**INFOSYS MILESTONE-4**

**About data:**

The dataset contains waste sensor readings collected from multiple sensors (identified by sensor\_id) over a period of time. Each record in the dataset represents a single sensor reading at a specific timestamp.

**Key Features**

* **sensor\_id:** A unique identifier for the sensor that recorded the reading.
* **timestamp:** The date and time when the sensor reading was taken.
* **waste\_type:** The type of waste detected by the sensor (e.g., recyclable, organic, non-recyclable).
* **inductive\_property:** A numerical measurement of the waste's inductive property.
* **capacitive\_property:** A numerical measurement of the waste's capacitive property.
* **moisture\_property:** A numerical measurement of the waste's moisture content.
* **infrared\_property:** A numerical measurement of the waste's infrared property.

**Purpose**

The dataset is likely used for analyzing waste characteristics and potentially predicting waste types based on sensor readings. Adjacent data (readings taken at consecutive timestamps) is significant for identifying trends, anomalies, and relationships between different waste properties over time.

**Shape and Size**

The dataset has 20,000 rows and 7 columns. This suggests a considerable amount of data collected over the given period.

**Potential Uses**

This data could be used for:

* **Waste Management:** Optimizing waste collection routes and schedules.
* **Recycling:** Identifying recyclable materials more efficiently.
* **Environmental Monitoring:** Tracking waste generation patterns.
* **Sensor Development:** Improving the accuracy and reliability of waste sensors.

**Code:**

import pandas as pd

# Load the Excel file into a pandas DataFrame

data = pd.read\_csv("waste\_sensor\_data.csv")

**Explanation:**

1. **import pandas as pd:** This line imports the pandas library, which is essential for working with data in Python. It's given the alias pd for easier use.
2. **data = pd.read\_csv("waste\_sensor\_data.csv"):** This line uses the read\_csv function from pandas to read the contents of the CSV file into a pandas DataFrame named data.

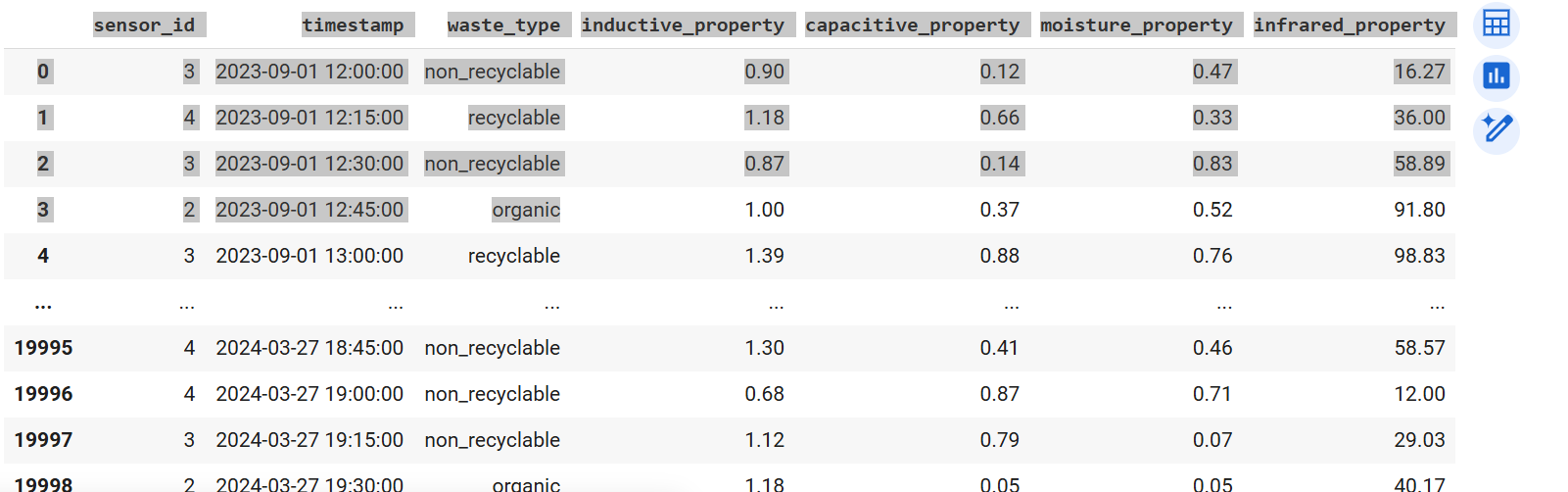
**Code:**

data

**Explanation:**

This command shows the data you loaded from the "waste\_sensor\_data.csv" file in a tabular format.

**Output:**

****

**Code:**

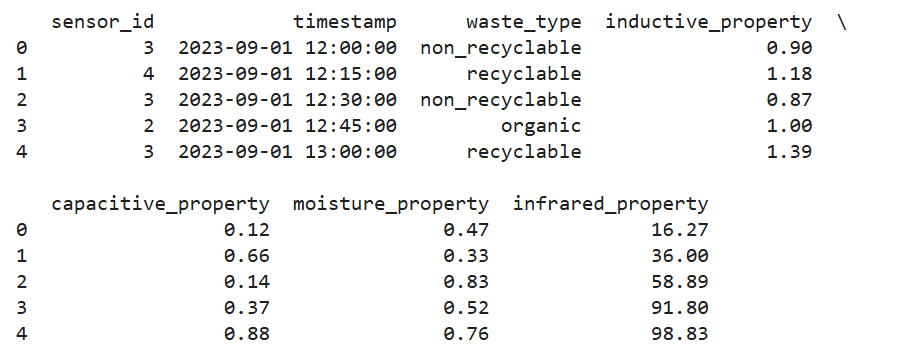
# Display the first few rows of the DataFrame

print(data.head())

**Explanation:**

1. **data.head()**: This part of the code uses the head() method on your DataFrame. The head() method is designed to extract the top rows of a DataFrame. By default, it shows the first 5 rows.

**Output:**

****

**Code:**

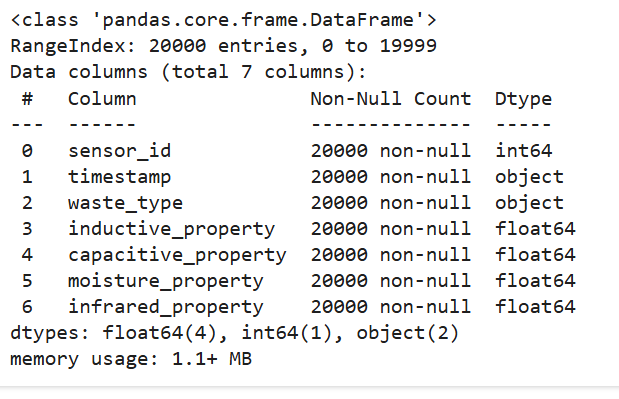
data.info()

**Explanation:**

Here's what data.info() does:

1. **Provides a concise summary of your DataFrame:** It gives you information about the columns, data types, non-null values, and memory usage of your dataset.
2. **Helps understand data structure:** You can quickly see the names of your columns, the type of data stored in each column (e.g., integer, float, object), and how many non-missing values you have for each column.
3. **Useful for data exploration:** data.info() is a fundamental tool when you start working with a new dataset, as it gives you an overview of its structure and potential issues like missing values.

**Output:**

****

**Code:**

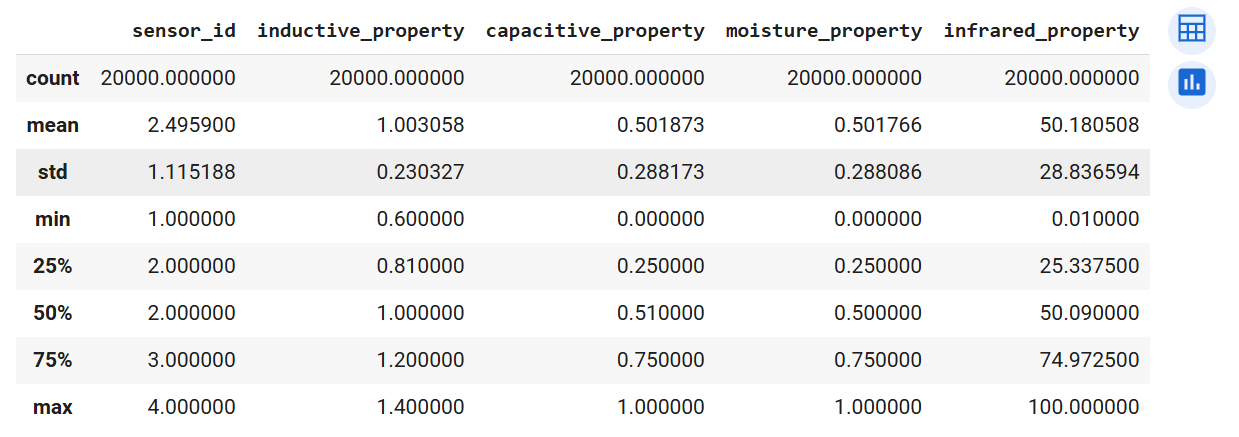
data.describe()

**Explanation:**

Here's a breakdown of what data.describe() does:

1. **Generates descriptive statistics:** It calculates and displays various summary statistics of your DataFrame's numerical columns. These statistics include:
   * count: The number of non-missing values.
   * mean: The average value.
   * std: The standard deviation (a measure of data spread).
   * min: The minimum value.
   * 25%: The 25th percentile (first quartile).
   * 50%: The 50th percentile (median).
   * 75%: The 75th percentile (third quartile).
   * max: The maximum value.
2. **Helps understand data distribution:** These statistics provide a quick overview of the central tendency, dispersion, and shape of the distribution of your numerical data.
3. **Useful for initial data exploration:** data.describe() is often one of the first steps in data analysis to get a general sense of your data's characteristics.

**Output:**

****

**Code:**

# Check data types of each column

print(data.dtypes)

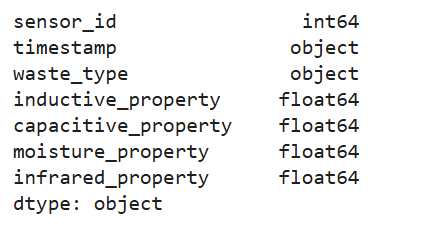
**Explanation:**

This code is used to check and display the data type of each column in your Pandas DataFrame called data.

Here's how it works:

1. **data.dtypes**: This part of the code accesses the dtypes attribute of your DataFrame. The dtypes attribute holds information about the data type of each column in the DataFrame (e.g., integer, float, object, datetime).
2. **print(...)**: This is a built-in Python function that takes whatever is inside the parentheses and displays it as output in your Colab cell. In this case, it takes the output of data.dtypes (which is a Series containing the data types of each column) and prints them out.

**Output:**

****

**Code:**

data['waste\_type'].unique()

**Explanation:**

This code is used to find and display the unique values in the 'waste\_type' column of your Pandas DataFrame called data.

Here's how it works:

1. **data['waste\_type']**: This part of the code selects the 'waste\_type' column from your DataFrame. It creates a Pandas Series containing only the values from that column.
2. **.unique()**: This method is applied to the Series and is designed to extract all the distinct (unique) values present in that column

**Output:**

array(['non\_recyclable', 'recyclable', 'organic'], dtype=object)

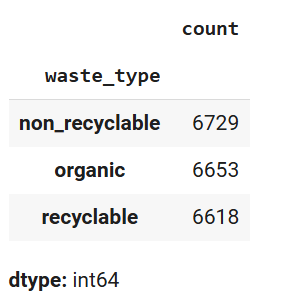
**Code:**

data['waste\_type'].value\_counts()

**Explanation:**

This code counts the occurrences of each waste type in your 'waste\_type' column. It shows you the distribution of different waste categories in your dataset. The output will be a table-like structure with waste types and their frequencies. This helps you understand the composition of your data based on waste types.

