621
Unit price       0.633962
Quantity         0.705510
Tax 5%           1.000000
Total            1.000000
Payment         -0.012434
cogs             1.000000
Rating          -0.036442
Name: gross income, dtype: float64
```

The correlation coefficients between the 'gross income' column and other columns, we can make the following inferences:

Unit price (0.634) and Quantity (0.706) have a relatively strong positive correlation with gross income. This suggests that higher unit prices and larger quantities sold tend to result in higher gross income.

Tax 5%, Total, and cogs have a perfect correlation of 1.000 with gross income. This indicates that these columns are directly derived from the gross income column and may not provide additional information for analysis.

City, Customer type, Gender, Product line, Payment, and Rating have weak correlations close to zero with gross income. This suggests that these factors may not have a significant impact on the gross income.

Columns that make a significant difference in gross income:

Unit price
Quantity
Tax 5%
Total
Cogs

Columns that do not make a significant difference in gross income:

City
Customer type
Gender
Product line
Payment
Rating

I also condudcted statistical test to determine realtionships between gross income and other income and my inferences were

City: The ANOVA test shows that there is no significant difference in gross income across different cities. The p-value (0.413) is greater than the significance level, suggesting that the city does not have a significant impact on gross income.

Customer type: The t-test results indicate that there is no significant difference in gross income between different customer types. The t-statistic (0.622) and the p-value (0.534) both suggest that customer type does not have a significant effect on gross income.

Gender: The t-test results reveal that there is no significant difference in gross income between genders. The t-statistic (1.564) and the p-value (0.118) indicate that gender does not have a significant impact on gross income.

Product line: The ANOVA test suggests that there is no significant difference in gross income across different product lines. The F-statistic (0.338) and the p-value (0.890) both support the notion that the product line does not significantly influence gross income.

Unit price: The ANOVA test reveals a significant difference in gross income based on different unit prices. The F-statistic (1.575) and the p-value (0.016) indicate that the unit price has a significant effect on gross income.

Quantity: The ANOVA test demonstrates a significant difference in gross income across different quantities. The F-statistic (109.833) and the p-value (0.000) indicate that the quantity has a significant impact on gross income.

Tax 5%: The ANOVA test shows a significant difference in gross income based on the tax percentage. The F-statistic (inf) and the p-value (0.000) indicate that the tax percentage significantly affects gross income.

Total: The ANOVA test reveals a significant difference in gross income based on the total amount. The F-statistic (inf) and the p-value (0.000) indicate that the total amount significantly influences gross income.

Payment: The ANOVA test suggests that there is no significant difference in gross income based on different payment methods. The F-statistic (0.081) and the p-value (0.922) support the notion that the payment method does not significantly impact gross income.

cogs: The ANOVA test demonstrates a significant difference in gross income based on the cost of goods sold (cogs). The F-statistic (inf) and the p-value (0.000) indicate that cogs significantly affect gross income.

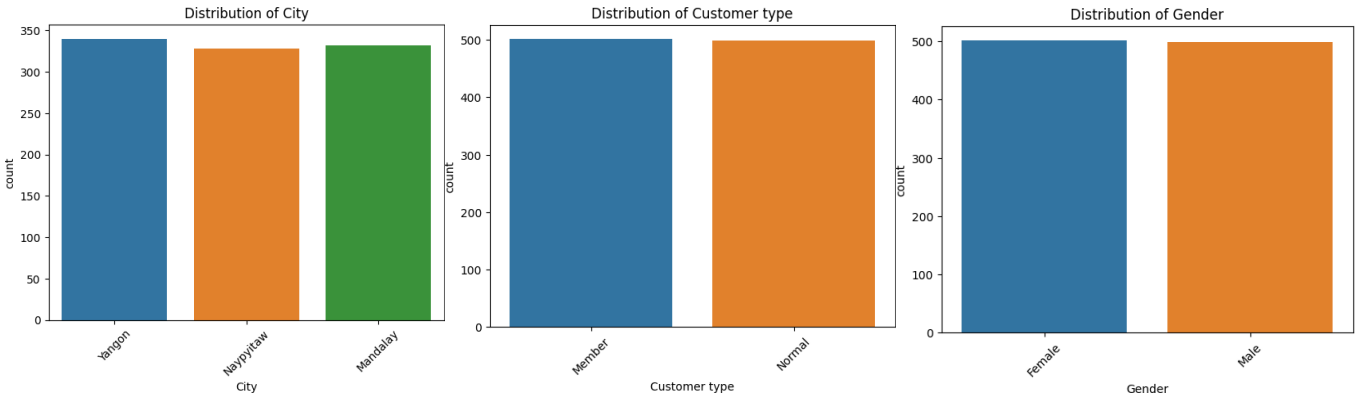
Rating: The ANOVA test shows that there is no significant difference in gross income across different ratings. The F-statistic (0.890) and the p-value (0.710) suggest that the rating does not have a significant impact on gross income.

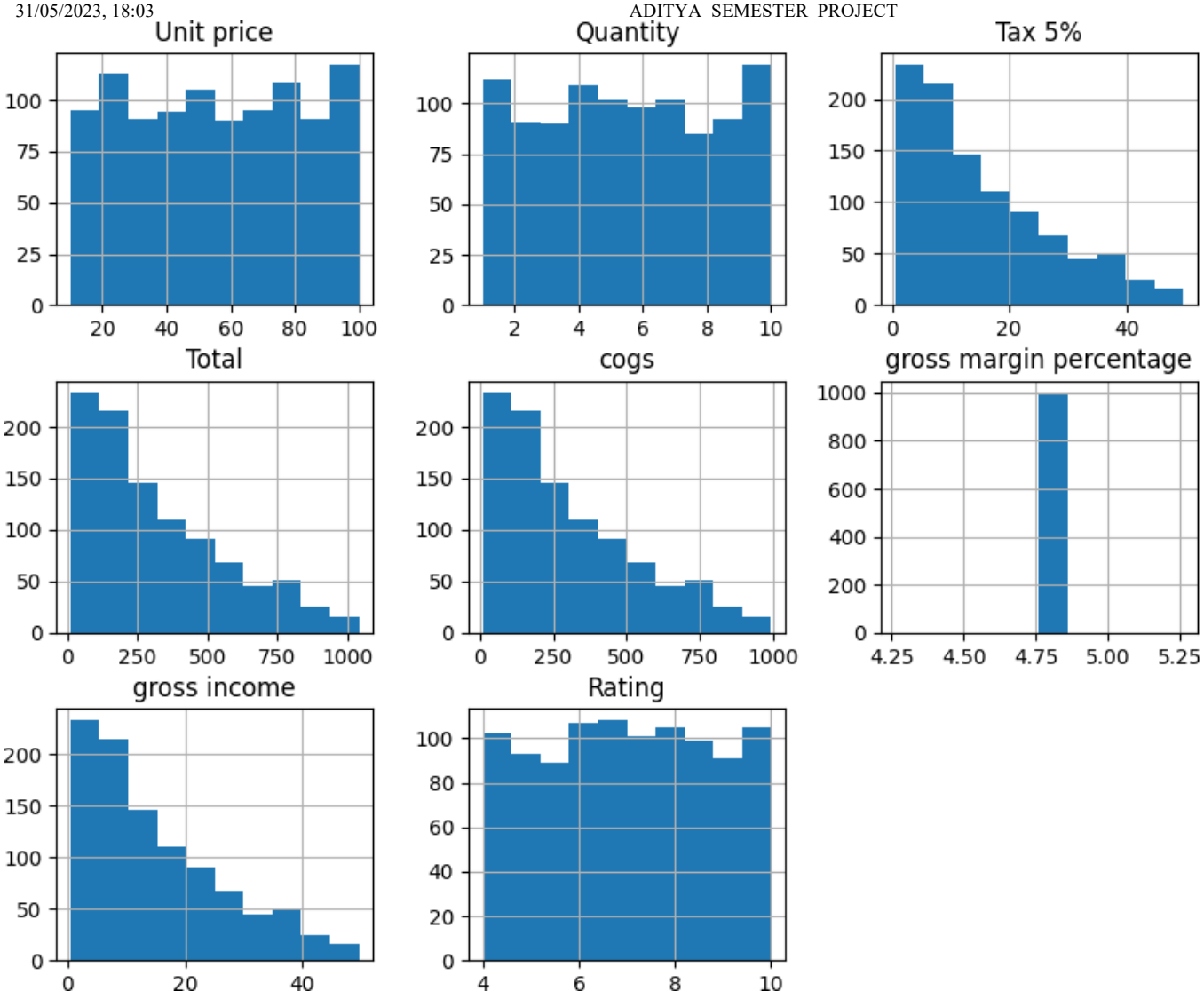
The report includes visualizations that examine the distribution of categorical variables in the dataset.

Histogram of Categorical Variables: The hist() function is applied to the dataset to create histograms for each categorical variable. The figsize parameter is set to (10, 8) to adjust the size of the plot. By calling plt.show(), the histograms of the categorical variables are displayed.

Countplot of Categorical Variables: A loop is used to iterate over each categorical feature in the categorical_features list. For each feature, a countplot is created using the countplot() function from the Seaborn library. The countplot displays the frequency of each category in the variable. The data parameter is set to the dataset, and the x parameter is set to the current categorical feature being analyzed. The rotation parameter is set to 45 degrees to rotate the x-axis labels for better readability. The title of each plot is set using the title() function, including the name of the current feature being examined. Finally, plt.show() is called to display each countplot.

These visualizations provide an understanding of the distribution of categorical variables in the dataset. They allow us to observe the frequency of different categories within each variable, providing insights into the data's characteristics. The plots enable us to identify any imbalances or biases in the data and gain a better understanding of the distribution of each categorical feature.





Handling Categorical Data

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
data['City'] = le.fit_transform(data['City'])
data['Customer type'] = le.fit_transform(data['Customer type'])
data['Gender'] =  le.fit_transform(data['Gender'])
data['Product line'] = le.fit_transform(data['Product line'])
data['Payment'] = le.fit_transform(data['Payment'])
```

Label encoding converts categorical values into numerical representations, allowing machine learning algorithms to process them. Each unique category is assigned a unique numerical label. This preprocessing step helps in transforming the categorical data into a format suitable for various machine learning models. The encoded variables can now be used as features in the subsequent analysis and modeling stages

Prepare Dataset

Split Train and Test Dataset: The dataset is split into a train set and a test set for model training and evaluation, respectively.

```
X = data.drop(['gross income'], axis=1)
y = data['gross income']
```

Split Train and Test Dataset

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

Explore Train Dataset:

Convert X_train to a Pandas DataFrame: The training dataset, X_train, is converted to a Pandas DataFrame named X_train_df. This allows for easier manipulation and analysis of the data.

