Function Documentation

Purpose

This document outlines a set of custom Python functions designed to facilitate various aspects of data preprocessing, modeling, and analysis. Each function is carefully developed to solve specific problems, improve workflow efficiency, and provide reusable code components that can be integrated into different projects. The document serves as a comprehensive guide to understanding, implementing, and utilizing these functions in practical applications.

Table of Contents

Function Overview

Below is a list of the functions described in this document, along with a brief explanation of their primary purpose:

Function Name	Description
API_Unlimited	Fetches unlimited records from a specified API.
FindMissingVal	Identifies missing values in a dataset.
HandleNullValues	Handles missing data by removing or filling them.
Combine_Dataset	Combines multiple datasets using a specified join mode.
Scale_data	Scales data using various scaling methods like 'minmax', 'zscore'.
PCA_Reduction	Performs Principal Component Analysis (PCA).
Encode_data	Encodes categorical columns using various encoding techniques.
Feature_Engineering	Applies transformations and generates new features in a dataset.
ExtractCoords	Extracts coordinates as tuples (latitude, longitude) from a DataFrame.
Min_Distance	Calculates the minimum geodesic distance between points.
Plot_Correlation_Heatmaps	Plots Pearson and Spearman correlation heatmaps for a dataset.
Optimal_K_Clusters	Determines the optimal number of clusters based on silhouette scores.
Find_Optimal_Clusters	Finds the optimal number of clusters using the elbow method.
Dimensionality_Reduction	Reduces data dimensionality using PCA or t-SNE.
Plot_Estimator	Plots decision boundaries of an estimator along with data points.
Linear_Regression	Fits a linear regression model to the data.

Polynomial_Regression	Fits a polynomial regression model and plots the fitted curve.
LogisticRegression Class	Implements a logistic regression model using gradient descent.
LinearSVM Class	Implements a linear Support Vector Machine using gradient descent.

API_Unlimited

Description:

The API_Unlimited function is designed to retrieve unlimited records from the City of Melbourne open data API. It fetches the dataset in CSV format and loads it into a pandas DataFrame. The function allows users to pass in the dataset name and API key (optional for now) to download the dataset.

Parameters:

- datasetname (string): The unique dataset identifier from the City of Melbourne open data portal.
- apikey (string, optional): The API key for authentication if required (commented out in current implementation).

Get Data No ApiKey

```python
def API_Unlimited(datasetname): # pass in dataset name and api key
    dataset_id = datasetname

    base_url = 'https://data.melbourne.vic.gov.au/api/explore/v2.1/catalog/datasets/'
    #apikey = api_key
    dataset_id = dataset_id
    format = 'csv'

    url = f'{base_url}{dataset_id}/exports/{format}'
    params = {
        'select': '*',
        'limit': -1,  # all records
        'lang': 'en',
        'timezone': 'UTC'
    }

    # GET request
    response = requests.get(url, params=params)

    if response.status_code == 200:
        # StringIO to read the CSV data
        url_content = response.content.decode('utf-8')
        datasetname = pd.read_csv(StringIO(url_content), delimiter=';')
        print(datasetname.sample(10, random_state=999)) # Test
        return datasetname
    else:
        return (print(f'Request failed with status code {response.status_code}'))
```

Returns:

- pandas.DataFrame: The function returns a DataFrame containing the dataset. If the API call fails, it prints an error message with the corresponding status code.

Edge Cases:

- If the API call fails, the function prints the error status code and returns None.
- The function assumes that the CSV is delimited by semicolons (;), so it may need adjustment if the dataset uses a different delimiter.

```python
if response.status_code == 200:
    # StringIO to read the CSV data
    url_content = response.content.decode('utf-8')
    datasetname = pd.read_csv(StringIO(url_content), delimiter=';')
    print(datasetname.sample(10, random_state=999)) # Test
    return datasetname
else:
    return (print(f'Request failed with status code {response.status_code}'))
```

Notes:

- You need the correct dataset identifier to download data from the API.
- This implementation fetches all records from the dataset using limit = -1. Adjust this value if you want to limit the number of records.

2.

FindMissingVal

Description:

The FindMissingVal function identifies and returns the number of missing (NaN) values for each feature in a pandas DataFrame. It returns a list of dictionaries, where each dictionary contains the feature name and the count of missing values.

Parameters:

- df (pandas.DataFrame): The DataFrame for which you want to check for missing values.

Returns:

- list: A list of dictionaries, where each dictionary contains:
    - Feature: The column name (string).
    - Number of Missing Values: The count of missing values (int) in that column.

Dealing with NULL Values (Finding Missing Data Count)

```
def FindMissingVal(df):
  #now lets have a array to store the feature with number of NAN values
  MissingFeaturenValues = []
  #now we check each column
  for column in df.columns:
    missingVals = np.sum(df[column].isnull()) # sum the number of NAN values into variable
    MissingFeaturenValues.append({'Feature':column ,'Number of Missing Values':missingVals}) #the array consist of dictionary with feature and its missing values
  return MissingFeaturenValues
```

Edge Cases:

- The function does not differentiate between columns that have 0 missing values and columns that have NaN values; all columns will be included in the result.
- The function assumes that missing values are represented as NaN in the DataFrame.

Notes:

- The function uses the isnull() method to identify missing values in the DataFrame.

3.

handle_null_values

Description:

The handle_null_values function processes missing (NaN) values in a specified dataset. It allows you to either remove rows containing null values or fill them with the mean, median, or mode of the column.

Parameters:

- dataset (pandas.DataFrame): The dataset (DataFrame) where missing values need to be handled.
- columns (list): A list of column names where missing values will be processed.
- action (string): The action to perform on the missing values. Options are:
    - 'remove': Removes rows with missing values in the specified columns.
    - 'mean': Fills missing values in the specified columns with the mean of the column.
    - 'median': Fills missing values in the specified columns with the median of the column.
    - 'mode': Fills missing values in the specified columns with the mode of the column.

Returns:

- pandas.DataFrame: The modified dataset with missing values either removed or filled based on the specified action.

```python
def handle_null_values(dataset, columns, action): # nested conditions
    if action == 'remove':
        modified_dataset = dataset.dropna(subset=columns)
    elif action in ['mean', 'median', 'mode']:
        for column in columns:
            if dataset[column].isnull().any():
                if action == 'mean':
                    fill_value = dataset[column].mean()
                elif action == 'median':
                    fill_value = dataset[column].median()
                elif action == 'mode':
                    fill_value = dataset[column].mode()[0]
                dataset[column] = dataset[column].fillna(fill_value)
        modified_dataset = dataset
    else:
        raise ValueError("Action must be 'remove', 'mean', 'median', or 'mode'")
    return modified_dataset
```

4.

Combine_Dataset

Description:

The Combine_Dataset function merges multiple pandas DataFrames based on common columns using a specified join mode. This function checks for common columns between datasets and combines them using the chosen merge strategy (default is outer join).