**Output:**

****

**Code:**

import matplotlib.pyplot as plt

data['waste\_type'].value\_counts().plot(kind='bar', color='skyblue')

plt.title('Distribution of Waste Types')

plt.xlabel('Waste Type')

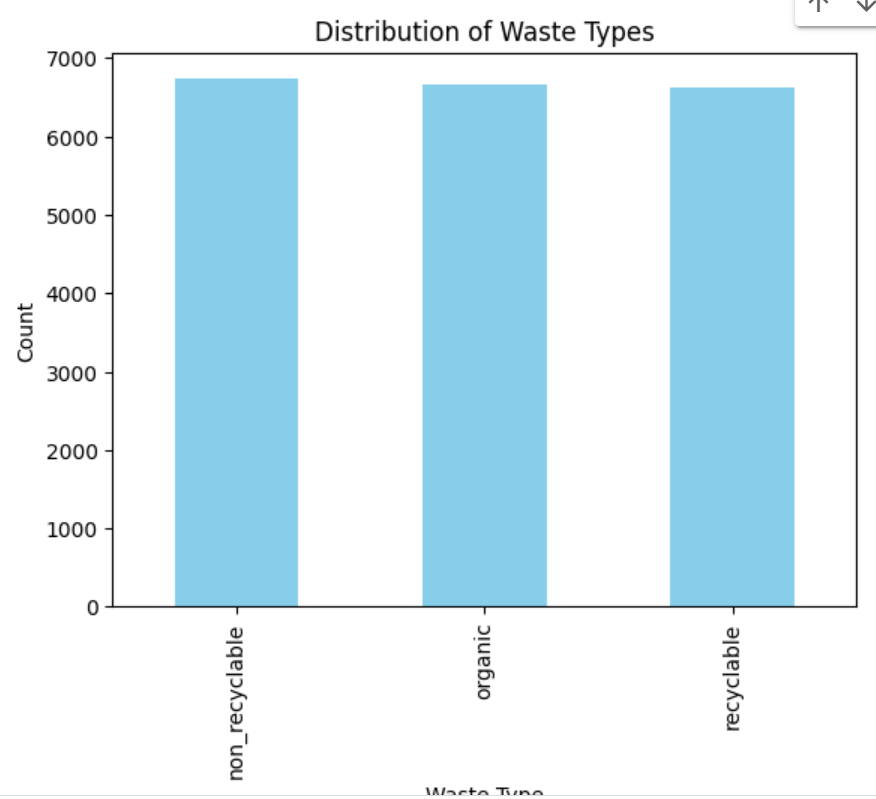
plt.ylabel('Count')

plt.show()

**Explanation:**

This code creates a bar chart showing how many times each waste type appears in your data. It's a visual representation of the data['waste\_type'].value\_counts() output.

**Output:**

****

**Code:**

sns.scatterplot(data=data, x='inductive\_property', y='capacitive\_property', hue='waste\_type')

plt.title('Inductive vs Capcitive Property')

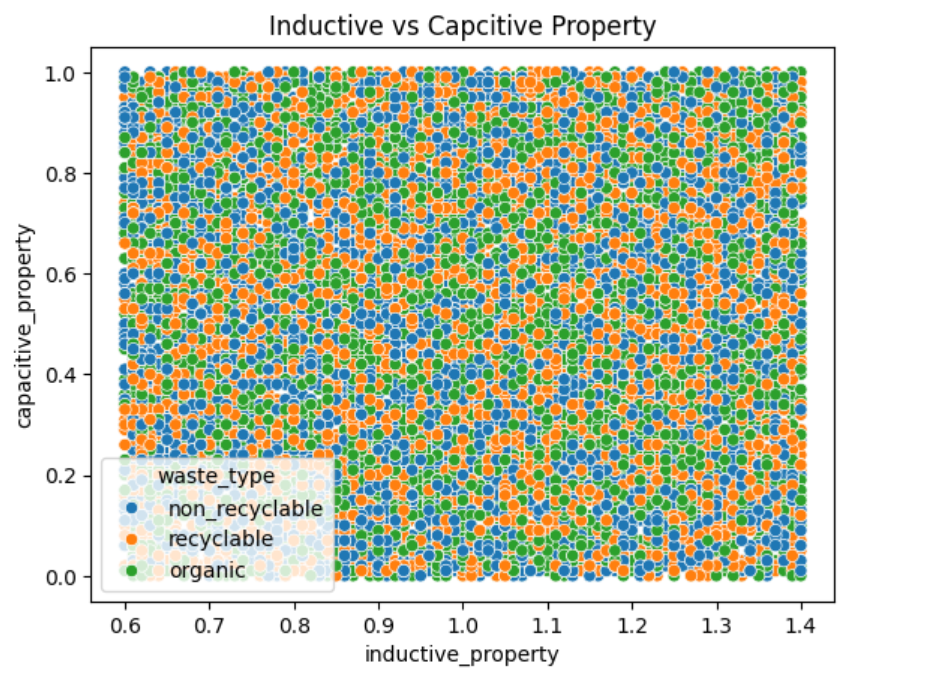
plt.show()

**Explanation:**

This code creates a scatter plot to visualize the relationship between 'inductive\_property' and 'capacitive\_property', with points colored by 'waste\_type'.

It uses the scatterplot function from the seaborn library. The hue parameter is used to color the points based on the 'waste\_type', making it easy to see if there are any patterns or clusters associated with different waste types.

**Output:**

****

**Code:**

import numpy as np

numeric\_data = data.select\_dtypes(include=np.number)

numeric\_data['waste\_type'] = data['waste\_type']

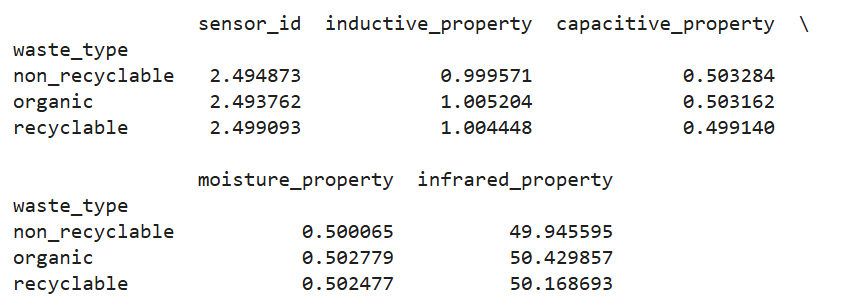
grouped\_data = numeric\_data.groupby('waste\_type').mean()

print(grouped\_data)

**Explanation:**

This code calculates the average values of numerical features for each waste type in your dataset. It first selects only the numerical columns, then adds the 'waste\_type' column back for grouping. Finally, it calculates and prints the mean values of each numerical feature for each waste type.

**Output:**

****

**Code:**

data['timestamp'] = pd.to\_datetime(data['timestamp'])

**Explanation:**

Okay, this line of code converts the 'timestamp' column in your DataFrame to datetime objects.

**Code:**

data.set\_index('timestamp').groupby('waste\_type')['sensor\_id'].resample('D').count().unstack(0).plot(figsize=(10, 6))

plt.title('Waste Type Counts Over Time')

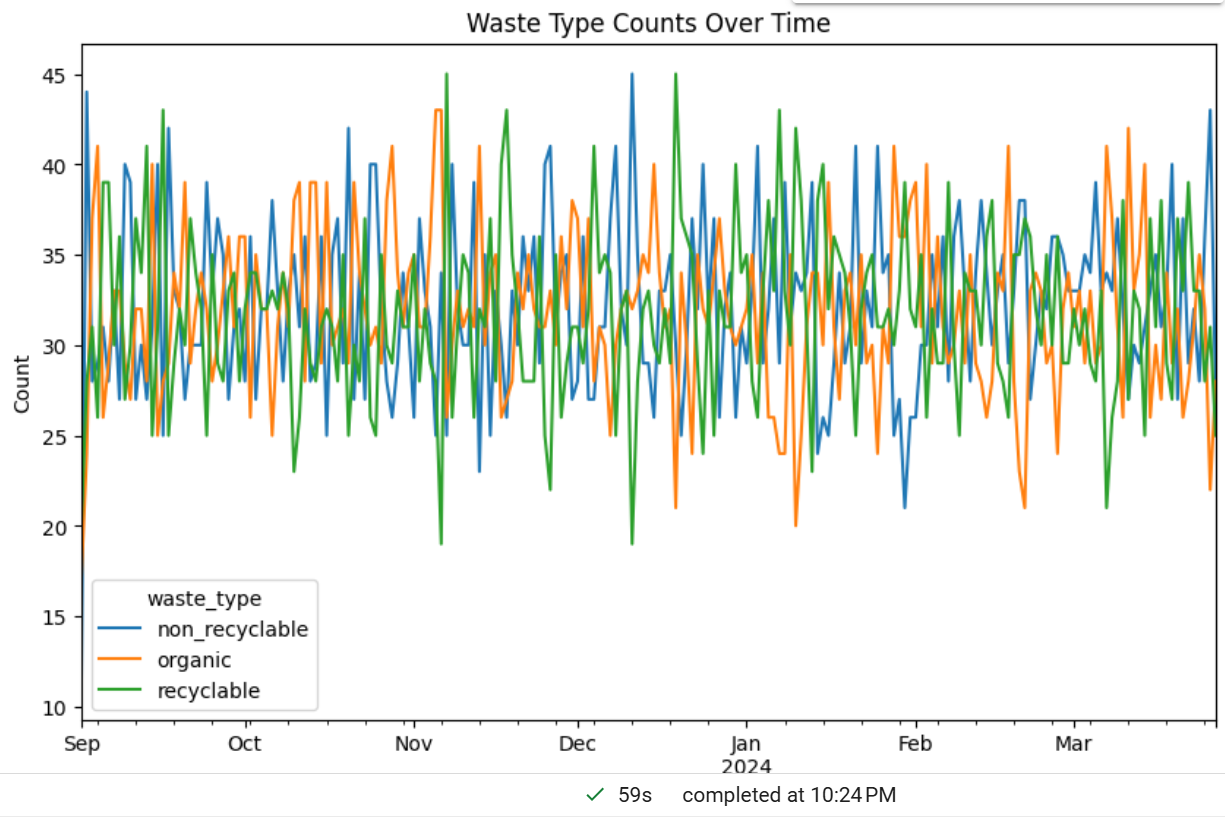
plt.ylabel('Count')

plt.show()

**Explanation:**

This code creates a line graph showing the daily counts of each waste type over time. It helps you visualize how the amounts of different waste types change daily.

**Output:**

****

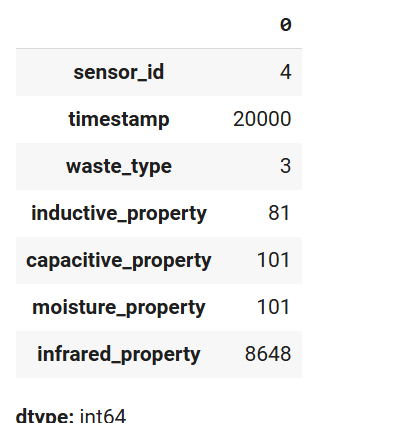
**Code:**

data.nunique()

**Explanation:**

This code calculates and displays the number of unique values in each column of your DataFrame. It helps you understand the variety of data within each column. This is useful for quickly identifying categorical features and assessing the diversity of your dataset.

