EN3150 Assignment 01: Learning from data and related challenges and linear models for regression

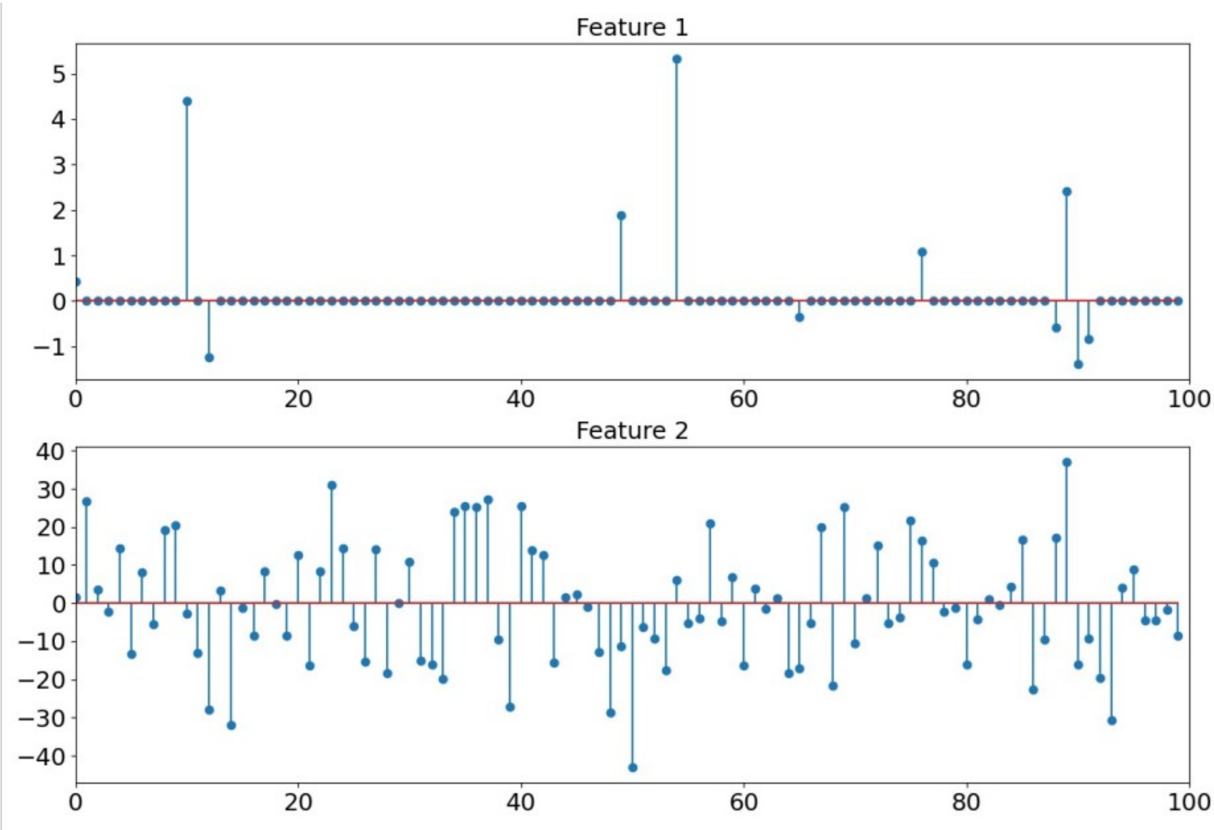
210174X

Gallella M.M.H.H.B

ENTC

1 Data pre-processing

The diagram shows a dataset with feature vector X = <x1, x2> with 2 features and 0 to 100 samples.



Answer:

(2) Max-Abs scaling is the best scaling option for scale both x1 and x2 into a similar feature spaces with same range from [-1,1].

Special reasons:

- Scaling shouldn't damage the negative-positive property which may a very important feature property (After scaling still negatives Should be negative and positives should be positive).
- Both x1, x2 are already centered at zero (Shouldn't move the zeros to another value)
- x1 is [-1,5] range and x2 is [40,-40] range. After proper scaling both x1, x2 should be in the same scale; can't use Robust Scaling or Standard Scaling.

Overview of all the Scaling methods;

1. **Min-Max Scaling**

- **Definition**: Min-max scaling (also known as normalization) transforms features by scaling them to a specific range; typically [0, 1].
- **Formula**:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

  where X is the original value, $X_{min}$is the minimum value in the feature, and X_{max} is the maximum value in the feature.
- **Use Case**: Min-max scaling is useful when the features have different units or scales and you want to maintain the relationships between the data points.

2. **Max Abs Scaling**

- **Definition**: Max Abs scaling scales each feature by its maximum absolute value, transforming the data such that the values range between [-1, 1].
- **Formula**:

$$X' = \frac{X}{max(|X|)}$$

  where $max(|X|)$ is the maximum absolute value of the feature.
- **Use Case**: This scaling is useful for data that is **already centered at zero** and is sparse, such as text data represented by word counts.

3. **Standard Scaling**

- **Definition**: Standard scaling (or standardization) transforms features by removing the mean and scaling to unit variance, resulting in the Z distribution with [μ = 0, σ = 1].
- **Formula**:

$$X' = \frac{X - \mu}{\sigma}$$

  where μ is the mean of the feature and σ is the standard deviation.
- **Use Case**: This scaling is useful when the data follows a Gaussian distribution $N[\mu, \sigma)$ and when features have different units or scales.

4. **Robust Scaling**

- **Definition**: Robust scaling scales features using statistics that are **robust to outliers**, such as the median and the interquartile range (IQR).
- **Formula**:

$$X' = \frac{X - \text{median}(X)}{\text{IQR}(X)}$$

  where the IQR = [Q1,Q2] is the difference between the 75th and 25th percentiles.
- **Use Case**: This scaling is particularly useful when the data contains many outliers, as it reduces the impact of extreme values.

2 Learning from data

2.1] Generate data.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# Generate 100 samples
n_samples = 100

# Generate X values (uniformly distributed between 0 and 10)
X = 10 * np.random.rand(n_samples, 1)

# Generate epsilon values (normally distributed with mean 0 and
standard deviation 15)
epsilon = np.random.normal(0, 15, n_samples)

# Generate Y values using the model Y = 3 + 2X + epsilon
Y = 3 + 2 * X + epsilon[:, np.newaxis]
```

Summary of the Generated Data;

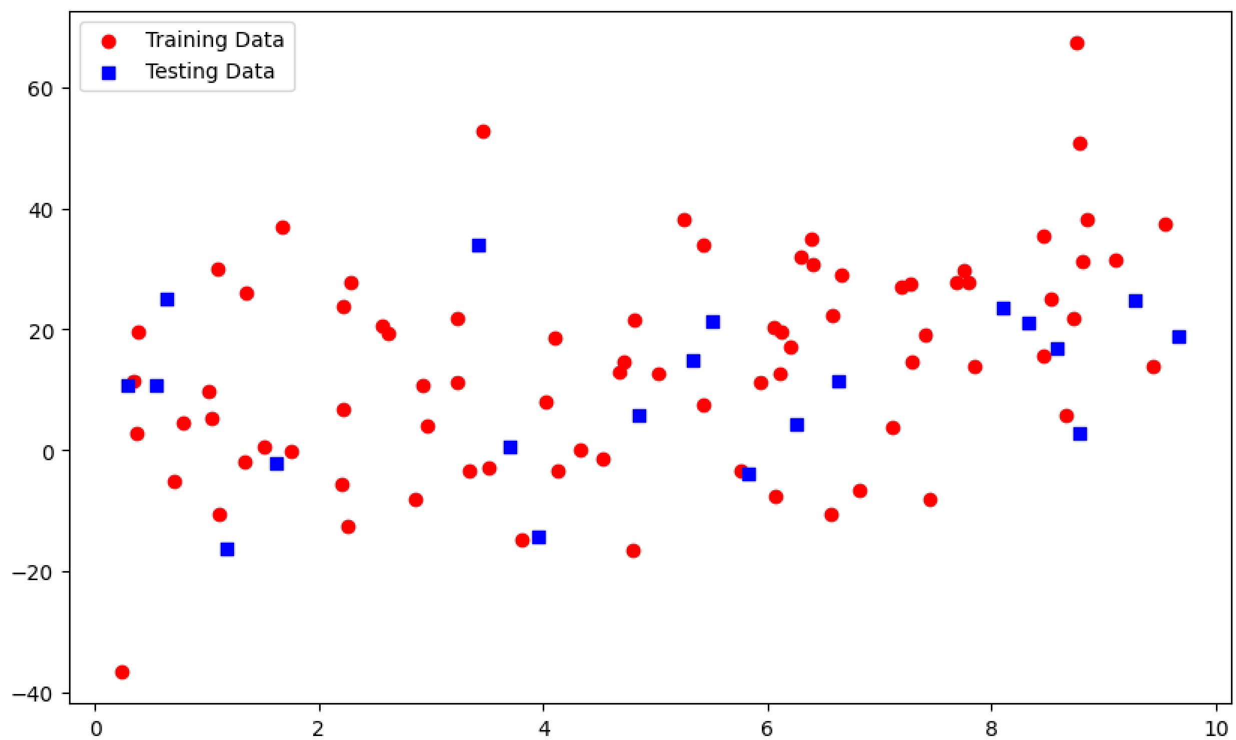
- $X_i = 10 \cdot U_i$, where $U_i \sim \text{Uniform}(0,1)$ for $i = 1, 2, \ldots, 100$
- $ \epsilon_i \sim \mathcal{N}(0, 15^2) $
- $ Y_i = 3 + 2X_i + \epsilon_i $

The dataset consists of 100 points $ (X_i, Y_i) $, where $ X_i $ is uniformly distributed between 0 and 10, and $ Y_i $ is generated by the linear model with added Gaussian noise $ \epsilon_i $.

2. Run the code given in listing 2 multiple times and write down your observation. Why training and testing data is different in each run?

2.2.1] If random_state=r is a fixed value we'll endup getting the same data plot for training and testing data in each run.

```
r =4

# Split the data into training and test sets (80% train, 20% test)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.2,random_state=r)

# Plot the data points
plt.figure(figsize=(10, 6))
plt.scatter(X_train, Y_train, alpha=1, marker='o', color='red',
label='Training Data')
plt.scatter(X_test, Y_test, alpha=1, marker='s', color='blue',
label='Testing Data')
plt.legend()
plt.show()
```



2.2.2] If random_state=r is a random value we'll endup getting different plots for training and testing data each time we run the code.

```
r = np.random.randint(104)
print(r)
```

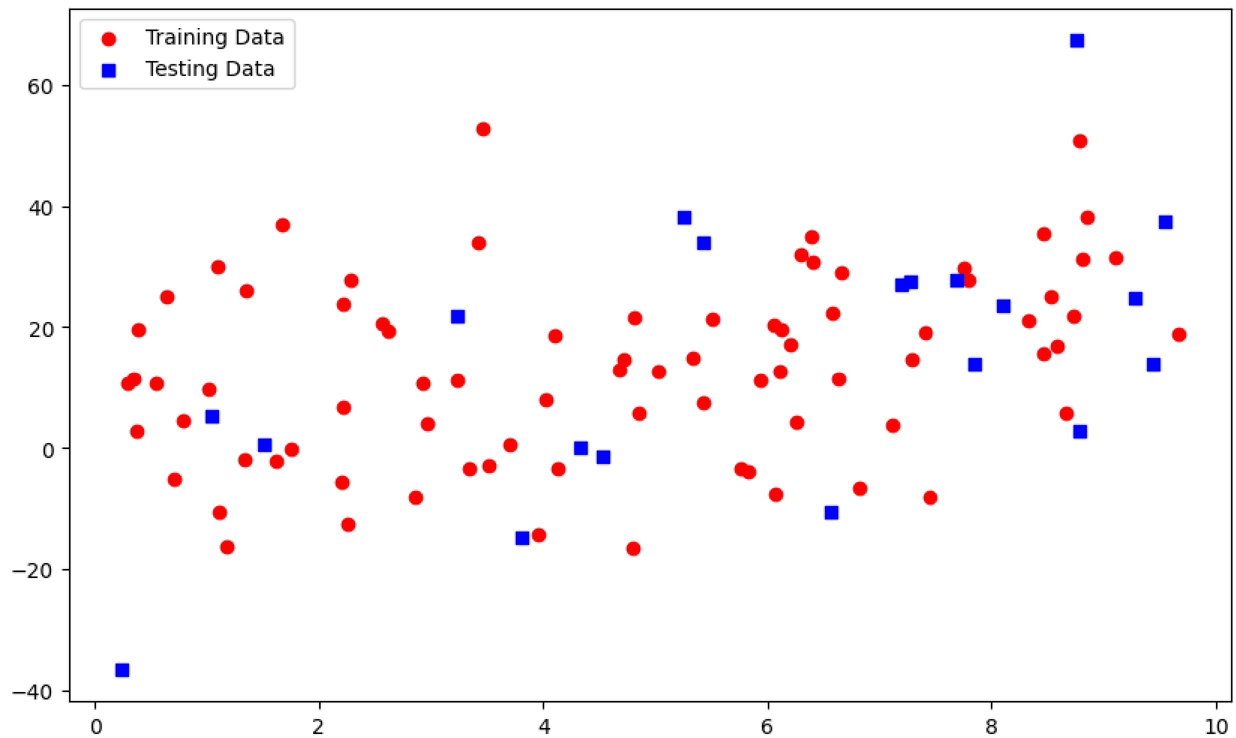
```
29
```

```
r = np.random.randint(104)
print(r)
```

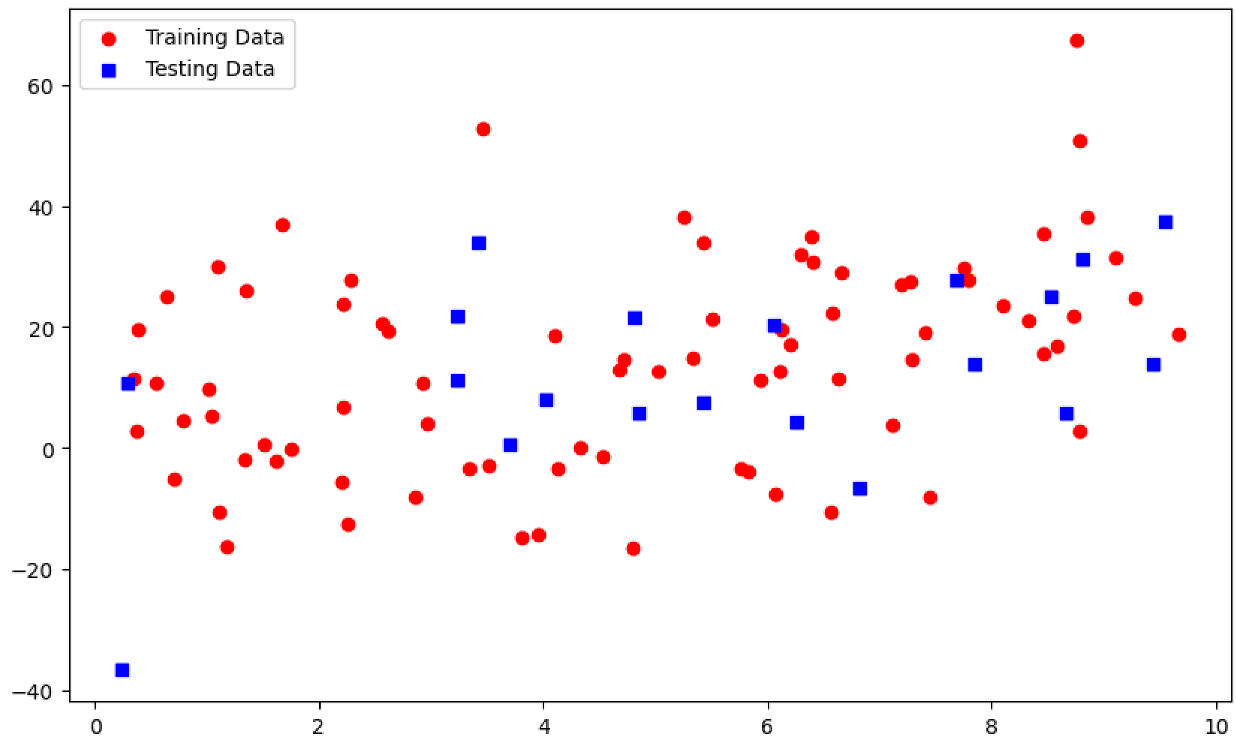
```
71
```

The above code generates a random integer between 0 and 103 using `np.random.randint(104)` in each run.