Parameters:

- datasets (list of pandas.DataFrame): A list of DataFrames to be combined.
- mode (string, optional): The type of join operation to use. Options are:
    - 'inner': Returns rows that have matching values in both datasets.
    - 'outer': Returns all rows, combining where possible and filling with NaN where no match is found (default).
    - 'left': Returns all rows from the left dataset and matching rows from the right.
    - 'right': Returns all rows from the right dataset and matching rows from the left.

Returns:

- pandas.DataFrame: The combined DataFrame based on the common columns and the specified join mode.

Data Combining / Intergration

```python
import pandas as pd

def Combine_Dataset(datasets, mode='outer'):
    # Check if no datset is given
    if not datasets:
        raise ValueError("No datasets provided for merging.")

    #We check if there are any common columns
    common_columns = set(datasets[0].columns) # making a SET
    for dataset in datasets[1:]:
        common_columns.intersection_update(dataset.columns) #Appending if we find any matching

    #Error if no common found
    if not common_columns:
        raise ValueError("No common columns available for combining the datasets. Please give datasets with common columns.")

    #Merge
    combined_dataset = datasets[0]
    for dataset in datasets[1:]:
        combined_dataset = pd.merge(combined_dataset, dataset, how=mode, on=list(common_columns))# combine with mode ( default is outer) with the common columns

    return combined_dataset
```

Edge Cases:

- If no datasets are provided, a ValueError is raised.
- If there are no common columns between the datasets, a ValueError is raised, alerting the user to provide datasets with common columns.

5.

Scale_data

Description:

The Scale_data function scales the specified columns in a pandas DataFrame using various scaling methods. It creates a scaled version of the DataFrame without modifying the original one.

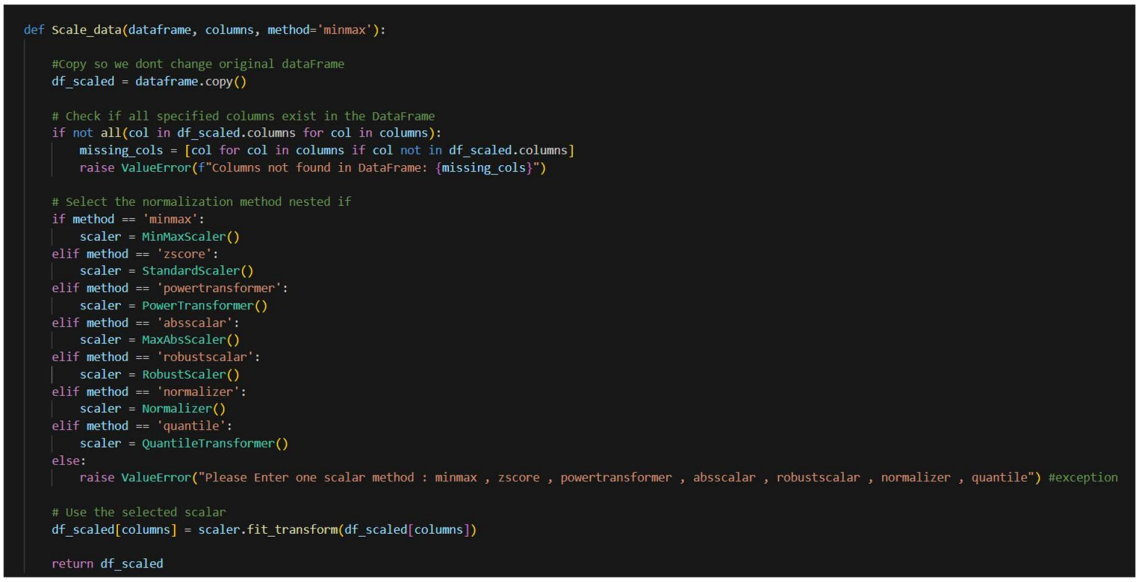
Parameters:

- dataframe (pandas.DataFrame): The input DataFrame to scale.
- columns (list): The list of columns to be scaled.
- method (string, optional): The scaling method to use. The available options are:
    - 'minmax': Scales the data to a range between 0 and 1 (default).
    - 'zscore': Standardizes the data by subtracting the mean and dividing by the standard deviation (Z-score normalization).

- 'powertransformer': Applies a power transformation to make the data more Gaussian-like.
- 'absscalar': Scales the data based on the absolute maximum value.
- 'robustscalar': Scales the data by removing the median and scaling according to the interquartile range (IQR).
- 'normalizer': Scales individual samples to unit norm.
- 'quantile': Transforms the data to follow a uniform or normal distribution.

Returns:

- pandas.DataFrame: A new DataFrame with the specified columns scaled using the selected method.



```
def Scale_data(dataframe, columns, method='minmax'):

    #Copy so we dont change original dataFrame
    df_scaled = dataframe.copy()

    # Check if all specified columns exist in the DataFrame
    if not all(col in df_scaled.columns for col in columns):
        missing_cols = [col for col in columns if col not in df_scaled.columns]
        raise ValueError(f"Columns not found in DataFrame: {missing_cols}")

    # Select the normalization method nested if
    if method == 'minmax':
        scaler = MinMaxScaler()
    elif method == 'zscore':
        scaler = StandardScaler()
    elif method == 'powertransformer':
        scaler = PowerTransformer()
    elif method == 'absscalar':
        scaler = MaxAbsScaler()
    elif method == 'robustscalar':
        scaler = RobustScaler()
    elif method == 'normalizer':
        scaler = Normalizer()
    elif method == 'quantile':
        scaler = QuantileTransformer()
    else:
        raise ValueError("Please Enter one scalar method : minmax , zscore , powertransformer , absscalar , robustscalar , normalizer , quantile") #exception

    # Use the selected scalar
    df_scaled[columns] = scaler.fit_transform(df_scaled[columns])

    return df_scaled
```

Edge Cases:

- If any column specified in columns is not found in the DataFrame, a ValueError is raised with the missing column names.
- If an invalid scaling method is provided, a ValueError is raised, prompting the user to enter one of the supported methods.

Notes:

- The function uses sklearn's scalers for different scaling techniques.

6.

