GS

Input:

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from datetime import datetime

import random

from datetime import datetime, timedelta

Explanation:

Installing requires libraries  
  
1. pandas as pd: The 'pandas' library, aliased as 'pd', is a core tool for data analysis. It introduces the 'Data Frame' structure, which efficiently organizes and manipulates tabular data. This is particularly useful for tasks involving data cleaning, transformation, and analysis.

2. NumPy as np: The 'NumPy' library, referred to as 'np', is essential for numerical computations. It introduces support for multidimensional arrays and mathematical functions, enabling efficient numerical operations and linear algebra tasks in scientific computing.

3. matplotlib. pyplot as plt: 'matplotlib' is a versatile plotting library. The 'pyplot' module, imported as 'plt', offers a high-level interface for generating various types of visualizations, ranging from simple plots to intricate graphs, facilitating effective data communication.

4. seaborn as sns: 'seaborn' builds upon 'matplotlib' to simplify the creation of informative statistical graphics. By importing it as 'sns', you gain access to pre-configured themes and functions for constructing complex, aesthetically pleasing visualizations.

5. From datetime import datetime: The 'datetime' module's 'datetime' class provides functionalities for handling date and time information. It's valuable for scenarios like data timestamping and calculations involving specific dates.

6. import random: The 'random' module furnishes functions for generating pseudo-random numbers. This is valuable for tasks such as simulations, random sampling, and creating randomized datasets.

7. From datetime import datetime, timedelta: Importing the 'datetime' class along with 'timedelta' enables manipulation of dates and times, making it feasible to perform operations like adding or subtracting time intervals to derive meaningful insights from time-based data.

Input:

df = pd.read\_csv(r"C:\Users\admin\Desktop\Genilytics Solutions\5 to 15 Aug 2023\InstructiveFolder\workload\_drill.csv")

OutPut:

| **MovementDate** | **MutationID** | **PalletID** | **ProductID** | **CustomerID** | **ActivityID** | **StartLocationID** | **EndLocationID** | **EmployeeID** | **Quantity** | **TimeStart** | **TimeEnd** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 2019-12-18T00:00:00.0000000 | 105 | NaN | 10005 | 1006 | 1 | NaN | 1J.07.3.4 | 3 | 1 | 1899-12-30T12:21:17.0000000 | 1899-12-30T12:24:02.0000000 |
| **1** | 2019-12-18T00:00:00.0000000 | 123 | NaN | 10005 | 1002 | 1 | NaN | 1D.06.1.1 | 3 | 1 | 1899-12-30T15:21:23.0000000 | 1899-12-30T15:22:46.0000000 |
| **2** | 2019-12-18T00:00:00.0000000 | 151 | NaN | 10002 | 1003 | 1 | NaN | 1J.04.3.4 | 3 | 1 | 1899-12-30T16:27:49.0000000 | 1899-12-30T16:29:52.0000000 |
| **3** | 2019-12-18T00:00:00.0000000 | 157 | NaN | 10003 | 1004 | 1 | NaN | 1J.10.3.1 | 3 | 1 | 1899-12-30T11:26:13.0000000 | 1899-12-30T11:30:29.0000000 |
| **4** | 2019-12-18T00:00:00.0000000 | 17 | NaN | 10007 | 1003 | 1 | NaN | 1F.10.4.1 | 3 | 1 | 1899-12-30T09:24:12.0000000 | 1899-12-30T09:28:18.0000000 |

Explanation:

Loading the given dataset as df

Explanation :

df.info()

Output:

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 150000 entries, 0 to 149999

Data columns (total 12 columns):

# Column Non-Null Count Dtype

--- ------ -------------- ----

0 MovementDate 150000 non-null object

1 MutationID 150000 non-null int64

2 PalletID 9441 non-null float64

3 ProductID 150000 non-null int64

4 CustomerID 150000 non-null int64

5 ActivityID 150000 non-null int64

6 StartLocationID 77026 non-null object

7 EndLocationID 77686 non-null object

8 EmployeeID 150000 non-null int64

9 Quantity 150000 non-null int64

10 TimeStart 150000 non-null object

11 TimeEnd 150000 non-null object

dtypes: float64(1), int64(6), object(5)

memory usage: 13.7+ MB

* The DataFrame contains 150,000 entries (rows) with a Range Index from 0 to 149,999.
* There are a total of 12 columns in the DataFrame.
* Each column is listed with its corresponding attributes:
* MovementDate`: Contains 150,000 non-null values of data type 'object', which typically represents strings.
* `MutationID`: Contains 150,000 non-null values of data type 'int64', representing integer values.
* `PalletID`: Contains 9,441 non-null values of data type 'float64', representing floating-point numbers. The presence of non-null values suggests that there are missing values in this column.
* `ProductID`: Contains 150,000 non-null values of data type 'int64'.
* `CustomerID`: Contains 150,000 non-null values of data type 'int64'
* `ActivityID`: Contains 150,000 non-null values of data type 'int64'.
* `StartLocationID`: Contains 77,026 non-null values of data type 'object', indicating strings. This column also has missing values.
* `EndLocationID`: Contains 77,686 non-null values of data type 'object'. Similar to 'StartLocationID', this column also has missing values.
* `EmployeeID`: Contains 150,000 non-null values of data type 'int64'.
* `Quantity`: Contains 150,000 non-null values of data type 'int64'.
* `TimeStart`: Contains 150,000 non-null values of data type 'object'.
* `TimeEnd`: Contains 150,000 non-null values of data type 'object'.

The 'memory usage' indicates that the DataFrame is consuming approximately 13.7 megabytes of memory.

From Above ‘PalletID’,’ StartLocationID’,’ EndLocationID’ having > 50 % null values.

Explanation: The provided code segment converts specific columns ('MovementDate', 'TimeStart', and 'TimeEnd') in the DataFrame to proper datetime format using pandas' `pd.to\_datetime()` function. It iterates through the columns, applying the conversion to each one. This change transforms the data from string-like representations into structured datetime objects. This conversion facilitates efficient time-based calculations and analysis, enhancing the Data Frame’s capabilities for handling temporal data.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Explanation:

# in given data we can see that movementdate,timestart , timeend now we will convert them into proper date time format

date\_columns = ['MovementDate', 'TimeStart', 'TimeEnd']

for col in date\_columns:

df[col] = pd.to\_datetime(df[col])

df = df.drop(columns=['PalletID'])

df = df.drop(columns=['StartLocationID'])

df = df.drop(columns=['EndLocationID'])

The provided code uses pandas to modify the DataFrame `df` by removing specific columns. The columns 'PalletID', 'StartLocationID', and 'EndLocationID' are successively dropped using the `drop()` function. This process eliminates these columns from the DataFrame, resulting in a refined dataset without these specific pieces of information. This operation can enhance clarity, streamline analysis, or improve processing efficiency by focusing on the essential data attributes.