```
# Generate a random integer as the seed for reproducibility
r = np.random.randint(104)

# Split the data into training and test sets (80% train, 20% test)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.2,random_state=r)

# Plot the data points
plt.figure(figsize=(10, 6))
plt.scatter(X_train, Y_train, alpha=1, marker='o', color='red',
label='Training Data')
plt.scatter(X_test, Y_test, alpha=1, marker='s', color='blue',
label='Testing Data')
plt.legend()
plt.show()
```




```
# Generate a random integer as the seed for reproducibility
r = np.random.randint(104)

# Split the data into training and test sets (80% train, 20% test)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.2,random_state=r)

# Plot the data points
plt.figure(figsize=(10, 6))
plt.scatter(X_train, Y_train, alpha=1, marker='o', color='red',
label='Training Data')
plt.scatter(X_test, Y_test, alpha=1, marker='s', color='blue',
label='Testing Data')
plt.legend()
plt.show()
```

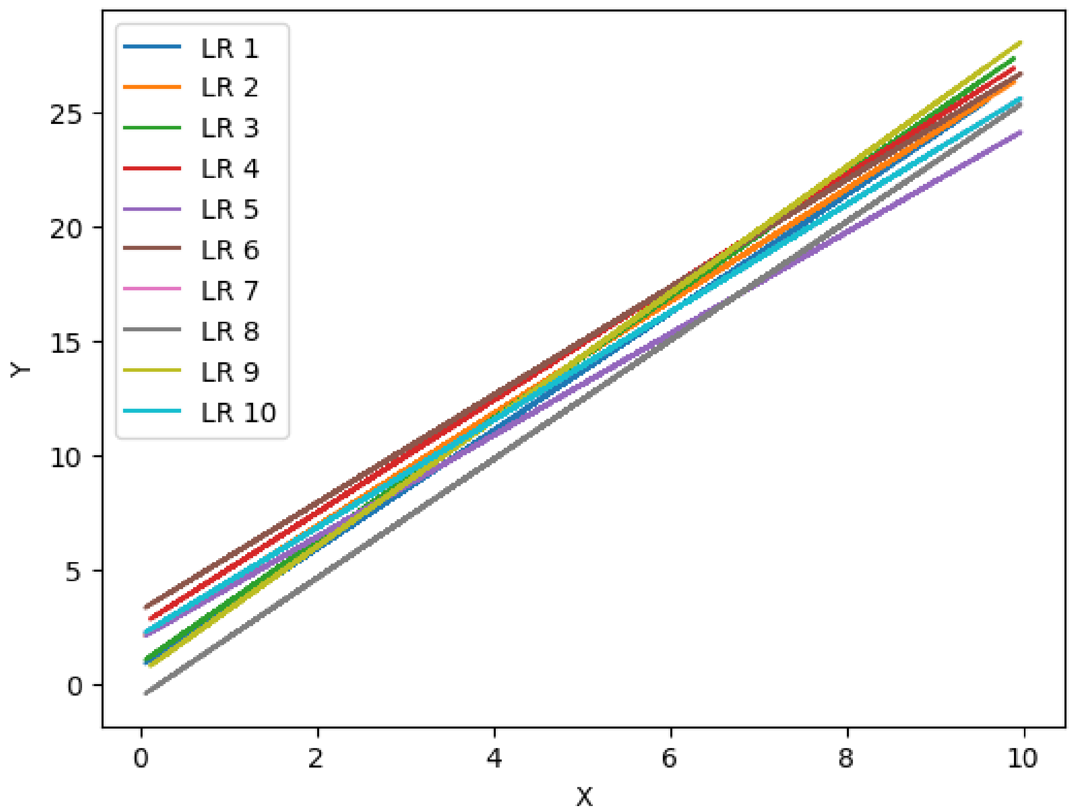


Observation: When we run multiple times, train_test_split generates different train and test data splits based on the random seed from r = np.random.randint(104) build differen splits, in seed state between [0,103]

2.3] Fitting linear regression model to each train_test_split generated through different random seeds.

```
# Plotting 10 different instances
for i in range(10):
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.2, random_state=np.random.randint(104))

    # Building the model
    model = LinearRegression()
    # Train the model
    model.fit(X_train, Y_train)

    # Predict on the training data
    Y_pred_train = model.predict(X_train)

    # Plot the predictions
    plt.plot(X_train, Y_pred_train, label=f'LR {i+1}')


# Label the axes
plt.xlabel('X')
plt.ylabel('Y')
```

```
# Display the legend
plt.legend()
plt.show()
```

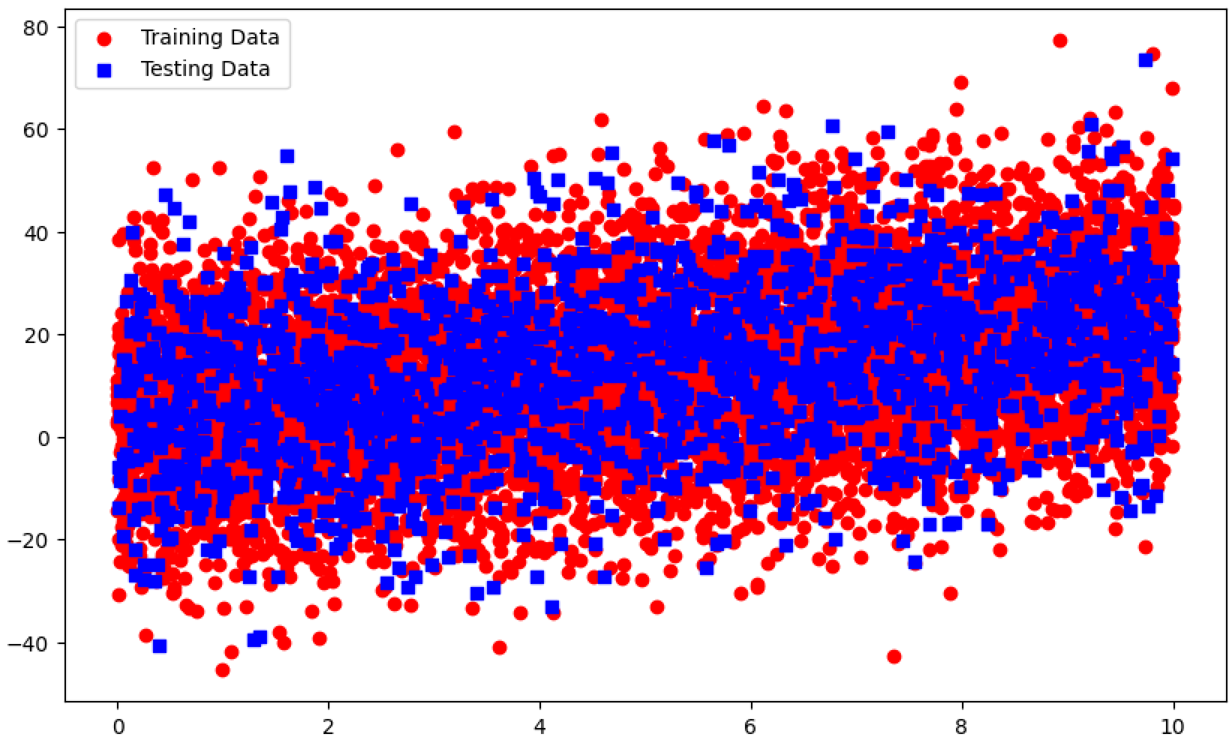


When running the linear regression model over 10 instances, the models vary because each time, a new training dataset is formed by randomly dividing the original data into training and testing sets. This random division impacts the resulting model parameters. Furthermore, since the dataset is relatively small (n_samples = 100), the limited data contributes to larger inconsistencies in model outcomes across different runs.

2.4] Observations After Increasing the Data Samples to 10,000

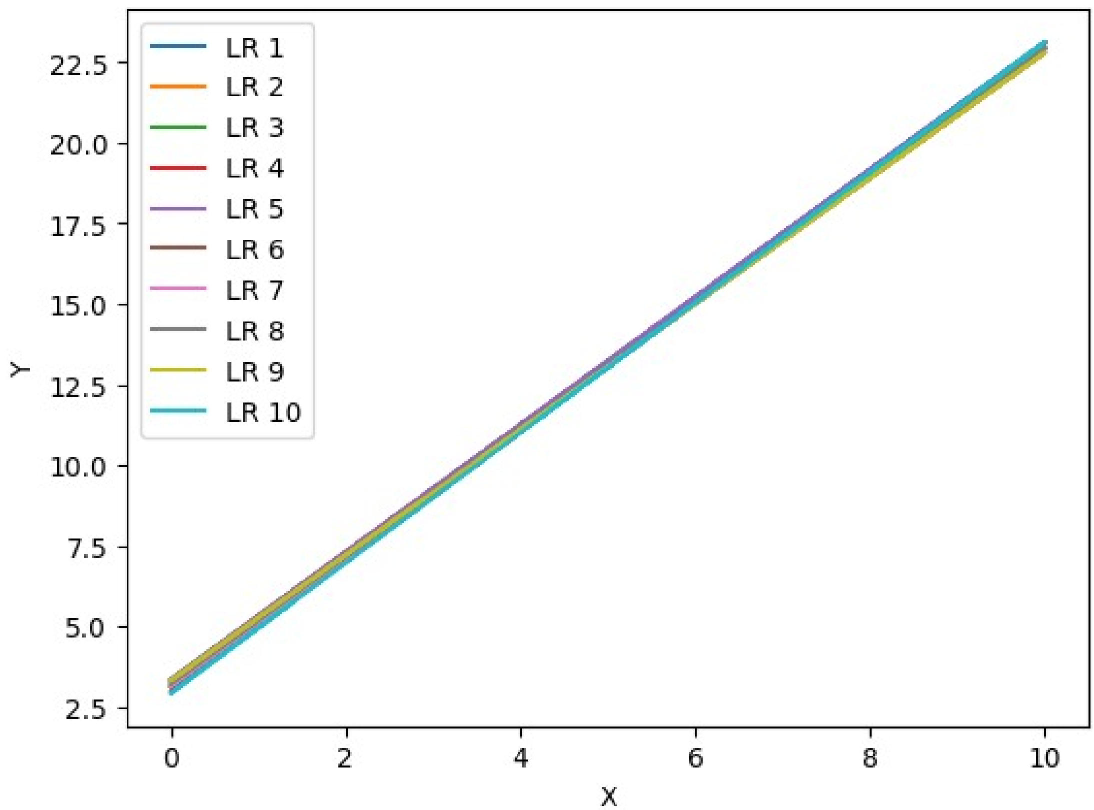
```
# Generate 100 samples
n_samples = 10000
# Generate X values ( uniformly distributed between 0and 10)
X = 10 * np . random . rand ( n_samples , 1)
# Generate epsilon values ( normally distributed with mean 0 and
standard deviation 15)
epsilon = np . random . normal (0 , 15 , n_samples )
# Generate Y values using the model Y = 3 + 3X + epsilon
Y = 3 + 2 * X + epsilon [: , np . newaxis ]
```

```python
# Generate a random integer as the seed for reproducibility
r = np.random.randint(104)

# Split the data into training and test sets (80% train, 20% test)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.2,random_state=r)

# Plot the data points
plt.figure(figsize=(10, 6))
plt.scatter(X_train, Y_train, alpha=1, marker='o', color='red',
label='Training Data')
plt.scatter(X_test, Y_test, alpha=1, marker='s', color='blue',
label='Testing Data')
plt.legend()
plt.show()
```




```python
# Plotting 10 different instances
for i in range(10):
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.2, random_state=np.random.randint(104))

    # Building the model
    model = LinearRegression()
    # Train the model
    model.fit(X_train, Y_train)
```

```
    # Predict on the training data
    Y_pred_train = model.predict(X_train)

    # Plot the predictions
    plt.plot(X_train, Y_pred_train, label=f'LR {i+1}')


# Label the axes
plt.xlabel('X')
plt.ylabel('Y')

# Display the legend
plt.legend()
plt.show()
```



Observation: When increasing the number of data samples to 10,000 (n_samples = 10000), the linear regression model becomes more stable and consistent across different instances of train-test data split. Compared to the 100 data samples, the variability in the model predictions decreases significantly because of optimal parameters. This is because with a larger dataset, the model can better capture the underlying patterns in the data, reducing the impact of random fluctuations from train-test splitting of the dataset. The larger sample size provides more data points for training, which results in more reliable model coefficients and improved generalization.

3] Linear regression on real world data

3.1] Load the dataset

Install ucimlrepo

```
! pip install ucimlrepo

Collecting ucimlrepo
  Downloading ucimlrepo-0.0.7-py3-none-any.whl.metadata (5.5 kB)
Requirement already satisfied: pandas>=1.0.0 in
/usr/local/lib/python3.10/dist-packages (from ucimlrepo) (2.1.4)
Requirement already satisfied: certifi>=2020.12.5 in
/usr/local/lib/python3.10/dist-packages (from ucimlrepo) (2024.8.30)
Requirement already satisfied: numpy<2,>=1.22.4 in
/usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0-
>ucimlrepo) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0-
>ucimlrepo) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0-
>ucimlrepo) (2024.1)
Requirement already satisfied: tzdata>=2022.1 in
/usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0-
>ucimlrepo) (2024.1)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2-
>pandas>=1.0.0->ucimlrepo) (1.16.0)
Downloading ucimlrepo-0.0.7-py3-none-any.whl (8.0 kB)
Installing collected packages: ucimlrepo
Successfully installed ucimlrepo-0.0.7
```

Fetch the dataset

```
# If the package is not installed, install it using: pip install
ucimlrepo
from ucimlrepo import fetch_ucirepo

# Fetch the dataset
infrared_thermography_temperature = fetch_ucirepo(id=925)

# Data (as pandas DataFrames)
X = infrared_thermography_temperature.data.features
y = infrared_thermography_temperature.data.targets

# Metadata
print(infrared_thermography_temperature.metadata)
```

```python
# Variable information
print(infrared_thermography_temperature.variables)
```

{'uci_id': 925, 'name': 'Infrared Thermography Temperature', 'repository_url': 'https://archive.ics.uci.edu/dataset/925/infrared+thermography+temperature+dataset', 'data_url': 'https://archive.ics.uci.edu/static/public/925/data.csv', 'abstract': 'The Infrared Thermography Temperature Dataset contains temperatures read from various locations of inferred images about patients, with the addition of oral temperatures measured for each individual. The 33 features consist of gender, age, ethnicity, ambiant temperature, humidity, distance, and other temperature readings from the thermal images. The dataset is intended to be used in a regression task to predict the oral temperature using the environment information as well as the thermal image readings. ', 'area': 'Health and Medicine', 'tasks': ['Regression'], 'characteristics': ['Tabular'], 'num_instances': 1020, 'num_features': 33, 'feature_types': ['Real', 'Categorical'], 'demographics': ['Gender', 'Age', 'Ethnicity'], 'target_col': ['aveOralF', 'aveOralM'], 'index_col': ['SubjectID'], 'has_missing_values': 'no', 'missing_values_symbol': None, 'year_of_dataset_creation': 2021, 'last_updated': 'Tue Dec 12 2023', 'dataset_doi': '10.13026/9ay4-2c37', 'creators': ['Quanzeng Wang', 'Yangling Zhou', 'Pejman Ghassemi', 'David McBride', 'J. Casamento', 'T. Pfefer', 'Quanzeng Wang', 'Yangling Zhou', 'Pejman Ghassemi', 'David McBride', 'J. Casamento', 'T. Pfefer'], 'intro_paper': {'title': 'Infrared Thermography for Measuring Elevated Body Temperature: Clinical Accuracy, Calibration, and Evaluation', 'authors': 'Quanzeng Wang, Yangling Zhou, Pejman Ghassemi, David McBride, J. Casamento, T. Pfefer', 'published_in': 'Italian National Conference on Sensors', 'year': 2021, 'url': 'https://www.semanticscholar.org/paper/443b9932d295ca3a014e7d874b4bd77a33a276bd', 'doi': None}, 'additional_info': {'summary': None, 'purpose': None, 'funded_by': None, 'instances_represent': None, 'recommended_data_splits': None, 'sensitive_data': None, 'preprocessing_description': None, 'variable_info': '- gender\n- age\n- ethnicity\n- ambiant temperature\n- humidity\n- distance\n- temperature readings from the thermal images', 'citation': None}, 'external_url': 'https://physionet.org/content/face-oral-temp-data/1.0.0/'}