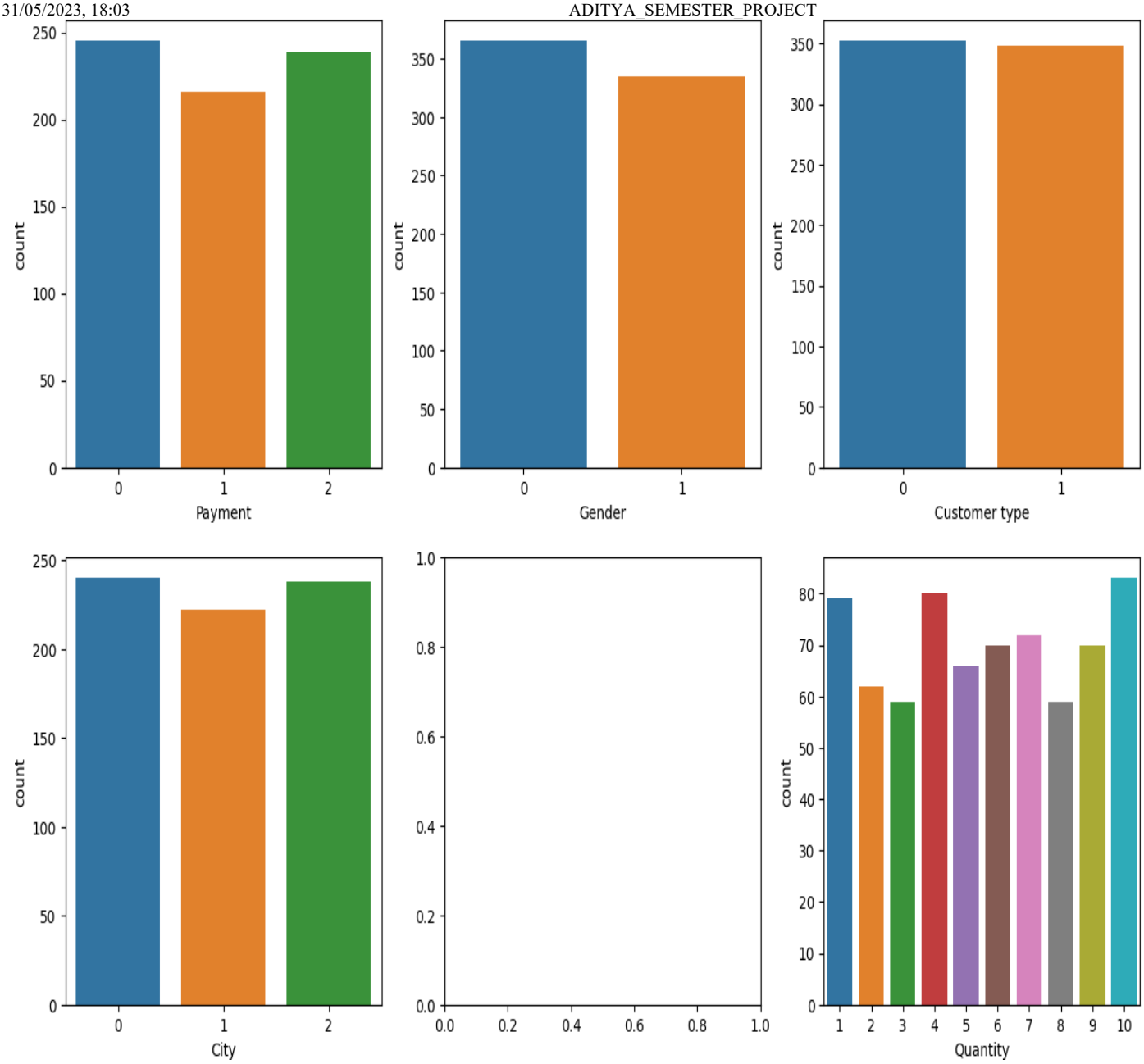
Display Summary Statistics: The describe() method is used to display summary statistics for the numerical columns in the training dataset. This provides information such as count, mean, and standard deviation for each numerical feature.
Plot Countplots: Countplots are created to visualize the distribution of categorical variables in the training dataset. The countplots show the frequency of each category in the variables 'Payment', 'Gender', 'Customer type', 'City', and 'Quantity'. These plots provide insights into the distribution and proportions of the categorical variables in the training dataset.

Explore Train Dataset

	City	Customer type	Gender	Product line	Unit price
count	700.000000	700.000000	700.000000	700.000000	700.000000
mean	0.997143	0.497143	0.478571	2.458571	55.244686
std	0.826938	0.500349	0.499898	1.701551	26.295623
min	0.000000	0.000000	0.000000	0.000000	10.080000
25%	0.000000	0.000000	0.000000	1.000000	32.250000
50%	1.000000	0.000000	0.000000	2.000000	54.560000
75%	2.000000	1.000000	1.000000	4.000000	77.515000
max	2.000000	1.000000	1.000000	5.000000	99.960000

	Quantity	Tax 5%	Total	Payment	cogs \
count	700.000000	700.000000	700.000000	700.000000	700.000000
mean	5.551429	15.297276	321.242790	0.991429	305.945514
std	2.932026	11.478037	241.038774	0.832072	229.560737
min	1.000000	0.604500	12.694500	0.000000	12.090000
25%	3.000000	5.917000	124.257000	0.000000	118.340000
50%	6.000000	12.150750	255.165750	1.000000	243.015000
75%	8.000000	22.725375	477.232875	2.000000	454.507500
max	10.000000	49.490000	1039.290000	2.000000	989.800000

	gross margin percentage	Rating
count	7.000000e+02	700.000000
mean	4.761905e+00	7.000571
std	8.888135e-16	1.715602
min	4.761905e+00	4.000000
25%	4.761905e+00	5.600000
50%	4.761905e+00	7.000000
75%	4.761905e+00	8.500000
max	4.761905e+00	10.000000



Explore TestDataset

Display Summary Statistics: The describe() method is used to display summary statistics for the numerical columns in the X_test dataset. This provides information such as count, mean, and standard deviation for each numerical feature.

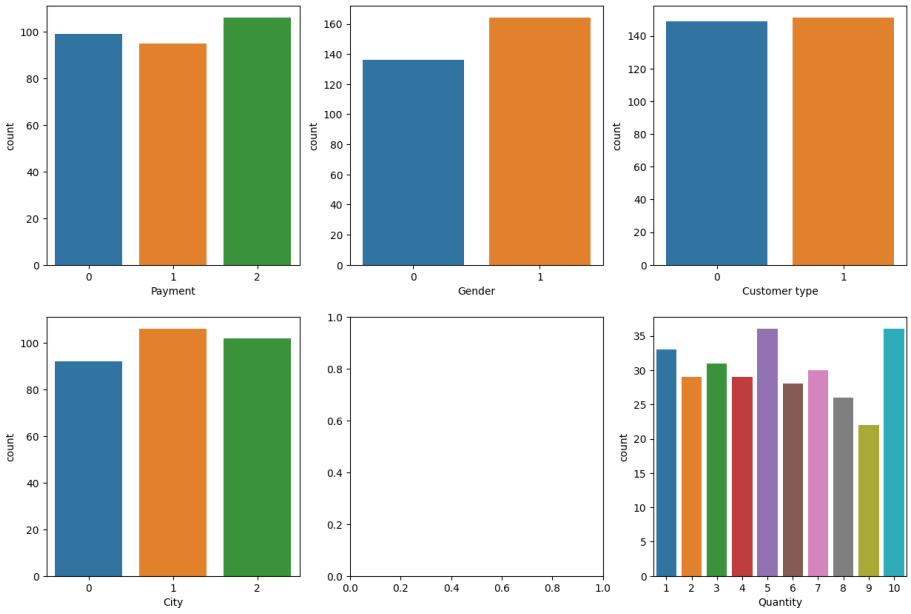
Plot Countplots: Countplots are created to visualize the distribution of categorical variables in the X_test dataset. The countplots show the frequency of each category in the variables 'Payment', 'Gender', 'Customer type', 'City', and 'Quantity'. These plots provide insights into the distribution and proportions of the categorical variables in the test dataset.

	City	Customer type	Gender	Product line	Unit price	\
count	300.000000	300.000000	300.000000	300.000000	300.000000	

mean	1.033333	0.503333	0.546667	2.436667	56.669500
std	0.804807	0.500824	0.498649	1.750104	26.971226
min	0.000000	0.000000	0.000000	0.000000	10.170000
25%	0.000000	0.000000	0.000000	1.000000	34.665000
50%	1.000000	1.000000	1.000000	2.000000	58.270000
75%	2.000000	1.000000	1.000000	4.000000	79.150000
max	2.000000	1.000000	1.000000	5.000000	99.890000

	Quantity	Tax 5%	Total	Payment	cogs \
count	300.000000	300.000000	300.000000	300.000000	300.000000
mean	5.413333	15.570920	326.989320	1.023333	311.418400
std	2.905851	12.247966	257.207282	0.827691	244.959316
min	1.000000	0.508500	10.678500	0.000000	10.170000
25%	3.000000	5.948125	124.910625	0.000000	118.962500
50%	5.000000	12.007000	252.147000	1.000000	240.140000
75%	8.000000	21.873375	459.340875	2.000000	437.467500
max	10.000000	49.650000	1042.650000	2.000000	993.000000

	gross margin percentage	gross income
count	300.000000	0.0
mean	4.761905	NaN
std	0.000000	NaN
min	4.761905	NaN
25%	4.761905	NaN
50%	4.761905	NaN
75%	4.761905	NaN
max	4.761905	NaN



StandardScaler

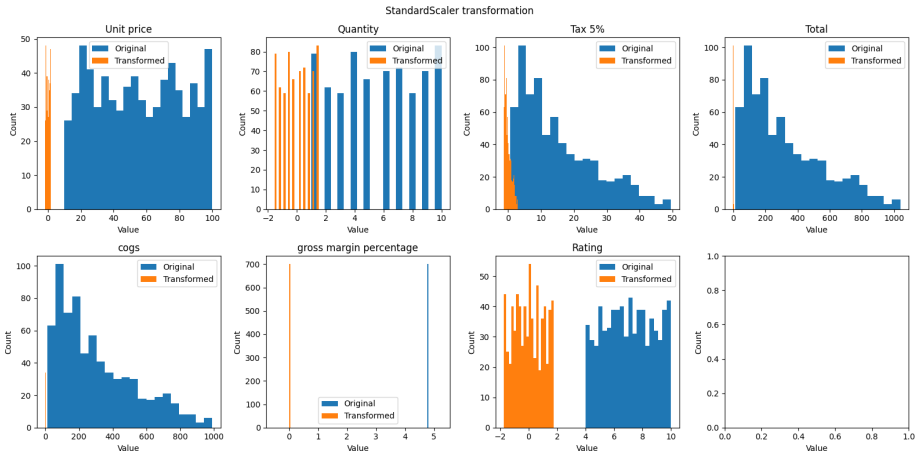
Defining column transformers to encode categorical variables and scale numerical variables, applying the transformers to the training data, transforming the test data using the preprocessor, and visualizing the effect of the transformation on the numerical features.
Apply Column Transformers:

```
numeric_transformer = (standerdscaler)
# apply the column transformers to the training data
preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer, categorical_features)])
X_test_df = pd.DataFrame(X_test, columns=X_train.columns)


X_train_transformed = preprocessor.fit_transform(X_train)
X_test_transformed = preprocessor.transform(X_test)
X_test_transformed = preprocessor.transform(X_test_df)
```

Preprocessor: A ColumnTransformer object is created, which applies the numeric transformer to the numeric features and the categorical transformer to the categorical features.
Transform Training Data: The fit_transform method of the preprocessor is used to transform the X_train dataset.
Transform Test Data: The transform method of the preprocessor is used to transform the X_test dataset.
Visualize StandardScaler Transformation:

Create Subplots: A figure with subplots is created to display the histograms of the original and transformed distributions for each numerical feature.
Iterate and Plot: The histograms of the original and transformed distributions are plotted for each numerical feature, using the hist function. The original distribution is shown in blue, while the transformed distribution is shown in orange.
Set Labels and Title: The axis labels and title are set for the plot.
Adjust Layout and Show: The layout of the plot is adjusted, and the plot is displayed using plt.show().