PCA_Reduction

Description:

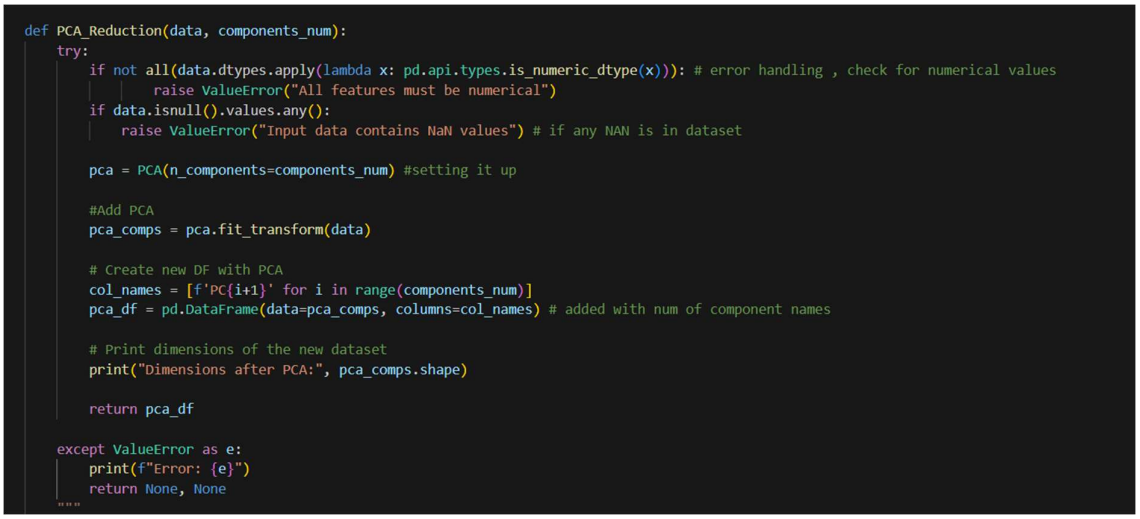
The PCA_Reduction function performs Principal Component Analysis (PCA) on a numerical dataset to reduce its dimensionality. It checks for non-numerical values and NaN values before applying PCA. The resulting components are returned in a new DataFrame, with the number of components specified by the user.

Parameters:

- data (pandas.DataFrame): The input DataFrame containing numerical features.
- components_num (int): The number of principal components to retain.

Returns:

- pandas.DataFrame: A new DataFrame containing the principal components (PC1, PC2, ...).
- None: Returned in case of an error, such as non-numerical or missing values in the data.



```
def PCA_Reduction(data, components_num):
    try:
        if not all(data.dtypes.apply(lambda x: pd.api.types.is_numeric_dtype(x))): # error handling , check for numerical values
                raise ValueError("All features must be numerical")
        if data.isnull().values.any():
            raise ValueError("Input data contains NaN values") # if any NAN is in dataset

        pca = PCA(n_components=components_num) #setting it up

        #Add PCA
        pca_comps = pca.fit_transform(data)

        # Create new DF with PCA
        col_names = [f'PC{i+1}' for i in range(components_num)]
        pca_df = pd.DataFrame(data=pca_comps, columns=col_names) # added with num of component names

        # Print dimensions of the new dataset
        print("Dimensions after PCA:", pca_comps.shape)

        return pca_df

    except ValueError as e:
        print(f"Error: {e}")
        return None, None
    """
```

Notes:

- The function only works with numerical data. It is important to handle categorical variables or missing values before passing the data to this function.
- The number of components (components_num) should be less than or equal to the number of features in the dataset.

Memory updated

Here’s the documentation for your encode_data function:

---

encode_data

Description:

The encode_data function encodes categorical columns in a dataset using different encoding methods, such as label encoding, one-hot encoding, and ordinal encoding. It allows users to specify which columns to encode and the encoding technique to apply.

Parameters:

- dataset (pandas.DataFrame): The input DataFrame containing the data to be encoded.
- encoding_type (string, optional): The type of encoding to use. Options are:
    - 'label': Uses label encoding to convert each unique category value into a numerical label.
    - 'one-hot': Uses one-hot encoding to create binary columns for each category.
    - 'ordinal': Uses ordinal encoding to assign ordered numerical values to each category. The default value is 'label'.
- columns (list, optional): A list of columns to encode. If None, all columns in the dataset will be encoded (default: None).
- target (string, optional): The target column to encode (currently not used, can be expanded for target-specific encoding).

Returns:

- pandas.DataFrame: A new DataFrame with the specified columns encoded based on the selected encoding type.

Encoding

```python
def encode_data(dataset, encoding_type='label', columns=None, target=None):
    if columns is None:
        columns = dataset.columns

    dataset_encoded = dataset.copy()

    if encoding_type == 'label':
        encoder = LabelEncoder()
        for col in columns:
            dataset_encoded[col] = encoder.fit_transform(dataset_encoded[col])

    elif encoding_type == 'one-hot':
        dataset_encoded = pd.get_dummies(dataset_encoded, columns=columns)

    elif encoding_type == 'ordinal':
        encoder = OrdinalEncoder()
        dataset_encoded[columns] = encoder.fit_transform(dataset_encoded[columns])
    else:
        raise ValueError(f"Unsupported encoding type: {encoding_type}")

    return dataset_encoded
```

Notes:

- Label encoding is useful for ordinal categorical data, where the categories have a meaningful order.
- One-hot encoding is useful when each category is independent and should be represented as a separate feature.
- Ordinal encoding is useful when categorical variables have an inherent order.

8.

Description:

The feature_engineering function applies various transformations to both numerical and categorical features of a DataFrame. It includes steps for imputation, scaling, and encoding, as well as the generation of new features such as squared and cubed terms for numerical features.

Parameters:

- df (pandas.DataFrame): The input DataFrame to be processed.
- numerical_features (list): A list of column names representing numerical features in the dataset.

- categorical_features (list): A list of column names representing categorical features in the dataset.

Returns:

- pd.DataFrame: A new DataFrame with transformed features, including the original processed features and newly generated ones (squared and cubed terms for numerical features).

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

def feature_engineering(df, numerical_features, categorical_features):
    """
    Perform feature engineering on a given dataframe.

    Parameters:
    df (pd.DataFrame): The input dataframe.
    numerical_features (list): List of numerical feature names.
    categorical_features (list): List of categorical feature names.

    Returns:
    pd.DataFrame: The transformed dataframe with engineered features.
    """

    # Define transformers for numerical and categorical features
    numerical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ])

    categorical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
        ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ])
```

```
    # Combine transformers into a single ColumnTransformer
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numerical_transformer, numerical_features),
            ('cat', categorical_transformer, categorical_features)
        ]
    )

    # Apply transformations to the data
    df_transformed = preprocessor.fit_transform(df)

    # Convert the result to a DataFrame
    df_transformed = pd.DataFrame(df_transformed, columns=[
        *numerical_features,
        *preprocessor.named_transformers_['cat']['onehot'].get_feature_names_out(categorical_features)
    ])

    # Generate new features (example: interaction terms)
    for num_feature in numerical_features:
        df_transformed[f'{num_feature}_squared'] = df_transformed[num_feature] ** 2
        df_transformed[f'{num_feature}_cubed'] = df_transformed[num_feature] ** 3

    return df_transformed

# Example usage

df = pd.DataFrame(AirQuality_df)

numerical_features = ['age', 'income']
categorical_features = ['city']