```
          name     role         type demographic  \
0    SubjectID       ID  Categorical        None
1     aveOralF   Target   Continuous        None
2     aveOralM   Target   Continuous        None
3       Gender  Feature  Categorical      Gender
4          Age  Feature  Categorical         Age
5    Ethnicity  Feature  Categorical   Ethnicity
6        T_atm  Feature   Continuous        None
7     Humidity  Feature   Continuous        None
8     Distance  Feature   Continuous        None
```

```
9     T_offset1  Feature   Continuous        None
10    Max1R13_1  Feature   Continuous        None
11    Max1L13_1  Feature   Continuous        None
12  aveAllR13_1  Feature   Continuous        None
13  aveAllL13_1  Feature   Continuous        None
14        T_RC1  Feature   Continuous        None
15    T_RC_Dry1  Feature   Continuous        None
16    T_RC_Wet1  Feature   Continuous        None
17    T_RC_Max1  Feature   Continuous        None
18        T_LC1  Feature   Continuous        None
19    T_LC_Dry1  Feature   Continuous        None
20    T_LC_Wet1  Feature   Continuous        None
21    T_LC_Max1  Feature   Continuous        None
22         RCC1  Feature   Continuous        None
23         LCC1  Feature   Continuous        None
24   canthiMax1  Feature   Continuous        None
25  canthi4Max1  Feature   Continuous        None
26       T_FHCC1  Feature   Continuous        None
27       T_FHRC1  Feature   Continuous        None
28       T_FHLC1  Feature   Continuous        None
29       T_FHBC1  Feature   Continuous        None
30       T_FHTC1  Feature   Continuous        None
31    T_FH_Max1  Feature   Continuous        None
32   T_FHC_Max1  Feature   Continuous        None
33       T_Max1  Feature   Continuous        None
34        T_OR1  Feature   Continuous        None
35    T_OR_Max1  Feature   Continuous        None

                                          description units  
missing_values  
0                                          Subject ID  None  
no  
1                 Oral temperature measured in fast mode  None  
no  
2              Oral temperature measured in monitor mode  None  
no  
3                                      Male or Female  None  
no  
4                         Age ranges in categories\n  None  
no  
5   American Indian or Alaska Native, Asian, Black...  None  
no  
6                                 Ambiant temperature  None  
no  
7                                   Relative humidity  None  
no  
8       Distance between the subjects and the IRTs.    None  
no  
9   Temperature difference between the set and mea...  None
```

```
no
10  Max value of a circle with diameter of 13 pixe...  None
no
11  Max value of a circle with diameter of 13 pixe...  None
no
12  Average value of a circle with diameter of 13 ...  None
no
13  Average value of a circle with diameter of 13 ...  None
no
14  Average temperature of the highest four pixels...  None
no
15  Average temperature of the highest four pixels...  None
no
16  Average temperature of the highest four pixels...  None
no
17  Max value of a square of 24x24 pixels around t...  None
no
18  Average temperature of the highest four pixels...  None
no
19  Average temperature of the highest four pixels...  None
no
20  Average temperature of the highest four pixels...  None
no
21  Max value of a circle with diameter of 13 pixe...  None
no
22  Average value of a square of 3x3 pixels center...  None
no
23  Average value of a square of 3x3 pixels center...  None
no
24                Max value in the extended canthi area  None
no
25  Average temperature of the highest four pixels...  None
no
26  Average value in the center point of forehead,...  None
no
27  Average value in the right point of the forehe...  None
no
28  Average value in the left point of the forehea...  None
no
29  Average value in the bottom point of the foreh...  None
no
30  Average value in the top point of the forehead...  None
no
31  Maximum temperature within the extended forehe...  None
no
32  Max value in the center point of forehead, a s...  None
no
33  Maximum temperature within the whole face region.  None
no
```

```
34  Average temperature of the highest four pixels...   None
no
35       Maximum temperature within the mouth region.   None
no
```

```
print(type(X))
```

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

X and y are Pandas DataFrame objects.

3.2] Independent and Dependent Variables in the data set

1] X DataFrame content;

```
import pandas as pd

X.head()
```

{"type":"dataframe","variable_name":"X"}

```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1020 entries, 0 to 1019
Data columns (total 33 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Gender       1020 non-null   object
 1   Age          1020 non-null   object
 2   Ethnicity    1020 non-null   object
 3   T_atm        1020 non-null   float64
 4   Humidity     1020 non-null   float64
 5   Distance     1018 non-null   float64
 6   T_offset1    1020 non-null   float64
 7   Max1R13_1    1020 non-null   float64
 8   Max1L13_1    1020 non-null   float64
 9   aveAllR13_1  1020 non-null   float64
 10  aveAllL13_1  1020 non-null   float64
 11  T_RC1        1020 non-null   float64
 12  T_RC_Dry1    1020 non-null   float64
 13  T_RC_Wet1    1020 non-null   float64
 14  T_RC_Max1    1020 non-null   float64
 15  T_LC1        1020 non-null   float64
 16  T_LC_Dry1    1020 non-null   float64
 17  T_LC_Wet1    1020 non-null   float64
 18  T_LC_Max1    1020 non-null   float64
 19  RCC1         1020 non-null   float64
 20  LCC1         1020 non-null   float64
 21  canthiMax1   1020 non-null   float64
```

```
 22  canthi4Max1  1020 non-null   float64
 23  T_FHCC1      1020 non-null   float64
 24  T_FHRC1      1020 non-null   float64
 25  T_FHLC1      1020 non-null   float64
 26  T_FHBC1      1020 non-null   float64
 27  T_FHTC1      1020 non-null   float64
 28  T_FH_Max1    1020 non-null   float64
 29  T_FHC_Max1   1020 non-null   float64
 30  T_Max1       1020 non-null   float64
 31  T_OR1        1020 non-null   float64
 32  T_OR_Max1    1020 non-null   float64
dtypes: float64(30), object(3)
memory usage: 263.1+ KB
```

2] y DataFrame contens;

```
y.head()
```


```
{"summary":"{\n  \"name\": \"y\",\n  \"rows\": 1020,\n  \"fields\": [\n    {\n      \"column\": \"aveOralF\",\n      \"properties\": {\n        \"dtype\": \"number\",\n        \"std\": 0.3864033248934766,\n        \"min\": 35.75,\n        \"max\": 39.6,\n        \"num_unique_values\": 53,\n        \"samples\": [\n          37.65,\n          39.2,\n          38.9\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"aveOralM\",\n      \"properties\": {\n        \"dtype\": \"number\",\n        \"std\": 0.5095022787133393,\n        \"min\": 35.54,\n        \"max\": 40.34,\n        \"num_unique_values\": 70,\n        \"samples\": [\n          36.39,\n          36.59,\n          37.59\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    }\n  ]\n}","type":"dataframe","variable_name":"y"}
```


```
y.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1020 entries, 0 to 1019
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   aveOralF  1020 non-null   float64
 1   aveOralM  1020 non-null   float64
dtypes: float64(2)
memory usage: 16.1 KB
```

```
print(X.shape, y.shape)
```

```
(1020, 33) (1020, 2)
```

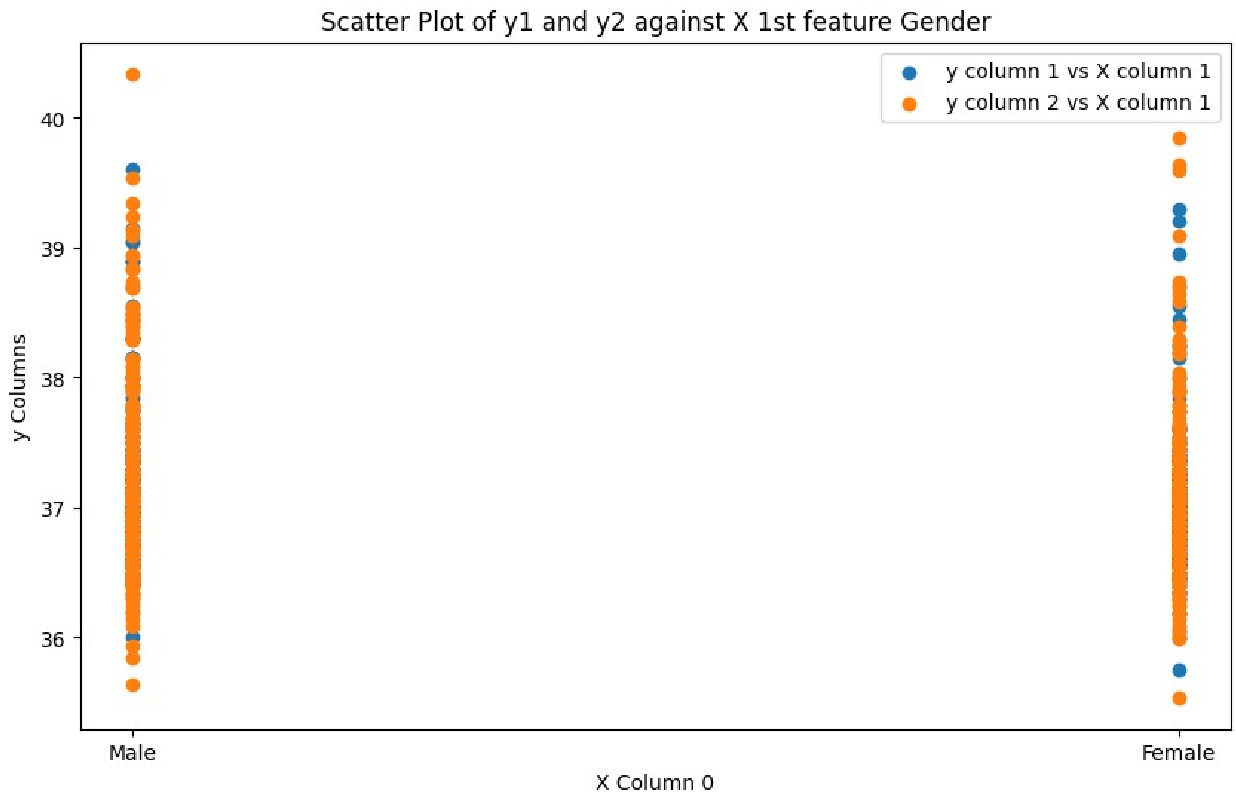
(3.2) Number of Independent and Dependent Variables in the data set

Answer:

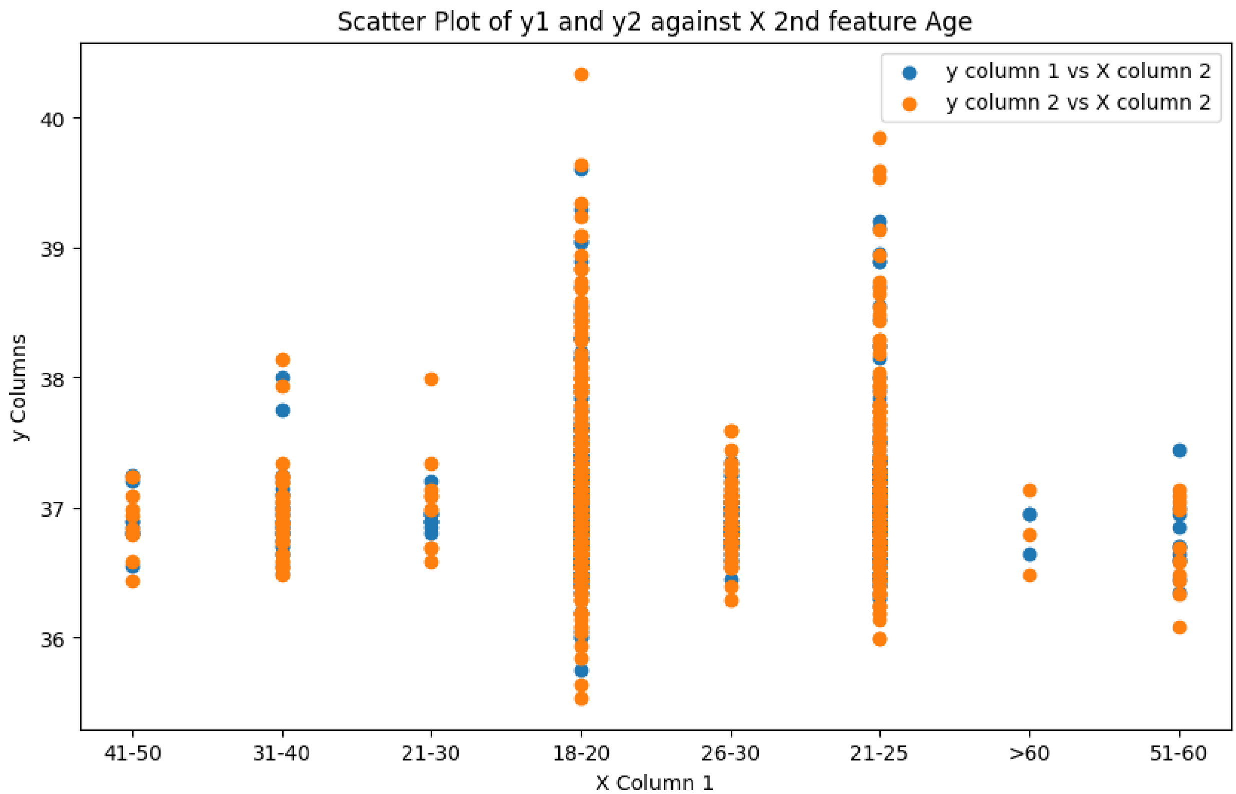
- **X Independent Variables:** 33 because there are 33 Data columns in DataFrame X.
- **Y Dependent Variables:** 2 because there are 2 Data columns in DataFrame y with float64.
- For both X and y there are 1020 data entries as rows which indicates we have 1020 samples to train and test.

Let's have a look on the datapoint distributions, y1 and y2 against each feature x1,x2.........to get a better overview on data

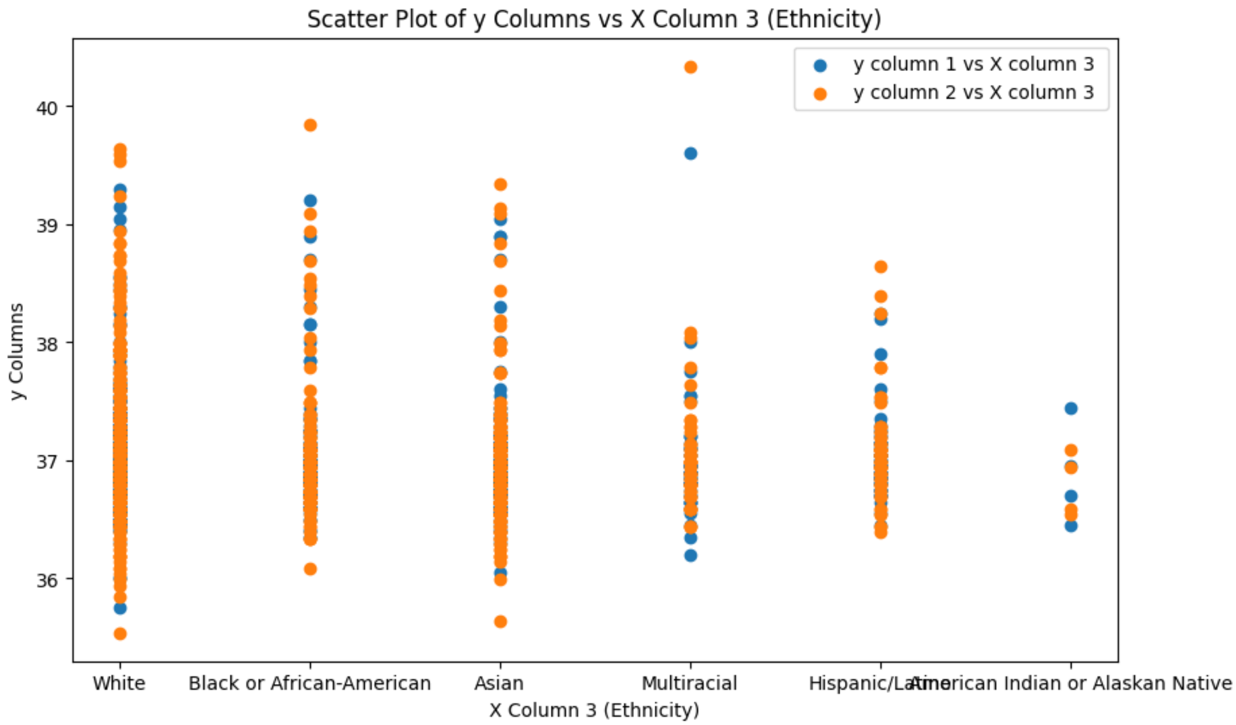
```
import matplotlib.pyplot as plt