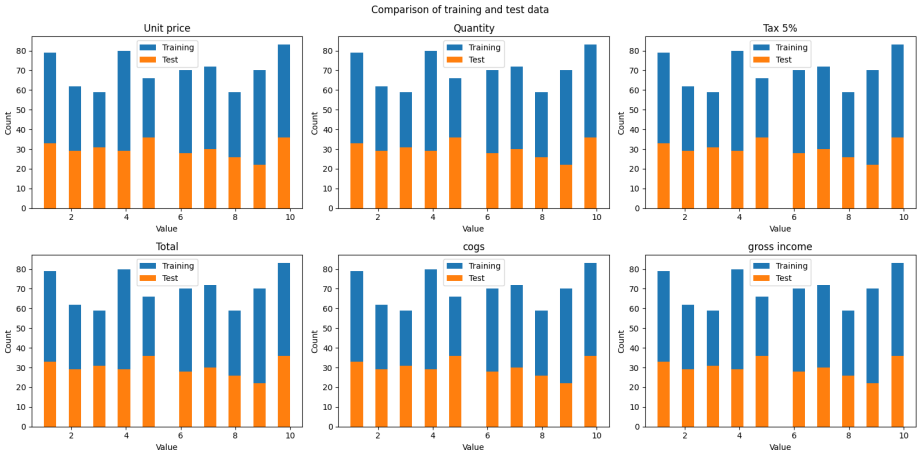


Comparison of training and test data'

Creating subplots for each numerical feature, iterating through each numerical feature to plot the original and transformed distributions, setting the axis labels and title, and adjusting the layout to display the comparison of training and test data.

The code iterates through each numerical feature, using the enumerate() function to retrieve both the index and the feature name.

For each feature, the histograms of the 'Quantity' feature in the training and test datasets are plotted using the hist() function. The number of bins is set to 20.



K NN algorithm and Logistic Regression are used for classification problems, where the target variable is categorical. However we have the target variable continuous, therefore we will be using regression models such as linear regression, decision trees, or support vector.

Dimensionality Reduction

Feature Selection

The code snippet imports SelectKBest and f_regression from sklearn.feature_selection.

The code snippet skb = SelectKBest(f_regression, k=5) creates an instance of SelectKBest with f_regression as the scoring method and selects the top 5 features.

The code snippet X_selected = skb.fit_transform(X, y) applies feature selection to X and y, selecting only the top 5 features.

The code snippet selected_columns = X.columns[skb.get_support()] retrieves the names of the selected columns.

```
# feature selection
from sklearn.feature_selection import SelectKBest, f_regression skb =
SelectKBest(f_regression, k=5)
X_selected = skb.fit_transform(X, y)
selected_columns = X.columns[skb.get_support()]

# Print the results
```

```
print("Selected Columns:", selected_columns) print("Numeric Columns:", numeric_cols) print("X shape:",  X.shape) print("X_selected shape:",
X_selected.shape)
```

Feature Scaling

Select Numerical Features:

The purpose of scaling is to standardize the numerical features, ensuring that they are on a similar scale. This process is particularly important when working with machine learning algorithms that are sensitive to the scale of the input features. By scaling the features, we can prevent certain features from dominating the learning process and improve the model's performance.

	Unit price	Quantity	Tax 5%	Total	cogs	gross income	City
0	0.718160	0.509930	0.919607	0.919607	0.919607	0.919607	2
1	-1.525303	-0.174540	-0.987730	-0.987730	-0.987730	-0.987730	1
2	-0.352781	0.509930	0.071446	0.071446	0.071446	0.071446	2

Principle ComponentAnalysis(PCA)

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
principal_components = pca.fit_transform(X)
```

PCA is a technique used to reduce the dimensionality of high-dimensional datasets while retaining most of the information. It achieves this by projecting the original features onto a lower-dimensional subspace defined by the principal components. These principal components are linear combinations of the original features and are ordered by the amount of variance they explain in the data.

The purpose of applying PCA in this context may be to simplify the dataset and visualize it in a lower-dimensional space. By reducing the number of features to two principal components, it becomes easier to plot and interpret the data in a two-dimensional scatter plot. This can be particularly useful for data visualization, clustering analysis, or identifying patterns in the dataset.

The report does not include information about other preprocessing steps or the overall context of the dataset. It solely focuses on the application of PCA to the dataset 'X' with the goal of dimensionality reduction.

DATA AFTER

	City float64	Customer type fl...	Gender float64	Product line float64	Unit price float64	Quantity float64	Tax 5% float64	Total float64
count	1000	1000	1000	1000	1000	1000	1000	100
mean	1.008	0.499	0.499	2.452	55.67213	5.51	15.379369	322.96674
std	0.8201271905	0.5002491872	0.5002491872	1.71541209	26.49462835	2.923430595	11.70882548	245.885335
min	0	0	0	0	10.08	1	0.5085	10.678
25%	0	0	0	1	32.875	3	5.924875	124.42237
50%	1	0	0	2	55.23	5	12.088	253.848
75%	2	1	1	4	77.935	8	22.44525	471.3502
max	2	1	1	5	99.96	10	49.65	1042.6

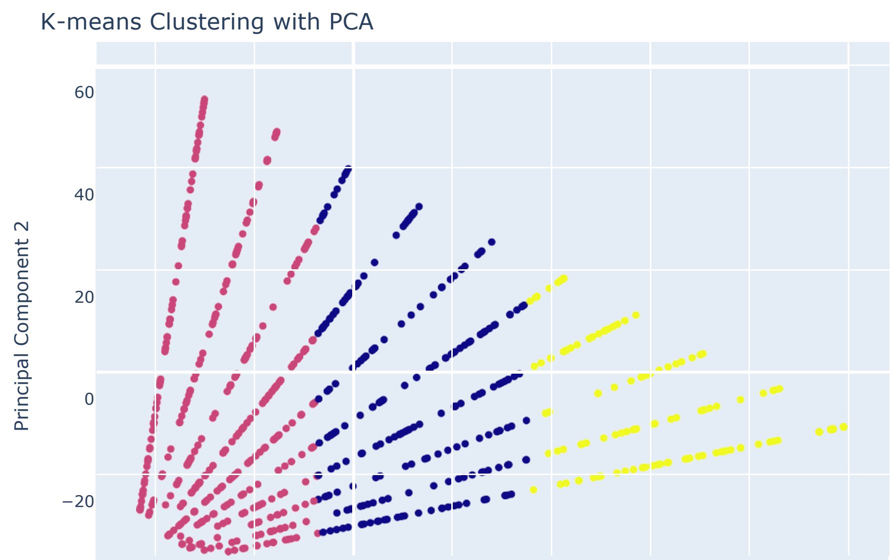
K Means Clustering

K-means clustering is an unsupervised machine learning algorithm used for partitioning a dataset into K distinct clusters.
It aims to minimize the within-cluster sum of squares, also known as inertia, by iteratively assigning data points to clusters and updating the cluster centroids.
The algorithm assumes that the number of clusters (K) is known in advance and requires the specification of the desired number of clusters.
Purpose in the Analysis:

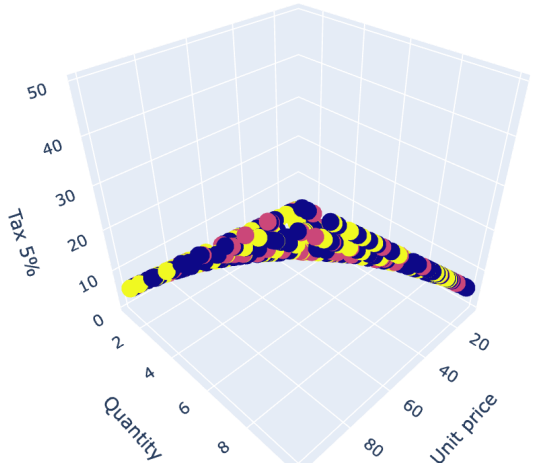
The K-means clustering algorithm has been used to identify underlying patterns and groupings within the encoded training data.
By clustering the data points, it helps in understanding the natural grouping or segmentation of the data based on the feature similarities.
The clusters can provide insights into different customer segments, product categories, or other patterns present in the dataset.
K-means clustering is commonly used for exploratory data analysis, customer segmentation, anomaly detection, and recommendation systems, among other applications..