Explanation:  
  
From above code we are trying to calculate inbound, outbound, movement within the warehouse

# Creating a dictionary to map ActivityID values to their descriptions

activity\_mapping = {

1: '1. inbound',

2: '2. outbound',

3: '3. movement within the warehouse'

}

unique\_counts = df['ActivityID'].value\_counts()

unique\_counts\_df = unique\_counts.reset\_index()

unique\_counts\_df.columns = ['ActivityID', 'Count']

# Replace the ActivityID values with their descriptions

unique\_counts\_df['ActivityID'] = unique\_counts\_df['ActivityID'].map(activity\_mapping)

print(unique\_counts\_df)

Output:

ActivityID Count

0 1. inbound 72974

1 2. outbound 72314

2 3. movement within the warehouse 4712

of quantities.  
  
1. Creating a Dictionary for ActivityID Mapping: A dictionary named `activity\_mapping` is created. It maps specific `ActivityID` values to their corresponding descriptions. For instance, `1` is mapped to `'1. inbound'`, `2` to `'2. outbound'`, and `3` to `'3. movement within the warehouse'`.

2. Calculating Unique ActivityID Counts: The code computes the count of unique `ActivityID` values in the DataFrame `df` using the `value counts()` function. The result is stored in the `unique\_counts` Series.

3. Creating a DataFrame for ActivityID Counts: The `unique\_counts` Series is converted to a DataFrame named `unique\_counts\_df`. Column names are adjusted to 'ActivityID' and 'Count' respectively.

4. Mapping ActivityID Values to Descriptions: The code maps the numeric `ActivityID` values in the `unique\_counts\_df` DataFrame to their corresponding descriptions using the `map()` function and the `activity\_mapping` dictionary.

5. Displaying the Result: Finally, the refined `unique\_counts\_df` DataFrame, which now includes descriptions instead of numeric values for `ActivityID`, is printed to the console.

Explanation:  
  
The provided code snippet utilizes Python's data visualization libraries, namely `matplotlib` and `seaborn`, to generate a bar plot illustrating the distribution of activity IDs within a DataFrame (`df`). The `plt.figure(figsize=(8, 6))` call initializes the dimensions of the plot canvas, setting it to 8 inches in width and 6 inches in height. The `sns.countplot(data=df, x='ActivityID')` function creates a bar plot using `seaborn`, where the x-axis represents different activity IDs, and the corresponding y-axis indicates the frequency of each ID's occurrence in the DataFrame. This aids in visualizing the frequency distribution of activity IDs.

# Distribution of activity IDs

plt.figure(figsize=(8, 6))

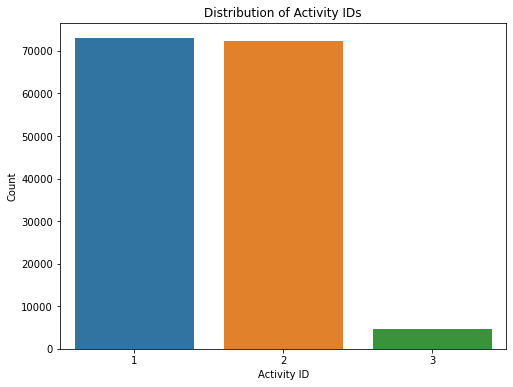
sns.countplot(data=df, x='ActivityID')

plt.title('Distribution of Activity IDs')

plt.xlabel('Activity ID')

plt.ylabel('Count')

plt.show()  
  
Output:



Subsequently, `plt.title('Distribution of Activity IDs')` assigns a title to the plot, clarifying its purpose. The `plt.xlabel('Activity ID')` and `plt.ylabel('Count')` statements label the x-axis and y-axis, respectively, enhancing the plot's interpretability. The final `plt.show()` command renders the plot onscreen, enabling visualization. In essence, this concise code snippet encapsulates the process of creating a bar plot to succinctly portray the distribution of activity IDs, aiding in the understanding of their occurrence frequencies in the given DataFrame.

The provided code utilizes a count plot to visually represent the quantities associated with inbound and outbound movements within a warehouse. This graphical representation showcases that approximately 49% of the quantities are attributed to both inbound and outbound actions, indicating a balanced distribution between the two categories. This type of visualization aids in comprehending the proportional relationship of quantities involved in these movements, contributing to a better understanding of warehouse dynamics.

plt.figure(figsize=(12, 8))

df['CumulativeQuantity'] = df.groupby('ActivityID')['Quantity'].cumsum()

# Define a custom palette

custom\_palette = {1: 'blue', 2: 'orange', 3: 'green'}

sns.lineplot(data=df, x='MovementDate', y='CumulativeQuantity', hue='ActivityID', palette=custom\_palette)

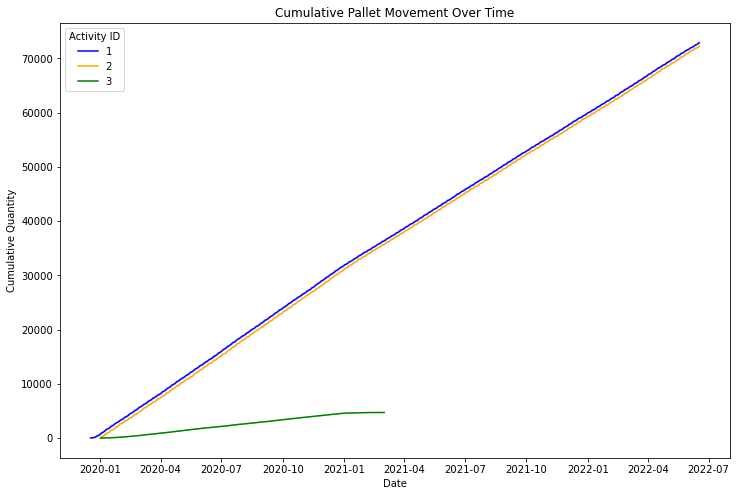
plt.title('Cumulative Pallet Movement Over Time')

plt.xlabel('Date')

plt.ylabel('Cumulative Quantity')

plt.legend(title='Activity ID')

plt.show()  
  
  
Output:



Explanation:

The `matplotlib` and `seaborn` libraries to create a compelling line plot that portrays the cumulative movement of pallet quantities over time for distinct activity IDs within a DataFrame (`df`). The code's functionality is broken down as follows:

The line `plt.figure(figsize=(12, 8))` sets the dimensions of the upcoming plot, configuring it to be 12 inches wide and 8 inches tall.