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

plt.scatter(X.iloc[:, 0], y.iloc[:, 0], label='y column 1 vs X column 1')
plt.scatter(X.iloc[:, 0], y.iloc[:, 1], label='y column 2 vs X column 1')

plt.xlabel('X Column 0')
plt.ylabel('y Columns')
plt.legend()
plt.title('Scatter Plot of y1 and y2 against X 1st feature Gender')
plt.show()
```



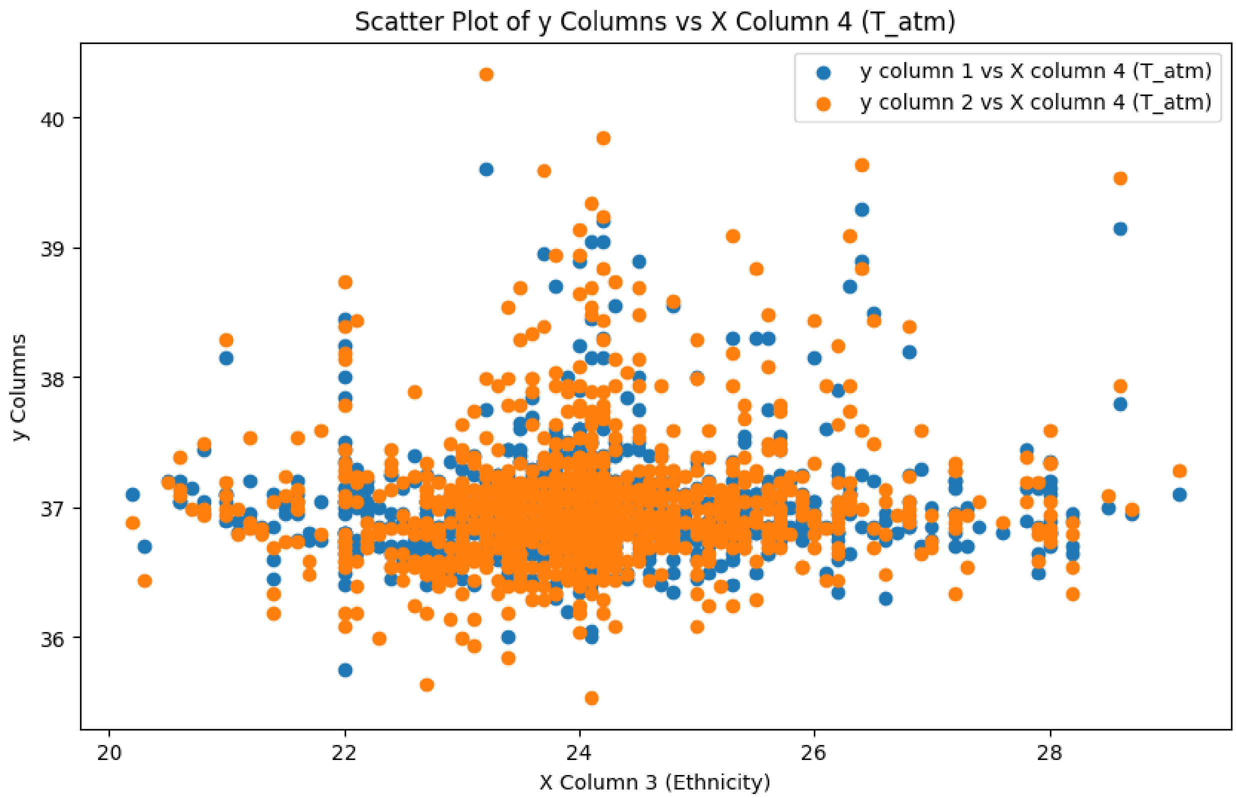

```
plt.figure(figsize=(10, 6))

plt.scatter(X.iloc[:, 1], y.iloc[:, 0], label='y column 1 vs X column
2')
plt.scatter(X.iloc[:, 1], y.iloc[:, 1], label='y column 2 vs X column
2')

plt.xlabel('X Column 1')
plt.ylabel('y Columns')
plt.legend()
plt.title('Scatter Plot of y1 and y2 against X 2nd feature Age')
plt.show()
```



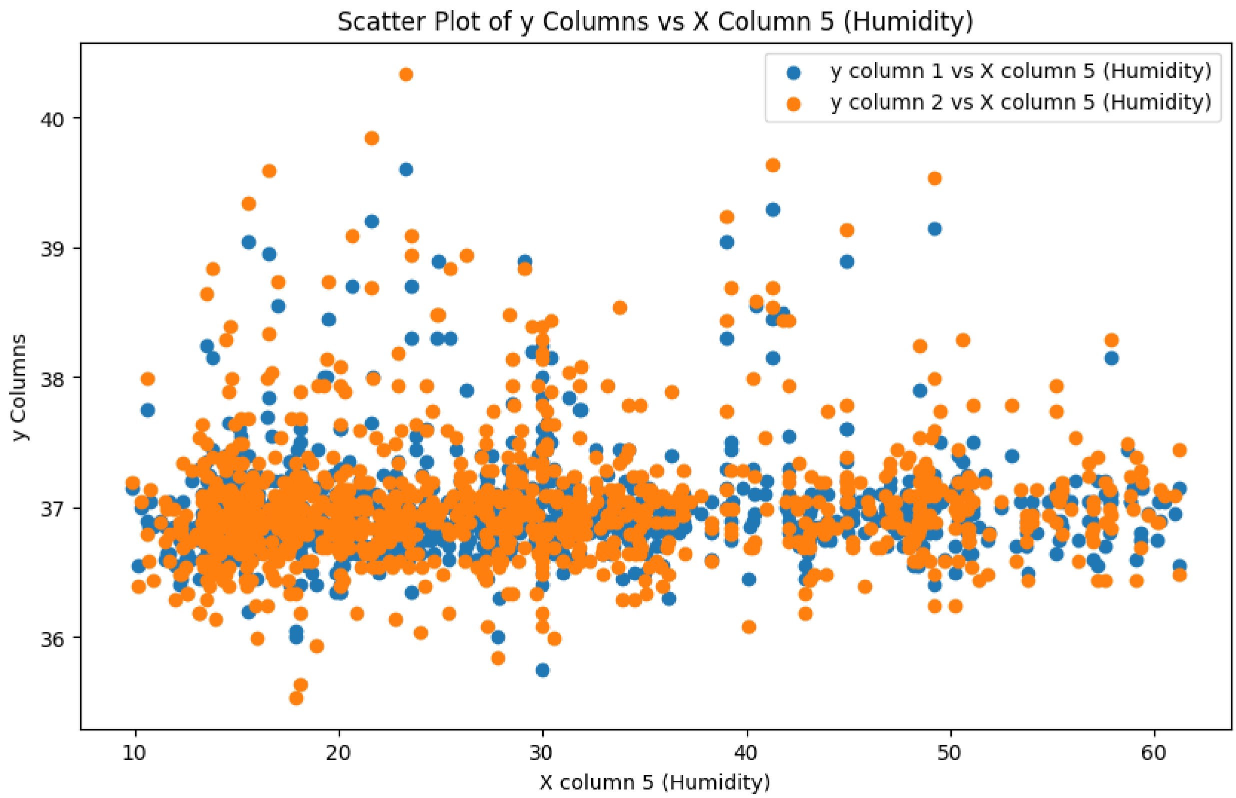

```
plt.figure(figsize=(10, 6))

plt.scatter(X.iloc[:, 2], y.iloc[:, 0], label='y column 1 vs X column
3 ')
plt.scatter(X.iloc[:, 2], y.iloc[:, 1], label='y column 2 vs X column
3 ')

plt.xlabel('X Column 3 (Ethnicity)')
plt.ylabel('y Columns')
plt.legend()
plt.title('Scatter Plot of y Columns vs X Column 3 (Ethnicity)')
plt.show()
```




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

plt.scatter(X.iloc[:, 3], y.iloc[:, 0], label='y column 1 vs X column
4 (T_atm)')
plt.scatter(X.iloc[:, 3], y.iloc[:, 1], label='y column 2 vs X column
4 (T_atm)')

plt.xlabel('X Column 3 (Ethnicity)')
plt.ylabel('y Columns')
plt.legend()
plt.title('Scatter Plot of y Columns vs X Column 4 (T_atm)')
plt.show()
```




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

plt.scatter(X.iloc[:, 4], y.iloc[:, 0], label='y column 1 vs X column 5 (Humidity)')
plt.scatter(X.iloc[:, 4], y.iloc[:, 1], label='y column 2 vs X column 5 (Humidity)')

plt.xlabel('X column 5 (Humidity)')
plt.ylabel('y Columns')
plt.legend()
plt.title('Scatter Plot of y Columns vs X Column 5 (Humidity)')
plt.show()
```



3.3 Feasibility of Applying Linear Regression

Not feasible to apply linear regression in this stage becase still,

- We didn't check for outliers
- Need to scale all the features into a same scale for better convergence in Gradient descend algorithms. -The dataset contains categorical variables, such as: (Categorical) Gender, Age, Ethnicity

Since linear regression works only with numerical data, it is necessary to transform these categorical variables into numerical format before proceeding. Two common methods to achieve this are:

- **One-Hot Encoding**: This technique generates binary columns for each category within the categorical variable. For instance, if the "Gender" feature has categories like "Male" and "Female", one-hot encoding would create two columns: "Gender Male" and "Gender Female", with binary values (0 or 1). The method `sklearn.preprocessing.OneHotEncoder` can be used to implement this.

- **Label Encoding**: This method assigns a distinct integer value to each category. For example, "Male" could be assigned as 0, and "Female" as 1. Although this method is simpler, it may create unintended ordinal relationships among the categories, which can be problematic for nominal variables.

3.4 Correcting NaN Handling Code

Wrongw Approach:

```
# Drop rows with missing values from both X and y
X = X.dropna()
y = y.dropna()

X.shape

(1018, 33)
```

1018 data entries; therefore 2 data entries were removed

```
y.shape

(1020, 2)
```

But here in y there's 1020 data entries

```
y.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1020 entries, 0 to 1019
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   aveOralF  1020 non-null   float64
 1   aveOralM  1020 non-null   float64
dtypes: float64(2)
memory usage: 16.1 KB
```

X has 1018 entrie but y has 1020 entrie

The given code snippet is not the correct approach to remove missing values from both `X` and `y`. Dropping rows with missing values independently from `X` and `y` can lead to a misalignment between the features and the corresponding target values, as the rows dropped from `X` might not correspond to the rows dropped from `y`.

Correct Approach:

The correct approach is to drop rows where there are missing values in either `X` or `y`, ensuring that the data remains aligned. Here's how you can do it:

```
# Combine X and y into a single DataFrame
data = pd.concat([X, y], axis=1)

# Drop rows with any missing values across all columns
data = data.dropna()
```

```
# Separate the combined data back into X and y
X = data.iloc[:, :-2]  # All columns except the last two
y = data.iloc[:, -2:]  # The last two columns (the two target columns)
```

Explanation:

- **Combining X and y:** By concatenating X and y into a single DataFrame, you ensure that the features and corresponding target values stay together.
- **Dropping Missing Values:** Dropping rows with missing values from the combined DataFrame ensures that only the rows with complete data in both X and y are retained.
- **Separating X and y:** After removing the rows with missing data, you split the DataFrame back into X (features) and y (target).

This approach preserves the alignment between the features and their corresponding target values.

```
# Combine X and y into a single DataFrame
data = pd.concat([X, y], axis=1)

# Drop rows with any missing values across all columns
data = data.dropna()

# Separate the combined data back into X and y
X = data.iloc[:, :-2]  # All columns except the last two
y = data.iloc[:, -2:]  # The last two columns (the two target columns)

X.shape
```

```
(1018, 33)
```

```
y.shape
```

```
(1018, 2)
```

Now the dataset is cleaned from NaN/missing values and we can clearly observe that X and y both have 1018 same size data entries.

3.5 Feature Selection

Select Age and four other features based on your preference.

Preparing the new DataFrame yM with the aveOralF

```
y.head()
```

{"summary":"{\n \"name\": \"y\",\n \"rows\": 1018,\n \"fields\": [\n   {\n   \"column\": \"aveOralF\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.38655667866453125,\n   \"min\": 35.75,\n   \"max\": 39.6,\n   \"num_unique_values\": 53,\n   \"samples\": [\n   37.65,\n

```
39.2,\n          38.9\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"aveOralM\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 0.509742705000394,\n        \"min\":
35.54,\n        \"max\": 40.34,\n        \"num_unique_values\": 70,\n
\"samples\": [\n          36.39,\n          36.59,\n          37.59\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    }\n  ]\n}","type":"dataframe","variable_name":"y"}
```


```
yM=y.drop(columns=["aveOralF"]) #features of our data set array
yM.head()
```


```
{"summary":"{\n  \"name\": \"yM\",\n  \"rows\": 1018,\n  \"fields\":
[\n    {\n      \"column\": \"aveOralM\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 0.509742705000394,\n
\"min\": 35.54,\n        \"max\": 40.34,\n
\"num_unique_values\": 70,\n        \"samples\": [\n          36.39,\n
36.59,\n          37.59\n        ],\n        \"semantic_type\": \"\",\
n        \"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"yM"}
```


yM is correct

Let's choose few features which seems to be better for the target, we can do a mean,std, and min,Q1,meadian,Q3,max analysis to choose features

```
X.describe()
```


```
{"type":"dataframe"}
```


```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1018 entries, 0 to 1019
Data columns (total 33 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Gender        1018 non-null   object
 1   Age           1018 non-null   object
 2   Ethnicity     1018 non-null   object
 3   T_atm         1018 non-null   float64
 4   Humidity      1018 non-null   float64
 5   Distance      1018 non-null   float64
 6   T_offset1     1018 non-null   float64
 7   Max1R13_1     1018 non-null   float64
 8   Max1L13_1     1018 non-null   float64
 9   aveAllR13_1   1018 non-null   float64
 10  aveAllL13_1   1018 non-null   float64
 11  T_RC1         1018 non-null   float64
 12  T_RC_Dry1     1018 non-null   float64
```