df_transformed = feature_engineering(df, numerical_features, categorical_features)
print(df_transformed)
```

Notes:

- The generated squared and cubed terms for numerical features can help capture non-linear relationships in models.
- One-hot encoding is applied to categorical features, ensuring that any unseen categories are ignored during transformation.

9.

ExtractCoords

Description:

The ExtractCoords function extracts geographic coordinates (latitude, longitude) from a DataFrame. It can handle both separate latitude and longitude columns, or a single combined column with coordinates. The function returns a list of tuples containing latitude and longitude as floating-point numbers.

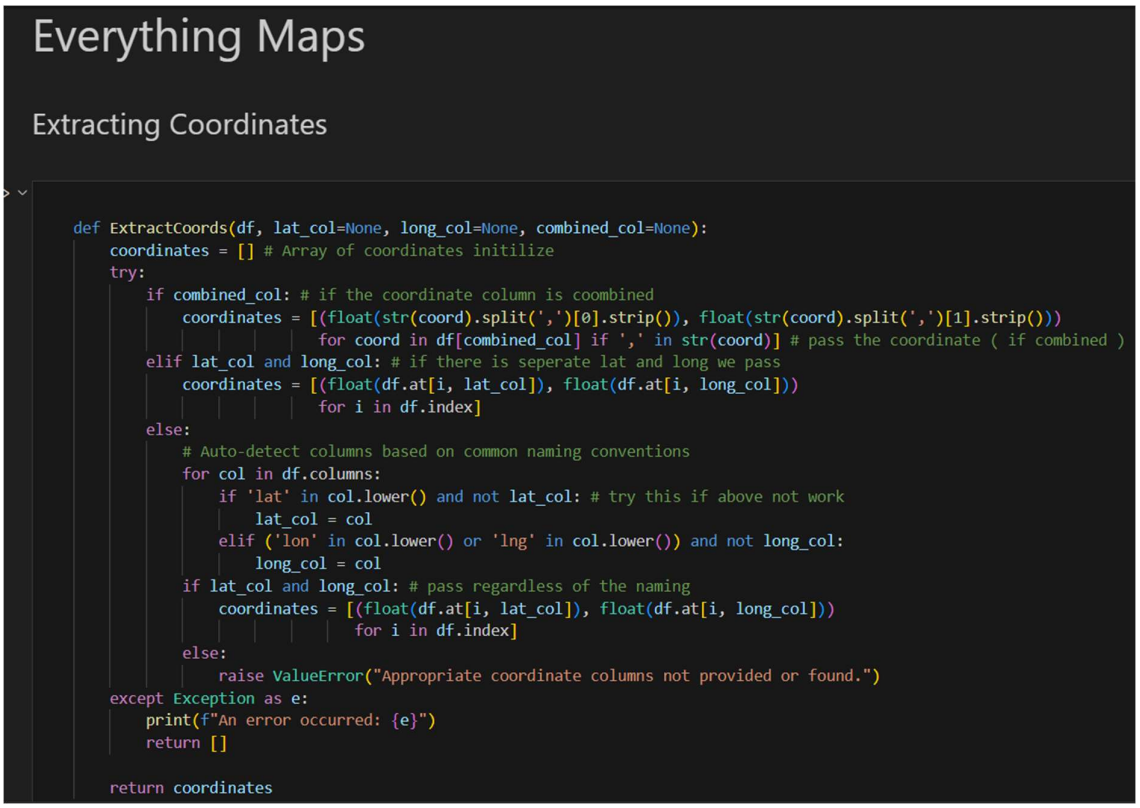
Parameters:

- df (pandas.DataFrame): The DataFrame containing the coordinate data.

- lat_col (string, optional): The column name for latitude. If not provided, the function will try to auto-detect it based on common naming conventions (e.g., columns containing 'lat').
- long_col (string, optional): The column name for longitude. If not provided, the function will try to auto-detect it based on common naming conventions (e.g., columns containing 'lon' or 'lng').
- combined_col (string, optional): The column name that contains both latitude and longitude in a combined format (e.g., "123.0, 123.5").

Returns:

- list: A list of tuples representing the extracted coordinates in the format (latitude, longitude). If the extraction fails, an empty list is returned.



10.

Memory updated

Here's the documentation for your min_distance function:

---

min_distance

Description:

The min_distance function calculates the minimum geodesic distance from a given point to a list of other points. It uses the geodesic distance in meters, making it suitable for calculating distances between geographic coordinates (latitude, longitude).

Parameters:

- point (tuple): A tuple representing the coordinates (latitude, longitude) of the point from which the distance is calculated.
- list_of_points (list of tuples): A list of tuples, each containing the coordinates (latitude, longitude) of comparison points.

Returns:

- float: The minimum geodesic distance (in meters) from the given point to the closest point in the list.

Notes:

- The function uses the geodesic distance from the geopy.distance module, which calculates the shortest path between two points on the Earth's surface.

```
def min_distance(point, list_of_points):
    return min([geodesic(point, pt).meters for pt in list_of_points]) #get min dis

#example :


row = {'lat': 40.7128, 'lon': -74.0060}
# Call the lambda function with the row as an argument
value = lambda row: min_distance((row['lat'], row['lon']), bbq_coords)
# Get the result by calling the lambda function
result = value(row)
# Print the result
print("test distance in meters :",result)

# example used in dataset :


litter_df['Nearest BBQ Distance (m)'] = litter_df.apply(lambda row: min_distance((row['lat'], row['lon']), bbq_coords), axis=1)
#creates a new column for nearest distance to a point
```

Notes:

- The function uses the geodesic distance from the geopy.distance module, which calculates the shortest path between two points on the Earth's surface.
- This function is suitable for geographic applications, such as finding the closest point of interest or calculating distances between two sets of coordinates.

plot_correlation_heatmaps

Description:

The plot_correlation_heatmaps function generates two heatmaps to visualize Pearson and Spearman correlations between variables in a dataset. It helps analyze linear and rank-based relationships between variables, respectively.

Parameters:

- data (2D numpy array or pandas.DataFrame): The input dataset for correlation analysis.
- labels (list of strings): A list of labels or column names corresponding to the variables in the dataset.
- order (list of integers, optional): A list of indices specifying the order of the variables in the heatmap for aesthetic purposes. If not provided, the default order will be used.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import spearmanr

def plot_correlation_heatmaps(data, labels, order=None):
    """
    Plots Pearson and Spearman correlation heatmaps for the given data.

    Parameters:
    - data: 2D numpy array or DataFrame containing the data to analyze.
    - labels: List of column names corresponding to the data.
    - order: List of indices specifying the order of columns for aesthetic purposes in the heatmap.

    The function creates a figure with two subplots: one for Pearson correlation and one for Spearman correlation.
    """
    if order is None:
        order = range(len(labels))  # Default order if none provided

    # Compute Pearson correlation coefficients
    R = np.corrcoef(data, rowvar=False)

    # Compute Spearman's rank correlation
    rho, pval = spearmanr(data, axis=0)

    # Create a figure with two subplots
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

    # Plot Pearson correlation heatmap
    ax1.set_title('Pearson Correlation')
    plt.sca(ax1)
    corrheatmap(R[np.ix_(order, order)], np.array(labels)[order])
```

```
    # Plot Spearman correlation heatmap
    ax2.set_title('Spearman Correlation')
    plt.sca(ax2)
    corrheatmap(rho[np.ix_(order, order)], np.array(labels)[order])

    plt.show()

ef corrheatmap(R, labels):
    """
    Helper function to draw a correlation heat map.
    """
    k = len(labels)
    plt.imshow(R, cmap='RdBu', vmin=-1, vmax=1)
    plt.xticks(np.arange(k), labels=labels, rotation=45)
    plt.yticks(np.arange(k), labels=labels)
    plt.colorbar()
    for i in range(k):
        for j in range(k):
            plt.text(j, i, f"{R[i, j]:.2f}", ha="center", va="center",
                     color="white" if np.abs(R[i, j]) > 0.5 else "black")
    plt.grid(False)
```

Example Output:

The function creates a figure with two subplots:

1. **Pearson Correlation Heatmap**: Visualizes linear correlations between variables.
2. **Spearman Correlation Heatmap**: Visualizes rank-based correlations between variables.