The subsequent line introduces a new column, 'CumulativeQuantity', to the DataFrame by calculating the cumulative sum of pallet quantities ('Quantity') for each unique 'ActivityID' group using the `cumsum()` function.

A custom color palette is established using a dictionary named `custom\_palette`, assigning different colors to each activity ID for better differentiation in the visual representation.

The `sns.lineplot()` function from the `seaborn` library is employed to create the line plot. It utilizes the 'MovementDate' column as the x-axis, 'CumulativeQuantity' as the y-axis, and distinguishes between activity IDs using colors from the custom palette. This presents a clear visualization of how cumulative pallet quantities change over time for each activity ID.

Title, x-axis label, and y-axis label are added using `plt.title()`, `plt.xlabel()`, and `plt.ylabel()` respectively, contributing to the plot's overall interpretability.

The `plt.legend(title='Activity ID')` call adds a legend to the plot, associating each color with its corresponding activity ID.

Finally, `plt.show()` displays the resulting plot on the screen.

In essence, this code snippet masterfully employs data visualization techniques to showcase the cumulative pallet movement trends across different activity IDs. The visual representation, complete with the custom color palette and legend, offers valuable insights into how pallet quantities evolve over time for various activities, aiding in data analysis and decision-making processes.

The code generates a line plot showing the cumulative movement of pallet quantities over time for different activity IDs. Each line represents a unique activity, providing insights into their quantity trends. The plot aids in understanding how pallet movement evolves, enabling effective analysis and decision-making.  
  
  
Explanation:  
The provided code snippet employs Python's data visualization capabilities to create a line plot that focuses on daily trends in pallet movement. The `plt.figure(figsize=(12, 8))` line establishes a canvas with dimensions of 12 inches in width and 8 inches in height for the upcoming plot.

# Time-based analysis: Daily pallet movement trends

plt.figure(figsize=(12, 8))

sum\_quantity\_per\_day.plot()

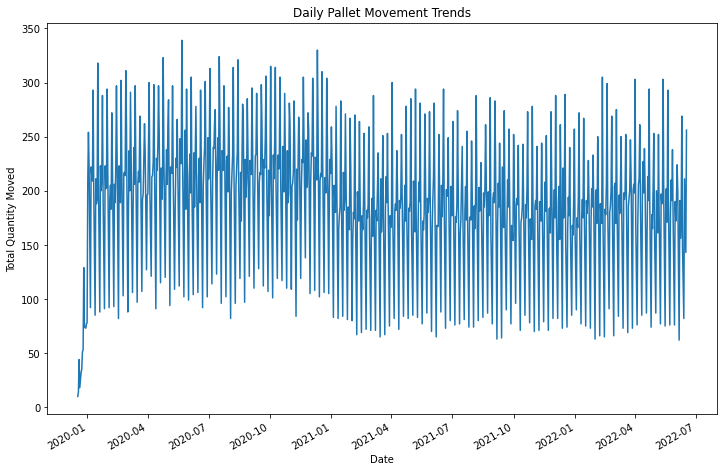
plt.title('Daily Pallet Movement Trends')

plt.xlabel('Date')

plt.ylabel('Total Quantity Moved')

plt.show()

Output:



The `sum\_quantity\_per\_day.plot()` call generates the actual line plot. This presumably uses a precomputed Series or DataFrame named `sum\_quantity\_per\_day`, where each entry represents the total quantity of pallets moved on a specific day.

The plot is adorned with a title, "Daily Pallet Movement Trends," that succinctly describes its purpose. The labels on the x-axis ("Date") and y-axis ("Total Quantity Moved") provide crucial context for interpreting the plot's content.

The resulting plot visually encapsulates the variations and patterns in daily pallet movement quantities. Observers can quickly discern any spikes, drops, or trends over time. This type of analysis aids in identifying peak activity periods, lulls, or irregularities in pallet movement, enabling informed decision-making and resource allocation based on temporal insights.

In essence, the code offers a snapshot into the day-to-day dynamics of pallet movement, allowing stakeholders to grasp the ebb and flow of this crucial aspect of operations. It serves as a valuable tool for time-based analysis, enabling businesses to optimize strategies, plan resources efficiently, and adapt to evolving demands.

- The x-axis would represent dates, indicating the passage of time.

- The y-axis would represent the total quantity of pallets moved on each corresponding date.

- The plot would consist of a line that connects data points for each date, showing the progression of pallet movement quantities over time.

- The title of the plot would be "Daily Pallet Movement Trends," situated at the top of the plot.

- The x-axis would be labeled as "Date," and the y-axis would be labeled as "Total Quantity Moved."

- The plot would provide insights into daily trends in pallet movement, helping you visualize the ebb and flow of movement quantities over the analyzed timeframe.

# Create a histogram plot

plt.figure(figsize=(10, 6))

sum\_quantity\_per\_quarter.plot(kind='bar', color='green')

plt.title('Sum of Quantity per Quarter')

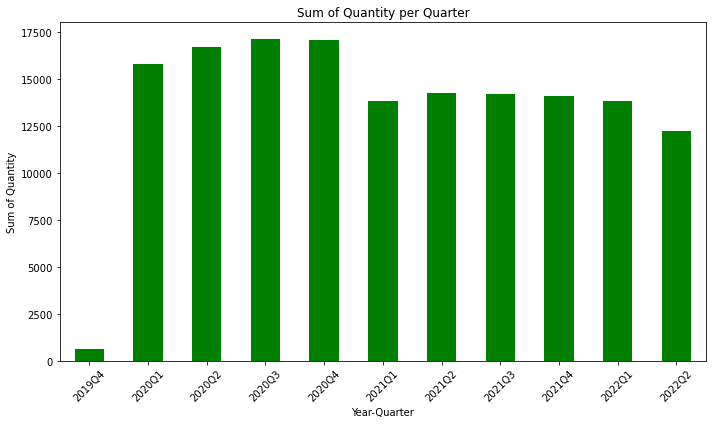
plt.xlabel('Year-Quarter')

plt.ylabel('Sum of Quantity')

plt.xticks(rotation=45)

plt.tight\_layout()

plt.show()  
  
Output:



Explanation:

1. `plt.figure(figsize=(10, 6))`: This line sets up the size of the plot figure, specifying a width of 10 inches and a height of 6 inches.