```
 13  T_RC_Wet1    1018 non-null   float64
 14  T_RC_Max1    1018 non-null   float64
 15  T_LC1        1018 non-null   float64
 16  T_LC_Dry1    1018 non-null   float64
 17  T_LC_Wet1    1018 non-null   float64
 18  T_LC_Max1    1018 non-null   float64
 19  RCC1         1018 non-null   float64
 20  LCC1         1018 non-null   float64
 21  canthiMax1   1018 non-null   float64
 22  canthi4Max1  1018 non-null   float64
 23  T_FHCC1      1018 non-null   float64
 24  T_FHRC1      1018 non-null   float64
 25  T_FHLC1      1018 non-null   float64
 26  T_FHBC1      1018 non-null   float64
 27  T_FHTC1      1018 non-null   float64
 28  T_FH_Max1    1018 non-null   float64
 29  T_FHC_Max1   1018 non-null   float64
 30  T_Max1       1018 non-null   float64
 31  T_OR1        1018 non-null   float64
 32  T_OR_Max1    1018 non-null   float64
dtypes: float64(30), object(3)
memory usage: 270.4+ KB
```

Here I decided to go with these features; "Age", "Distance" ,"T_offset1", "RCC1" , "LCC1"

```
XM = X[["Age", "Distance" ,"T_offset1", "RCC1" , "LCC1"]]
XM.head()
```

{"summary":"{\n \"name\": \"XM\",\n \"rows\": 1018,\n \"fields\": [\n   {\n   \"column\": \"Age\",\n   \"properties\": {\n   \"dtype\": \"category\",\n   \"num_unique_values\": 8,\n   \"samples\": [\n   \"31-40\",\n   \"21-25\",\n   \"41-50\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Distance\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 2.4564860370861084,\n   \"min\": 0.54,\n   \"max\": 79.0,\n   \"num_unique_values\": 33,\n   \"samples\": [\n   0.57,\n   0.63,\n   0.67\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"T_offset1\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.36285963777446373,\n   \"min\": -0.59,\n   \"max\": 2.875,\n   \"num_unique_values\": 467,\n   \"samples\": [\n   0.615,\n   0.95,\n   0.8525\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"RCC1\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.6065528156401492,\n   \"min\": 33.6175,\n   \"max\": 38.155,\n   \"num_unique_values\": 606,\n   \"samples\": [\n

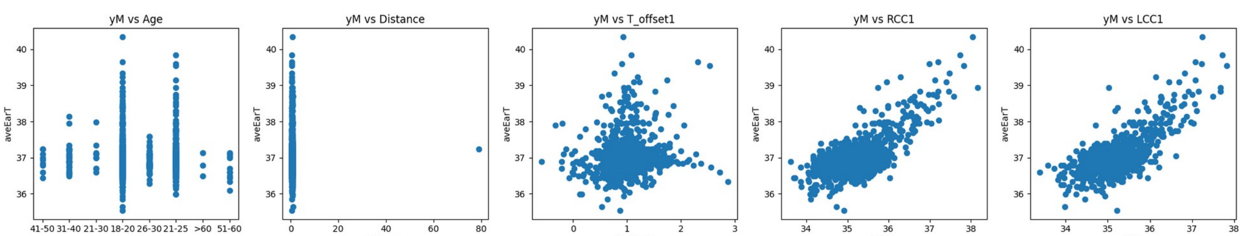
```
35.525,\n          34.94,\n          35.18\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"LCC1\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 0.5849279083765018,\n
\"min\": 33.385,\n        \"max\": 37.8275,\n
\"num_unique_values\": 616,\n        \"samples\": [\n          36.56,\
n          35.395,\n          35.0975\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe","variable_name":"XM"}
```


New (XM, yM) set seems to be better for the training session

```
print(XM.shape, yM.shape)

(1018, 5) (1018, 1)
```

Analysis for outlier detection and feture scaling:

```
# prompt: Give me 5 graphs which plots for y targets against X
features x1, x2, x3,x4 and x5

# Plotting yM against each of the five features in XM
fig, axes = plt.subplots(1, 5, figsize=(20, 4))

for i, feature in enumerate(XM.columns):
  axes[i].scatter(XM[feature], yM)
  axes[i].set_xlabel(feature)
  axes[i].set_ylabel('aveEarT')
  axes[i].set_title(f'yM vs {feature}')

plt.tight_layout()
plt.show()
```



Preparing the 1st feature "Age"

Preparing Age features: string to numpy.float64 conversion

```
# Extract the x1 to a separate dataframe from XM

x1 = XM[["Age"]]
x1.head()
```

```
{"summary":"{\n  \"name\": \"x1\",\n  \"rows\": 1018,\n  \"fields\": [\n    {\n      \"column\": \"Age\",\n      \"properties\": {\n        \"dtype\": \"category\",\n        \"num_unique_values\": 8,\n        \"samples\": [\n          \"31-40\",\n          \"21-25\",\n          \"41-50\"\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    }\n  ]\n}","type":"dataframe","variable_name":"x1"}
```


```
x1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1018 entries, 0 to 1019
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Age     1018 non-null   object
dtypes: object(1)
memory usage: 15.9+ KB
```

```
print(x1['Age'].unique())
```

```
['41-50' '31-40' '21-30' '18-20' '26-30' '21-25' '>60' '51-60']
```

```
# prompt: To convert x1 object dtype to float64; for each row contains
41-50 give 45.5 float64, for each row contains 31-40 give 35.5
float64, for each row contains 21-30 give 25.5 float64, for each row
contains 18-20 give 19 float64, for each row contains 26-30 give 28,
for each row contains 21-25 give 23, for each row contains >60 give
65, for each row contains 51-60 give 55.5

def convert_age(age_str):
  if age_str == '41-50':
    return 45.5
  elif age_str == '31-40':
    return 35.5
  elif age_str == '21-30':
    return 25.5
  elif age_str == '18-20':
    return 19.0
  elif age_str == '26-30':
    return 28.0
  elif age_str == '21-25':
    return 23.0
  elif age_str == '>60':
    return 65.0
  elif age_str == '51-60':
    return 55.5
  else:
    return None  # Handle unexpected values if needed

x1['Age'] = x1['Age'].apply(convert_age)
```

```
x1.head()
```

```
<ipython-input-21-eeac38d08b20>:23: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  x1['Age'] = x1['Age'].apply(convert_age)
```


{"summary":"{\n \"name\": \"x1\",\n \"rows\": 1018,\n \"fields\": [\n   {\n   \"column\": \"Age\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 5.956780440641162,\n   \"min\": 19.0,\n   \"max\": 65.0,\n   \"num_unique_values\": 8,\n   \"samples\": [\n   35.5,\n   23.0,\n   45.5\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe","variable_name":"x1"}


```
x1['Age'][1]
```

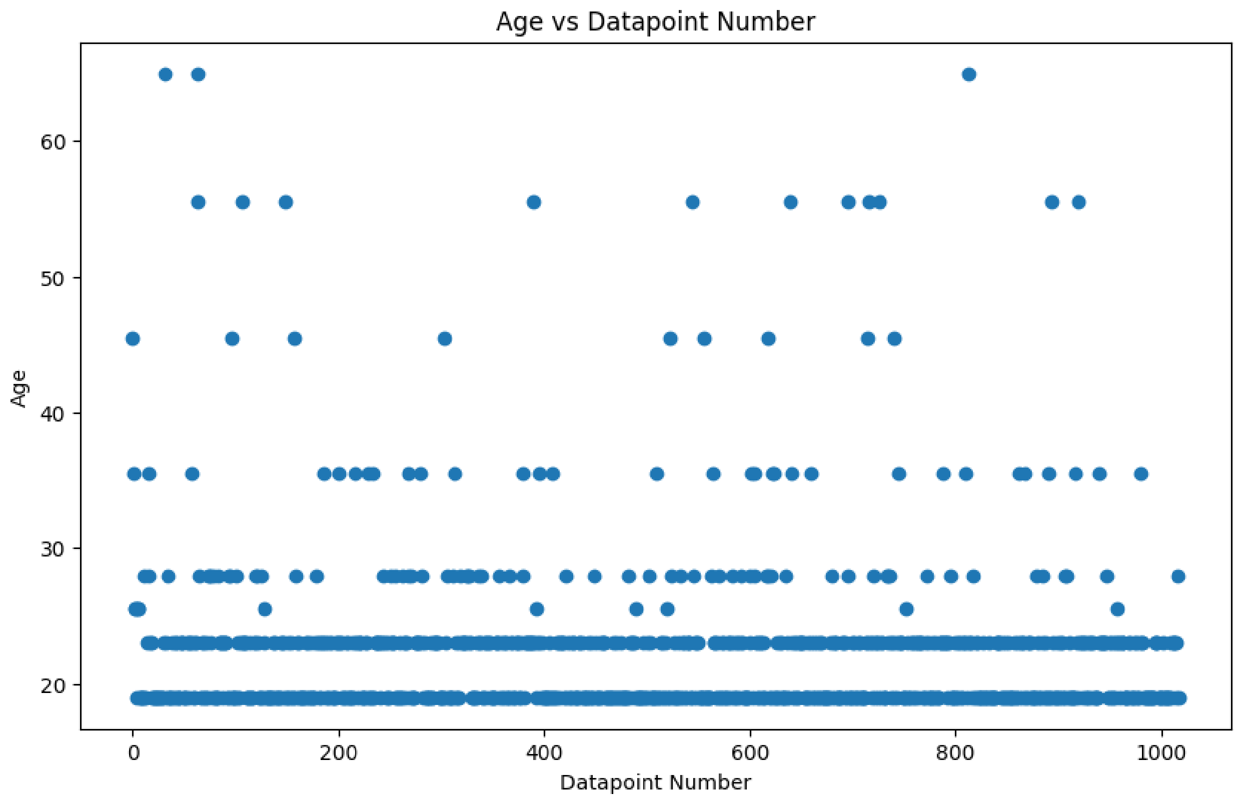
```
35.5
```

```
type(x1['Age'][1])
```

```
numpy.float64
```

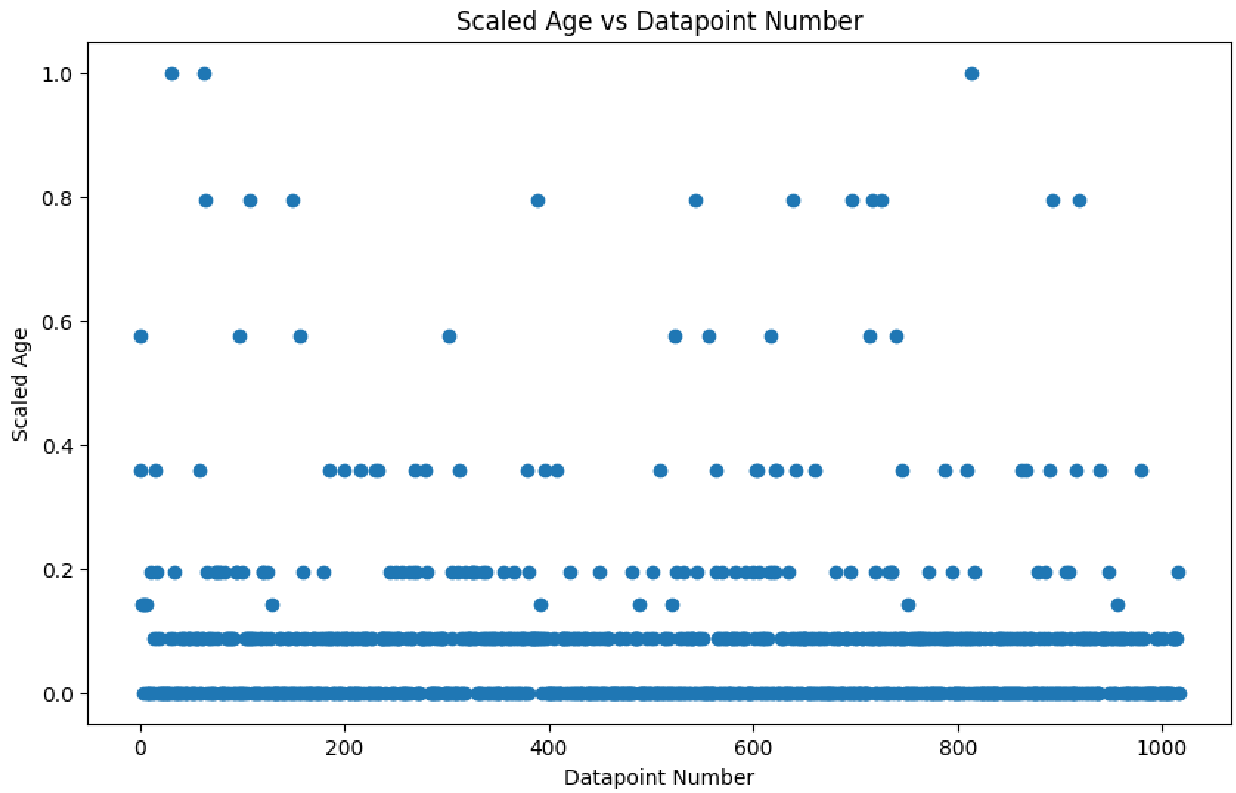
It is sucsessfuly converted in to numpy.float64

```
# prompt: plot new x1 features against datapoint number from 0 to 1018
as a dot graph

plt.figure(figsize=(10, 6))
plt.plot(range(len(x1)), x1['Age'], 'o')
plt.xlabel('Datapoint Number')
plt.ylabel('Age')
plt.title('Age vs Datapoint Number')
plt.show()
```



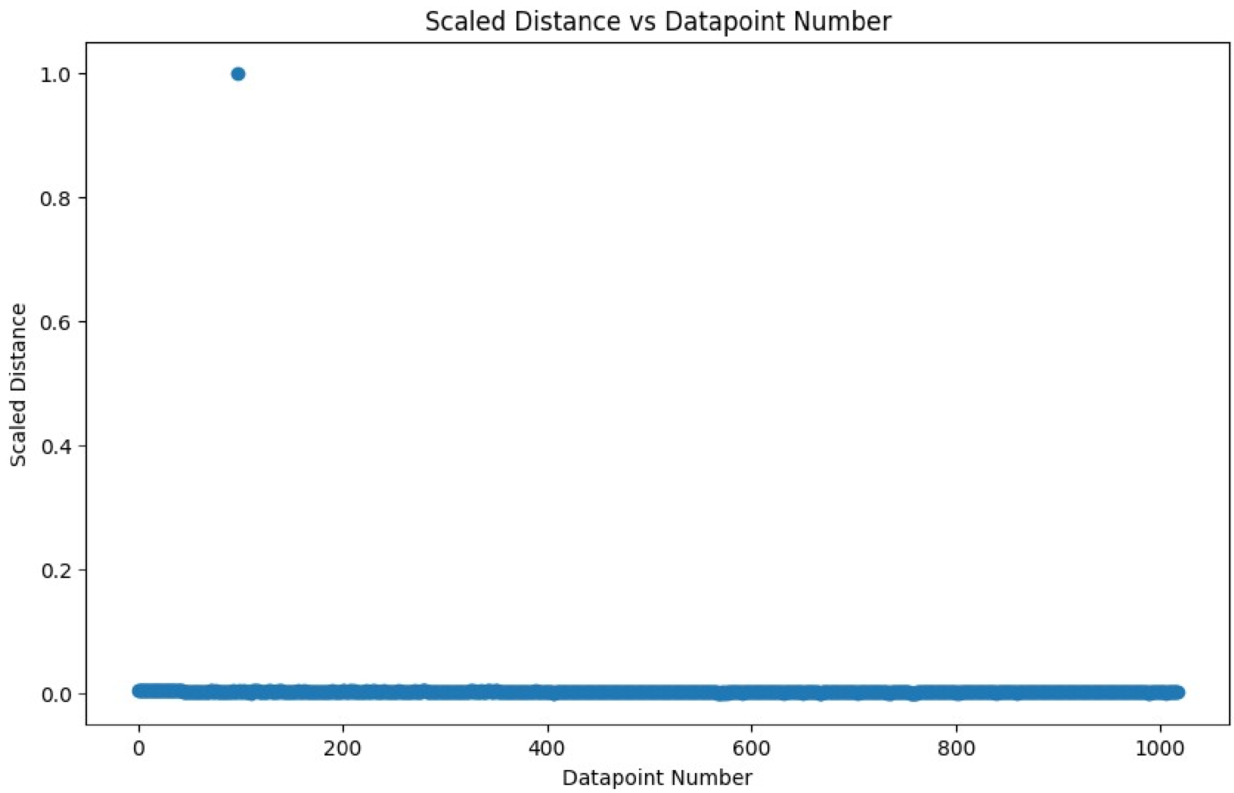
Scale the new 1st feature; Age = x1 with Min-Max-Scaling between 0 to 1 range, and plot scaled x1 feature in a plot against the datapoint number from 0 to 1018