**Output:**

****

**Code:**

sns.pairplot(data, hue='waste\_type', diag\_kind='kde', palette='Set2')

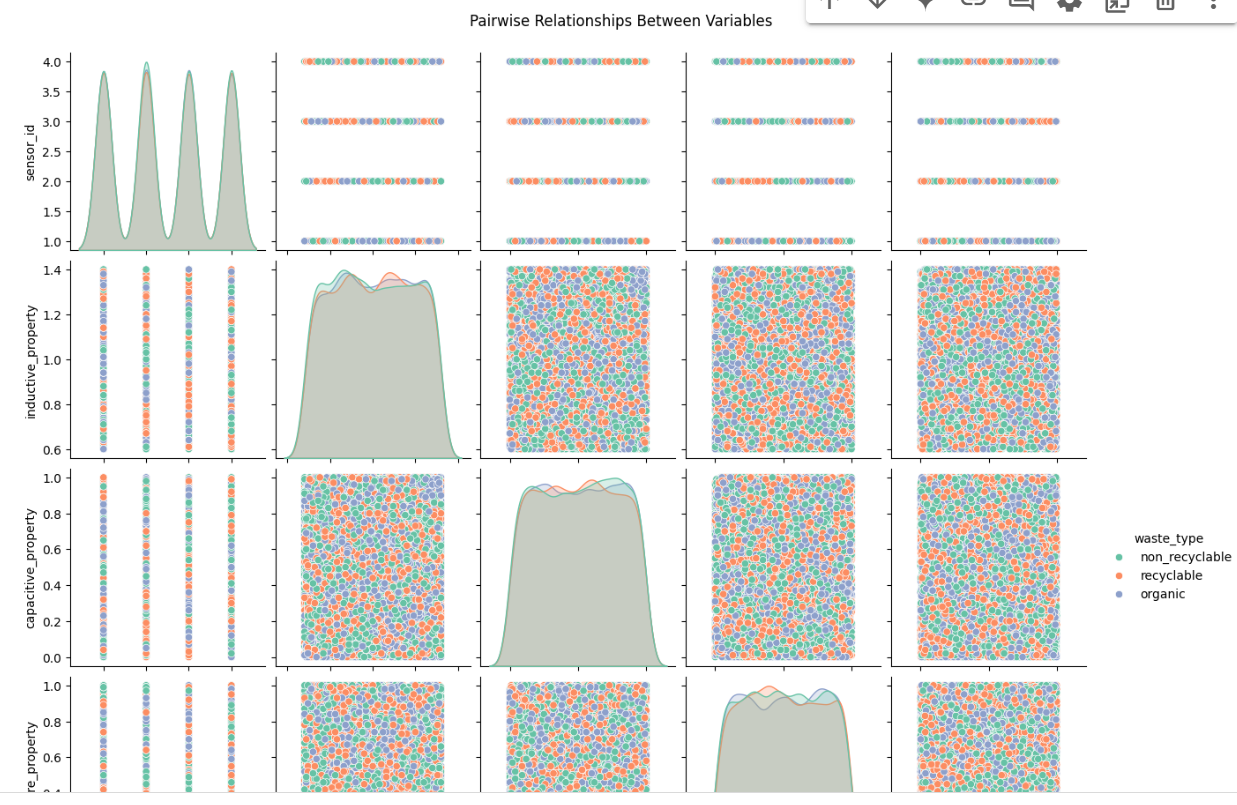
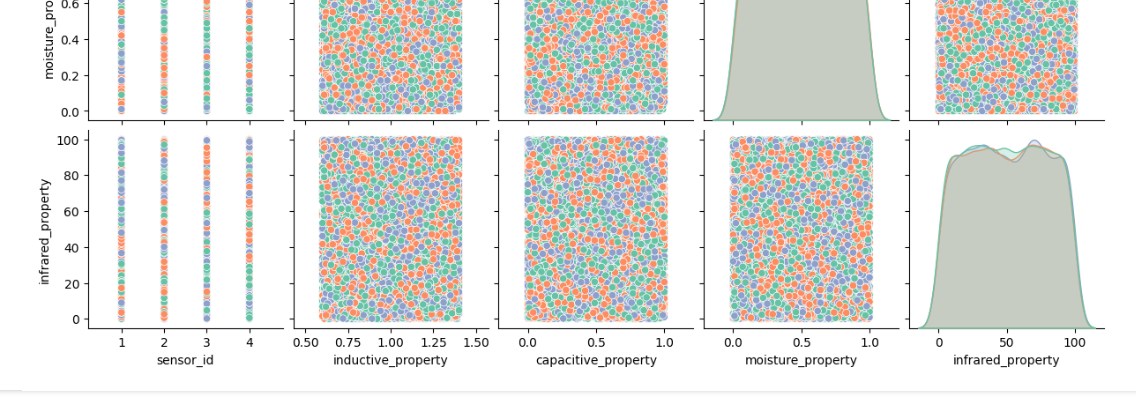
plt.suptitle("Pairwise Relationships Between Variables", y=1.02)

plt.show()

**Explanation:**

It uses the pairplot function from the seaborn library. The hue parameter colors the points based on waste type. diag\_kind='kde' displays kernel density estimations on the diagonal for a better understanding of the distribution of each feature. palette='Set2' uses a specific color palette for better visual distinction.

**Output:**

**** ****

**Code:**

sns.boxplot(data=data, x='waste\_type', y='moisture\_property', palette='pastel')

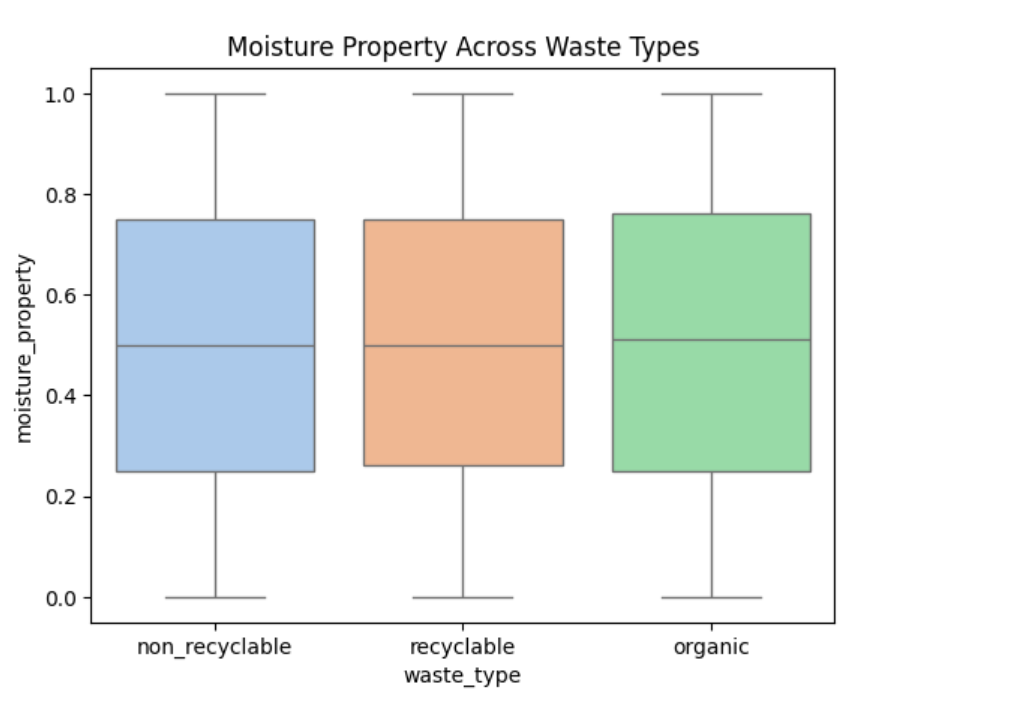
plt.title('Moisture Property Across Waste Types')

plt.show()

**Explanation:**

This code creates a box plot to visualize the distribution of 'moisture\_property' for each waste type. It uses the boxplot function from the seaborn library. The x and y parameters specify the columns to use for the plot, and palette='pastel' sets a color scheme.

**Output:**

****

**Code:**

sns.violinplot(data=data, x='waste\_type', y='infrared\_property', palette='muted')

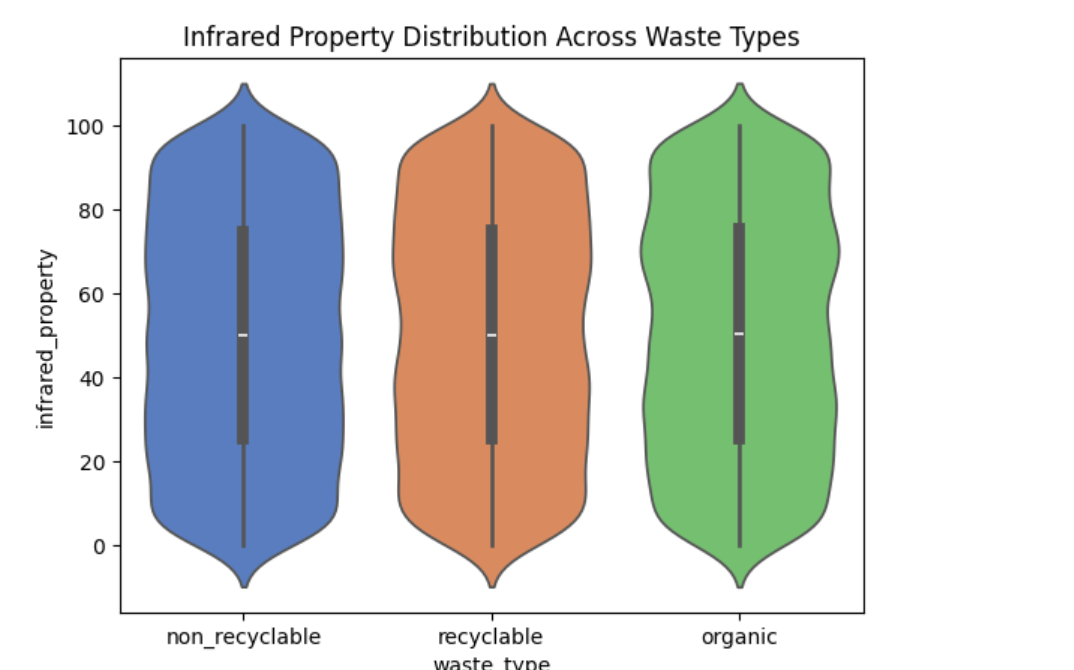
plt.title('Infrared Property Distribution Across Waste Types')

plt.show()

**Explanation:**

This code creates a violin plot to visualize the distribution of 'infrared\_property' for each waste type. It uses the violinplot function from the seaborn library. Similar to the box plot, it helps you compare the distribution of infrared property across different waste types, showing the density and spread of the data.