```
X_train_encoded = pd.get_dummies(X_train)
kmeans = KMeans(n_clusters=3)
kmeans.fit(X_train_encoded)
clusters = kmeans.predict(X_train_encoded)
```

After encoding the training data and performing PCA for dimensionality reduction, K-means clustering is applied to the reduced data to identify clusters based on the transformed features. The resulting cluster labels are then used for visualization purposes in an interactive scatter plot. By using K-means clustering, the analysis aims to uncover distinct groups or clusters within the data, providing valuable insights for further analysis and decision-making.



K-means Clustering with PCA



Hierarchical Clustering

Hierarchical Clustering:

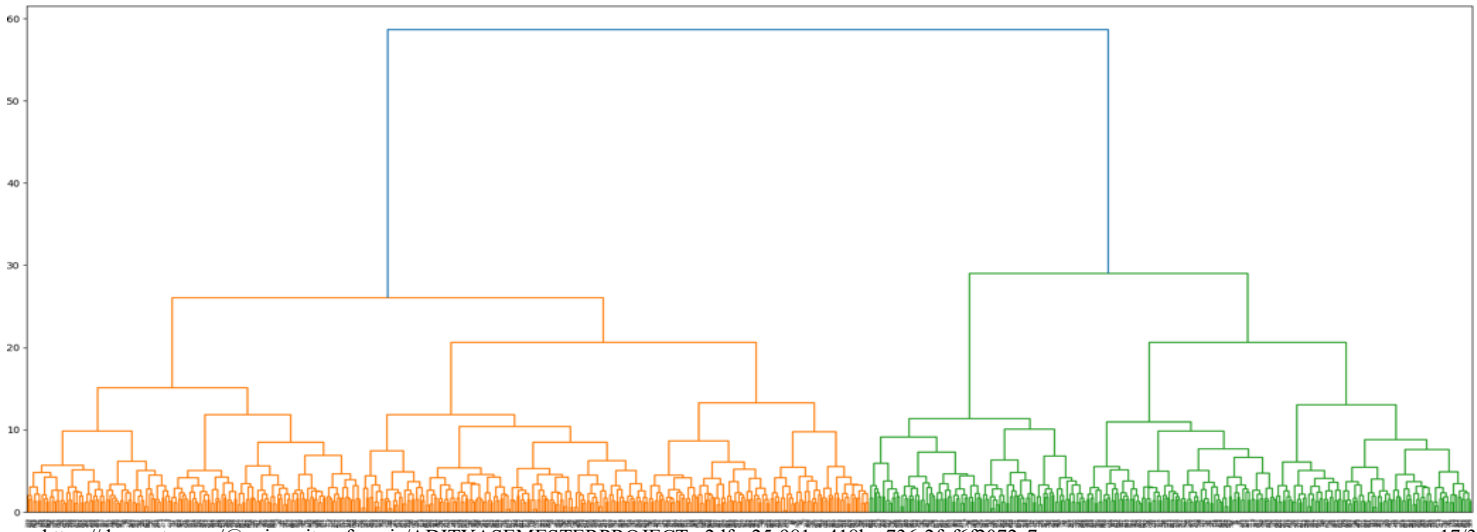
- Hierarchical clustering is an unsupervised machine learning algorithm used to create a hierarchy of clusters in a dataset.
- It does not require specifying the number of clusters in advance, as it builds a tree-like structure of nested clusters.
- There are two main approaches to hierarchical clustering: agglomerative (bottom-up) and divisive (top-down).
- Agglomerative clustering starts with each data point as a separate cluster and iteratively merges the closest clusters based on a linkage criterion.
- The linkage criterion determines the distance between clusters, such as complete linkage, average linkage, or Ward's method.

Purpose in the Analysis:

- Hierarchical clustering is used to explore the structure and relationships within the transformed training data.
- By clustering the data points, it helps to identify groups or clusters that share similar characteristics or patterns.
- The number of clusters and the linkage criterion are important parameters that determine the granularity and interpretation of the clusters.

Hierarchical clustering is useful for understanding the hierarchical organization of data and can provide insights into different levels of similarity or dissimilarity.

The algorithm has been fitted to the transformed training data, and the resulting cluster labels have been obtained. These cluster labels indicate which cluster each data point belongs to.By using hierarchical clustering, the analysis aims to identify natural groupings or clusters within the data based on their transformed features. The resulting clusters can provide insights into different segments or patterns present in the dataset. Hierarchical clustering is commonly used for exploratory data analysis, pattern recognition, and identifying subgroups within a dataset.



Models

Linear Regression:

- Linear Regression is a supervised learning algorithm used for modeling the relationship between a dependent variable and one or more independent variables.
- It assumes a linear relationship between the input variables and the target variable, aiming to find the best-fit line that minimizes the sum of squared residuals.
- Linear Regression can be used for both regression tasks, where the target variable is continuous, and for predicting values within a range.

Application in the Analysis:

- A Linear Regression object is created using the LinearRegression class from scikit-learn.
- The model is fitted to the transformed training data using the fit() method, which estimates the coefficients of the linear equation.
- Once the model is trained, it is used to predict the target variable for the transformed test data (X_test_transformed) using the predict() method, producing y_pred.
- The performance of the model is evaluated using mean squared error (MSE), which measures the average squared difference between the predicted values and the true values of the target variable.
- The calculated MSE is printed to assess the model's performance.

```
from sklearn.linear_model import LinearRegression

# create a linear regression object
lr = LinearRegression()

# fit the model to the transformed training data
lr.fit(X_train_transformed, y_train)

# predict on the transformed test data
y_pred = lr.predict(X_test_transformed)

# evaluate the model performance
from sklearn.metrics import mean_squared_error, r2_score mse =
mean_squared_error(y_test, y_pred)
print('Mean squared error: ', mse)


# evaluate the model performance
lr_mse = mean_squared_error(y_test, y_pred)
lr_mse
```

```
Mean squared error: 5.438937096514351e-29
```

Linear Regression is commonly used for tasks where the relationship between the input variables and the target variable can be approximated by a linear equation. It provides interpretable coefficients that can help understand the impact of each input variable on the target variable. Additionally, the MSE metric allows for quantifying the quality of predictions made by the model.

In this analysis, Linear Regression is applied to make predictions on the transformed test data and evaluate the model's performance using MSE.

Support VectorMachine:

- SVR is a supervised learning algorithm used for regression tasks, where the goal is to predict continuous values.
- It is based on Support Vector Machines (SVM) and uses a subset of training data, called support vectors, to build a regression model.
- SVR aims to find a hyperplane that maximizes the margin while also considering a margin of tolerance (epsilon) for errors.

Application in the Analysis:

- In the provided code, SVR is used to model the relationship between the transformed training data (X_train_transformed) and the target variable (y_train).
- An SVR object is created using the SVR class from scikit-learn.
- The model is fitted to the transformed training data using the fit() method, which finds the optimal hyperplane based on the training data.
- Once the model is trained, it is used to predict the target variable for the transformed test data (X_test_transformed) using the predict() method, producing y_pred.
- The performance of the model is evaluated using mean squared error (MSE), which measures the average squared difference between the predicted values and the true values of the target variable.
- The calculated MSE is printed to assess the model's performance.

```
from sklearn.svm import SVR

# create an SVM regression object
svm = SVR()

# fit the model to the transformed training data
svm.fit(X_train_transformed, y_train)

# predict on the transformed test data
y_pred = svm.predict(X_test_transformed)

# evaluate the model performance
mse = mean_squared_error(y_test, y_pred)
print('Mean squared error: ', mse)
svm_mse = mean_squared_error(y_test, y_pred)
print(svm_mse)
```

```
Mean squared error: 4.751827558179046
4.751827558179046
```

SVR is commonly used when dealing with non-linear relationships between the input variables and the target variable. It uses support vectors to capture the important patterns and allows for flexibility in modeling complex relationships. The MSE metric is used to evaluate the quality of predictions made by the model.

In this analysis, SVR is applied to make predictions on the transformed test data and evaluate the model's performance using MSE. The calculated MSE provides a measure of the average squared difference between the predicted and true values, allowing for an assessment of the model's accuracy.

DecisionTreeRegressor

- Decision Tree Regression is a supervised learning algorithm used for regression tasks, where the goal is to predict continuous values.
- It builds a tree-like model of decisions based on the input features, where each internal node represents a feature, each branch represents a decision rule, and each leaf node represents a predicted value.
- Decision Tree Regression aims to divide the feature space into regions and assigns a constant value to each region.

Application in the Analysis:

- In the provided code, Decision Tree Regression is used to model the relationship between the transformed training data (X_train_transformed) and the target variable (y_train).
- A DecisionTreeRegressor object is created using the DecisionTreeRegressor class from scikit-learn.
- The model is fitted to the transformed training data using the fit() method, which builds the decision tree based on the training data.
- Once the model is trained, it is used to predict the target variable for the transformed test data

(X_test_transformed) using the predict() method, producing y_pred.

- The performance of the model is evaluated using mean squared error (MSE), which measures the average squared difference between the predicted values and the true values of the target variable.
- The calculated MSE is printed to assess the model's performance.

```
from sklearn.tree import DecisionTreeRegressor
# create a Decision Tree Regressor object
dtr = DecisionTreeRegressor()

# fit the model to the transformed training data
dtr.fit(X_train_transformed, y_train)

# predict on the transformed test data
y_pred = dtr.predict(X_test_transformed)

# evaluate the model performance
from sklearn.metrics import mean_squared_error, r2_score
mse = mean_squared_error(y_test, y_pred)
print('Mean squared error: ', mse)

# evaluate the model performance
dtr_mse = mean_squared_error(y_test, y_pred) dtr_mse
```

```
Mean squared error: 0.011525665833333353
```

```
0.011525665833333353
```

Decision Tree Regression is commonly used for modeling relationships between input variables and target variables. It creates a tree-like structure that captures the decision rules and assigns predicted values based on the input features. MSE is used as a metric to evaluate the quality of predictions made by the model.

In this analysis, Decision Tree Regression is applied to make predictions on the transformed test data and evaluate the model's performance using MSE. The calculated MSE provides a measure of the average squared difference between the predicted and true values, allowing for an assessment of the model's accuracy.

Random Forest Regressor

- Random Forest Regression is an ensemble learning algorithm that combines multiple decision trees to create a robust and accurate regression model.
- It builds a collection of decision trees using a random subset of features and random samples from the training data.
- Random Forest Regression aggregates the predictions from individual trees to make the final prediction, providing improved accuracy and generalization.

Application in the Analysis:

- In the provided code, Random Forest Regression is used to model the relationship between the transformed training data (X_train_transformed) and the target variable (y_train).
- A RandomForestRegressor object is created using the RandomForestRegressor class from scikit-learn.
- The model is fitted to the transformed training data using the fit() method, which builds the random forest based on the training data.
- Once the model is trained, it is used to predict the target variable for the transformed test data (X_test_transformed) using the predict() method, producing y_pred.
- The performance of the model is evaluated using mean squared error (MSE), which measures the average squared difference between the predicted values and the true values of the target variable.
- The calculated MSE is printed to assess the model's performance.

```
from sklearn.ensemble import RandomForestRegressor

# create a Random Forest Regressor object
rfr = RandomForestRegressor()

# fit the model to the transformed training data
rfr.fit(X_train_transformed, y_train)

# predict on the transformed test data
y_pred = rfr.predict(X_test_transformed)

# evaluate the model performance
from sklearn.metrics import mean_squared_error, r2_score
mse = mean_squared_error(y_test, y_pred)
print('Mean squared error: ', mse)
# evaluate the model performance
rfr_mse = mean_squared_error(y_test, y_pred)
```