```
#Scale the  new 1st feature; Age = x1 with Min-Max-Scaling between 0 to 1 range, and plot scaled x1 feature in a plot against the datapoint number from 0 to 1018

from sklearn.preprocessing import MinMaxScaler

# Reshape the data for the scaler (it expects a 2D array)
x1_values = x1['Age'].values.reshape(-1, 1)

# Create a MinMaxScaler object
scaler = MinMaxScaler()

# Fit the scaler to the data and transform it
x1_scaled = scaler.fit_transform(x1_values)

# Plot the scaled data
plt.figure(figsize=(10, 6))
plt.plot(range(len(x1_scaled)), x1_scaled, 'o')
plt.xlabel('Datapoint Number')
plt.ylabel('Scaled Age')
plt.title('Scaled Age vs Datapoint Number')
plt.show()
```



Preparing the 2nd feature "Distance"

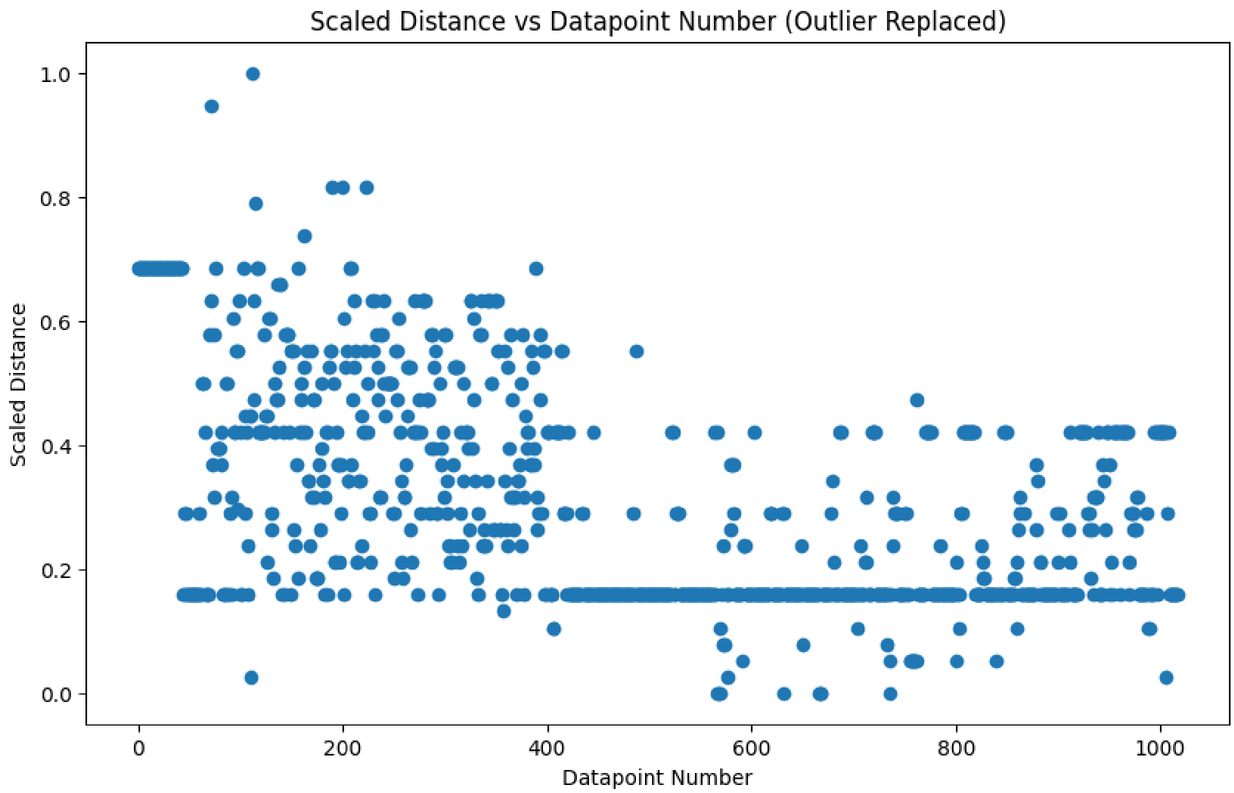
```
# Scale the 2nd feature; Distance = x2 in XM with Min-Max-Scaling
between 0 to 1 range, and plot scaled x2 feature in a plot against the
datapoint number from 0 to 1018

# Extract the 'Distance' feature
x2 = XM['Distance'].values.reshape(-1, 1)  # Reshape for scaler

# Create a MinMaxScaler object (if not already created)
# scaler = MinMaxScaler()  # Uncomment if scaler not defined earlier

# Fit and transform the 'Distance' feature
x2_scaled = scaler.fit_transform(x2)

# Plot the scaled feature against datapoint number
plt.figure(figsize=(10, 6))
plt.scatter(range(len(x2_scaled)), x2_scaled)
plt.xlabel('Datapoint Number')
plt.ylabel('Scaled Distance')
plt.title('Scaled Distance vs Datapoint Number')
plt.show()
```



Clearely we have a one outlier which affected the scaling

Let's remove the outlier;

```
# Identify the outlier index (where x2_scaled is very close to 1)
outlier_index = (x2_scaled > 0.99).flatten()

# Calculate the mean of non-outlier 'Distance' values
mean_distance = XM.loc[~outlier_index, 'Distance'].mean()

# Replace the outlier with the mean value
XM.loc[outlier_index, 'Distance'] = mean_distance

# Re-plot the updated data with another reshape in (-1,1)
x2_scaled_updated =
scaler.fit_transform(XM['Distance'].values.reshape(-1, 1))
plt.figure(figsize=(10, 6))
plt.scatter(range(len(x2_scaled_updated)), x2_scaled_updated)
plt.xlabel('Datapoint Number')
plt.ylabel('Scaled Distance')
plt.title('Scaled Distance vs Datapoint Number (Outlier Replaced)')
plt.show()
```

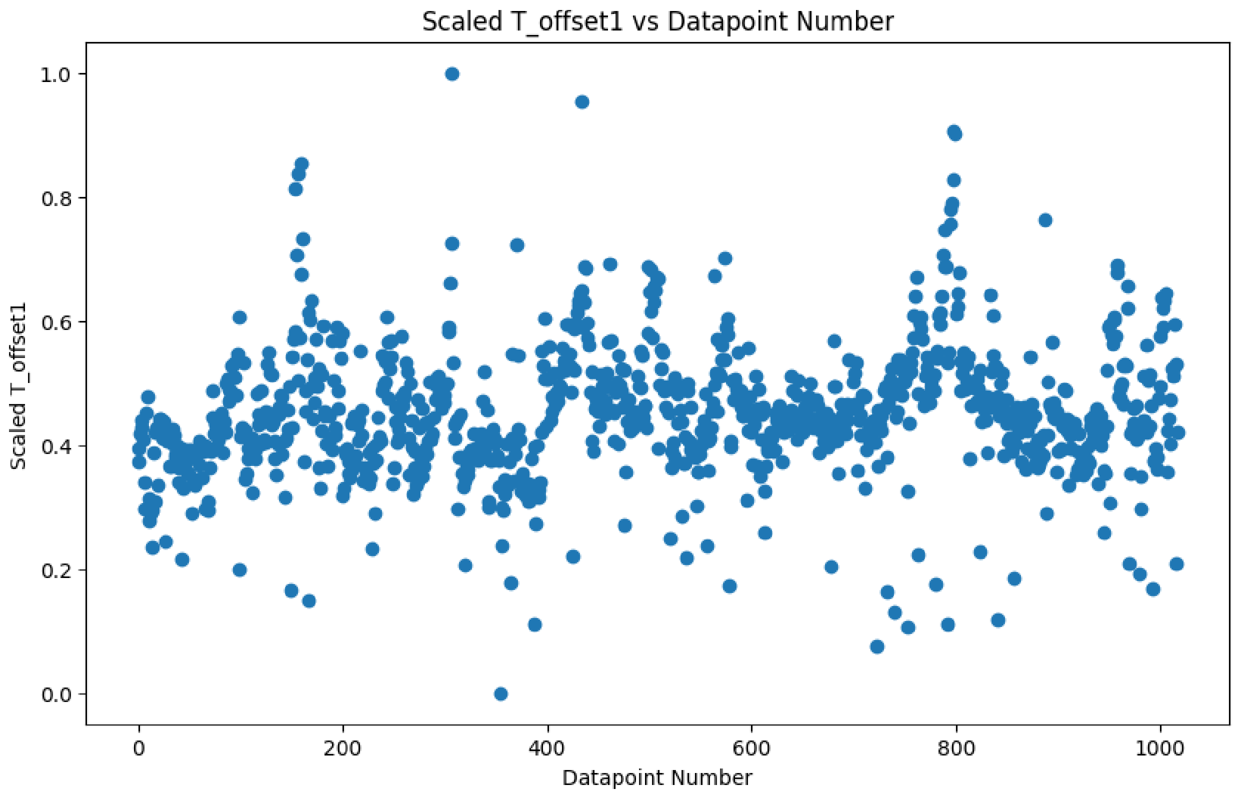



```
x2.shape
```

```
(1018, 1)
```

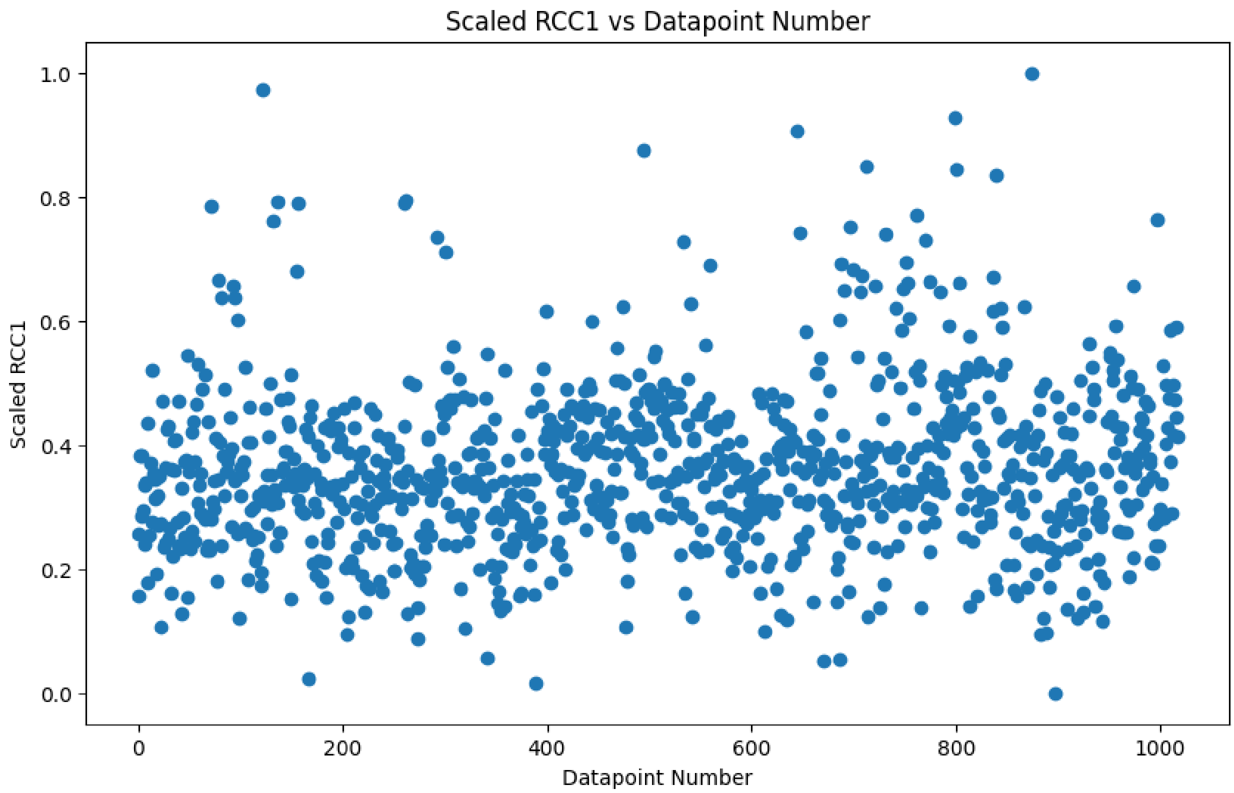
Preparing the 3rd feature "T_offset1"

```
# prompt: Scale the 3rd feature; T_offset1 = x3 in XM with Min-Max-
Scaling between 0 to 1 range, and plot scaled x3 feature in a plot
against the datapoint number from 0 to 1018

# Extract the 'T_offset1' feature
x3 = XM['T_offset1'].values.reshape(-1, 1)  # Reshape for scaler

# Create a MinMaxScaler object (if not already created)
# scaler = MinMaxScaler()  # Uncomment if scaler not defined earlier

# Fit and transform the 'T_offset1' feature
x3_scaled = scaler.fit_transform(x3)

# Plot the scaled feature against datapoint number
plt.figure(figsize=(10, 6))
plt.scatter(range(len(x3_scaled)), x3_scaled)
plt.xlabel('Datapoint Number')
plt.ylabel('Scaled T_offset1')
plt.title('Scaled T_offset1 vs Datapoint Number')
plt.show()
```



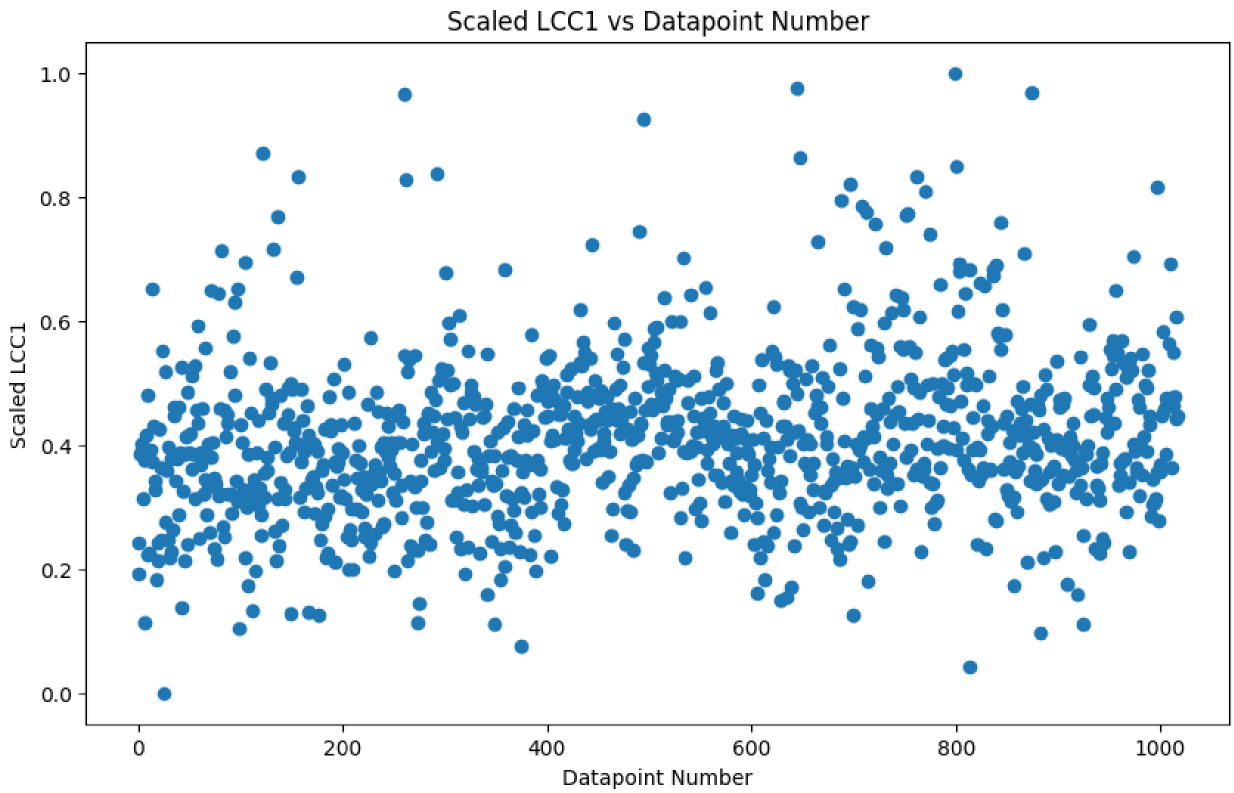
Preparing the 4th feature "RCC1"