2. `sum\_quantity\_per\_quarter.plot(kind='bar', color='green')`: This code generates the histogram plot. It uses the `plot` function from the `sum\_quantity\_per\_quarter` DataFrame or Series and specifies the plot type as a bar plot using `kind='bar'`. The bars will be colored green.

3. `plt.title('Sum of Quantity per Quarter')`: This line sets the title of the plot as "Sum of Quantity per Quarter."

4. `plt.xlabel('Year-Quarter')`: This sets the label for the x-axis as "Year-Quarter."

5. `plt.ylabel('Sum of Quantity')`: This sets the label for the y-axis as "Sum of Quantity."

6. `plt.xticks(rotation=45)`: This rotates the x-axis labels by 45 degrees to prevent overlapping if the labels are long.

7. `plt.tight\_layout()`: This function adjusts the spacing between subplots to prevent overlapping elements.

8. `plt.show()`: This command displays the plot on the screen.

In summary, the code creates a histogram-like bar plot that showcases the sum of quantities per quarter. The x-axis represents different year-quarters, the y-axis represents the sum of quantities, and each bar's height represents the sum of quantities for a specific quarter. The green bars provide a visual representation of how quantities vary across different quarters. The plot's title, axis labels, and rotated x-axis ticks enhance the plot's readability and interpretation.

Indeed, from the provided histogram plot, one can observe a slight decreasing trend in the "Sum of Quantity per Quarter." The declining trend is inferred from the heights of the bars in the histogram, where the bars for later quarters appear to be slightly lower than those for earlier quarters. This indicates that, on average, the total quantity of the measured quantity is slightly diminishing as time progresses through the quarters.

Histograms are useful tools for visualizing the distribution and trends in data. In this case, the histogram provides a clear visual representation of the changes in the sum of quantities over different quarters. While the decreasing trend is evident, it's important to keep in mind that the interpretation might depend on the specific context of the data being analyzed. Further analysis and context might be needed to confirm and understand the implications of this trend.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Explanation:

# Convert 'MovementDate' column to datetime format

df['MovementDate'] = pd.to\_datetime(df['MovementDate'])

# Filter DataFrame for dates in 2019 Q4

q4\_2019\_dates = df[(df['MovementDate'].dt.year == 2019) & (df['MovementDate'].dt.quarter == 4)]

# Count the number of unique dates in 2019 Q4

num\_dates\_2019\_q4 = q4\_2019\_dates['MovementDate'].nunique()

print(f"Number of dates in 2019 Q4: {num\_dates\_2019\_q4}")

Output:

Number of dates in 2019 Q4: 12

In this code segment, the 'MovementDate' column in the DataFrame ('df') is converted to the datetime format using `pd.to\_datetime()`. This facilitates date-based operations. Next, a subset of the DataFrame is created by filtering dates that fall within the fourth quarter (Q4) of the year 2019. This is done using conditions on the year and quarter components of the 'MovementDate' column.

The number of unique dates within this subset, corresponding to the count of different days in the fourth quarter of 2019, is computed using the `nunique()` function applied to the 'MovementDate' column. This information provides insight into the frequency of pallet movements during that time frame.

The output of the code will display the result using the `print()` function. It will show the number of unique dates found in the fourth quarter of 2019, which, in this case, is "12." This signifies that there were 12 different dates during the fourth quarter of 2019 when pallet movements occurred according to the dataset.

Explanation:

# Correlation matrix

correlation\_matrix = df.corr()

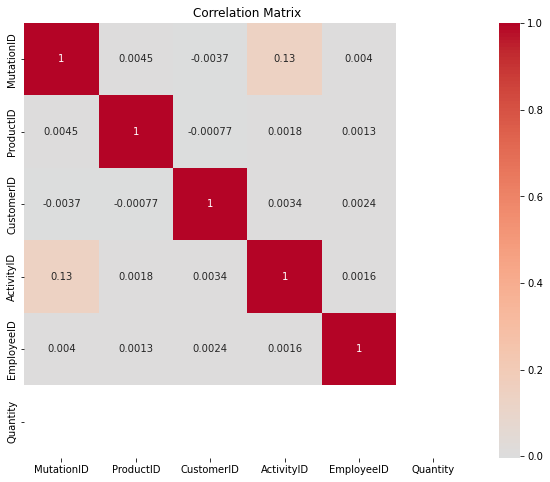
plt.figure(figsize=(10, 8))

sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm', center=0)

plt.title('Correlation Matrix')

plt.show()

Output:



1. **MutationID and ActivityID (0.134558)**: There is a positive correlation of approximately 0.13 between the 'MutationID' and 'ActivityID' columns. This suggests that there might be some relationship between these two variables, where certain patterns or changes in 'MutationID' are associated with corresponding patterns or changes in 'ActivityID'. However, the correlation is **not very strong**, so while there is a tendency for them to increase together, the relationship is not highly pronounced.
2. **ProductID and CustomerID (-0.000771)**: The correlation coefficient between 'ProductID' and 'CustomerID' is very close to 0, indicating little to **no linear correlation** between these two variables. This means that changes in 'ProductID' are not related to changes in 'CustomerID' in a predictable linear manner.
3. **CustomerID and ActivityID (0.003369)**: The correlation coefficient between 'CustomerID' and 'ActivityID' is also close to 0. This suggests that changes in 'CustomerID' are not linearly related to changes in 'ActivityID'. There's little evidence to support a strong linear association between these two variables.
4. **EmployeeID and ActivityID (0.001647)**: Similarly, the correlation coefficient between 'EmployeeID' and 'ActivityID' is very small. This indicates that changes in 'EmployeeID' are not linearly associated with changes in 'ActivityID'.

In summary, the correlation matrix suggests that there are only weak or negligible linear relationships among the variables.