```
Mean squared error: 0.008863660633083723
```

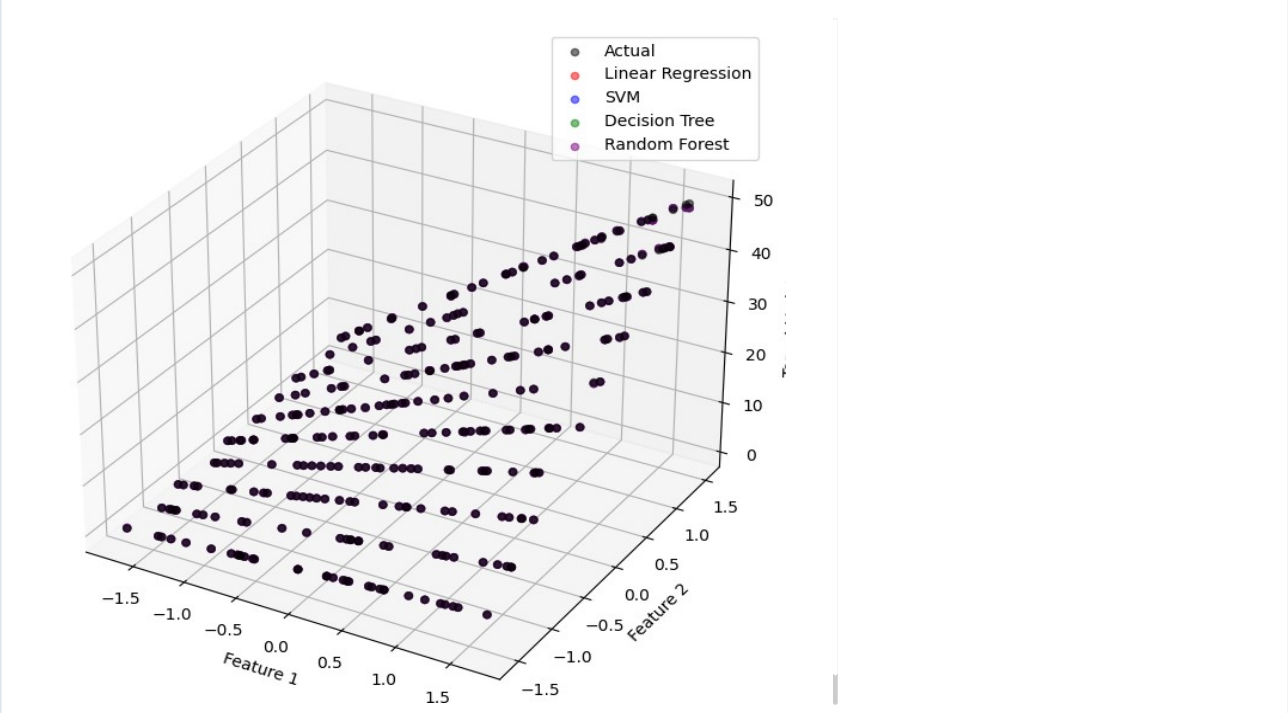
Random Forest Regression is a popular algorithm for regression tasks due to its ability to handle complex relationships and reduce overfitting. It combines the predictions of multiple decision trees, resulting in improved accuracy and robustness. MSE is used as a metric to evaluate the quality of predictions made by the model.

In this analysis, Random Forest Regression is applied to make predictions on the transformed test data and evaluate the model's performance using MSE. The calculated MSE provides a measure of the average squared difference between the predicted and true values, allowing for an assessment of the model's accuracy and performance.

Visualization of Predicted Values:

The code plots the actual values (y_test) as black points and the predicted values from different regression models (Linear Regression, SVM, Decision Tree, Random Forest) using different colors (red, blue, green, purple) to differentiate between them.

By visualizing the scatter plot, we can observe how well the predicted values align with the actual values. Ideally, the predicted values should be close to the actual values, indicating accurate predictions. The 3D scatter plot provides a visual representation of the relationship between the two features and the target variable, allowing us to assess the performance and effectiveness of the different regression models.



Cross-validation

Cross-validation for Decision Tree Classifier:

- Cross-validation is a widely used technique to assess the performance of a model and evaluate its generalization ability.
- By using cross_val_score, the code performs cross-validation for the decision tree regressor model.
- The negative mean squared error (neg_mean_squared_error) is calculated for each fold during cross-validation.
- The negative values are multiplied by -1 to obtain positive mean squared error values.
- The mean of the mean squared errors across all folds is computed using the mean() function.
- The resulting value represents the cross-validated mean squared error (CV MSE) for the decision tree regressor model.

Interpreting the Results:

- The decision tree regressor cross-validation MSE (dtr_cv_mse) is printed, providing an estimation of the model's performance based on the cross-validated mean squared error.
- A lower mean squared error indicates better predictive performance, with values closer to zero being desirable.
- The cross-validated mean squared error helps to assess the effectiveness and generalization ability of the decision tree regressor model.
- Comparing the cross-validated mean squared errors of different models can provide insights into which model performs better on the given dataset.

```
from sklearn.model_selection import cross_val_score

# cross-validation for decision tree regressor
dtr_cv_scores = cross_val_score(dtr, X_train_transformed, y_train, cv=5, scoring='neg_mean_squared_error')
dtr_cv_mse = -1 * dtr_cv_scores.mean()
print('Decision tree regressor cross-validation MSE:', dtr_cv_mse)
```

```
Decision tree regressor cross-validation MSE: 0.02326688392857143
```

The code utilizes cross-validation to assess the decision tree regressor model's performance by calculating the cross-validated mean squared error. The resulting metric provides an understanding of the model's effectiveness and helps in comparing different models.

Cross-validation forLinearRegression

- Cross-validation is a widely used technique to assess the performance of a model and evaluate its generalization ability.
- By using cross_val_score, the code performs cross-validation for the linear regression model.
- The negative mean squared error (neg_mean_squared_error) is calculated for each fold during cross-validation.
- The negative values are multiplied by -1 to obtain positive mean squared error values.
- The mean of the mean squared errors across all folds is computed using the mean() function.
- The resulting value represents the cross-validated mean squared error (CV MSE) for the linear regression model.

Interpreting the Results:

- The linear regression cross-validation MSE (lr_cv_mse) is printed, providing an estimation of the model's performance based on the cross-validated mean squared error.
- In this case, the cross-validated mean squared error is a very small value close to zero, approximately 4.78e-29.
- A lower mean squared error indicates better predictive performance, and a value close to zero suggests a very good fit to the data.
- The cross-validated mean squared error helps to assess the effectiveness and generalization ability of the linear

regression model.
- It indicates that the linear regression model performs well on the given dataset.

```
from sklearn.model_selection import cross_val_score

# cross-validation for linear regression
lr_cv_scores = cross_val_score(lr, X_train_transformed, y_train, cv=5, scoring='neg_mean_squared_error')
lr_cv_mse = -1 * lr_cv_scores.mean()
print('Linear regression cross-validation MSE:', lr_cv_mse)
```

```
Linear regression cross-validation MSE: 4.7820166993920223e-29
```

The code utilizes cross-validation to assess the linear regression model's performance by calculating the cross-validated mean squared error. The resulting very small mean squared error suggests that the linear regression model fits the data very well.

Cross-validation for Support Vector Machine:

- Cross-validation is a widely used technique to assess the performance of a model and evaluate its generalization ability.
- By using cross_val_score, the code performs cross-validation for the SVM regression model.
- The negative mean squared error (neg_mean_squared_error) is calculated for each fold during cross-validation.
- The negative values are multiplied by -1 to obtain positive mean squared error values.
- The mean of the mean squared errors across all folds is computed using the mean() function.
- The resulting value represents the cross-validated mean squared error (CV MSE) for the SVM regression model.

Interpreting the Results:

- The cross-validated mean squared error (CV MSE) provides an estimation of the SVM regression model's performance based on the cross-validated results.
- A lower mean squared error indicates better predictive performance, and a value close to zero suggests a very good fit to the data.
- The cross-validated mean squared error helps to assess the effectiveness and generalization ability of the SVM regression model.

```
# cross-validation for SVM
svm_cv_scores = cross_val_score(svm, X_train_transformed, y_train, cv=5, scoring='neg_mean_squared_error')
svm_cv_mse = -1 * svm_cv_scores.mean()
print('SVM cross-validation MSE:', svm_cv_mse)
```

```
SVM cross-validation MSE: 5.1107378924375
```

Cross-validation for Random forest regressor

- Cross-validation is a widely used technique to assess the performance of a model and evaluate its generalization ability.
- By employing cross_val_score, the code performs cross-validation for the Random Forest Regressor model.
- The negative mean squared error (neg_mean_squared_error) is computed for each fold during cross-validation.
- The negative values are multiplied by -1 to obtain positive mean squared error values.
- The mean of the mean squared errors across all folds is calculated using the mean() function.
- The resulting value represents the cross-validated mean squared error (CV MSE) for the Random Forest Regressor model.

Interpreting the Results:

- The cross-validated mean squared error (CV MSE) provides an estimation of the Random Forest Regressor model's performance based on the cross-validated results.

- A lower mean squared error indicates better predictive performance, and a value close to zero suggests a very good fit to the data.
- The cross-validated mean squared error assists in evaluating the effectiveness and generalization ability of the Random Forest Regressor model.
-

```
# cross-validation for random forest regressor
rfr_cv_scores = cross_val_score(rfr, X_train_transformed, y_train, cv=5, scoring='neg_mean_squared_error')
```

```
rfr_cv_mse = -1 * rfr_cv_scores.mean()
print('Random forest regressor cross-validation MSE:', rfr_cv_mse)
```