**Output:**

****

**Code:**

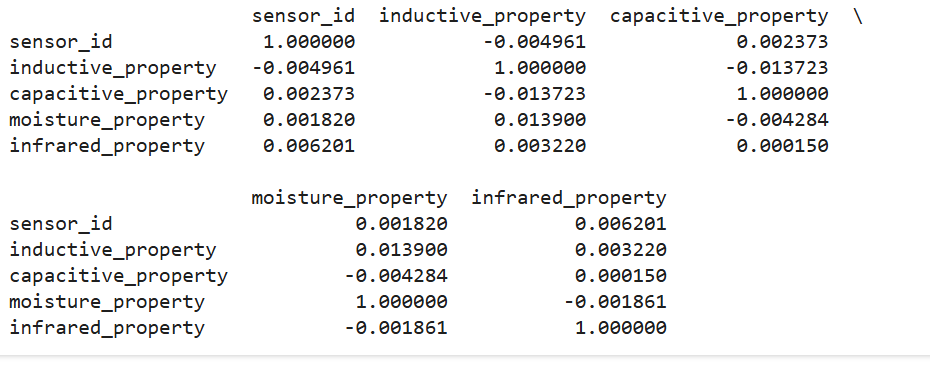
numeric\_data = data.select\_dtypes(include=np.number)

correlation\_matrix = numeric\_data.corr()

print(correlation\_matrix)

**Explanation:**

This code calculates and displays a table showing how strongly the numerical features in your data are related to each other. High positive values mean a strong positive relationship, high negative values mean a strong negative relationship, and values near 0 indicate a weak or no relationship.

**Output:** ****

**Code:**

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

sns.heatmap(correlation\_matrix, annot=True, fmt='.2f', cmap='coolwarm', linewidths=0.5)

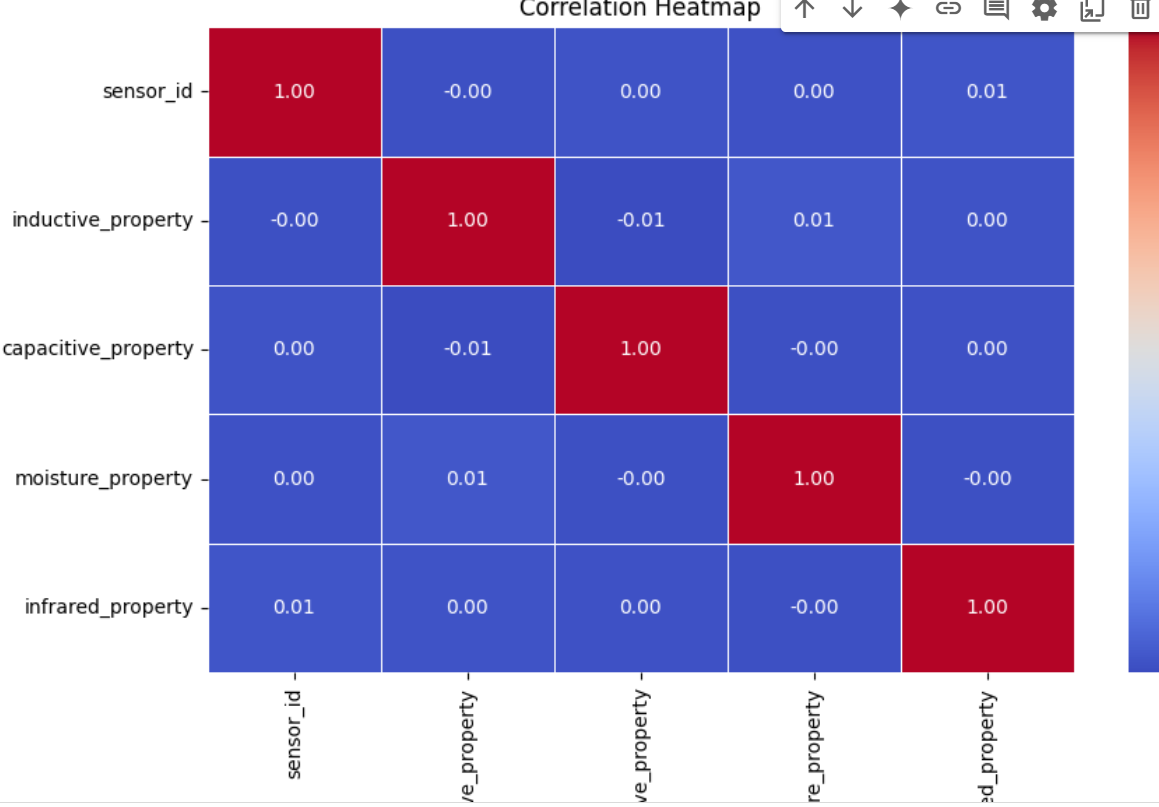
plt.title('Correlation Heatmap')

plt.show()

**Explanation:**

This code creates a colorful visualization of the correlation table, making it easier to spot relationships between numerical features. Red indicates strong positive relationships, blue indicates strong negative relationships, and lighter colors represent weaker relationships.

**Output:**

****

**Code:**

sns.scatterplot(data=data, x='capacitive\_property', y='moisture\_property', hue='waste\_type', palette='Set1')

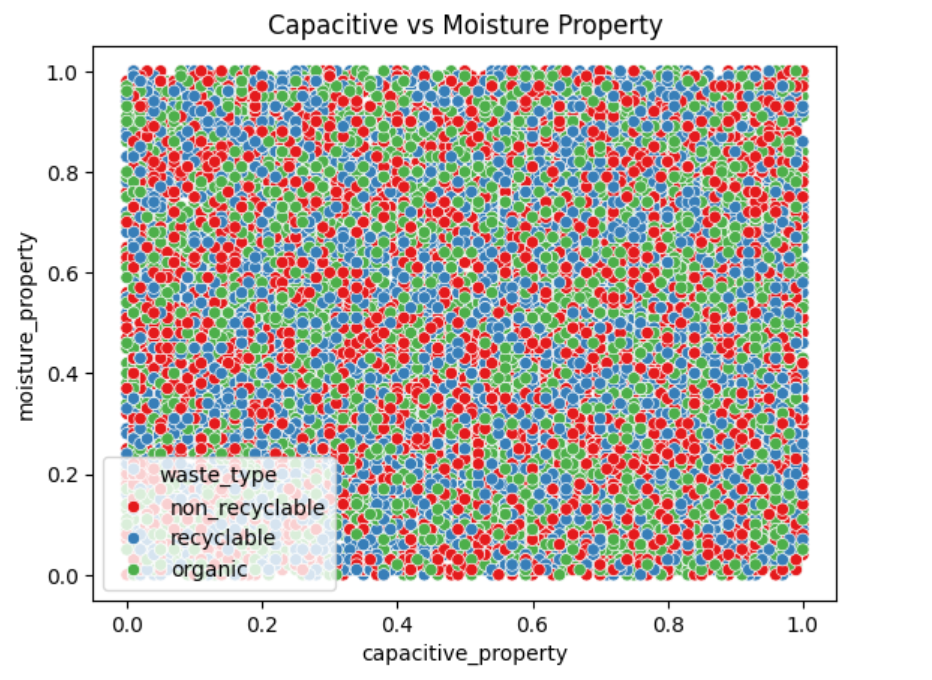
plt.title('Capacitive vs Moisture Property')

plt.show()

**Explanation:**

This code creates a scatter plot to visualize the relationship between 'capacitive\_property' and 'moisture\_property', colored by 'waste\_type'. Similar to previous scatter plots, it helps you see how these two properties relate to each other and whether different waste types exhibit distinct patterns in these properties.

**Output:**

****

**Code:**

import matplotlib.pyplot as plt

import seaborn as sns

# Example: Histogram of 'inductive\_property' for 'recyclable' waste

recyclable\_data = data[data['waste\_type'] == 'recyclable']['inductive\_property']

plt.hist(recyclable\_data, bins=10)

plt.xlabel('Inductive Property')

plt.ylabel('Frequency')

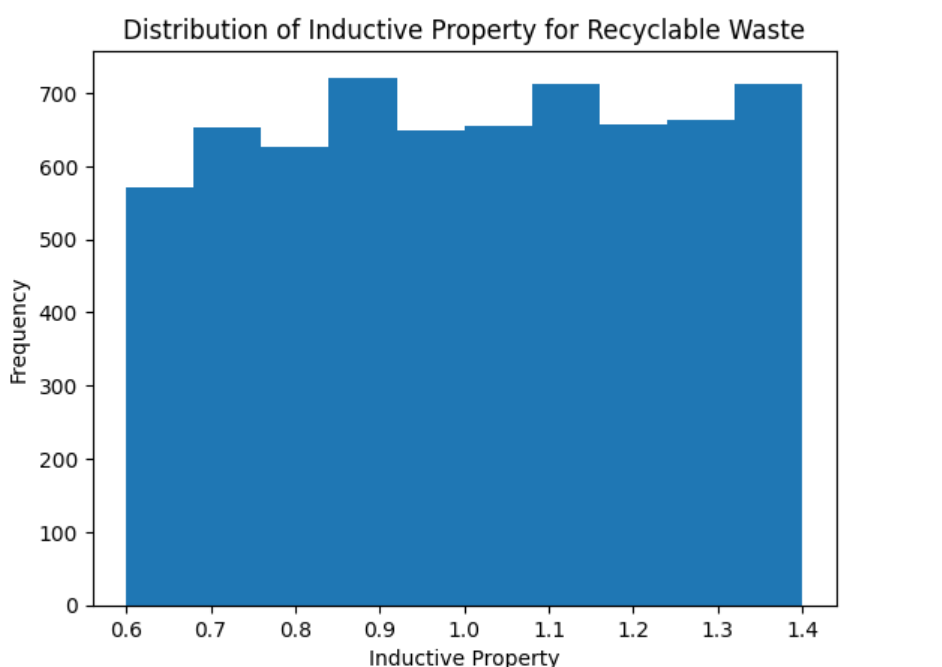
plt.title('Distribution of Inductive Property for Recyclable Waste')

plt.show()

**Explanation:**

This code creates a histogram specifically for the 'inductive\_property' of 'recyclable' waste. It filters the data to include only recyclable waste, then creates a histogram to show the distribution of the 'inductive\_property' values within that subset.

**Output:**

****

**Code:**

# Example: Line plot of 'infrared\_property' over time for a specific sensor

sensor\_data = data[data['sensor\_id'] == 3]['infrared\_property']

plt.plot(sensor\_data.index, sensor\_data.values)

plt.xlabel('Time')

plt.ylabel('Infrared Property')

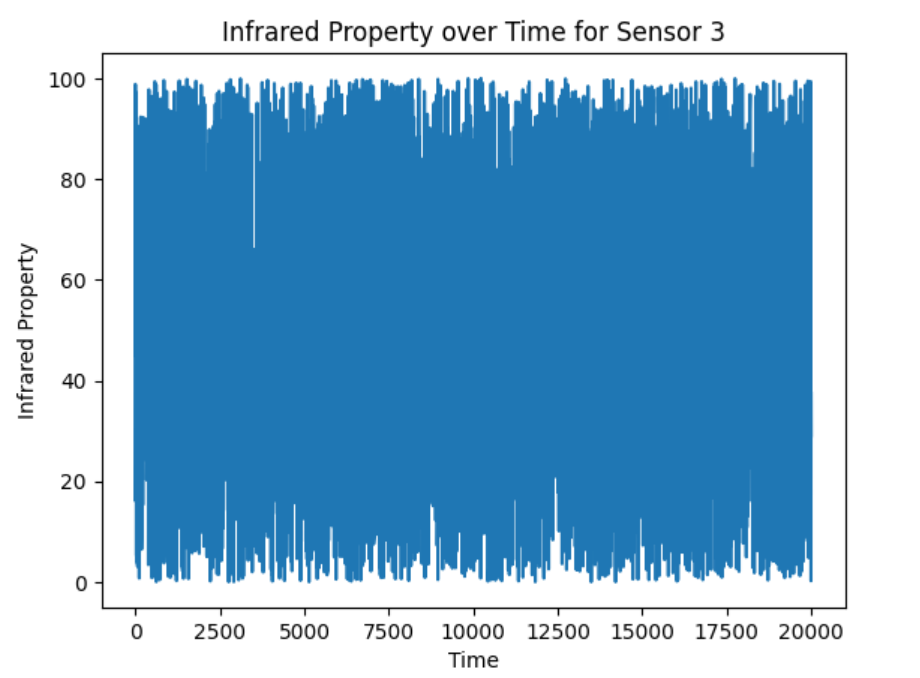
plt.title('Infrared Property over Time for Sensor 3')

plt.show()