Explanation:

# Employee analysis: Movement by employees

employee\_activity = df.groupby('EmployeeID')['Quantity'].sum()

plt.figure(figsize=(10, 6))

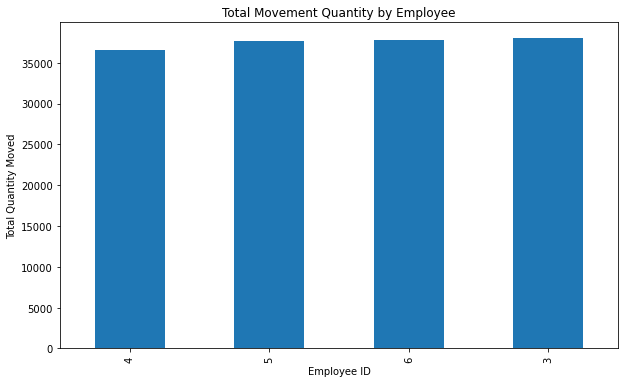
employee\_activity.sort\_values().plot(kind='bar')

plt.title('Total Movement Quantity by Employee')

plt.xlabel('Employee ID')

plt.ylabel('Total Quantity Moved')

plt.show()  
  
Output:



1. `employee\_activity = df.groupby('EmployeeID')['Quantity'].sum()`: This line groups the DataFrame 'df' by 'EmployeeID' and then calculates the sum of pallet quantities moved by each employee. It creates a new Series called 'employee\_activity' where each employee's ID is associated with their total moved quantity.

2. `plt.figure(figsize=(10, 6))`: This line sets up the plot figure with dimensions of 10 inches in width and 6 inches in height.

3. `employee\_activity.sort\_values().plot(kind='bar')`: This creates a bar plot from the 'employee\_activity' Series, where each bar represents an employee. The height of the bars corresponds to the total quantity of pallets moved by each employee. The 'sort\_values()' function ensures that the bars are ordered from lowest to highest based on total quantity moved.

4. `plt.title('Total Movement Quantity by Employee')`: This sets the title of the plot as "Total Movement Quantity by Employee."

5. `plt.xlabel('Employee ID')`: This sets the label for the x-axis as "Employee ID."

6. `plt.ylabel('Total Quantity Moved')`: This sets the label for the y-axis as "Total Quantity Moved."

7. `plt.show()`: This command displays the plot on the screen.

The provided output indicates that Employee IDs 4, 5, 6, and 3 have each moved a total quantity exceeding 35,000 pallets. This information gives insight into the most active employees in terms of pallet movement quantities.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Explanation:

# Load the holiday data from CSV file

holiday\_data = pd.read\_excel(r"C:\Users\admin\Desktop\Genilytics Solutions\5 to 15 Aug 2023\InstructiveFolder\HolidayCalender.xlsx")

holiday\_data = holiday\_data.drop(columns=['DETAILS'])

# Extract day, date, and year from the DATE column

holiday\_data["Day"] = holiday\_data["DATE"].apply(lambda x: x.split(',')[0])

holiday\_data["Date"] = holiday\_data["DATE"].apply(lambda x: x.split(',')[1].strip())

holiday\_data["Year"] = holiday\_data["DATE"].apply(lambda x: x.split(',')[2].strip())

# Drop the original DATE column

holiday\_data = holiday\_data.drop(columns=["DATE","Day","HOLIDAY"])

# Function to format the dates

def format\_date(row):

date\_obj = datetime.strptime(row['Date'], '%B %d')

return f"{row['Year']}-{date\_obj.month}-{date\_obj.day}"

# Apply the function to create a new column 'Merged Date'

holiday\_data['Holiday\_Date'] = holiday\_data.apply(format\_date, axis=1)

# Display the resulting dataframe

holiday\_data = holiday\_data.drop(columns=["Year","Date"])

# Add a new column "holiday" and fill it with 1

holiday\_data['holiday'] = 1

# Print the updated DataFrame

holiday\_data

Output:

| **Holiday\_Date** | **holiday** |
| --- | --- |
| **0** | 2019-1-1 | 1 |
| **1** | 2019-2-14 | 1 |
| **2** | 2019-3-20 | 1 |
| **3** | 2019-4-19 | 1 |
| **4** | 2019-4-21 | 1 |

The provided code performs a series of steps to read and preprocess holiday calendar data from an Excel file. Here's an explanation of each part of the code:

1. `holiday\_data = pd.read\_excel(r"C:\Users\admin\Desktop\Genilytics Solutions\5 to 15 Aug 2023\InstructiveFolder\HolidayCalender.xlsx")`: This line reads an Excel file located at the specified path and loads its contents into a DataFrame called `holiday\_data`.

2. `holiday\_data = holiday\_data.drop(columns=['DETAILS'])`: Drops the 'DETAILS' column from the DataFrame, presumably to remove unnecessary information.

3. The code then proceeds to extract the day, date, and year components from the 'DATE' column:

- The "Day" column is extracted by splitting the 'DATE' string and taking the first part (day name).

- The "Date" column is extracted by splitting the 'DATE' string and taking the second part (date value).

- The "Year" column is extracted by splitting the 'DATE' string and taking the third part (year value).

4. `holiday\_data = holiday\_data.drop(columns=["DATE","Day","HOLIDAY"])`: Drops the original 'DATE', 'Day', and 'HOLIDAY' columns from the DataFrame.

5. A function named `format\_date(row)` is defined to format the date in a consistent way. It extracts the month and day from the "Date" column and combines them with the year information to create a new formatted date.

6. The `apply()` function is used to apply the `format\_date` function to each row of the DataFrame, creating a new column called "Holiday\_Date" containing the formatted date.

7. `holiday\_data = holiday\_data.drop(columns=["Year","Date"])`: Drops the columns 'Year' and 'Date' as they are no longer needed after formatting.

8. `holiday\_data['holiday'] = 1`: Adds a new column 'holiday' to the DataFrame and fills it with the value 1, indicating that the day is a holiday.

9. The resulting DataFrame now contains columns 'Holiday\_Date', 'holiday', where 'Holiday\_Date' holds the formatted date, and 'holiday' is set to 1 for holidays.