Notes:

- The Pearson correlation coefficient measures the strength of the linear relationship between variables.
- The Spearman correlation coefficient measures the strength of the rank-based relationship (non-linear, monotonic relationship).
- The order parameter allows reordering the variables to improve the readability or aesthetics of the heatmaps.

corrheatmap

Description:

The corrheatmap function is a helper function used to plot a heatmap of correlation values. It visualizes a matrix of correlation coefficients with annotated values and color scaling.

Parameters:

- R (2D numpy array): The correlation matrix to visualize.
- labels (list of strings): The labels for the rows and columns of the heatmap.

Example Usage:

This function is called within plot_correlation_heatmaps to display the heatmaps.

Example Output:

The function displays a heatmap with:

- Color scaling from red (negative correlation) to blue (positive correlation).
- Annotated correlation coefficients inside each cell.
- Labels for both axes based on the provided labels.

optimal_k_clusters using silhouette score

Description:

The optimal_k_clusters function determines the optimal number of clusters for K-means clustering by evaluating silhouette scores across a range of cluster numbers. It plots the silhouette scores for different values of k and returns the number of clusters with the highest silhouette score.

Parameters:

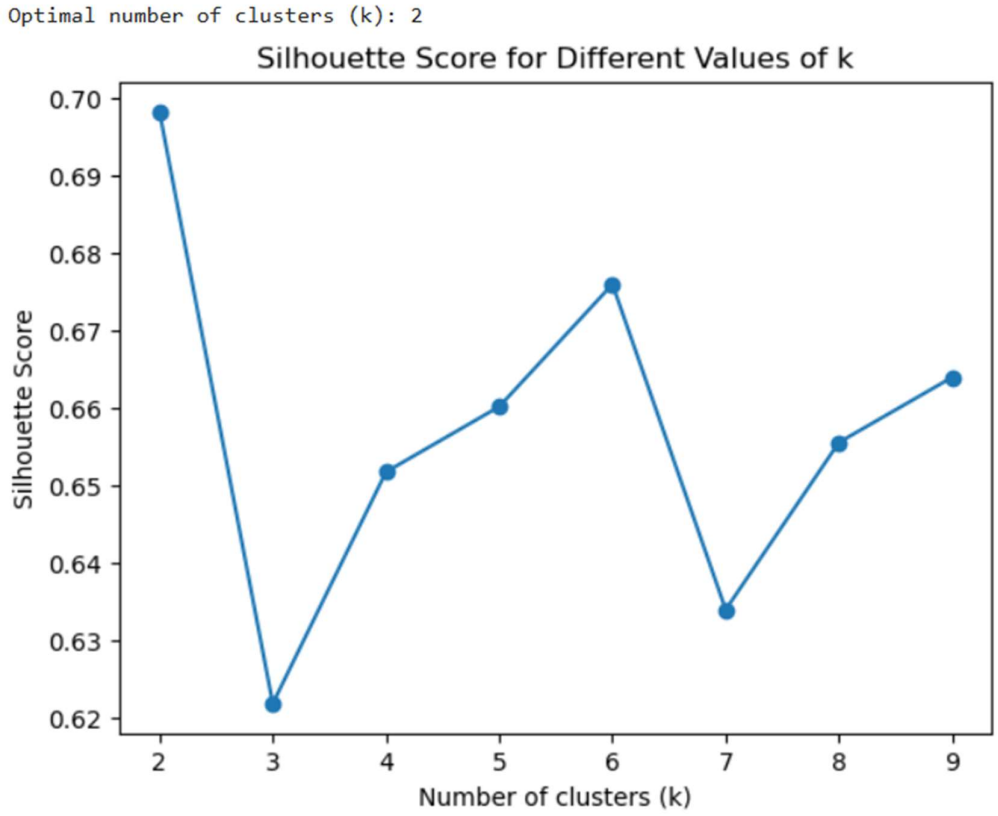
- data (numpy array or pandas.DataFrame): The dataset on which K-means clustering will be performed.
- k_range (range or list of integers): A range of values for the number of clusters (k) to test.

Returns:

- optimal_k (int): The optimal number of clusters, which corresponds to the highest silhouette score.

```python
import numpy as np
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt

def optimal_k_clusters(data, k_range):
    """
    Determines the optimal number of clusters for K-means clustering based on silhouette scores.

    Parameters:
    - data: The dataset on which clustering is to be performed.
    - k_range: A range of k values to test. Typically, this is a range object.

    Returns:
    - optimal_k: The optimal number of clusters with the highest silhouette score.
    - Plots the silhouette scores for each k in k_range.
    """
    # List to store silhouette scores for each value of k
    silh_scores = []

    # Iterate over each value of k in the range provided
    for k in k_range:
        # Fit KMeans clustering model to the data with 'k' clusters
        kmeans = KMeans(n_clusters=k, n_init=10) # n_init=10 to ensure consistency across initializations
        cluster_labels = kmeans.fit_predict(data)

        # Calculate the silhouette score for the current number of clusters
        silhouette_avg = silhouette_score(data, cluster_labels)
        silh_scores.append(silhouette_avg)

    # Determine the value of k that has the maximum silhouette score
    optimal_k = k_range[np.argmax(silh_scores)]
    print("Optimal number of clusters (k):", optimal_k)
```

```python
    # Plot the silhouette scores against the number of clusters
    plt.figure(figsize=(10, 6))
    plt.plot(k_range, silh_scores, marker='o')
    plt.xlabel('Number of clusters (k)')
    plt.ylabel('Silhouette Score')
    plt.title('Silhouette Score for Different Values of k')
    plt.grid(True)
    plt.show()

    return optimal_k
```



Plot:

- **X-axis**: Number of clusters (k).
- **Y-axis**: Silhouette Score.
- **Marker**: Indicates the silhouette score for each value of k.

Notes:

- **Silhouette Score**: A metric used to measure how similar an object is to its own cluster compared to other clusters. A higher score indicates better-defined clusters

find_optimal_clusters using elbow method

Description:

The find_optimal_clusters function determines the optimal number of clusters for K-means clustering using the elbow method. It visualizes the elbow plot to help identify the point at which increasing the number of clusters yields diminishing returns in terms

of reduced distortion (sum of squared distances from points to their assigned cluster center).