**Explanation:**

This code creates a line plot showing how the 'infrared\_property' changes over time for a specific sensor (sensor\_id 3). It filters the data to include only readings from that sensor, then plots the 'infrared\_property' values against time

**Output:**

****

**Code:**

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

for waste\_type in data['waste\_type'].unique():

    subset = data[data['waste\_type'] == waste\_type]

    ax.scatter(subset['inductive\_property'], subset['capacitive\_property'], subset['moisture\_property'], label=waste\_type)

ax.set\_xlabel('Inductive Property')

ax.set\_ylabel('Capacitive Property')

ax.set\_zlabel('Moisture Property')

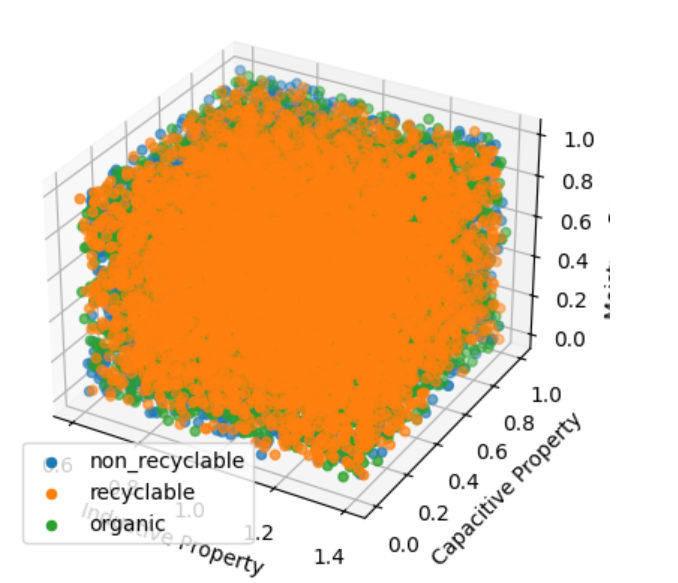
ax.legend()

plt.show()

**Explanation:**

This code creates a 3D scatter plot to visualize the relationship between 'inductive\_property', 'capacitive\_property', and 'moisture\_property', with points colored by 'waste\_type'. It uses matplotlib to create the 3D plot and iterates through each waste type to plot its data points separately.

**Output:**

****

**Code:**

import pandas as pd

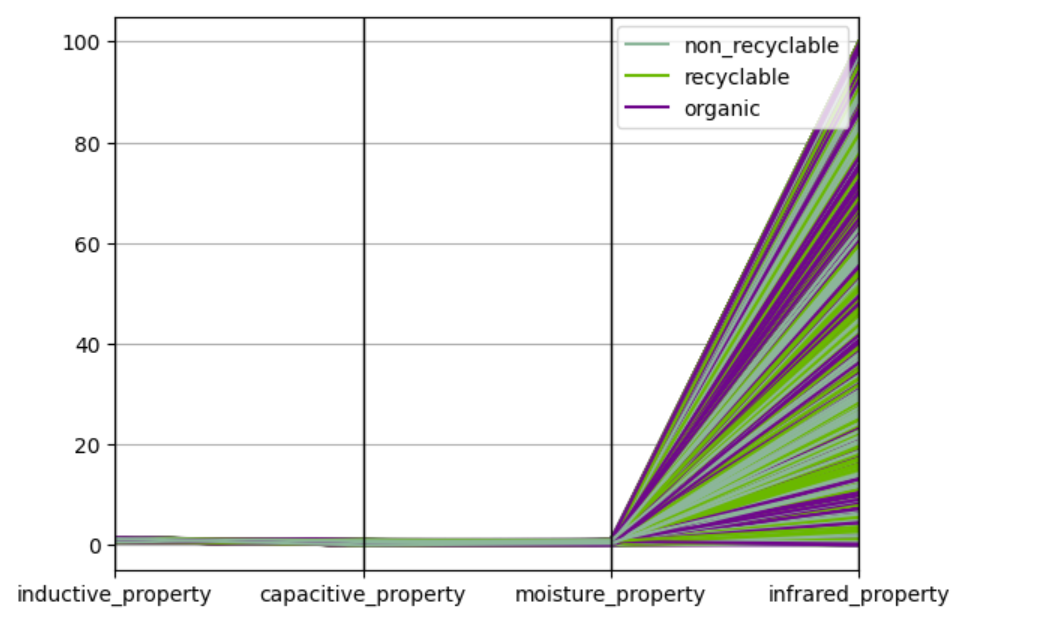
pd.plotting.parallel\_coordinates(data, 'waste\_type', cols=['inductive\_property', 'capacitive\_property', 'moisture\_property', 'infrared\_property'])

plt.show()

**Explanation:**

This code creates a parallel coordinates plot to visualize the relationship between 'waste\_type' and the properties: 'inductive\_property', 'capacitive\_property', 'moisture\_property', and 'infrared\_property'.

**Output:**

****

**Code:**

import plotly.express as px

# Assuming your DataFrame is named 'data'

fig = px.scatter\_3d(data,

                    x='inductive\_property',

                    y='capacitive\_property',

                    z='moisture\_property',

                    color='waste\_type',

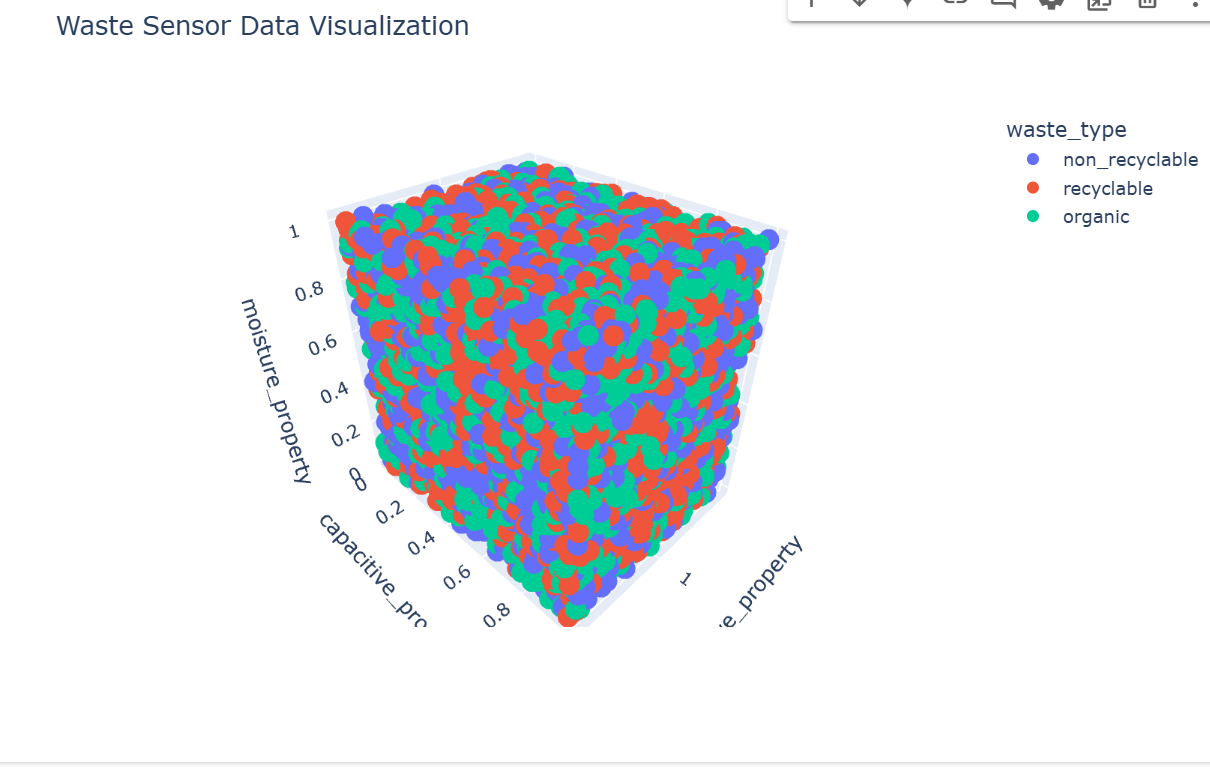
                    title="Waste Sensor Data Visualization")

fig.show()

**Explanation:**

This code creates an interactive 3D scatter plot using the plotly.express library. It's similar to the previous 3D scatter plot but provides an interactive visualization where you can rotate, zoom, and hover over data points for more details.

**Output:**

****

**Code:**

sns.boxplot(data=data, y='capacitive\_property', color='blue')

plt.title('Outliers in Capacitive Property')

plt.show()

sns.boxplot(data=data, y='inductive\_property', color='blue')

plt.title('Outliers in Inductivee Property')

plt.show()

sns.boxplot(data=data, y='moisture\_property', color='blue')

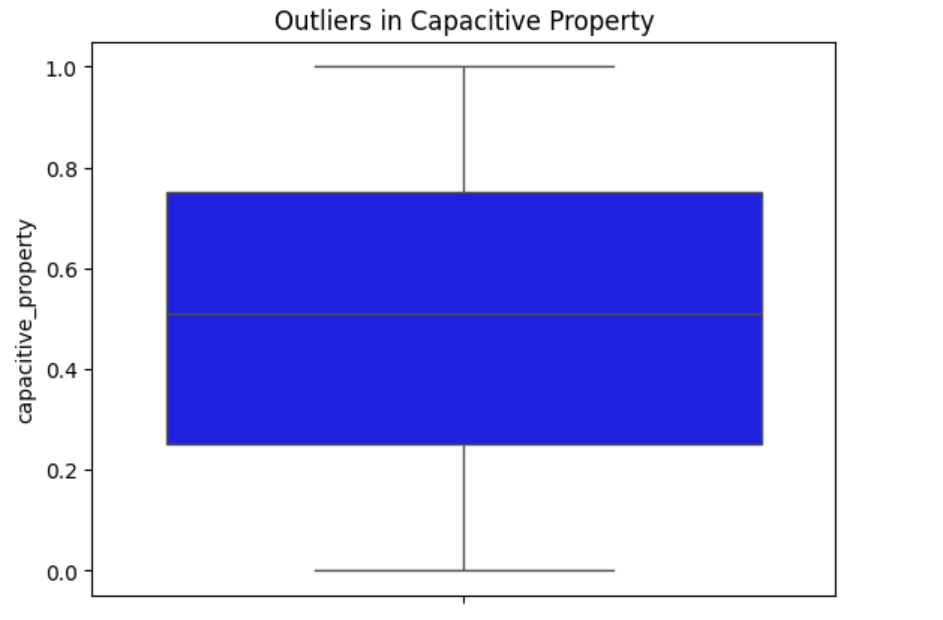
plt.title('Outliers in Moisturee Property')

plt.show()

**Explanation:**

In essence, these box plots help you identify potential outliers in each of these properties. Outliers are data points that fall significantly outside the typical range of values and are often represented as individual points beyond the whiskers of the box plot.