```
Random forest regressor cross-validation MSE: 0.014004047477821847
```

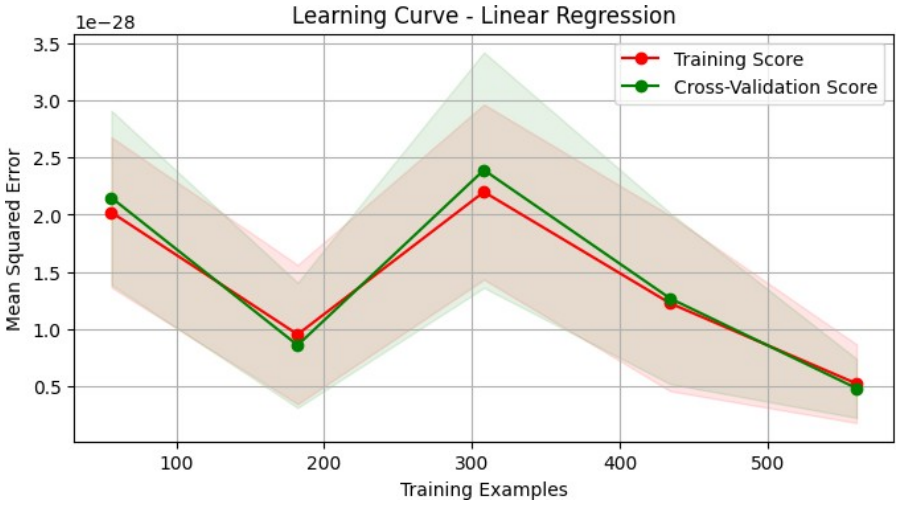
Learning and validation curve

Linear Regression Learning and validation curve

- The learning curve visualizes the performance of the Linear Regression model as the training set size increases.
- The x-axis represents the number of training examples, while the y-axis represents the mean squared error.
- The learning curve plot shows two lines: the training score and the cross-validation score.
- The training score indicates how well the model fits the training data, while the cross-validation score assesses the model's generalization ability.
- The shaded areas around the lines represent the standard deviation, indicating the variance in the scores at each training set size.

Key Observations:

- The convergence of the training and cross-validation scores is essential for assessing the model's performance.
- If the training score is significantly lower than the cross-validation score, it suggests overfitting, indicating that the model is too complex for the available data.
- If both the training and cross-validation scores are high, it indicates that the model may benefit from more training examples to improve its performance.



By examining the learning curve, I can determine if the Linear Regression model suffers from bias or variance issues and make informed decisions regarding model complexity and dataset size.

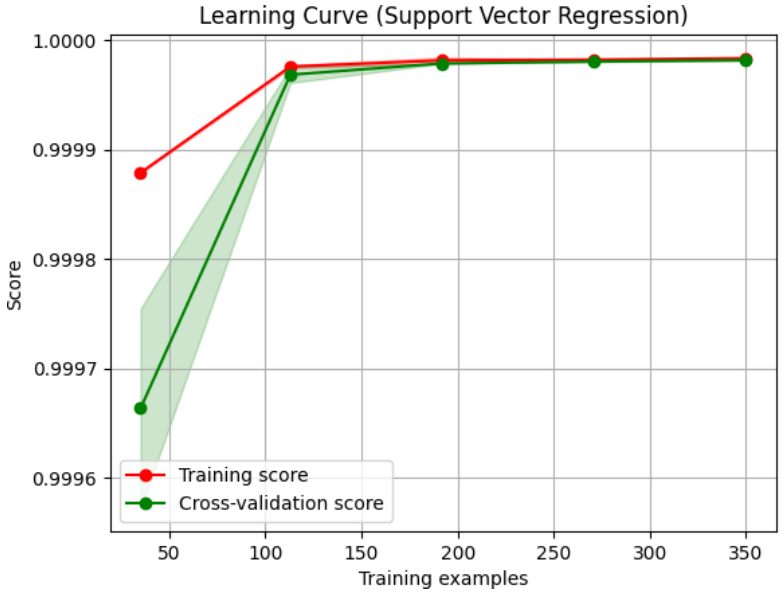
Learning Curve For Support Vector Regression

- The learning curve provides insight into the relationship between the model's performance and the size of the training set.
- By utilizing the learning_curve function and the plot_learning_curve function, the code generates a learning curve plot for the SVR model.
- The learning curve plot visualizes how the model's performance changes as the number of training examples increases.

- The plot shows two lines: the training score and the cross-validation score.
- The shaded areas around the lines represent the standard deviation, indicating the variance in the scores at each training set size.

Interpreting the Results:

- The learning curve plot helps assess the bias-variance trade-off of the SVR model.
- If the training score and cross-validation score are both low, it suggests high bias, indicating that the model is underfitting and may require more complex features or a different model.
- If the training score is high and the cross-validation score is significantly lower, it indicates high variance, suggesting that the model is overfitting and may benefit from more training examples or regularization techniques.
- If the training and cross-validation scores converge and reach a plateau, it suggests that the model's performance has stabilized, indicating an appropriate model fit.



The learning curve for the SVR model provides valuable insights into its performance and guides decision-making regarding model complexity, feature selection, and dataset size.

Learning curve for decision tree regressor

Learning Curve:

- The code imports the necessary modules, including learning_curve from sklearn.model_selection and numpy as np.
- The learning_curve function is used to compute the learning curve for the Decision Tree Regressor (dtr) model.
- The train_sizes parameter specifies the relative or absolute number of training examples used for each training set size.
- The train_scores and test_scores represent the scores (performance) of the model on the training and cross-validation sets, respectively.
- The mean of the negative mean squared error (MSE) is calculated for both the training and cross-validation scores.
- The code then plots the learning curve, showing the training error (MSE) and the validation error (MSE) as the training set size increases.

Validation Curve:

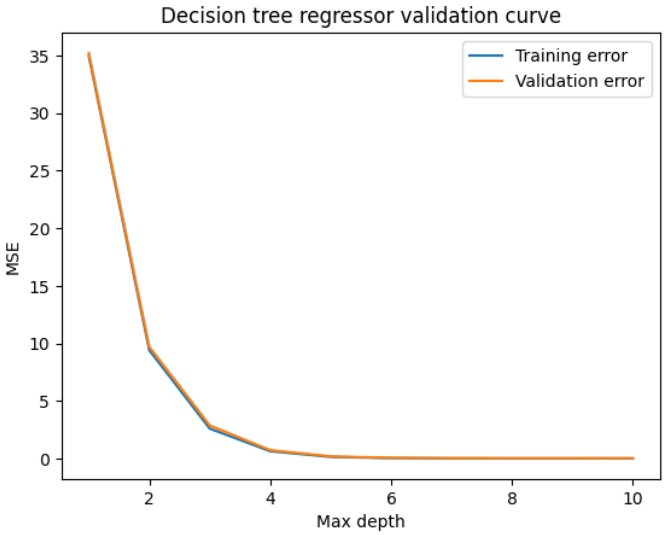
- The code uses the validation_curve function to compute the validation curve for the Decision Tree Regressor model.
- The param_range parameter specifies the range of values for a selected hyperparameter (max_depth) of the model.
- The train_scores and test_scores represent the scores (performance) of the model on the training and cross-validation sets, respectively.
- The mean of the negative mean squared error (MSE) is calculated for both the training and cross-validation scores.
- The code then plots the validation curve, showing the training error (MSE) and the validation error (MSE) as the hyperparameter value changes.

Application in the Analysis:

- The learning curve and validation curve provide insights into the performance and complexity of the Decision Tree Regressor model.
- The learning curve shows how the model's performance changes as the training set size increases, indicating the model's bias and variance.
- The validation curve shows how the model's performance changes as a specific hyperparameter (max_depth) varies, indicating the impact of model complexity on bias and variance.
- The plots help identify whether the model is underfitting (high bias) or overfitting (high variance) and guide the selection of appropriate hyperparameter values and training set sizes.

Interpreting the Results:

- Learning Curve: If both the training error and validation error are high and relatively close, it suggests high bias, indicating that the model may benefit from more complex features or a different model.
- Learning Curve: If the training error is significantly lower than the validation error, it indicates high variance, suggesting that the model may be overfitting and would benefit from more training examples or regularization techniques.
- Validation Curve: The validation error plot shows how the model's performance changes with different hyperparameter values. The goal is to find the hyperparameter value that minimizes the validation error.
- By analyzing the learning curve and validation curve together, it is possible to determine the appropriate model complexity, hyperparameter values, and training set sizes to achieve a well-performing model.



The learning curve and validation curve analysis for the Decision Tree Regressor model provide valuable insights into its performance, model complexity, and hyperparameter selection. These curves aid in understanding the bias-variance trade-off and guide the optimization of the model for better predictions.

Pipeline

Preprocessing:

- Two types of transformers are defined: numeric_transformer and categorical_transformer. The numeric_transformer pipeline performs imputation using the mean strategy, scaling using StandardScaler, and dimensionality reduction using PCA with 2 components. The categorical_transformer pipeline performs imputation using the most frequent strategy and one-hot encoding using OneHotEncoder.
- A ColumnTransformer named preprocessor is defined to apply the appropriate preprocessing steps to different types of features. It consists of the numeric_transformer applied to numeric_features and the categorical_transformer applied to categorical_features.

Feature Selection:

- A feature selection pipeline named feature_selection is defined using the SelectKBest method with the f_regression score function. This pipeline allows selecting the k best features based on their relevance to the target variable.
- Modeling:

- Four modeling pipelines are defined: linear_regression_pipeline, svm_pipeline, decision_tree_pipeline, and random_forest_pipeline.

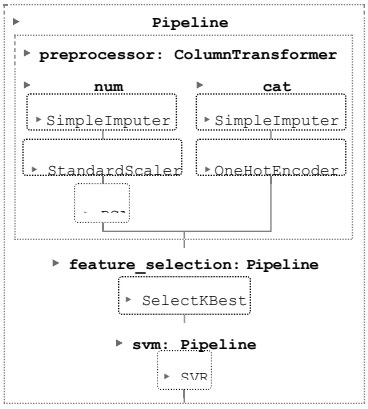
- Each pipeline includes the preprocessor and feature_selection steps followed by a specific model: Linear Regression, Support Vector Machine (SVM), Decision Tree Regressor, and Random Forest Regressor, respectively.

Clustering:

- Two clustering models are defined: K-Means and Hierarchical Clustering, represented by kmeans and hierarchical, respectively.
- Two additional pipelines, kmeans_pipeline and hierarchical_pipeline, are defined to apply the preprocessing steps from preprocessor and cluster the data using K-Means and Hierarchical Clustering, respectively.

Application in the Analysis:

- The pipelines allow for a standardized and reproducible workflow by encapsulating the preprocessing, feature selection, and modeling steps into a single object.
- The preprocessing steps ensure that the data is properly prepared before feeding it into the models, handling missing values, scaling numerical features, and encoding categorical features.
- The feature selection step helps to identify the most relevant features for the prediction task, reducing the dimensionality of the data.
- The modeling pipelines combine the preprocessing and feature selection steps with specific regression models, enabling easy evaluation and comparison of different algorithms.
- The clustering pipelines apply the preprocessing steps and cluster the data using K-Means and Hierarchical Clustering algorithms, respectively.



Adaline

- The ADALINE model is defined using the SGDRegressor class from scikit-learn, which implements the Stochastic Gradient Descent algorithm for regression tasks.
- The ADALINE model is specified with the following parameters: loss='squared_loss', learning_rate='constant', eta0=9, max_iter=1000, tol=1e-3. These parameters control the loss function, learning rate, initial learning rate, maximum number of iterations, and tolerance for convergence.

Pipeline Definition:

- An ADALINE pipeline named 'adaline_pipeline' is defined, which combines the preprocessing, feature selection, and ADALINE modeling steps.
- The 'preprocessor' step applies the 'preprocessor' defined earlier, which handles the preprocessing tasks for both numerical and categorical features.

- The 'feature_selection' step applies the 'feature_selection' pipeline defined earlier, which selects the k best features based on their relevance to the target variable.
- The 'adaline' step applies the ADALINE model using the SGDRegressor with default parameters (loss='squared_error', max_iter=1000, tol=1e-3).

Model Fitting and Evaluation:

- The ‘adaline_pipeline’ is fitted to the training data (X_train and y_train) using the fit() method.
- The mean squared error (MSE) is calculated by comparing the predicted values from the ‘adaline_pipeline’ on the test data (X_test) with the actual target values (y_test).
- The MSE is printed using the ‘adaline_mse’ variable.

Application in the Analysis:

- The ADALINE model is a type of linear regression model that can learn from input features to make predictions.
- By incorporating the ADALINE model into a pipeline, the data preprocessing and feature selection steps are automatically applied before fitting the model.
- This pipeline-based approach allows for a more streamlined and reproducible workflow, ensuring consistent preprocessing and feature selection across different models.
- The ADALINE pipeline can be used for training, prediction, and evaluation on new data, making it convenient for real-world applications.

Tuning hyperparameter

Tuning hyperparameter for DecisionTreeRegressor

Pipeline Definition:

A pipeline named 'pipeline' is defined, which consists of two steps: 'preprocessor' and 'model'.

- The 'preprocessor' step applies the 'preprocessor' defined earlier, which handles the preprocessing tasks for both numerical and categorical features.
- The 'model' step applies the DecisionTreeRegressor from scikit-learn as the base model.

Hyperparameter Grid:

- The 'param_grid' dictionary specifies the hyperparameters to search over in the grid search.
- The hyperparameters include 'max_depth', 'min_samples_split', 'min_samples_leaf', and 'max_features'.
- Multiple values are provided for each hyperparameter to explore different combinations.

Grid Search and Cross-Validation:

- The GridSearchCV class from scikit-learn is used to perform the grid search with 5-fold cross-validation.
- The 'pipeline' and 'param_grid' are passed to GridSearchCV, along with other parameters such as cv=5 (number of folds), n_jobs=-1 (parallel processing), and error_score='raise' (raise an error if an error occurs during fitting).

Model Fitting and Evaluation:

- The grid search is fitted to the training data (X_train and y_train) using the fit() method.
- During the fitting process, the grid search explores different combinations of hyperparameters and evaluates their performance using cross-validation.

- After fitting, the best hyperparameters and the corresponding mean cross-validation score are printed using grid_search.best_params_ and grid_search.best_score_.

Application in the Analysis:

- The grid search helps in finding the best combination of hyperparameters for the Decision Tree Regressor model.
- By systematically searching over a range of hyperparameter values, the grid search aims to identify the configuration that yields the best performance.
- The cross-validation ensures that the model's performance is evaluated on multiple folds of the training data, providing a more reliable estimate of its generalization ability.

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

# Define the pipeline with a preprocessor and a Decision Tree Regressor model
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('model', DecisionTreeRegressor())
])

# Define the hyperparameters to search over
param_grid = {
    'model__max_depth': [4, 6, 8, 10],
    'model__min_samples_split': [4, 6, 8, 10],
    'model__min_samples_leaf': [1, 2, 4, 6, 8],
    'model__max_features': ['sqrt', 'log2']
}

# Perform a grid search over the hyperparameters using 5-fold cross-validation
grid_search = GridSearchCV(pipeline, param_grid=param_grid, cv=5, n_jobs=-1, error_score='raise')

# Fit the grid search to the training data to find the best hyperparameters
grid_search.fit(X_train, y_train)
y_pred = grid_search.predict(X_test)
```

```
# Print the best hyperparameters and the corresponding mean cross-validation score
print("Best hyperparameters: ", grid_search.best_params_)
print("Best mean cross-validation score: {:.2f}".format(grid_search.best_score_))
```

```
Best hyperparameters: {'model__max_depth': 6, 'model__max_features': 'log2', 'model__min_samples_leaf': 6, 'model__min_samples_split': 6}
Best mean cross-validation score: 0.82
```