Parameters:

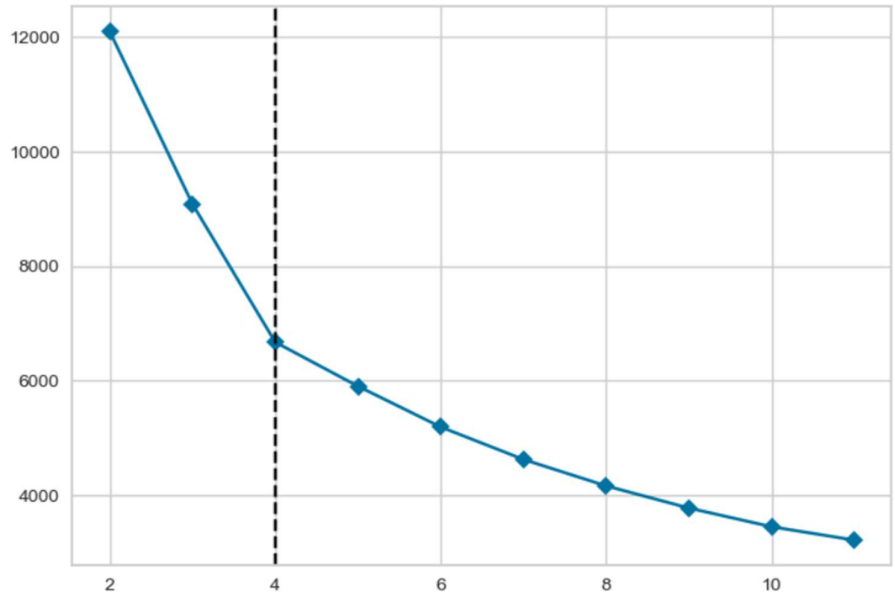
- data (numpy array or pandas.DataFrame): The dataset on which K-means clustering will be performed. It is recommended that the data be preprocessed, such as being PCA-transformed.
- k_range (tuple of integers, optional): A range specifying the minimum and maximum number of clusters (k) to test. Default is (2, 12).

Returns:

- The function generates and displays an elbow plot, showing the distortion for each value of k in the specified range. The optimal number of clusters is identified by the "elbow point," where the distortion starts to decrease more slowly.

finding the optimal K using the elbow method

```python
from sklearn.cluster import KMeans
from yellowbrick.cluster import KElbowVisualizer

def find_optimal_clusters(data, k_range=(2, 12)):
    """
    Determines the optimal number of clusters for K-means clustering using the elbow method and plots the results.

    Parameters:
    - data: The dataset on which clustering is to be performed, typically preprocessed (e.g., PCA-transformed).
    - k_range: A tuple indicating the range of k values to test (inclusive). Default is (2, 12).

    Returns:
    - Plots the elbow plot showing the distortion for each k, helping to identify the optimal number of clusters.
    """
    # Initialize the KMeans model with a fixed number of initializations to avoid random seed variability
    model = KMeans(n_init=10)

    # Initialize the KElbowVisualizer with the KMeans model, specifying the range of k and the metric 'distortion'
    visualizer = KElbowVisualizer(
        model, k=k_range, metric='distortion', timings=False
    )

    # Fit the visualizer to the data
    visualizer.fit(data)

    # Finalize and render the figure
    visualizer.show()

# Example of how to use the function
```



The number of clusters here from our graph is 4

dimensionality_reduction

Description:

The dimensionality_reduction function applies dimensionality reduction techniques such as Principal Component Analysis (PCA) or t-Distributed Stochastic Neighbor Embedding (t-SNE) to reduce the number of dimensions in a dataset. This is particularly useful for visualizing high-dimensional data or improving the performance of machine learning models.

Parameters:

- X (numpy array or pandas.DataFrame): The input data to be reduced, typically a 2D array where each row is a data point and each column is a feature.
- method (string, optional): The method to use for dimensionality reduction. Options are:
    - 'pca': Principal Component Analysis (PCA).
    - 'tsne': t-SNE for nonlinear dimensionality reduction. Default is 'pca'.

- n_components (int, optional): The number of dimensions to reduce the data to. Default is 2.
- kwargs (optional): Additional arguments to be passed to the selected dimensionality reduction method (PCA or t-SNE).

Returns:

- X_reduced (numpy array): The reduced dataset, where each data point has been projected into a lower-dimensional space of n_components dimensions.

pca and TSNE function

```python
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
import numpy as np

def dimensionality_reduction(X, method='pca', n_components=2, **kwargs):
    """
    Applies PCA or t-SNE to reduce the dimensionality of the input data.

    Parameters:
    - X: np.array or pd.DataFrame
        The input data to be reduced.
    - method: str
        The method to use for dimensionality reduction. Options are 'pca' or 'tsne'.
    - n_components: int
        The number of dimensions to reduce to. Default is 2.
    - kwargs: dict
        Additional arguments to be passed to the selected method.

    Returns:
    - X_reduced: np.array
        The reduced dataset.
    """
    if method == 'pca':
        reducer = PCA(n_components=n_components, **kwargs)
    elif method == 'tsne':
        reducer = TSNE(n_components=n_components, **kwargs)
    else:
        raise ValueError(f"Unsupported reduction method: {method}")

    X_reduced = reducer.fit_transform(X)
    return X_reduced
```

encode_data

Description:

The encode_data function encodes categorical features in a DataFrame using different encoding techniques. It supports label encoding, one-hot encoding, ordinal encoding, and target encoding based on the specified encoding type.

Parameters:

- X (pandas.DataFrame): The input DataFrame containing features to be encoded.
- encoding_type (string, optional): The encoding method to use. Options include:
    - 'label': Converts each unique category into an integer.
    - 'one-hot': Creates binary columns for each category.
    - 'ordinal': Assigns ordered numerical values to categories.
    - 'target': Encodes categories based on the mean value of the target variable for each category. Default is 'label'.
- columns (list of strings, optional): The list of column names to be encoded. If None, all columns will be encoded.
- target (pandas.Series, optional): The target variable used for target encoding. Required only when encoding_type is set to 'target'.

Returns:

- X_encoded (pandas.DataFrame): A DataFrame with the specified columns encoded.

(Label Encoding: One-Hot Encoding: Ordinal Encoding: Target Encoding:) functions

```python
def encode_data(X, encoding_type='label', columns=None, target=None):
    """
    Encodes categorical features in a DataFrame using the specified encoding type.

    Parameters:
    - X: pd.DataFrame
        The input data containing features to be encoded.
    - encoding_type: str
        The type of encoding to use. Options are 'label', 'one-hot', 'ordinal', 'target'.
    - columns: list of str (optional)
        List of column names to be encoded. If None, all columns are encoded.
    - target: pd.Series (optional)
        The target variable used for target encoding.

    Returns:
    - X_encoded: pd.DataFrame
        DataFrame with encoded features.
    """
    if columns is None:
        columns = X.columns

    X_encoded = X.copy()

    if encoding_type == 'label':
        le = LabelEncoder()
        for col in columns:
            X_encoded[col] = le.fit_transform(X_encoded[col])
```

```python
    elif encoding_type == 'one-hot':
        X_encoded = pd.get_dummies(X_encoded, columns=columns)

    elif encoding_type == 'ordinal':
        oe = OrdinalEncoder()
        X_encoded[columns] = oe.fit_transform(X_encoded[columns])

    elif encoding_type == 'target':
        if target is None:
            raise ValueError("Target must be provided for target encoding")
        for col in columns:
            mean_encodings = X_encoded.groupby(col)[target].mean()
            X_encoded[col] = X_encoded[col].map(mean_encodings)
    else:
        raise ValueError(f"Unsupported encoding type: {encoding_type}")

    return X_encoded
```

Notes:

- **Label Encoding**: Each category in a column is assigned an integer based on alphabetical order.
- **One-Hot Encoding**: Creates new columns for each category, and assigns binary values (1 or 0).