**Output:**

**** ****

****

**Code:**

sns.countplot(data=data, x='waste\_type', palette='Set3')

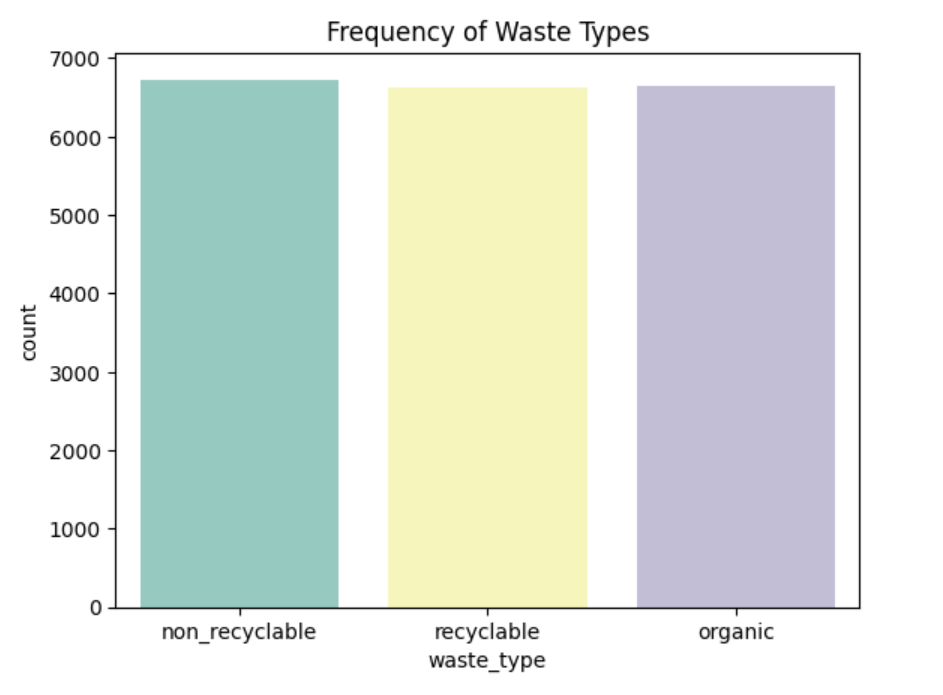
plt.title('Frequency of Waste Types')

plt.show()

**Explanation:**

This code creates a count plot to visualize the frequency of each waste type in your dataset. It uses the countplot function from the seaborn library, which is specifically designed for visualizing the counts of categorical data.

**Output:**

****

**Code:**

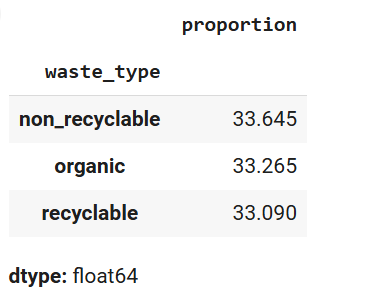
data['waste\_type'].value\_counts(normalize=True) \* 100

**Explanation:**

Here's how it works:

1. **data['waste\_type'].value\_counts():** This part gets the frequency of each waste type.
2. **normalize=True:** This argument within value\_counts() calculates the relative frequencies (proportions) instead of raw counts.
3. **\* 100:** This multiplies the proportions by 100 to express them as percentages.

**Output:**

****

**Code:**

sns.ecdfplot(data['moisture\_property'], color='green')

plt.title('Cumulative Distribution of Moisture Property')

plt.show()

sns.ecdfplot(data['inductive\_property'],color='blue')

plt.title('Cumulative Distribution of Inductive Property')

plt.show()

sns.ecdfplot(data['capacitive\_property'],color='red')

plt.title('Cumulative Distribution of Capacitive Property')

plt.show()

sns.ecdfplot(data['infrared\_property'],color='orange')

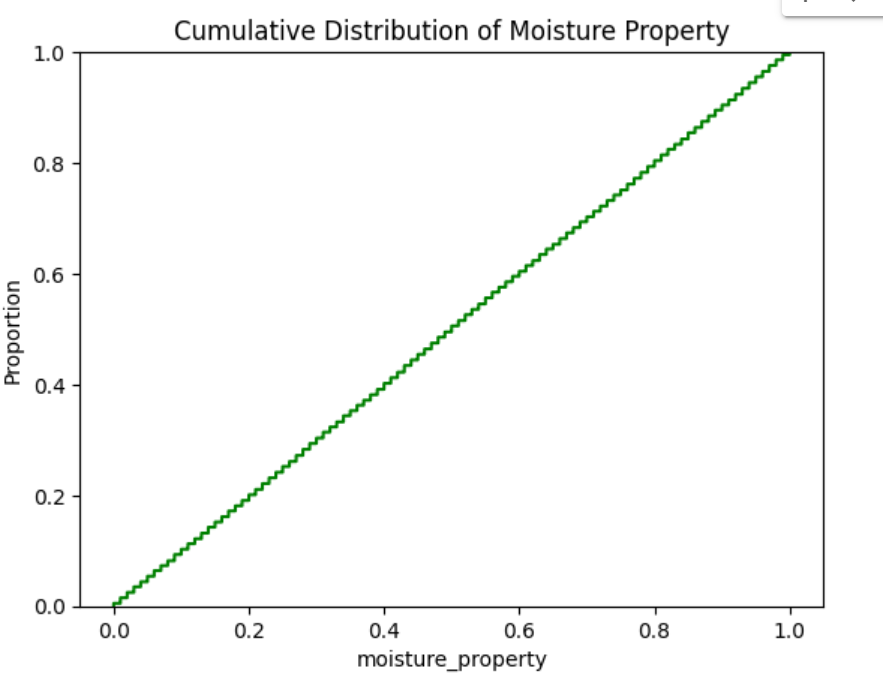
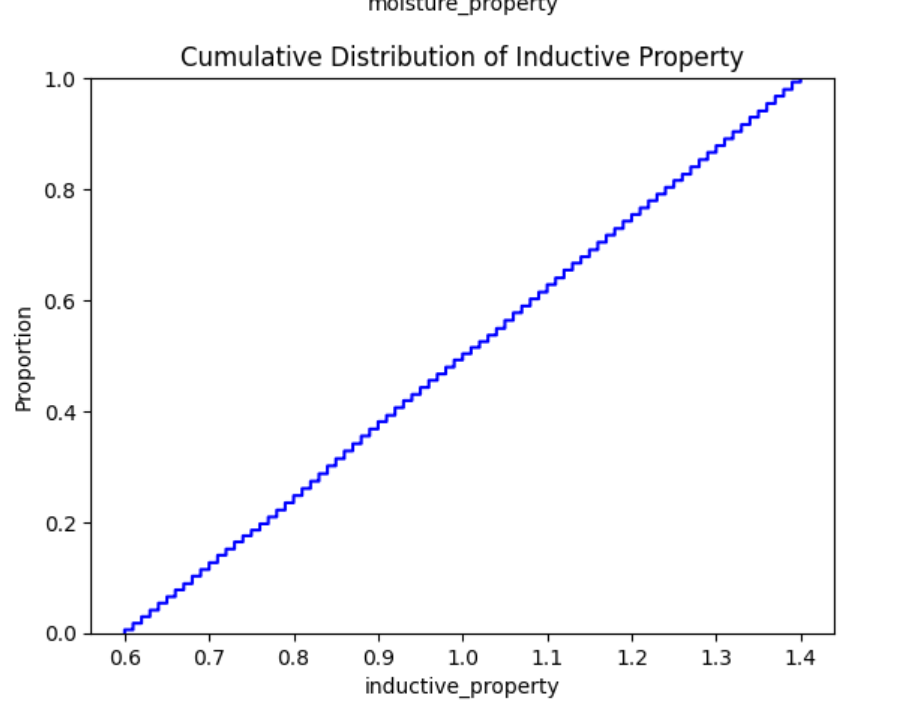
plt.title('Cumulative Distribution of Infrared Property')

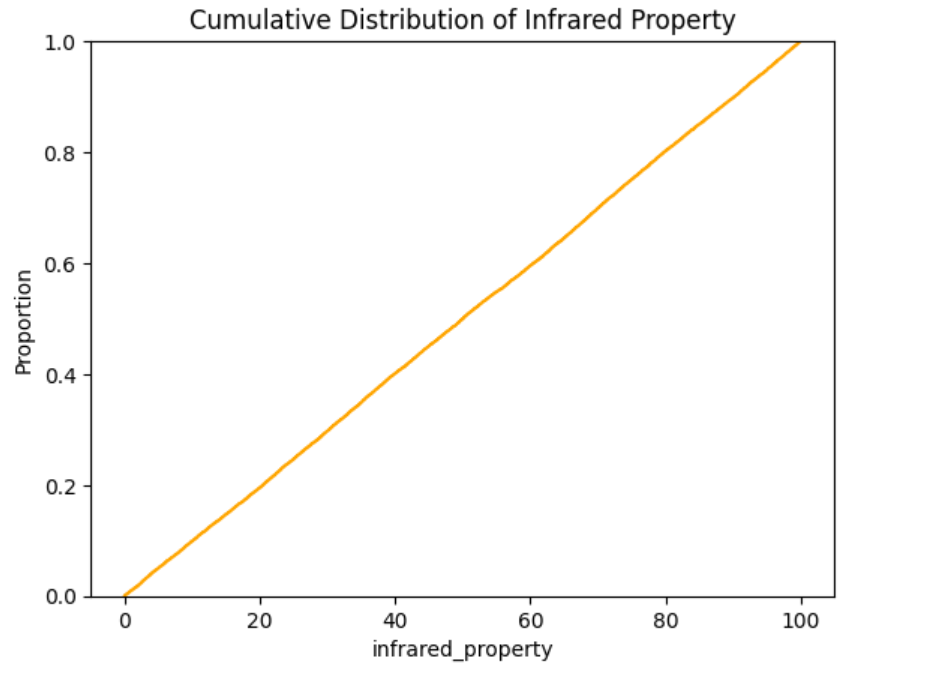
plt.show()

**Explanation:**

This code creates four separate Empirical Cumulative Distribution Function (ECDF) plots to visualize the cumulative distribution of 'moisture\_property', 'inductive\_property', 'capacitive\_property', and 'infrared\_property'. It uses the ecdfplot function from the seaborn library.

**Output:**

**** 

****

**Code:**

from statsmodels.stats.outliers\_influence import variance\_inflation\_factor

X = data.select\_dtypes(include='number')

vif = pd.DataFrame()

vif['Feature'] = X.columns

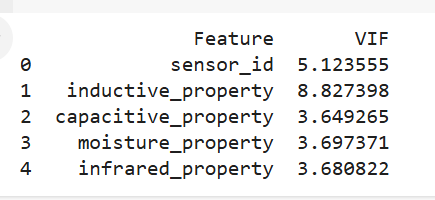
vif['VIF'] = [variance\_inflation\_factor(X.values, i) for i in range(X.shape[1])]

print(vif)

**Explanation:**

This code calculates and displays a table showing the Variance Inflation Factor (VIF) for each numerical feature. VIF helps detect if some features are too strongly related to each other, which can be problematic for certain types of analysis. Higher VIF values indicate a stronger relationship and potential issues.