```
# prompt: Scale the 4th feature; RCC1 = x4 in XM with Min-Max-Scaling
between 0 to 1 range, and plot scaled x4 feature in a plot against the
datapoint number from 0 to 1018

# Extract the 'RCC1' feature
x4 = XM['RCC1'].values.reshape(-1, 1)  # Reshape for scaler

# Create a MinMaxScaler object (if not already created)
# scaler = MinMaxScaler()  # Uncomment if scaler not defined earlier

# Fit and transform the 'RCC1' feature
x4_scaled = scaler.fit_transform(x4)

# Plot the scaled feature against datapoint number
plt.figure(figsize=(10, 6))
plt.scatter(range(len(x4_scaled)), x4_scaled)
plt.xlabel('Datapoint Number')
plt.ylabel('Scaled RCC1')
plt.title('Scaled RCC1 vs Datapoint Number')
plt.show()
```



Preparing the 5th feature "LCC1"

```
# prompt: Scale the 5th feature; LCC1 = x5 in XM with Min-Max-Scaling
between 0 to 1 range, and plot scaled x5 feature in a plot against the
datapoint number from 0 to 1018

# Extract the 'LCC1' feature
x5 = XM['LCC1'].values.reshape(-1, 1)  # Reshape for scaler

# Create a MinMaxScaler object (if not already created)
# scaler = MinMaxScaler()  # Uncomment if scaler not defined earlier

# Fit and transform the 'LCC1' feature
x5_scaled = scaler.fit_transform(x5)

# Plot the scaled feature against datapoint number
plt.figure(figsize=(10, 6))
plt.scatter(range(len(x5_scaled)), x5_scaled)
plt.xlabel('Datapoint Number')
plt.ylabel('Scaled LCC1')
plt.title('Scaled LCC1 vs Datapoint Number')
plt.show()
```




```
# prompt: Now combine scaled x1 to x5 features in to a new dataframe

# Create a new DataFrame from the scaled features
X_scaled_df = pd.DataFrame({
    'Age_scaled': x1_scaled.flatten(),
    'Distance_scaled': x2_scaled_updated.flatten(),
    'T_offset1_scaled': x3_scaled.flatten(),
    'RCC1_scaled': x4_scaled.flatten(),
    'LCC1_scaled': x5_scaled.flatten()
})

# Display the new DataFrame
X_scaled_df.head()
```

{"summary":"{\n \"name\": \"X_scaled_df\",\n \"rows\": 1018,\n \"fields\": [\n   {\n   \"column\": \"Age_scaled\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.12949522697046,\n   \"min\": 0.0,\n   \"max\": 1.0,\n   \"num_unique_values\": 8,\n   \"samples\": [\n   0.358695652173913,\n   0.08695652173913043,\n   0.576086956521739\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Distance_scaled\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.17806986543665435,\n   \"min\": 0.0,\n   \"max\": 1.0000000000000002,\n

```
\"num_unique_values\": 33,\n        \"samples\": [\n
0.07894736842105243,\n          0.23684210526315774,\n
0.3421052631578947\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"T_offset1_scaled\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 0.10472139618310641,\n        \"min\":
0.0,\n        \"max\": 1.0000000000000002,\n
\"num_unique_values\": 467,\n        \"samples\": [\n
0.3477633477633478,\n          0.4444444444444444,\n
0.41630591630591635\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"RCC1_scaled\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 0.13367555165623116,\n        \"min\":
0.0,\n        \"max\": 0.9999999999999991,\n
\"num_unique_values\": 606,\n        \"samples\": [\n
0.4203856749311292,\n          0.2914600550964179,\n
0.3443526170798892\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"LCC1_scaled\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 0.1316663834274623,\n        \"min\":
0.0,\n        \"max\": 1.0,\n        \"num_unique_values\": 616,\n
\"samples\": [\n          0.7146876758581886,\n
0.45244794597636595,\n          0.38548114800225086\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe","variable_name":"X_scaled_df"}
```


3.6 Split the data into training and testing sets with 80% of data points for training and 20% of data points for testing.

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score


# Split the data into training and test sets
X_train, X_test, Y_train, Y_test = train_test_split(X_scaled_df, yM,
test_size=0.2, random_state=3)
```

3.7 Train a linear regression model and estimate the coefficient corresponds to independent variables.

Building amd training the model

```
# Building the model
model = LinearRegression()
```

```
# Train the model
model.fit(X_train, Y_train)
```

```
LinearRegression()
```

Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2$$

Measures the average squared difference between actual ((y_i)) and predicted ((\hat{y}_i)) values. Lower MSE indicates better accuracy.

R-squared (R^2):

$$R^2 = 1 - \frac{\sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2}{\sum_{i=1}^{n} \left(y_i - \acute{y}\right)^2}$$

Indicates the proportion of variance in (y_i) explained by the model. Closer to 1 means a better fit.

These metrics assess **regression models**.

```
# Predict on the training data
Y_pred_train = model.predict(X_train)

# Predict on the test data
Y_pred_test = model.predict(X_test)

# Evaluate the model
train_mse = mean_squared_error(Y_train, Y_pred_train)
test_mse = mean_squared_error(Y_test, Y_pred_test)
train_r2 = r2_score(Y_train, Y_pred_train)
test_r2 = r2_score(Y_test, Y_pred_test)

# Print the evaluation metrics
print(f"Training MSE: {train_mse}")
print(f"Test MSE: {test_mse}")
print(f"Training R^2: {train_r2}")
print(f"Test R^2: {test_r2}")
```

```
Training MSE: 0.11635506930348234
Test MSE: 0.12546687552553934
Training R^2: 0.5268354374859148
Test R^2: 0.5976173840368082
```

MSE is close to 0 and R^2 is also higher, Predictions are good, but not a perfect model; the reason is the selected features.

Coefficient corresponds to independent variables:

```
# Extract the coefficients for the independent variables
coefficients = model.coef_

# Create a DataFrame to display the coefficients alongside the feature names
coef_df = pd.DataFrame({
    'Feature': X_scaled_df.columns,
    'Coefficient': coefficients.flatten()
})

# Display the coefficients
print(coef_df)
```

```
             Feature  Coefficient
0         Age_scaled     0.067448
1    Distance_scaled     0.062485
2   T_offset1_scaled    -0.512615
3        RCC1_scaled     1.606298
4        LCC1_scaled     1.372426
```

3.8 Which independent variable contributes highly for the dependent feature.

Since we used MinMaxScaler() for all features, the coefficients are directly comparable, making RCC1_scaled the strongest contributor. The independent variable that contributes the most to the dependent feature is RCC1_scaled with a coefficient of 1.606298.

3.9 Linear Regression Model Training Using selected (`T_OR1`, `T_OR_Max1`, `T_FHC_Max1`, `T_FH_Max1`) Infrared Temperature Features

```
y.head()
```


```
{"summary":"{\n  \"name\": \"y\",\n  \"rows\": 1018,\n  \"fields\": [\n    {\n      \"column\": \"aveOralF\",\n      \"properties\": {\n        \"dtype\": \"number\",\n        \"std\": 0.38655667866453125,\n        \"min\": 35.75,\n        \"max\": 39.6,\n        \"num_unique_values\": 53,\n        \"samples\": [\n          37.65,\n          39.2,\n          38.9\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"aveOralM\",\n      \"properties\": {\n        \"dtype\": \"number\",\n        \"std\": 0.509742705000394,\n        \"min\": 35.54,\n        \"max\": 40.34,\n        \"num_unique_values\": 70,\n        \"samples\": [\n          36.39,\n          36.59,\n          37.59\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    }\n  ]\n}","type":"dataframe","variable_name":"y"}
```

```
yNew=y.drop(columns=["aveOralF"])  # Target variable after dropping
'aveOralF'
yNew.head()
```


{"summary":"{\n \"name\": \"yNew\",\n \"rows\": 1018,\n \"fields\":
[\n   {\n   \"column\": \"aveOralM\",\n   \"properties\": {\n
\"dtype\": \"number\",\n   \"std\": 0.509742705000394,\n
\"min\": 35.54,\n   \"max\": 40.34,\n
\"num_unique_values\": 70,\n   \"samples\": [\n   36.39,\n
36.59,\n   37.59\n   ],\n   \"semantic_type\": \"\",\
n   \"description\": \"\"\n   }\n   }\n]\
n}","type":"dataframe","variable_name":"yNew"}


```
X.head()
```

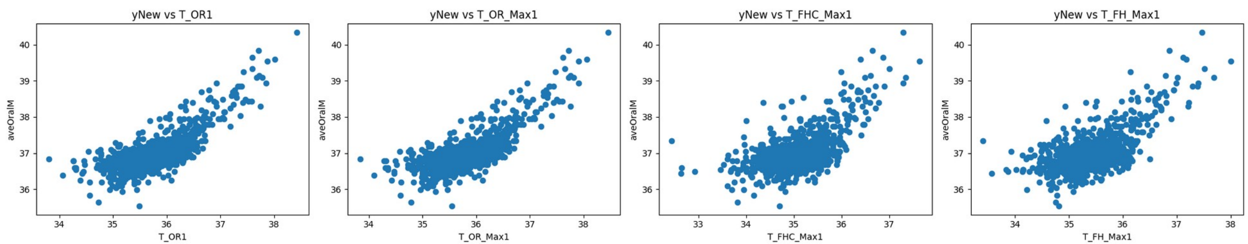

{"type":"dataframe","variable_name":"X"}


```
XNew = X[['T_OR1', 'T_OR_Max1', 'T_FHC_Max1', 'T_FH_Max1']]  #
Independent features
XNew.head()
```


{"summary":"{\n \"name\": \"XNew\",\n \"rows\": 1018,\n \"fields\":
[\n   {\n   \"column\": \"T_OR1\",\n   \"properties\": {\n
\"dtype\": \"number\",\n   \"std\": 0.5597552369624542,\n
\"min\": 33.8025,\n   \"max\": 38.4175,\n
\"num_unique_values\": 583,\n   \"samples\": [\n
36.9225,\n   35.26,\n   35.9925\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"T_OR_Max1\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
0.5598116134653163,\n   \"min\": 33.835,\n   \"max\":
38.455,\n   \"num_unique_values\": 587,\n   \"samples\": [\n
36.3275,\n   35.6425,\n   37.0575\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"T_FHC_Max1\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
0.576283115077168,\n   \"min\": 32.44,\n   \"max\":
37.6325,\n   \"num_unique_values\": 611,\n   \"samples\": [\
n   35.0475,\n   34.5375,\n   35.0175\
n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"T_FH_Max1\",\n   \"properties\": {\n   \"dtype\":
\"number\",\n   \"std\": 0.5237026164644623,\n   \"min\":
33.4075,\n   \"max\": 38.0025,\n   \"num_unique_values\":
558,\n   \"samples\": [\n   34.925,\n   34.565,\n
35.355\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   }\n]\
n}","type":"dataframe","variable_name":"XNew"}


```
# Plotting yNew against each of the four features in XNew
fig, axes = plt.subplots(1, 4, figsize=(20, 4))  # Adjusted to 4
```

```python
features

for i, feature in enumerate(XNew.columns):
    axes[i].scatter(XNew[feature], yNew)
    axes[i].set_xlabel(feature)
    axes[i].set_ylabel('aveOralM')
    axes[i].set_title(f'yNew vs {feature}')

plt.tight_layout()
plt.show()
```



The **RobustScaler** scales the features by removing the median and dividing by the interquartile range (IQR), making it robust to outliers. Since the original features are in the range of 0 to 40, this scaling ensures that the features are standardized without being affected by outliers, and the resulting scaled values can be effectively compared.

For each feature X_i:

$$X_{\text{scaled}} = \frac{X_i - \text{median}(X_i)}{\text{IQR}(X_i)}$$

Where IQR (Interquartile Range) is the range between the 25th and 75th percentiles.

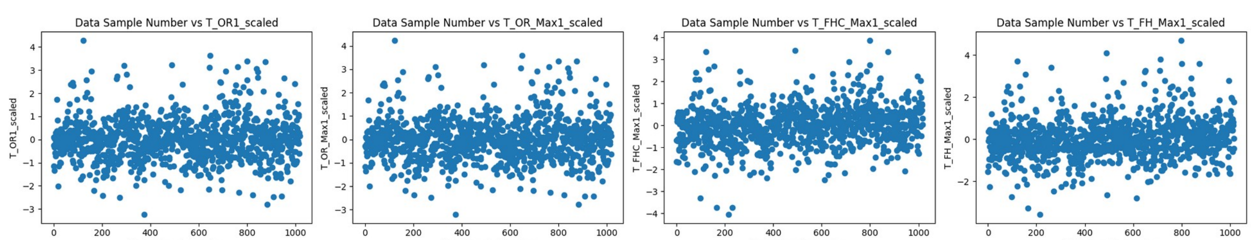
```python
from sklearn.preprocessing import RobustScaler
import pandas as pd

# Initialize RobustScaler
scaler = RobustScaler()

# Apply RobustScaler to XNew
XNew_scaled = pd.DataFrame(scaler.fit_transform(XNew),
columns=XNew.columns)

# Display the first few rows of the scaled dataframe
XNew_scaled.head()
```


```
{"summary":"{\n  \"name\": \"XNew_scaled\",\n  \"rows\": 1018,\n
\"fields\": [\n    {\n      \"column\": \"T_OR1\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.9110970286265748,\n        \"min\": -3.232960325534063,\n
\"max\": 4.2787385554425,\n        \"num_unique_values\": 583,\n
\"samples\": [\n          1.8453713123092483,\n          -
```

```
0.8606307222787387,\n         0.331637843336722\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"T_OR_Max1\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.9001995794417106,\n        \"min\": -3.1999999999999886,\n
\"max\": 4.229145728643189,\n        \"num_unique_values\": 587,\n
\"samples\": [\n          0.8080402010050177,\n          -
0.29346733668342306,\n          1.9819095477386757\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"T_FHC_Max1\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.8773101656740909,\n        \"min\": -4.049476688867754,\n
\"max\": 3.8553758325404393,\n        \"num_unique_values\": 611,\n
\"samples\": [\n          -0.07992388201712965,\n          -
0.8563273073263566,\n          -0.12559467174120373\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"T_FH_Max1\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.9414878498237547,\n        \"min\": -3.5685393258427047,\n
\"max\": 4.692134831460685,\n        \"num_unique_values\": 558,\n
\"samples\": [\n          -0.8404494382022514,\n          -
1.4876404494382072,\n          -0.06741573033708138\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe","variable_name":"XNew_scaled"}
```