- **Ordinal Encoding**: Assigns ordered numerical values based on category order.
- **Target Encoding**: Encodes categories based on the mean target value for each category.
    - Requires a target variable for mapping.

plot_estimator

Description:

The plot_estimator function visualizes the decision boundaries of a trained estimator (e.g., classifier) on a 2D feature space. It generates a mesh grid to plot the decision regions and overlays the data points to show how well the estimator separates the classes.

Parameters:

- estimator (object): The model to plot, which must follow the scikit-learn API and have .fit() and .predict() methods.
- X (numpy array): The feature data (2D array) used to train the estimator, where each row is a data point and each column is a feature.
- y (numpy array): The target labels corresponding to the feature data.
- mesh_step (float, optional): The step size for the mesh grid. A smaller value provides a finer grid but increases computational time. Default is 0.1.
- cmap_light (ListedColormap, optional): Colormap for the background decision regions. Default is ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF']).
- cmap_bold (ListedColormap, optional): Colormap for the data points. Default is ListedColormap(['#FF0000', '#00FF00', '#0000FF']).

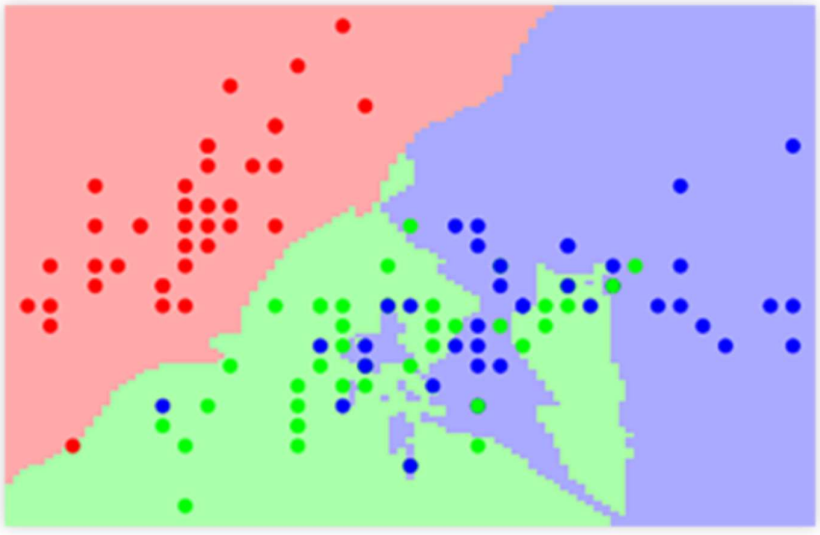
Returns:

- None: The function directly plots the decision boundary and the data points.

DEcision Boundary for KNN classifier

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def plot_estimator(estimator, X, y, mesh_step=0.1, cmap_light=None, cmap_bold=None):
    """
    Plots the decision boundaries of an estimator alongside the data points.

    Parameters:
    - estimator: object
        The model that follows scikit-learn API, should have a .fit() and .predict() method.
    - X: np.array
        The features data (2D array).
    - y: np.array
        The target labels.
    - mesh_step: float (optional)
        The step size for mesh grid. Default is 0.1.
    - cmap_light: ListedColormap (optional)
        The colormap for mesh background. Default is a preset colormap.
    - cmap_bold: ListedColormap (optional)
        The colormap for scatter plot. Default is a preset colormap.

    Returns:
    - None
        Displays the decision boundary plot.
    """
```

```
    Displays the decision boundary plot.
"""
# Set default colormaps if not provided
if cmap_light is None:
    cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
if cmap_bold is None:
    cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

estimator.fit(X, y)

# Determine the maximum and minimum mesh as a boundary
x_min, x_max = X[:, 0].min() - mesh_step, X[:, 0].max() + mesh_step
y_min, y_max = X[:, 1].min() - mesh_step, X[:, 1].max() + mesh_step

# Generate points on the mesh
xx, yy = np.meshgrid(np.arange(x_min, x_max, mesh_step),
                     np.arange(y_min, y_max, mesh_step))

# Make predictions on the grid points
Z = estimator.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision boundary
plt.figure()
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

# Plot the original data points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, edgecolor='k', s=20)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Decision Boundary Plot')
plt.show()
```



Example Output:

A plot displaying:

- **Decision Boundaries**: The background regions are colored based on the predicted class of the model in those regions.
- **Data Points**: The original data points are plotted with a distinct color for each class and a black edge for visual clarity.

Notes:

- The function assumes that the dataset X has exactly two features, as the decision boundary is plotted in a 2D plane.
- The colormaps can be customized by passing cmap_light and cmap_bold if specific color schemes are required.
- For models like KNeighborsClassifier, LogisticRegression, DecisionTreeClassifier, etc., the function will plot the decision regions based on how well the model fits the data

linear_regression using a function

Description:

The linear_regression function performs simple linear regression on a set of data. It calculates the best-fit line by determining the slope and y-intercept that minimize the sum of squared residuals between the predicted and actual values.

Parameters:

- x (array-like): The independent variable (input feature).
- y (array-like): The dependent variable (target output).

Returns:

- beta (float): The slope of the best-fit line.
- alpha (float): The y-intercept of the best-fit line.

Example Usage:

linear regression

```python
import numpy as np

def linear_regression(x, y):
    """
    Performs linear regression on a set of data.

    Parameters:
    x (array-like): The independent variable (input feature).
    y (array-like): The dependent variable (target output).

    Returns:
    float: slope of the best fit line
    float: y-intercept of the best fit line
    """
    # Converting lists to numpy arrays if not already
    x = np.array(x)
    y = np.array(y)

    # Calculating the means of x and y
    x_mean = np.mean(x)
    y_mean = np.mean(y)

    # Calculating the terms needed for the numerator and denominator of the slope
    numerator = np.sum((x - x_mean) * (y - y_mean))
    denominator = np.sum((x - x_mean)**2)

    # Calculating the slope (beta)
    beta = numerator / denominator

    # Calculating the y-intercept (alpha)
    alpha = y_mean - beta * x_mean
```

```python
    y = np.array(y)

    # Calculating the means of x and y
    x_mean = np.mean(x)
    y_mean = np.mean(y)

    # Calculating the terms needed for the numerator and denominator of the slope
    numerator = np.sum((x - x_mean) * (y - y_mean))
    denominator = np.sum((x - x_mean)**2)

    # Calculating the slope (beta)
    beta = numerator / denominator

    # Calculating the y-intercept (alpha)
    alpha = y_mean - beta * x_mean

    return beta, alpha
```

polynomial_regression

Description:

The polynomial_regression function fits a polynomial regression model to the provided data. It calculates the best-fit polynomial of a specified degree and plots the polynomial curve along with the original data points.

Parameters:

- x (array-like): The independent variable (input feature).
- y (array-like): The dependent variable (target output).
- degree (int): The degree of the polynomial to fit to the data.

Returns:

- p (numpy.poly1d): A polynomial object that represents the fitted regression model.

Polynomial regression

```
import numpy as np
import matplotlib.pyplot as plt

def polynomial_regression(x, y, degree):
    """
    Fits a polynomial regression model to the given data.