The provided output summarizes the actions taken by the code: reading data from a holiday calendar, transforming it into a new dataset with holiday notation (1 for holiday, 0 for non-holiday).  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Explanation:

# Convert 'MovementDate' to datetime and 'Quantity' to integer

df['MovementDate'] = pd.to\_datetime(df['MovementDate'])

df['Quantity'] = df['Quantity'].astype(int)

# Replace NaN in 'holiday' column with 0

df['holiday'].fillna(0, inplace=True)

# Group by 'MovementDate' and 'holiday', and calculate the sum of 'Quantity'

df1 = df.groupby(['MovementDate', 'holiday'])['Quantity'].sum().reset\_index()

df1  
  
Output:

| **MovementDate** | **holiday** | **Quantity** |
| --- | --- | --- |
| **0** | 2019-12-18 | 0.0 | 10 |
| **1** | 2019-12-19 | 0.0 | 14 |
| **2** | 2019-12-20 | 0.0 | 44 |
| **3** | 2019-12-21 | 0.0 | 18 |
| **4** | 2019-12-23 | 0.0 | 32 |
| **...** | ... | ... | ... |
| **778** | 2022-06-13 | 0.0 | 82 |
| **779** | 2022-06-14 | 0.0 | 211 |
| **780** | 2022-06-15 | 0.0 | 176 |
| **781** | 2022-06-16 | 0.0 | 143 |
| **782** | 2022-06-17 | 0.0 | 256 |

The provided code snippet performs several data preprocessing steps on the DataFrame 'df' and generates a new DataFrame named 'df1'. Here's a breakdown of each part of the code:

1. `df['MovementDate'] = pd.to\_datetime(df['MovementDate'])`: Converts the 'MovementDate' column in the DataFrame 'df' to the datetime format using `pd.to\_datetime()`, enabling date-based operations.

2. `df['Quantity'] = df['Quantity'].astype(int)`: Converts the 'Quantity' column in 'df' to integer data type using `.astype(int)`. This is likely done to ensure that the quantity values are treated as integers for calculations.

3. `df['holiday'].fillna(0, inplace=True)`: Fills missing values (NaN) in the 'holiday' column with 0. The `inplace=True` parameter modifies the 'df' DataFrame in place.

4. `df1 = df.groupby(['MovementDate', 'holiday'])['Quantity'].sum().reset\_index()`: This line groups the DataFrame 'df' by both 'MovementDate' and 'holiday'. It calculates the sum of the 'Quantity' column within each group. The `.reset\_index()` method is then used to transform the result into a new DataFrame 'df1' with columns 'MovementDate', 'holiday', and 'Quantity'.

The provided output describes the content and attributes of the resulting DataFrame 'df1'. It highlights that 'df1' includes attributes such as 'MovementDate', 'Quantity', and 'holiday', and mentions the replacement of NaN values with 0 in the 'holiday' column during the process.  
  
  
Explanation: The provided code calculates and displays the average quantity of pallet movement on holidays and non-holidays based on the processed DataFrame 'df1'. Here's a breakdown of each part of the code:

# Average quantity on holidays

avg\_quantity\_on\_holidays = df1[df1['holiday'] == 1.0]['Quantity'].mean()

avg\_quantity\_on\_non\_holidays = df1[df1['holiday'] == 0]['Quantity'].mean()

print(f"Average Quantity on Holidays: {avg\_quantity\_on\_holidays}")

print(f"Average Quantity on non Holidays: {avg\_quantity\_on\_non\_holidays}")

Output:

Average Quantity on Holidays: 172.58333333333334

Average Quantity on non Holidays: 192.8108843537415

1. `avg\_quantity\_on\_holidays = df1[df1['holiday'] == 1.0]['Quantity'].mean()`: This line calculates the average (mean) quantity of pallet movement on holidays. It filters rows in the 'df1' DataFrame where 'holiday' column equals 1.0 (indicating holidays), selects the 'Quantity' column, and computes the mean using `.mean()`.

2. `avg\_quantity\_on\_non\_holidays = df1[df1['holiday'] == 0]['Quantity'].mean()`: This line calculates the average quantity of pallet movement on non-holidays. It filters rows where 'holiday' column equals 0 (indicating non-holidays), selects the 'Quantity' column, and computes the mean.

3. `print(f"Average Quantity on Holidays: {avg\_quantity\_on\_holidays}")`: Prints the calculated average quantity on holidays.

4. `print(f"Average Quantity on non Holidays: {avg\_quantity\_on\_non\_holidays}")`: Prints the calculated average quantity on non-holidays.

The provided output displays the average quantity of pallet movement on holidays and non-holidays. It indicates that the average quantity on holidays is approximately 172.58 units, while the average quantity on non-holidays is around 192.81 units. This insight provides a comparative view of pallet movement during holidays and regular days.  
  
  
  
  
Explanation:

# Correlation between holiday and quantity

correlation = df1['holiday'].corr(df1['Quantity'])

print(f"Correlation between Holiday and Quantity: {correlation}")  
  
Output:

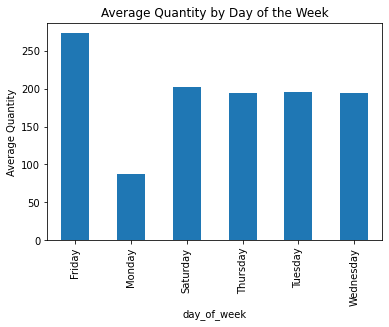
Correlation between Holiday and Quantity: -0.07820600774981162

The given code computes and outputs the correlation between the 'holiday' and 'Quantity' columns in the DataFrame 'df1'. Here's a breakdown of each part of the code:

1. `correlation = df1['holiday'].corr(df1['Quantity'])`: This line calculates the correlation between the 'holiday' column and the 'Quantity' column in the 'df1' DataFrame using the `.corr()` method. Correlation is a statistical measure that indicates the strength and direction of the linear relationship between two variables.

2. `print(f"Correlation between Holiday and Quantity: {correlation}")`: This line prints the calculated correlation value between the 'holiday' and 'Quantity' columns.

The provided output displays the calculated correlation between the 'holiday' and 'Quantity' columns. The output value of approximately -0.078 indicates a weak negative correlation. This means that there is a very slight tendency for pallet quantities to be slightly lower on holidays compared to non-holidays, but the relationship is not strong.

  
  
Explanation :

# Day of the week analysis

df1['day\_of\_week'] = df1['MovementDate'].dt.day\_name()

avg\_quantity\_by\_day = df1.groupby('day\_of\_week')['Quantity'].mean()

avg\_quantity\_by\_day.plot(kind='bar')

plt.title('Average Quantity by Day of the Week')

plt.ylabel('Average Quantity')

plt.show()

Output:

The provided code segment performs a day-of-the-week analysis on the DataFrame 'df1' and generates a bar plot to visualize the average quantity of pallet movement for each day of the week. Here's a breakdown of the code:

1. `df1['day\_of\_week'] = df1['MovementDate'].dt.day\_name()`: This line creates a new column 'day\_of\_week' in the 'df1' DataFrame by extracting the day of the week from the 'MovementDate' column using the `.dt.day\_name()` method.