**Output:**



**Code:**

data.groupby('waste\_type').agg(

    max\_moisture=('moisture\_property', 'max'),

    avg\_moisture=('moisture\_property', 'mean'),

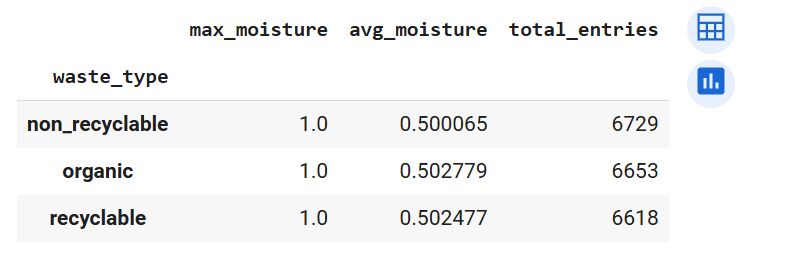
    total\_entries=('moisture\_property', 'count')

)

**Explanation:**

This code calculates and displays a table showing the maximum, average, and total count of moisture levels for each waste type. This helps you compare moisture characteristics across different waste categories.

**Output:**

****

**Code:**

import pandas as pd

from sklearn.preprocessing import LabelEncoder, StandardScaler

from sklearn.model\_selection import train\_test\_split

**Explanation:**

1. **import pandas as pd:** Imports the Pandas library for data manipulation.
2. **from sklearn.preprocessing import LabelEncoder, StandardScaler:** Imports LabelEncoder for converting categorical labels to numerical and StandardScaler for feature scaling.
3. **from sklearn.model\_selection import train\_test\_split:** Imports train\_test\_split for splitting data into training and testing sets.

**Code:**

import pandas as pd

import numpy as np

# Assuming your DataFrame is named 'data'

data['waste\_type\_organic'] = np.where(data['waste\_type'] == 'organic', 1, 0)

data['waste\_type\_recyclable'] = np.where(data['waste\_type'] == 'recyclable', 1, 0)

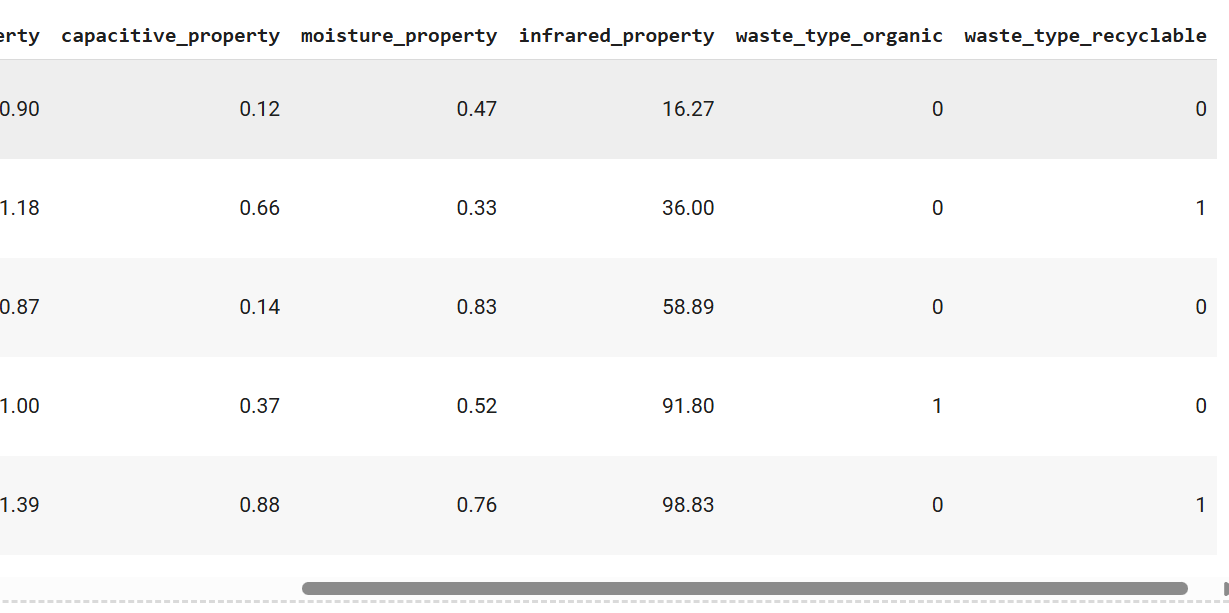
# Display the updated DataFrame

data.head()

**Explanation:**

This code creates two new columns, 'waste\_type\_organic' and 'waste\_type\_recyclable', which indicate whether the waste is organic or recyclable using 1s and 0s. This prepares the data for machine learning models.

**Output:**

****

**Code:**

# Assuming your DataFrame is named 'data'

data = data.drop('waste\_type', axis=1)

# Display the updated DataFrame

data.head()

**Explanation:**

This code is removing the column named 'waste\_type' from your DataFrame. It's like deleting a column from a spreadsheet. Then, it shows you the top few rows of the DataFrame so you can see what it looks like without that column.

**Output:**

****

**Code:**

data['timestamp'] = pd.to\_datetime(data['timestamp'])

data['hour'] = data['timestamp'].dt.hour

data['day\_of\_week'] = data['timestamp'].dt.dayofweek

data.drop(columns=['timestamp'], inplace=True)

**Explanation:**

Timestamps often contain a lot of information, but you might only need specific parts of it for your task. Extracting features like hour and day of the week can be useful for identifying patterns or trends related to time in your data. Removing the original timestamp after extracting the relevant information can help simplify your dataset and potentially improve model performance.

**Code:**

data.fillna(data.mean(), inplace=True)

**Explanation:**

This code finds any missing values (NaN) in your data and fills them with the average (mean) of each column. It does this directly in your original data, without creating a new copy. This prepares your data for analysis by dealing with missing entries.

**Code:**

scaler = StandardScaler()

numerical\_columns = ['inductive\_property', 'capacitive\_property', 'moisture\_property', 'infrared\_property', 'hour', 'day\_of\_week']

data[numerical\_columns] = scaler.fit\_transform(data[numerical\_columns])

**Explanation:**

Okay, let's break this code snippet down:

This code snippet is performing **feature scaling** on the numerical features of your dataset using StandardScaler from scikit-learn. Here's a breakdown:

1. **scaler = StandardScaler()**: Creates a StandardScaler object. This object will be used to transform the data.
2. **numerical\_columns = [...]**: Defines a list of the numerical columns in your dataset that you want to scale.
3. **data[numerical\_columns] = scaler.fit\_transform(data[numerical\_columns])**: This is where the magic happens:
   * **fit\_transform()**: This method first calculates the mean and standard deviation of each numerical feature in your data (the fit part).
   * Then, it transforms the data by subtracting the mean and dividing by the standard deviation for each feature (the transform part). This centers the data around 0 with a standard deviation of 1.
   * Finally, the scaled data is assigned back to the original numerical columns in your DataFrame (data[numerical\_columns]).

**Code:**

X = data.drop(columns=['waste\_type\_recyclable'])

y = data['waste\_type\_recyclable']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

**Explanation:**

This code is preparing your data for machine learning by creating **features (X)** and **target (y)** variables, and then splitting the data into **training and testing sets**.

**Code:**

print(f"Number of duplicate rows: {data.duplicated().sum()}")

data = data.drop\_duplicates()

print(f"Number of duplicate rows after removal: {data.duplicated().sum()}")

**Explanation:**

Okay, let's break this code down into simpler terms:

This code is designed to find and remove duplicate rows from your data:

1. **print(f"Number of duplicate rows: {data.duplicated().sum()}")**: This line first finds duplicate rows in your data using data.duplicated(), which returns True for each duplicate row and False otherwise. Then, .sum() counts the number of True values, giving you the total number of duplicate rows. Finally, it prints this number to the console.

**Output:**

Number of duplicate rows: 0

Number of duplicate rows after removal: 0

**Code:**

data.fillna(data.mean(), inplace=True)

categorical\_columns = data.select\_dtypes(include=['object']).columns

for col in categorical\_columns:

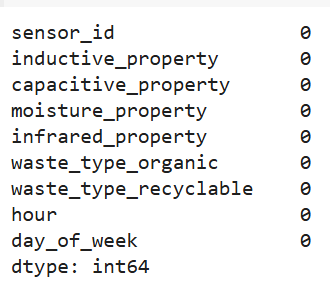
    data[col].fillna(data[col].mode()[0], inplace=True)

print(data.isnull().sum())

**Explanation:**

This code handles missing values by filling numerical columns with the mean and categorical columns with the mode. It then verifies the success of the imputation by checking for any remaining missing values.

**Output:**

****

**Code:**

for col in numerical\_columns:

    Q1 = data[col].quantile(0.25)

    Q3 = data[col].quantile(0.75)

    IQR = Q3 - Q1

    lower\_bound = Q1 - 1.5 \* IQR

    upper\_bound = Q3 + 1.5 \* IQR

    data[col] = data[col].clip(lower=lower\_bound, upper=upper\_bound)

print(data[numerical\_columns].describe())

**Explanation:**

This code identifies and handles outliers in numerical data using the IQR method. It replaces extreme values with less extreme values, making the data more robust for analysis and modeling**.**

**Output:**

****

**Code:**

print(data['waste\_type\_recyclable'].value\_counts())

from imblearn.over\_sampling import SMOTE

smote = SMOTE(random\_state=42)

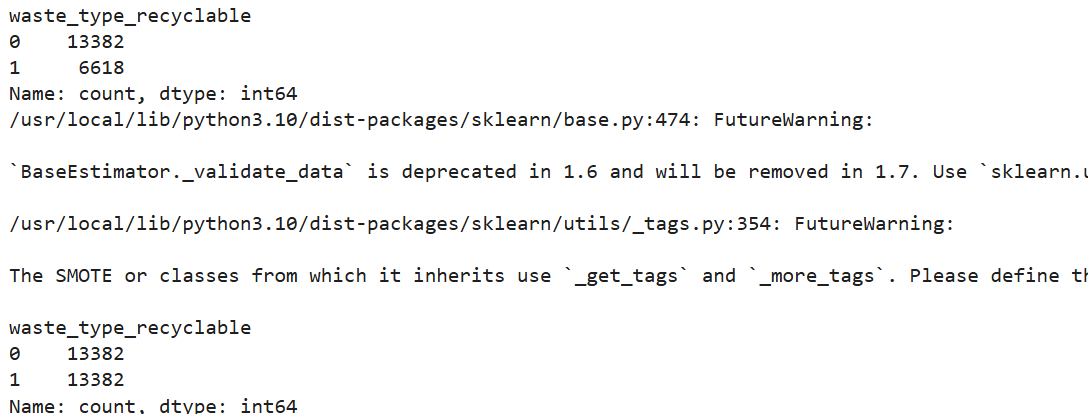
X, y = smote.fit\_resample(X, y)

print(pd.Series(y).value\_counts())

**Explanation:**

This code uses the SMOTE technique to address class imbalance by generating synthetic samples of the minority class, resulting in a more balanced dataset for training machine learning models**.**

**Output:**

****

**Code:**

data['inductive\_capacitive'] = data['inductive\_property'] \* data['capacitive\_property']

data['moisture\_infrared'] = data['moisture\_property'] \* data['infrared\_property']

**Explanation:**

This code creates two new features that represent interactions between existing features. This can potentially improve the performance of machine learning models by providing them with more information about the relationships between variables.

**Code:**

correlation\_matrix = data.corr()

high\_corr\_features = [col for col in correlation\_matrix.columns if any(abs(correlation\_matrix[col]) > 0.9) and col != 'waste\_type']

data.drop(columns=high\_corr\_features, inplace=True)

print(f"Dropped features: {high\_corr\_features}")

**Explanation:**

This code identifies and removes highly correlated features from your dataset. This can help to reduce redundancy and improve the performance and interpretability of machine learning models. By removing redundant features, you prevent multicollinearity and model overfitting, simplifying the model and improving its generalization ability.