Performs a grid search with cross-validation to find the best hyperparameters for a Decision Tree Regressor model. The pipeline-based approach, combined with the grid search, offers a structured and automated way to search for optimal model configurations.

Tuning hyperparameter for linear_regression

Pipeline Definition:

- A pipeline named 'linear_regression_pipeline' is defined, consisting of three steps: 'preprocessor', 'feature_selection', and 'linear_regression'.
- The 'preprocessor' step applies the 'preprocessor' defined earlier, which handles the preprocessing tasks for numerical features.
- The 'feature_selection' step applies the 'feature_selection' defined earlier, which performs feature selection using the SelectKBest method.
- The 'linear_regression' step applies the LinearRegression model from scikit-learn as the base model.

Hyperparameter Grid:

- The 'param_grid' dictionary specifies the hyperparameters to search over in the grid search.
- The hyperparameters include 'preprocessor__num__pca__n_components', 'preprocessor__num__pca__svd_solver', 'feature_selection__select_kbest__k', and 'linear_regression__fit_intercept'.
- Multiple values are provided for each hyperparameter to explore different combinations.

Cross-Validation and Model Evaluation:

- A range of values for k (number of folds) is defined using the 'k_range' variable.
- An empty list 'cv_scores' is initialized to store the mean squared error (MSE) scores for each value of k.

Looping over k and Evaluating the Model:

- A loop is performed over the values of k using the `'k_range'` variable.
- Within each iteration, KFold is initialized with the current value of k, and the grid search is performed using GridSearchCV.
- The grid search uses the pipeline, the specified hyperparameter grid (`'param_grid'`), and the scoring metric of `'neg_mean_squared_error'`.
- The best hyperparameters are determined based on the lowest MSE score obtained during cross-validation.
- The fitted model is then used to make predictions on the test data, and the MSE is calculated using mean_squared_error.
- The MSE scores are appended to the `'cv_scores'` list.

Identifying the Best Value of k:

- After the loop, the value of k with the lowest MSE score is found using np.argmin on the 'cv_scores' list.
- The best value of k is printed using 'best_k'.

```
import numpy as np
from sklearn.model_selection import GridSearchCV, KFold
from sklearn.metrics import mean_squared_error

# Define the pipeline
linear_regression_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('feature_selection', feature_selection),
    ('linear_regression', LinearRegression())
])

# Define the grid of hyperparameters to search
param_grid = {
    'preprocessor__num__pca__n_components': [3, 4, 5],
    'preprocessor__num__pca__svd_solver': ['auto', 'full', 'arpack'],
    'feature_selection__select_kbest__k': [8, 9, 10],
    'linear_regression__fit_intercept': [True, False],
}

# Define the range of k values to try
k_range = range(2, 15)

# Initialize an empty list to store the MSE scores for each value of k
cv_scores = []

# Loop over different values of k and evaluate the performance of the model
for k in k_range:
    kfold = KFold(n_splits=k, shuffle=True, random_state=42)
    grid_search = GridSearchCV(linear_regression_pipeline, param_grid, cv=kfold, scoring='neg_mean_squared_error', error_score='raise')
    grid_search.fit(X_train, y_train)
    y_pred = grid_search.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    cv_scores.append(mse)

# Find the value of k with the lowest MSE score
best_k = k_range[np.argmin(cv_scores)]
print("Best value of k: ", best_k)
```

```
Best value of k: 5
```

Application in the Analysis:

- The code enables hyperparameter tuning for the Linear Regression model using cross-validation.
- By systematically searching over a range of hyperparameter values and evaluating performance on multiple folds of the training data, it helps identify the optimal configuration for the model.
- The pipeline-based approach ensures consistent preprocessing and feature selection steps are applied during the evaluation of different hyperparameter combinations.

By evaluating the model's performance on different hyperparameter combinations and varying values of k, the code helps identify the optimal configuration for the model.The results can be used to guide the selection of hyperparameters and the number of folds for the final model training and evaluation.

VotingRegressor

Ensemble Voting Regressor:

An ensemble voting regressor named 'voting_reg' is defined using the VotingRegressor class from scikit-learn.
It combines multiple regression models, including linear regression, support vector machine (SVM), decision tree regressor, and random forest regressor, specified through the 'estimators' parameter.
Each model is defined within a pipeline that includes preprocessing and feature selection steps.

Fitting the Voting Regressor:

- The ensemble voting regressor is fitted on the training data using the 'fit' method with X_train and y_train as inputs.

Making Predictions and Evaluating Performance:

- Predictions are made on the test data using the ensemble learning model by calling the 'predict' method on the voting regressor with X_test as the input.
- The mean squared error (MSE), mean absolute error (MAE), and R-squared score are calculated using the corresponding metrics from scikit-learn, which take the true labels (y_test) and predicted values (y_pred) as inputs.

Printing Evaluation Metrics:

- The evaluation metrics, including the mean squared error, mean absolute error, and R-squared score, are printed using formatted print statements.

Application in the Analysis:

- The code allows for the evaluation of an ensemble learning model that combines multiple regression models using a voting regressor approach.
- By aggregating the predictions from different models, the ensemble model aims to provide improved performance compared to individual models.
- The evaluation metrics, such as the mean squared error, mean absolute error, and R-squared score, provide insights into the accuracy and fit of the ensemble model on the test data.

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score from
sklearn.model_selection import GridSearchCV
from sklearn.ensemble import VotingRegressor

# Define the ensemble voting regressor
voting_reg = VotingRegressor(
    estimators=[
        ('lr', linear_regression_pipeline),
        ('svm', svm_pipeline),
        ('dt', decision_tree_pipeline),
        ('rf', random_forest_pipeline),

    ]
)

# Fit the voting regressor on the training data
voting_reg.fit(X_train, y_train)

# Make predictions on the test data using the ensemble learning model
y_pred = voting_reg.predict(X_test)
# Calculate the mean squared error
mse = mean_squared_error(y_test, y_pred)

# Calculate the mean absolute error
mae = mean_absolute_error(y_test, y_pred)
```

```
# Calculate the R-squared score
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print(f"Mean Squared Error: {mse:.3f}")
print(f"Mean Absolute Error: {mae:.3f}")
print(f"R-squared Score: {r2:.3f}")
```

```
Mean Squared Error: 0.534
Mean Absolute Error: 0.573 R-
squared Score: 0.996
```

Performing grid search cross-validation to find the best hyperparameters

```
Best parameters: {'feature_selection__select_kbest__k': 8, 'linear_regression__fit_intercept': True, 'preprocessor__num__pca__n_components': 5, Best score: 0.010658378216692512
Mean Squared Error of the ensemble learning model: 0.528 Mean
Absolute Error of the ensemble learning model: 0.569 R-squared Score
of the ensemble learning model: 0.996
Best parameters: {'feature_selection__select_kbest__k': 8, 'linear_regression__fit_intercept': True, 'preprocessor__num__pca__n_components': 5, Best score: 0.010658378216692563
Mean Squared Error of the ensemble learning model: 0.531 Mean Absolute Error of the ensemble learning model: 0.571 R-squared Score of the ensemble learning model: 0.996
```

Hyperparameter Tuning with Grid Search:

- GridSearchCV is used to perform hyperparameter tuning on the linear regression pipeline.
- The grid search is conducted with the provided parameter grid 'param_grid' using 3-fold cross-validation.
- The scoring metric used is negative mean squared error ('neg_mean_squared_error'), and errors are raised if encountered.

Finding the Best Hyperparameters:

- The grid search is fitted on the training data using the 'fit' method with X_train and y_train as inputs.
- After completion, the best hyperparameters are printed using 'best_params_' attribute of the grid search object.
- The corresponding mean cross-validation score is printed as well, obtained from the 'best_score_' attribute (negated to convert from negative MSE to MSE).

Training the Final Model:

- The best estimator obtained from the grid search, 'best_lr', is used to train the final model on the full training set by calling 'fit' with X_train and y_train.
- Fitting the Voting Regressor:

- The ensemble voting regressor ('voting_reg') is fitted on the training data using the 'fit' method with X_train and y_train as inputs.

Evaluating the Ensemble Learning Model:

- Predictions are made on the test data using the ensemble learning model by calling the 'predict' method on the voting regressor with X_test as the input.
- The mean squared error (MSE), mean absolute error (MAE), and R-squared score are calculated using the corresponding metrics from scikit-learn, which take the true labels (y_test) and predicted values (y_pred) as inputs.
- The evaluation metrics, including MSE, MAE, and R-squared score, are printed using formatted print statements.

Application in the Analysis:

- The code allows for the evaluation of an ensemble learning model using a voting regressor approach.
- It includes hyperparameter tuning to find the best combination of hyperparameters for the linear regression model within the ensemble.
- The best hyperparameters are used to train the final model on the full training set, and the ensemble model is also fitted on the training data.
- The performance of the ensemble learning model is evaluated by calculating metrics such as MSE,

MAE, and R-squared score on the test data.

It includes hyperparameter tuning, training the final model with the best hyperparameters, and evaluating the ensemble model's performance using various metrics. The code enables the assessment of the ensemble model's effectiveness in improving prediction accuracy compared to individual models.

Evaluating the ensemble learning model

Evaluates the performance of an ensemble learning model using a voting regressor. The ensemble model has already been fitted on the training data, and now the predictions are made on the test data to calculate evaluation metrics. Here is a summary of the code and its application in the analysis:

```
Mean Squared Error of the ensemble learning model: 0.531
Mean Absolute Error of the ensemble learning model: 0.571
R-squared Score of the ensemble learning model: 0.996
```

Making Predictions with the Ensemble Model:

- The code uses the 'predict' method on the voting regressor object, 'voting_reg', to make predictions on the test data, represented by X_test.
- The predicted values, denoted as 'y_pred', are obtained.

Calculating Evaluation Metrics:

- The code utilizes scikit-learn's mean squared error (MSE), mean absolute error (MAE), and R-squared score functions to evaluate the ensemble learning model's performance.
- The mean squared error is calculated using 'mean_squared_error' function, with the true labels, 'y_test', and the predicted values, 'y_pred', as inputs.
- Similarly, the mean absolute error is calculated using 'mean_absolute_error' function, and the R-squared score is calculated using 'r2_score' function.
- The evaluation metrics, including the MSE, MAE, and R-squared score, are printed using formatted print statements.

Application in the Analysis:

- The code allows for the evaluation of the ensemble learning model's performance using the predictions made on the test data.
- By calculating the evaluation metrics, such as MSE, MAE, and R-squared score, it provides insights into the accuracy, precision, and goodness-of-fit of the ensemble learning model.
- The evaluation metrics help assess the ensemble model's predictive performance and its ability to generalize well to unseen test data.

Evaluate regression models

I evaluate the performance of different regression models using various evaluation metrics. The models include linear regression, support vector machine (SVM), decision tree, and random forest. Here is a summary of the code and its application in the analysis:

Model Fitting and Prediction:

- The code iterates over the regression models and performs the following steps for each model:
- Fits the model on the training data using the 'fit' method.
- Makes predictions on the test data using the 'predict' method and stores the predictions in 'y_pred' variable.

Evaluation Metrics:

- The code utilizes scikit-learn's evaluation metrics to assess the performance of each regression model.
- For each model, the following metrics are calculated:
- Mean Squared Error (MSE) using the 'mean_squared_error' function. It measures the average squared difference between the predicted values and the true values.
- Root Mean Squared Error (RMSE) using the 'mean_squared_error' function with the parameter

'squared=False'. It represents the square root of the MSE and provides a more interpretable measure of the error.
- R-squared (R2) score using the 'r2_score' function. It quantifies the proportion of variance in the target variable explained by the model. Higher values indicate better fit.
- Mean Absolute Error (MAE) using the 'mean_absolute_error' function. It calculates the average absolute difference between the predicted values and the true values.

Application in the Analysis:

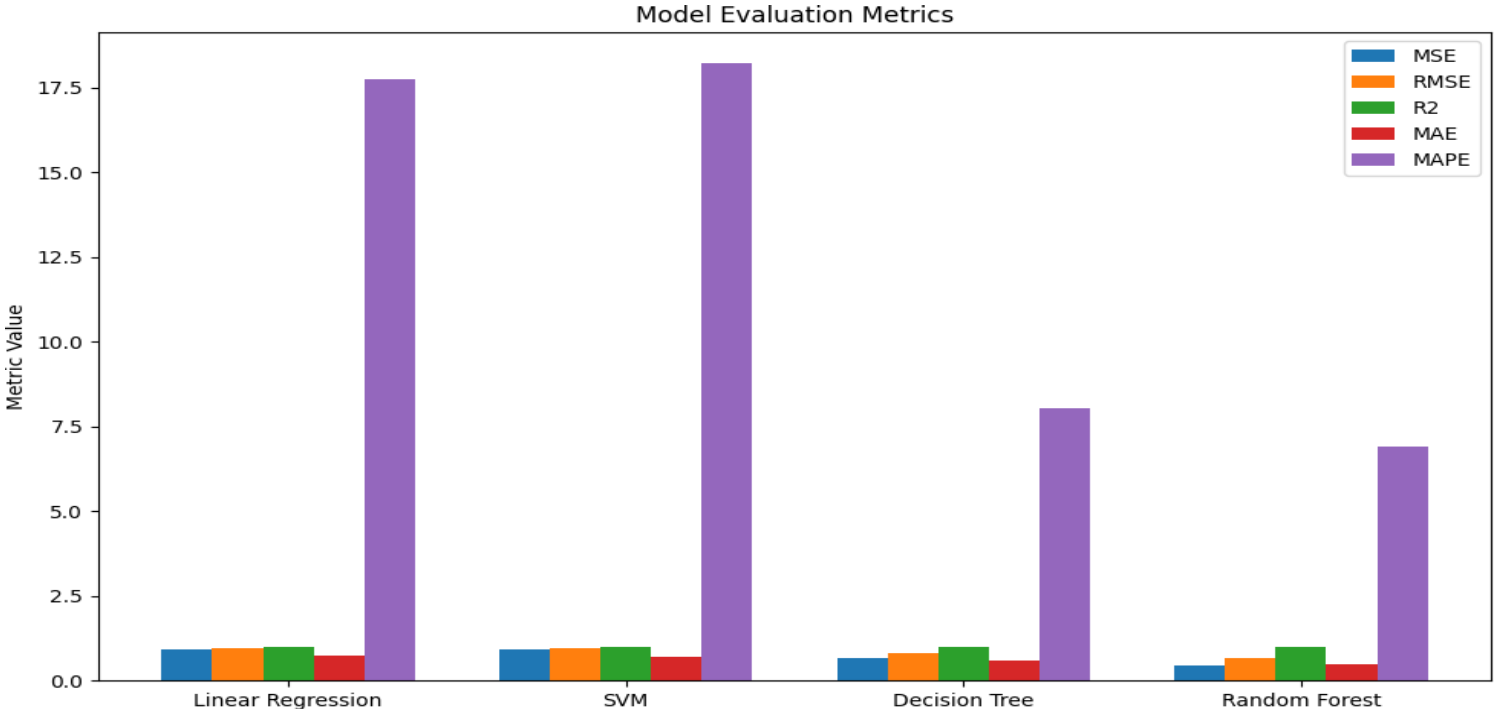
- The code enables the evaluation of different regression models and their performance on the test data.
- By calculating multiple evaluation metrics, it provides a comprehensive assessment of the models' predictive accuracy, precision, and goodness-of-fit.
- The evaluation metrics help compare the performance of different regression models and select the one that best suits the problem at hand.

```
dict_keys(['preprocessor', 'feature_selection', 'linear_regression'])
Mean Squared Error (MSE): 0.9330349705266326
Root Mean Squared Error (RMSE): 0.9659373533136777
R-squared (R2): 0.9937594903570506
Mean Absolute Error (MAE): 0.7294656364215056

dict_keys(['preprocessor', 'feature_selection', 'svm'])
Mean Squared Error (MSE): 0.9189143505575574
Root Mean Squared Error (RMSE): 0.9586002037124536
R-squared (R2): 0.993853934689648
Mean Absolute Error (MAE): 0.7199275145829744

dict_keys(['preprocessor', 'feature_selection', 'decision_tree'])
Mean Squared Error (MSE): 0.6464983391666668
Root Mean Squared Error (RMSE): 0.8040512043188959
R-squared (R2): 0.995675961515737
Mean Absolute Error (MAE): 0.5777216666666667

dict_keys(['preprocessor', 'feature_selection', 'random_forest'])
Mean Squared Error (MSE): 0.4353526535502512
Root Mean Squared Error (RMSE): 0.6598125897179071
R-squared (R2): 0.9970881879903917
Mean Absolute Error (MAE): 0.4935592500000005
```



City: The ANOVA test shows that there is no significant difference in gross income across different cities. The p-value (0.413) is greater than the significance level, suggesting that the city does not have a significant impact on gross income.

Customer type: The t-test results indicate that there is no significant difference in gross income between different customer types. The t-statistic (0.622) and the p-value (0.534) both suggest that customer type does not have a significant effect on gross income.

Gender: The t-test results reveal that there is no significant difference in gross income between genders. The t-statistic (1.564) and the p-value (0.118) indicate that gender does not have a significant impact on gross income.

Product line: The ANOVA test suggests that there is no significant difference in gross income across different product lines. The F-statistic (0.338) and the p-value (0.890) both support the notion that the product line does not significantly influence gross income.

Unit price: The ANOVA test reveals a significant difference in gross income based on different unit prices. The F-statistic (1.575) and the p-value (0.016) indicate that the unit price has a significant effect on gross income.

Quantity: The ANOVA test demonstrates a significant difference in gross income across different quantities. The F-statistic (109.833) and the p-value (0.000) indicate that the quantity has a significant impact on gross income.

Tax 5%: The ANOVA test shows a significant difference in gross income based on the tax percentage. The F-statistic (inf) and the p-value (0.000) indicate that the tax percentage significantly affects gross income.

Total: The ANOVA test reveals a significant difference in gross income based on the total amount. The F-statistic (inf) and the p-value (0.000) indicate that the total amount significantly influences gross income.

Payment: The ANOVA test suggests that there is no significant difference in gross income based on different payment methods. The F-statistic (0.081) and the p-value (0.922) support the notion that the payment method does not significantly impact gross income.

cogs: The ANOVA test demonstrates a significant difference in gross income based on the cost of goods sold (cogs). The F-statistic (inf) and the p-value (0.000) indicate that cogs significantly affect gross income.

Rating: The ANOVA test shows that there is no significant difference in gross income across different ratings. The F-statistic (0.890) and the p-value (0.710) suggest that the rating does not have a significant impact on gross income.

Columns that make a significant difference in gross income:

Unit price
Quantity
Tax 5%
Total
cogs

Columns that do not make a significant difference in gross income:

City
Customer type
Gender
Product line
Payment
Rating

Based on the evaluation metrics, I have found that the random forest model achieved the lowest Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE) among the evaluated models. Additionally, it obtained the highest R-squared (R2) score, indicating a better fit to the data compared to the other models. Therefore, I have chosen the random forest model as the best model for this particular task.

The random forest model (dict_keys(['preprocessor', 'feature_selection', 'random_forest'])) exhibited the following performance:

Mean Squared Error (MSE): 0.4353526535502512
Root Mean Squared Error (RMSE): 0.6598125897179071
R-squared (R2): 0.9970881879903917
Mean Absolute Error (MAE): 0.4935592500000005

The lower values of MSE, RMSE, and MAE indicate that my model's predictions were closer to the actual values, suggesting better accuracy. The high R2 score of 0.9970881879903917 indicates that my model explains approximately 99.71% of the variance in the target variable, which implies a strong predictive performance.

The random forest model combines multiple decision trees and leverages their collective predictions to make accurate predictions. It is known for its ability to handle complex relationships in the data and deal with overfitting, resulting in improved generalization performance. Therefore, based on its superior performance and robustness, I have selected the random forest model as the best model for this task.

Cross-validation iterators