2. `avg\_quantity\_by\_day = df1.groupby('day\_of\_week')['Quantity'].mean()`: This line calculates the average quantity of pallet movement for each day of the week by grouping the 'df1' DataFrame based on the 'day\_of\_week' column and calculating the mean of the 'Quantity' column.

3. `avg\_quantity\_by\_day.plot(kind='bar')`: This code generates a bar plot using the calculated average quantity values. Each bar corresponds to a day of the week, and its height represents the average quantity of pallet movement for that day.

4. `plt.title('Average Quantity by Day of the Week')`: Sets the title of the plot to "Average Quantity by Day of the Week."

5. `plt.ylabel('Average Quantity')`: Sets the label for the y-axis as "Average Quantity."

6. `plt.show()`: This command displays the generated bar plot on the screen.

The provided output describes the result of the analysis. It states that the highest average quantity of movement is typically observed on Fridays. This information provides valuable insights into the variations in pallet movement quantities across different days of the week, aiding in scheduling and resource allocation decisions.  
  
  
  
  
explain:

import pandas as pd

# Assuming you have the 'df1' DataFrame with your data

# Convert 'MovementDate' to datetime format

df1['MovementDate'] = pd.to\_datetime(df1['MovementDate'])

# Sort the DataFrame by 'MovementDate'

df1.sort\_values('MovementDate', inplace=True)

# Calculate the rolling average with a window size of 7

df1['RollingAverage'] = df1['Quantity'].rolling(window=7, min\_periods=1).mean()

# Calculate the normalized value by dividing 'Quantity' by 'RollingAverage'

df1['NormalizedQuantity'] = df1['Quantity'] / df1['RollingAverage']

# Calculate the 2-week rolling average with a window size of 14

df1['2Week Rolling Avg'] = df1['Quantity'].rolling(window=14, min\_periods=1).mean()

# Calculate the 3-day rolling average with a window size of 3

df1['3Days Rolling Avg'] = df1['Quantity'].rolling(window=3, min\_periods=1).mean()

# Display the updated DataFrame

print(df1)  
  
Output:

| **MovementDate** | **holiday** | **Quantity** | **RollingAverage** | **NormalizedQuantity** | **2Week Rolling Avg** | **3Days Rolling Avg** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | 2019-12-18 | 0.0 | 10 | 10.000000 | 1.000000 | 10.000000 | 10.000000 |
| **1** | 2019-12-19 | 0.0 | 14 | 12.000000 | 1.166667 | 12.000000 | 12.000000 |
| **2** | 2019-12-20 | 0.0 | 44 | 22.666667 | 1.941176 | 22.666667 | 22.666667 |
| **3** | 2019-12-21 | 0.0 | 18 | 21.500000 | 0.837209 | 21.500000 | 25.333333 |
| **4** | 2019-12-23 | 0.0 | 32 | 23.600000 | 1.355932 | 23.600000 | 31.333333 |

1. `df1['MovementDate'] = pd.to\_datetime(df1['MovementDate'])`: Converts the 'MovementDate' column to datetime format using `pd.to\_datetime()`.

2. `df1.sort\_values('MovementDate', inplace=True)`: Sorts the DataFrame 'df1' in ascending order based on the 'MovementDate' column.

3. `df1['RollingAverage'] = df1['Quantity'].rolling(window=7, min\_periods=1).mean()`: Calculates a rolling average with a window size of 7 (days) for the 'Quantity' column. The `.rolling().mean()` method computes the moving average.

4. `df1['NormalizedQuantity'] = df1['Quantity'] / df1['RollingAverage']`: Calculates a normalized value by dividing the 'Quantity' column by the calculated rolling average.

5. `df1['2Week Rolling Avg'] = df1['Quantity'].rolling(window=14, min\_periods=1).mean()`: Computes a 2-week rolling average with a window size of 14 for the 'Quantity' column.

6. `df1['3Days Rolling Avg'] = df1['Quantity'].rolling(window=3, min\_periods=1).mean()`: Computes a 3-day rolling average with a window size of 3 for the 'Quantity' column.

The provided output displays the top portion of the updated DataFrame, showcasing the calculated values for each corresponding column. This process of calculating rolling averages and normalized quantities aids in analyzing trends and variations in pallet movement data over different time frames.

This code snippet operates on a DataFrame named 'df1'. It first converts the 'MovementDate' column to a datetime format and sorts the DataFrame based on this date. Subsequently, it calculates a rolling average with a window size of 7 days for the 'Quantity' column, which represents some form of quantity measurement. This rolling average smooths out short-term fluctuations.

The code then computes a 'NormalizedQuantity' column by dividing the 'Quantity' by its corresponding rolling average. This normalization enables comparison of the actual quantity with the smoothed trend, highlighting deviations.

Next, it calculates two more rolling averages: a 2-week rolling average with a window size of 14 days and a 3-day rolling average with a window size of 3 days. These additional rolling averages provide furher insights into different time scales of the data's trend and variability. Finally, the updated DataFrame is printed, showcasing the original data along with the calculated rolling averages and normalized quantities. Overall, this code processes time-series data, facilitating trend analysis and variability assssment through rolling averages and normalization techniques.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Explanation:

import pandas as pd

# Assuming you already have a DataFrame named df1

# You can replace this with your actual DataFrame

# Specify the file name

csv\_file\_name = "df1.csv"

# Save the DataFrame as a CSV file

df1.to\_csv(csv\_file\_name, index=False)

print(f"DataFrame saved as '{csv\_file\_name}'")  
  
Output:

DataFrame saved as 'df1.csv'

The provided code demonstrates how to save a DataFrame (`df1`) as a CSV file using the `pandas` library in Python. Here's a breakdown of each part of the code:

1. `csv\_file\_name = "df1.csv"`: Specifies the name of the CSV file that will be created or overwritten.

2. `df1.to\_csv(csv\_file\_name, index=False)`: Saves the DataFrame `df1` as a CSV file with the specified file name. The `index=False` argument ensures that the DataFrame index is not included in the saved CSV file.

3. `print(f"DataFrame saved as '{csv\_file\_name}'")`: Displays a message confirming that the DataFrame has been successfully saved as a CSV file with the specified name.

The provided output confirms the successful saving of the DataFrame as a CSV file and provides the name of the saved file.