**Code:**

from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=2, include\_bias=False)

X\_poly = poly.fit\_transform(X)

print(f"Shape of feature matrix before: {X.shape}, after: {X\_poly.shape}")

**Explanation:**

This code creates new features by combining existing ones. It uses a technique called Polynomial Features to do this. It squares features and multiplies them together. This helps machine learning models capture complex patterns. The result is a new feature matrix with more features. This can improve the model's ability to make predictions.

**Output:**

**Shape of feature matrix before: (26764, 8), after: (26764, 44)**

**Code:**

from sklearn.decomposition import PCA

pca = PCA(n\_components=0.95)

X\_reduced = pca.fit\_transform(X)

print(f"Shape before PCA: {X.shape}, after PCA: {X\_reduced.shape}")

**Explanation:**

In essence, this code uses PCA to reduce the number of features while retaining most of the important information in your data. This can help to simplify your model, reduce overfitting, and improve performance**.** By selecting principal components that explain a high percentage of variance, PCA retains the essential information while discarding noise and redundant features. This dimensionality reduction can lead to faster model training, better generalization, and easier interpretation.

**Output:**

Shape before PCA: (26764, 8), after PCA: (26764, 7)

**Code:**

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

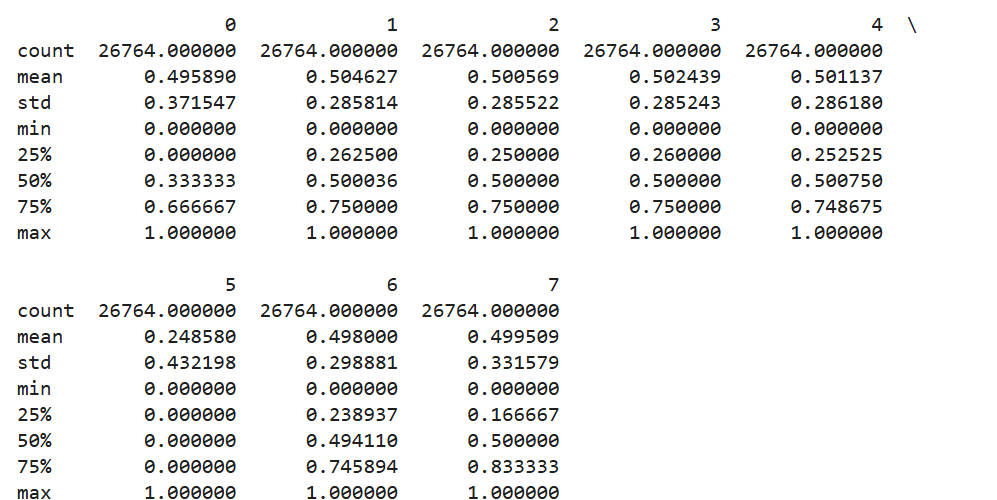
X\_scaled = scaler.fit\_transform(X)

print(pd.DataFrame(X\_scaled).describe())

**Explanation:**

In essence, this code scales your features to a specific range (0 to 1 by default). This can be beneficial for many machine learning algorithms that are sensitive to the scale of features, especially distance-based algorithms or those that use gradient descent**.** MinMaxScaler ensures that all features have the same range, preventing features with larger values from dominating the model's learning process. By scaling features, you improve the model's stability, convergence speed, and overall performance. Additionally, it can help in comparing and interpreting feature importance more effectively.

**Output:**

****

**Code:**

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42, stratify=y)

print(f"Training size: {X\_train.shape}, Testing size: {X\_test.shape}")

**Explanation:**

In simpler words, imagine you have a deck of cards. This code shuffles the deck and deals 80% of the cards to the 'training' pile and 20% to the 'testing' pile. You'll use the 'training' pile to teach your model how to play the game, and then you'll use the 'testing' pile to see how well it learned. This separation is crucial to ensure your model can generalize to new data and not just memorize the data it was trained on. The stratify argument helps in maintaining the balance of different categories (like suits in a deck of cards) in both the training and testing piles to prevent bias.

**Output:**

Training size: (21411, 8), Testing size: (5353, 8)

**Code:**

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

rf\_classifier = RandomForestClassifier(

    n\_estimators=50,

    max\_depth=None,

    random\_state=42,

    min\_samples\_split= 5,

    min\_samples\_leaf= 1,

    class\_weight='balanced'

)

rf\_classifier.fit(X\_train, y\_train)

y\_pred = rf\_classifier.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy \* 100:.2f}%")

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

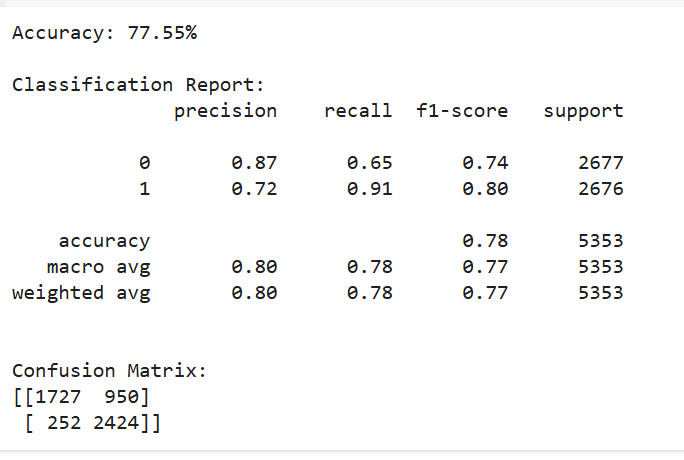
print("\nConfusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

**Explanation:**

In essence, this code builds a Random Forest model, trains it on your data, makes predictions, and then assesses how well the model performs using various metrics. The Random Forest is an ensemble learning method that combines multiple decision trees to make more robust and accurate predictions. The code uses common evaluation metrics to understand the model's overall performance, its performance on different classes, and the types of errors it makes. This comprehensive evaluation provides valuable insights into the model's strengths and weaknesses, helping you to refine and improve it.

**Output:**

****

**Code:**

importances = rf\_classifier.feature\_importances\_

feature\_importances = pd.DataFrame({

    'Feature': X.columns,

    'Importance': importances

}).sort\_values(by='Importance', ascending=False)

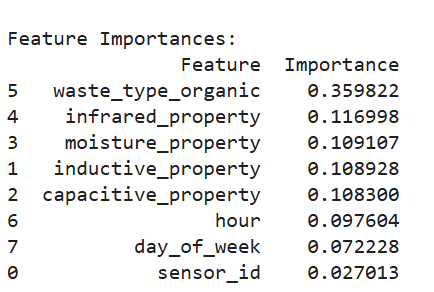
print("\nFeature Importances:")

print(feature\_importances)

**Explanation:**

In essence, this code helps you understand which features are most important for your Random Forest model in making predictions. This information can be valuable for feature selection, model interpretation, and gaining insights into the underlying relationships in your data. By identifying the most important features, you can potentially focus on those features for further analysis, data collection, or model improvement. It can also help in understanding the key drivers behind the model's predictions and in explaining the model's behavior.

**Output:**

****

**Code:**

!pip install xgboost

import xgboost as xgb

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

xgb\_classifier = xgb.XGBClassifier(

    objective='binary:logistic',

    random\_state=42,

    eval\_metric='logloss'

)

xgb\_classifier.fit(X\_train, y\_train)

y\_pred = xgb\_classifier.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy \* 100:.2f}%")

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

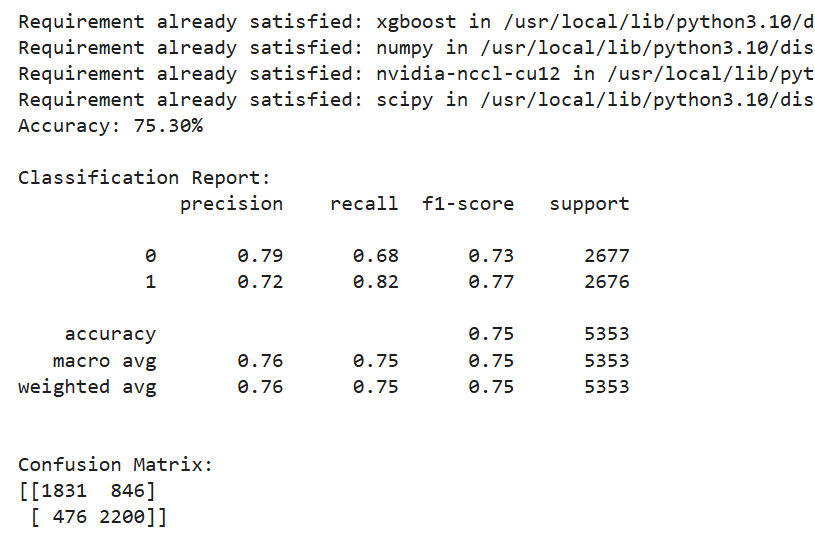
print("\nConfusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

**Explanation:**

In essence, this code installs XGBoost, builds an XGBoost model, trains it on your data, makes predictions, and then assesses how well the model performs using various metrics. XGBoost is a gradient boosting algorithm known for its high performance and efficiency in many machine learning tasks. The code follows a standard workflow for training and evaluating classification models, providing a comprehensive evaluation using accuracy, precision, recall, F1-score, and the confusion matrix. This detailed evaluation helps you understand the model's overall performance, its performance on different classes, and the types of errors it makes.

**Output:**

****

**Code:**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.metrics import accuracy\_score

from sklearn.ensemble import RandomForestClassifier

import xgboost as xgb

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.neighbors import KNeighborsClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.naive\_bayes import GaussianNB

classifiers = {

    'Logistic Regression': LogisticRegression(max\_iter=1000, random\_state=42),

    'SVM': SVC(random\_state=42),

    'K-Nearest Neighbors': KNeighborsClassifier(),

    'Decision Tree': DecisionTreeClassifier(random\_state=42),

    'Naive Bayes': GaussianNB(),

    'Random Forest': RandomForestClassifier(random\_state=42),

    'XGBoost': xgb.XGBClassifier(random\_state=42, eval\_metric='logloss')

}

accuracies = {}

for name, clf in classifiers.items():

    try:

        clf.fit(X\_train, y\_train)

        y\_pred = clf.predict(X\_test)

        accuracy = accuracy\_score(y\_test, y\_pred)

        accuracies[name] = accuracy

        print(f"{name}: Accuracy = {accuracy}")

    except Exception as e:

        print(f"Error training {name}: {e}")

        accuracies[name] = 0

plt.figure(figsize=(12, 6))

plt.bar(accuracies.keys(), accuracies.values(), color='skyblue')

plt.xlabel("Classifier")

plt.ylabel("Accuracy")

plt.title("Classifier Accuracies")

plt.xticks(rotation=45, ha='right')

plt.tight\_layout()

plt.show()

**Explanation:**

In essence, this code trains and evaluates multiple classification models on your data, stores their accuracies, and then creates a bar plot to visually compare their performance**.** This approach helps you identify the best-performing model for your specific dataset and task. By evaluating a range of models, you can make a more informed decision about which model to use for your final predictions. The bar plot provides a clear and concise way to compare the accuracies of different models, making it easier to choose the most suitable one.

**Output:**

****

**Colab link:**

**https://colab.research.google.com/drive/1ZeYQKMesHDN1T7--YusVv8ZN\_IsPqDc0?usp=sharing**