    Parameters:
    x (array-like): The independent variable (input feature).
    y (array-like): The dependent variable (target output).
    degree (int): The degree of the polynomial to fit.

    Returns:
    np.poly1d: A polynomial that represents the regression model.
    """
    # Fit the polynomial model
    coeffs = np.polyfit(x, y, degree)

    # Create a polynomial from the coefficients
    p = np.poly1d(coeffs)

    # Plotting the original data and the polynomial curve
    xp = np.linspace(min(x), max(x), 100)
    plt.scatter(x, y, label='Data Points')
    plt.plot(xp, p(xp), 'r-', label=f'Polynomial Degree {degree}')
    plt.title('Polynomial Regression Fit')
    plt.xlabel('Independent variable (x)')
    plt.ylabel('Dependent variable (y)')
    plt.legend()
    plt.show()

    return p
```

LogisticRegression Class

Description:

This LogisticRegression class implements logistic regression using gradient descent for binary classification tasks. It includes methods for fitting the model to data and predicting class labels based on the learned parameters (weights and bias). The model uses the sigmoid activation function to map the linear combination of features to a probability.

Parameters:

- learning_rate (float, optional): The learning rate for gradient descent. Default is 0.01.
- num_iterations (int, optional): The number of iterations for gradient descent optimization. Default is 1000.

Methods:

sigmoid(z)

- **Description**: Computes the sigmoid of z, mapping real numbers to the range (0, 1).
- **Parameters**:
    - z (array-like): Input values.
- **Returns**: The sigmoid transformation of z.

fit(X, y)

- **Description**: Fits the logistic regression model to the training data using gradient descent.
- **Parameters**:
    - X (numpy array): The training data, with shape (n_samples, n_features).
    - y (numpy array): The target labels, with shape (n_samples,).
- **Operation**:
    - Initializes the weights and bias to zero.
    - Performs gradient descent for the specified number of iterations, updating weights and bias to minimize the loss.
- **Returns**: None (updates internal weights and bias).

predict(X)

- **Description**: Predicts the class labels for the input data using the learned weights and bias.

- **Parameters**:
    - X (numpy array): The data for which predictions are made, with shape (n_samples, n_features).
- **Returns**: numpy array of predicted class labels (0 or 1).

Logistic regression

```python
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

class LogisticRegression:
    def __init__(self, learning_rate=0.01, num_iterations=1000):
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.weights = None
        self.bias = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def fit(self, X, y):
        # Number of features
        n_samples, n_features = X.shape

        # Init parameters
        self.weights = np.zeros(n_features)
        self.bias = 0

        # Gradient descent
        for _ in range(self.num_iterations):
            # Linear model
            linear_model = np.dot(X, self.weights) + self.bias
            # Apply sigmoid to linear model
            y_predicted = self.sigmoid(linear_model)
```

```python
            # Compute gradients
            dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / n_samples) * np.sum(y_predicted - y)

            # Update parameters
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_predicted = self.sigmoid(linear_model)
        y_predicted_cls = [1 if i > 0.5 else 0 for i in y_predicted]
        return np.array(y_predicted_cls)
```

Notes:

- **Sigmoid Function**: The sigmoid function is used to map the linear combination of the input features to a probability between 0 and 1.
- **Gradient Descent**: The model uses gradient descent to minimize the logistic loss function by adjusting the weights and bias.
- **Prediction Threshold**: The class labels are predicted based on a threshold of 0.5. If the predicted probability is greater than 0.5, the class label is 1; otherwise, it is 0.

LinearSVM Class

Description:

The LinearSVM class implements a linear Support Vector Machine (SVM) for binary classification using gradient descent optimization. It uses a hinge loss function and incorporates L2 regularization to penalize large weights. The model adjusts weights and bias through gradient updates over multiple iterations.

Parameters:

- learning_rate (float, optional): The learning rate for gradient descent. Default is 0.001.
- lambda_param (float, optional): The regularization parameter (L2 penalty). Default is 0.01.
- n_iters (int, optional): The number of iterations for gradient descent. Default is 1000.

Methods:

fit(X, y)

- **Description**: Fits the SVM model to the training data using gradient descent. The method minimizes the hinge loss function while regularizing the model using the L2 norm of the weights.
- **Parameters**:
    - X (numpy array): Training data, shape (n_samples, n_features).
    - y (numpy array): Target labels, shape (n_samples,), where the labels are expected to be binary (0 or 1).
- **Operation**:

- Converts the target labels y into values of -1 and 1 (SVM requires this format).
- Initializes the weights and bias to zero.
- Performs gradient updates based on whether the sample satisfies the hinge loss condition.

- **Returns**: None (updates internal weights w and bias b).

predict(X)

- **Description**: Predicts the class labels for input data using the learned SVM model.
- **Parameters**:
  - X (numpy array): Input data, shape (n_samples, n_features).
- **Returns**:
  - numpy array: Predicted class labels (either -1 or 1), based on the sign of the linear output.

Support vector Machine (SVM)

```
class LinearSVM:
    def __init__(self, learning_rate=0.001, lambda_param=0.01, n_iters=1000):
        self.learning_rate = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        # Convert labels to -1 and 1
        y_ = np.where(y <= 0, -1, 1)

        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)
        self.b = 0

        # Gradient descent
        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                condition = y_[idx] * (np.dot(x_i, self.w) - self.b) >= 1
                if condition:
                    self.w -= self.learning_rate * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.learning_rate * (2 * self.lambda_param * self.w - np.dot(x_i, y_[idx]))
                    self.b -= self.learning_rate * y_[idx]

    def predict(self, X):
        linear_output = np.dot(X, self.w) - self.b
        return np.sign(linear_output)
```

Notes:

- **Hinge Loss**: The hinge loss function penalizes points that lie inside the margin or on the wrong side of the decision boundary.
- **Regularization**: The regularization term (lambda_param) helps to prevent overfitting by penalizing large weight values.
- **Binary Classification**: This implementation of SVM is designed for binary classification. It converts target labels of 0 to -1 as required by SVM.

Limitations:

- The LinearSVM class is designed for binary classification and will not work out-of-the-box for multiclass problems.
- The gradient descent approach may require careful tuning of the learning rate and number of iterations for convergence.