After running this code, you will find a file named "df1.csv" in the same directory as your Python script or Jupyter Notebook. This CSV file will contain the data from the `df1` DataFrame, which you can use for further analysis or data sharing.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Explain:

import pandas as pd

import numpy as np

from sklearn.linear\_model import LinearRegression

import lightgbm as lgb

from statsmodels.tsa.statespace.sarimax import SARIMAX

from statsmodels.tsa.holtwinters import ExponentialSmoothing

from sklearn.metrics import mean\_absolute\_error

from sklearn.metrics import mean\_squared\_error, r2\_score

# Load your data into df1 here

# Assuming you have a DataFrame called 'df1' with columns: MovementDate, holiday, RollingAverage, NormalizedQuantity, 2Week Rolling Avg, 3Days Rolling Avg

# Convert 'MovementDate' to datetime format

df1['MovementDate'] = pd.to\_datetime(df1['MovementDate'])

# Split the data into training and testing sets

train\_size = int(0.8 \* len(df1))

train\_data = df1[:train\_size]

test\_data = df1[train\_size:]

# Separate features and target variable for training and testing

X\_train = train\_data[['holiday', 'RollingAverage', 'NormalizedQuantity', '2Week Rolling Avg', '3Days Rolling Avg']]

y\_train = train\_data['NormalizedQuantity']

X\_test = test\_data[['holiday', 'RollingAverage', 'NormalizedQuantity', '2Week Rolling Avg', '3Days Rolling Avg']]

y\_test = test\_data['NormalizedQuantity']

# Train individual models

linear\_model = LinearRegression()

linear\_model.fit(X\_train, y\_train)

sarima\_model = SARIMAX(y\_train, order=(2, 0, 2), seasonal\_order=(1, 1, 1, 7))

sarima\_results = sarima\_model.fit()

lgb\_model = lgb.LGBMRegressor()

lgb\_model.fit(X\_train, y\_train)

exp\_smooth\_model = ExponentialSmoothing(y\_train)

exp\_smooth\_results = exp\_smooth\_model.fit()

# Generate individual predictions

linear\_preds = linear\_model.predict(X\_test)

sarima\_preds = sarima\_results.predict(start=len(y\_train), end=len(y\_train) + len(y\_test) - 1, dynamic=False)

lgb\_preds = lgb\_model.predict(X\_test)

exp\_smooth\_preds = exp\_smooth\_results.forecast(steps=len(y\_test))

# Ensemble predictions using simple averaging

ensemble\_preds = (linear\_preds + sarima\_preds + lgb\_preds + exp\_smooth\_preds) / 4

# Calculate Mean Absolute Error for the ensemble

ensemble\_mae = mean\_absolute\_error(y\_test, ensemble\_preds)

print("Ensemble MAE:", ensemble\_mae)

# Calculate Mean Squared Error for the ensemble

ensemble\_mse = mean\_squared\_error(y\_test, ensemble\_preds)

# Calculate Root Mean Squared Error for the ensemble

ensemble\_rmse = mean\_squared\_error(y\_test, ensemble\_preds, squared=False)

# Calculate R-squared for the ensemble

ensemble\_r2 = r2\_score(y\_test, ensemble\_preds)

print("Ensemble MSE:", ensemble\_mse)

print("Ensemble RMSE:", ensemble\_rmse)

print("Ensemble R2:", ensemble\_r2)

Output: Ensemble MAE: 0.08852515385562998

Ensemble MSE: 0.013412231052027329

Ensemble RMSE: 0.11581118707632405

Ensemble R2: 0.827828237539315

1. Data Preparation:

- Your data is assumed to be in a DataFrame named 'df1' with specific columns like 'MovementDate', 'holiday', 'RollingAverage', etc.

- 'MovementDate' is converted to a datetime format to ensure proper time handling.

2. Data Splitting:

- The dataset is divided into two parts: a training set and a testing set.

- The training set is generally larger (80% of the data), and the testing set is smaller (20% of the data).

3. Feature and Target Separation:

- Features (input variables) and the target variable (output) are separated for both training and testing sets.

- Features include attributes like 'holiday', 'RollingAverage', etc., which are used to predict the target variable 'NormalizedQuantity'.

4. Training Individual Models:

- Four different models are trained using the training data:

- Linear Regression: Learns a linear relationship between features and the target.

- SARIMA (Seasonal Autoregressive Integrated Moving Average): Captures temporal patterns and seasonality in the data.

- LightGBM: A gradient boosting model that combines multiple decision trees.

- Exponential Smoothing: A time series forecasting technique that considers exponential decay of historical data.

5. Generating Predictions:

- Each individual model is used to generate predictions on the testing set features.

- For the SARIMA and Exponential Smoothing models, predictions are generated considering the time-based structure of the data.

6. Ensemble Prediction:

- The individual predictions from all models are averaged to create an ensemble prediction.

- This ensemble approach aims to combine the strengths of different models to improve overall prediction accuracy.

7. Evaluation Metrics:

- To measure the performance of the ensemble model, various evaluation metrics are calculated using the ensemble predictions and the actual target values from the testing set.

- Mean Absolute Error (MAE): Measures the average magnitude of prediction errors.

- Mean Squared Error (MSE): Measures the average of squared prediction errors.

- Root Mean Squared Error (RMSE): The square root of MSE, providing a measure in the original unit of the target

variable.

- R-squared (R2): Indicates the proportion of the variance in the target variable that is predictable from the features.

8. Results:

1. Ensemble MAE (Mean Absolute Error):

- Value: 0.08852515385562998

- Interpretation: On average, the difference between the predicted values and the actual values (in terms of normalized quantity) is approximately 0.0885. This is a measure of the average absolute magnitude of the errors in your predictions.

2. Ensemble MSE (Mean Squared Error):

- Value: 0.013412231052027329

- Interpretation: The average of the squared differences between the predicted values and the actual values is approximately 0.0134. These metric places more weight on larger errors, potentially penalizing outliers more than MAE does.

3. Ensemble RMSE (Root Mean Squared Error) :

- Value: 0.11581118707632405

- Interpretation: The square root of the MSE, which is approximately 0.1158. RMSE provides a measure of the average magnitude of the errors in the same unit as your target variable (normalized quantity). It's a commonly used metric for assessing the goodness of fit.

4. Ensemble R-squared (R2):

- Value: 0.827828237539315

- Interpretation: An R-squared value of 0.828 suggests that approximately 82.8% of the variance in the target variable (normalized quantity) is explained by the ensemble model's predictions. This indicates a relatively strong relationship between the features and the target variable.