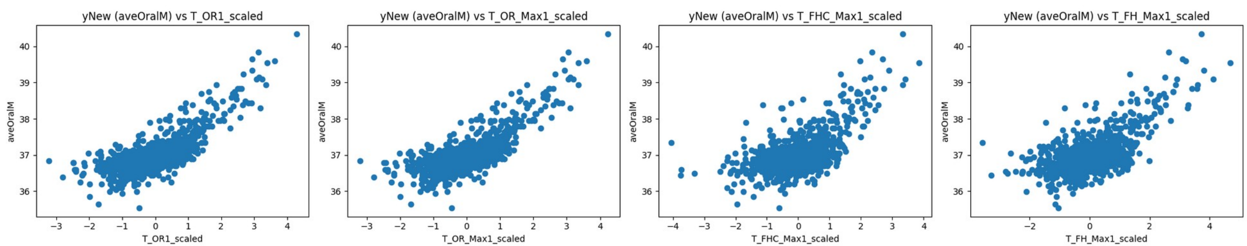

Plot (Feature vs. Data Sample Number: the row index)

```
# Plotting each feature in XNew_scaled against its data sample number
(index)
fig, axes = plt.subplots(1, 4, figsize=(20, 4))  # Adjusted to 4
features

for i, feature in enumerate(XNew_scaled.columns):
    axes[i].scatter(XNew_scaled.index, XNew_scaled[feature])
    axes[i].set_xlabel('Data Sample Number')
    axes[i].set_ylabel(f'{feature}_scaled')
    axes[i].set_title(f'Data Sample Number vs {feature}_scaled')

plt.tight_layout()
plt.show()
```



Plot (Feature vs. aveOralM):

```
# Plotting yNew (aveOralM) against each of the scaled features in
XNew_scaled
fig, axes = plt.subplots(1, 4, figsize=(20, 4))  # Adjusted to 4
features

for i, feature in enumerate(XNew_scaled.columns):
    axes[i].scatter(XNew_scaled[feature], yNew)
    axes[i].set_xlabel(feature + '_scaled')
    axes[i].set_ylabel('aveOralM')
    axes[i].set_title(f'yNew (aveOralM) vs {feature}_scaled')

plt.tight_layout()
plt.show()
```



Train Test Split

```
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets (80% training, 20%
testing)
X_train, X_test, y_train, y_test = train_test_split(XNew_scaled, yNew,
test_size=0.2, random_state=42)
```

Building the model

```
from sklearn.linear_model import LinearRegression

# Initialize the linear regression model
model = LinearRegression()
```

Training the model

```
# Train the model using the training data
model.fit(X_train, y_train)

LinearRegression()
```

Evaluation of the model

```python
from sklearn.metrics import mean_squared_error, r2_score
# Predict on the training data
y_pred_train = model.predict(X_train)

# Predict on the test data
y_pred_test = model.predict(X_test)

# Evaluate the model
train_mse = mean_squared_error(y_train, y_pred_train)
test_mse = mean_squared_error(y_test, y_pred_test)
train_r2 = r2_score(y_train, y_pred_train)
test_r2 = r2_score(y_test, y_pred_test)

# Print the evaluation metrics
print(f"Training MSE: {train_mse}")
print(f"Test MSE: {test_mse}")
print(f"Training R^2: {train_r2}")
print(f"Test R^2: {test_r2}")
```

```
Training MSE: 0.0957917577746669
Test MSE: 0.07805588114890863
Training R^2: 0.6513378726673114
Test R^2: 0.6076047374475761
```

Clearly, the new model performs better than the previous model. The previous model had a **Training MSE** of **0.1164** and **Test MSE** of **0.1255**, with **Training R^2** of **0.5268** and **Test R^2** of **0.5976**. In contrast, the new model shows improved performance with a **Training MSE** of **0.0958**, **Test MSE** of **0.0781**, **Training R^2** of **0.6513**, and **Test R^2** of **0.6076**. The new feature selection (`T_OR1`, `T_OR_Max1`, `T_FHC_Max1`, `T_FH_Max1`) is more effective than the previous one (`Age`, `Distance`, `T_offset1`, `RCC1`, `LCC1`).

The coefficients of each feature

```python
import pandas as pd

# Extract the coefficients for the independent variables and flatten
them
coefficients = model.coef_.flatten()

# Create a DataFrame to display the coefficients alongside the feature
names
coef_df = pd.DataFrame({
    'Feature': XNew_scaled.columns,
    'Coefficient': coefficients
})

# Display the coefficients
print(coef_df)
```

```
      Feature  Coefficient
0        T_OR1     0.126228
1    T_OR_Max1     0.216535
2   T_FHC_Max1    -0.054993
3    T_FH_Max1     0.209464
```

3.10 Calculating the evaluation parameters

```
import numpy as np
import statsmodels.api as sm

# Reset indices of X_train and y_train to ensure alignment
X_train_sm = sm.add_constant(X_train.reset_index(drop=True))  #
Resetting the index of X_train
y_train_sm = y_train.reset_index(drop=True)  # Resetting the index of
y_train

# Fit the model using OLS (Ordinary Least Squares) from statsmodels
ols_model = sm.OLS(y_train_sm, X_train_sm).fit()

# Predictions on the test set (for RSS and RSE calculations)
y_pred_test = model.predict(X_test)

# Summary of the OLS model (for standard error, t-statistic, p-value)
print("\nOLS Model Summary:")
print(ols_model.summary())

# Extracting Standard Error, t-statistics, and p-values for each
feature
standard_errors = ols_model.bse
t_statistics = ols_model.tvalues
p_values = ols_model.pvalues
```

```
OLS Model Summary:
                            OLS Regression Results                            
======================================================================
========
Dep. Variable:               aveOralM   R-squared:                       
0.651
Model:                            OLS   Adj. R-squared:                  
0.650
Method:                 Least Squares   F-statistic:                     
377.8
Date:                Mon, 09 Sep 2024   Prob (F-statistic):          
2.11e-183
Time:                        09:07:00   Log-Likelihood:                
-200.37
No. Observations:                 814   AIC:
```

```
410.7
Df Residuals:                     809   BIC:
434.2
Df Model:                           4

Covariance Type:            nonrobust

==============================================================================
========
                 coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
--------
const         37.0098      0.012   3208.977      0.000      36.987
37.032
T_OR1          0.1262      0.548      0.230      0.818      -0.950
1.203
T_OR_Max1      0.2165      0.554      0.391      0.696      -0.870
1.303
T_FHC_Max1    -0.0550      0.029     -1.921      0.055      -0.111
0.001
T_FH_Max1      0.2095      0.027      7.756      0.000       0.156
0.262
==============================================================================
========
Omnibus:                       49.892   Durbin-Watson:
2.020
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
76.197
Skew:                           0.481   Prob(JB):
2.84e-17
Kurtosis:                       4.149   Cond. No.
116.
==============================================================================
========

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.

import numpy as np
import statsmodels.api as sm

# Reset indices of X_train and y_train to ensure alignment
X_train_sm = sm.add_constant(X_train.reset_index(drop=True))  #
Resetting the index of X_train
y_train_sm = y_train.reset_index(drop=True)  # Resetting the index of
y_train

# Fit the model using OLS (Ordinary Least Squares) from statsmodels
```

```
ols_model = sm.OLS(y_train_sm, X_train_sm).fit()

# Predictions on the test set (for RSS and RSE calculations)
y_pred_test = model.predict(X_test)
```

---

1. Residual Sum of Squares (RSS)

Equation and Short Explanation:

$$\mathrm{RSS} = \sum_{i=1}^{n} \left(y_i - \hat{y}_i \right)^2$$

- **Explanation:** RSS is the sum of the squared differences between the actual values ((y_i)) and the predicted values ((\hat{y}_i)). It measures the total deviation of the response variable from the fitted regression line. Lower RSS means a better fit.

- **Implementation:**

```
# Residual Sum of Squares (RSS)
RSS = np.sum((y_test - y_pred_test) ** 2)
print(f"Residual Sum of Squares (RSS): {RSS}")

Residual Sum of Squares (RSS): aveOralM    15.9234
dtype: float64

/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:86:
FutureWarning: The behavior of DataFrame.sum with axis=None is
deprecated, in a future version this will reduce over both axes and
return a scalar. To retain the old behavior, pass axis=0 (or do not
pass axis)
  return reduction(axis=axis, out=out, **passkwargs)
```

---

2. Residual Standard Error (RSE)

Equation and Short Explanation:

$$\mathrm{RSE} = \sqrt{\frac{\mathrm{RSS}}{n - p - 1}}$$

- **Explanation:** RSE provides an estimate of the standard deviation of the residuals (errors). It adjusts RSS by the number of data points ((n)) and the number of features ((p)) to give a measure of how much the predicted values typically deviate from the actual values.

- **Implementation:**

```python
# Residual Standard Error (RSE)
n = len(y_test)  # Number of data points
p = X_test.shape[1]  # Number of features
RSE = np.sqrt(RSS / (n - p - 1))
print(f"Residual Standard Error (RSE): {RSE}")
```

```
Residual Standard Error (RSE): aveOralM    0.282873
dtype: float64
```

---

3. Mean Squared Error (MSE)

Equation and Short Explanation:

$$\mathrm{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{y}_i \right)^2$$

- **Explanation:** MSE is the average of the squared differences between the actual and predicted values. It provides a measure of how well the regression model is performing, with lower values indicating better performance.

- **Implementation:**

```python
# Mean Squared Error (MSE)
MSE = np.mean((y_test - y_pred_test) ** 2)
print(f"Mean Squared Error (MSE): {MSE}")
```

```
Mean Squared Error (MSE): 0.07805588114890863
```

---

4. R-squared (R^2) Statistic

Equation and Short Explanation:

$$R^2 = 1 - \frac{\mathrm{RSS}}{\mathrm{TSS}}, \mathrm{TSS} = \sum_{i=1}^{n} \left(y_i - \acute{y} \right)^2$$

- **Explanation:** R^2 measures the proportion of the variance in the dependent variable that is predictable from the independent variables. A value closer to 1 indicates a better fit.

- **Implementation:**

```python
# R-squared (R²) Statistic
TSS = np.sum((y_test - np.mean(y_test)) ** 2)
```

```
R_squared = 1 - (RSS / TSS)
print(f"R-squared (R²): {R_squared}")

R-squared (R²): aveOralM    0.607605
dtype: float64
```

---

5. Standard Error for Each Feature

Equation and Short Explanation:

$$SE\left(\hat{\beta}_j\right)=\frac{\hat{\sigma}}{\sqrt{\sum\left(X_j-\acute{X}_j\right)^2}}$$

- **Explanation:** The standard error measures the accuracy with which a sample distribution represents the true population. A smaller standard error indicates more precise estimates for the regression coefficients.

- **Implementation:**

```
# Standard Error for each feature
standard_errors = ols_model.bse
print("\nStandard Errors for each feature:")
for feature, se in zip(XNew_scaled.columns, standard_errors[1:]):  #
[1:] skips the intercept
    print(f"{feature}: {se}")


Standard Errors for each feature:
T_OR1: 0.5483826352425681
T_OR_Max1: 0.5537006858888935
T_FHC_Max1: 0.028622526603597744
T_FH_Max1: 0.027006212574507676
```

---

6. t-statistic for Each Feature

Equation and Short Explanation:

$$t_j=\frac{\hat{\beta}_j}{SE\left(\hat{\beta}_j\right)}$$

- **Explanation:** The t-statistic tests whether a coefficient is significantly different from zero. A larger absolute t-statistic indicates that the feature is more likely to be a meaningful predictor.

- **Implementation:**

```
# t-statistics for each feature
t_statistics = ols_model.tvalues
print("\nt-statistics for each feature:")
for feature, t_stat in zip(XNew_scaled.columns, t_statistics[1:]):  #
[1:] skips the intercept
    print(f"{feature}: {t_stat}")


t-statistics for each feature:
T_OR1: 0.23018254988083564
T_OR_Max1: 0.3910685273806129
T_FHC_Max1: -1.9213037621542028
T_FH_Max1: 7.75613813658415
```

---

7. p-value for Each Feature

Equation and Short Explanation:

- The **p-value** is computed from the t-statistic and indicates the probability that the observed data could occur under the null hypothesis (that the coefficient is zero). A smaller p-value (usually < 0.05) suggests that the feature is statistically significant.

- **Implementation:**

```
# p-values for each feature
p_values = ols_model.pvalues
print("\np-values for each feature:")
for feature, p_value in zip(XNew_scaled.columns, p_values[1:]):  #
[1:] skips the intercept
    print(f"{feature}: {p_value}")


p-values for each feature:
T_OR1: 0.8180081112907768
T_OR_Max1: 0.6958495517079573
T_FHC_Max1: 0.05504450887252779
T_FH_Max1: 2.6339932537041413e-14
```

3.11 Discarding features based on p-value:

Explanation:

In regression analysis, the **p-value** helps us determine whether a feature is statistically significant in predicting the dependent variable. Typically, a p-value threshold of **0.05** is used:

- If the **p-value < 0.05**, the feature is considered **statistically significant**.
- If the **p-value ≥ 0.05**, the feature is **not statistically significant**, meaning it may not contribute much to the model's predictive power and could potentially be discarded.

Analysis of p-values:

- **T_OR1 (p-value = 0.8180):** This feature has a very high p-value (0.818), indicating that it is not statistically significant. It could likely be discarded from the model.
- **T_OR_Max1 (p-value = 0.6958):** This feature also has a high p-value (0.6958), suggesting it is not statistically significant and could potentially be discarded.
- **T_FHC_Max1 (p-value = 0.0550):** The p-value here is close to 0.05. While it is not quite statistically significant at the strict 0.05 level, it is marginally close. It may still hold some value depending on the context.
- **T_FH_Max1 (p-value = 2.63e-14):** This feature has an extremely low p-value, far below 0.05. It is highly statistically significant and should be kept in the model.

Answer:

- **T_OR1** and **T_OR_Max1** have high p-values and can likely be discarded as they are not statistically significant.
- **T_FHC_Max1** is borderline, so it could be further evaluated based on other model performance metrics (e.g., R^2) or domain knowledge.
- **T_FH_Max1** is statistically significant and should **definitely be retained** in the model.

---

4. Performance Evaluation of Linear Regression

1] The two linear regression models, A and B:

In this section, we compare A and B models using residual standard error (RSE) and R-squared (R^2) values based on the given SSE and TSS table.

2] Residual Standard Error (RSE)

RSE is calculated using the formula:

$$RSE=\sqrt{\frac{SSE}{N-d-1}}$$

Where:

- SSE is the Sum of Squared Errors.
- N is the number of data samples.
- d is the number of independent features.

Using the given values:

- **For Model A :** $SSE=9$, $N = 10000$, and $d = 2$ (since there are two independent variables).

$$RSE_A=\sqrt{\frac{9}{10000-2-1}}=\sqrt{\frac{9}{9997}}\approx 0.03$$

- **For Model B :** $SSE = 2$, $N = 10000$, and $d = 4$ (since there are four independent variables).

$$RSE_B=\sqrt{\frac{2}{10000-4-1}}=\sqrt{\frac{2}{9995}}\approx 0.014$$

Thus, **Model B** has a lower RSE and performs better based on this metric.

3] R-squared (R^2)

R^2 is calculated as:

$$R^2=1-\frac{SSE}{TSS}$$

Where:

- TSS is the Total Sum of Squares.

Using the given values:

- **For Model A :** $SSE = 9$, $TSS = 90$

$$R_A^2=1-\frac{9}{90}=1-0.1=0.9$$

- **For Model B :** $SSE = 2$, $TSS = 10$

$$R_B^2=1-\frac{2}{10}=1-0.2=0.8$$

Based on R^2, **Model A** performs better.

4] Conclusion:

- RSE suggests that **Model B** performs better as it has a lower error.
- R^2 suggests that **Model A** performs better, capturing more variance.
- When comparing two models with the same large dataset size (10,000 samples in this case), **R-squared (R^2)** is the more fair and reliable metric. This is because R^2 accounts for the proportion of variance explained by the model, making it independent of dataset size and the scale of errors. On the other hand, **RSE** does not account for the inherent size of the data and can result in higher error values for larger datasets, leading to potentially misleading conclusions. Thus Model A can cosider as the better one, since R^2 provides a better comparative insight into model performance, especially for large datasets.

5. Linear Regression Impact on Outliers

Linear regression is sensitive to outliers because it minimizes the sum of squared residuals. Large residuals (outliers) can disproportionately influence the fitting the model.

1] Modified Loss Functions

To reduce the impact of outliers, two modified loss functions $ L1(w) $ and $ L2(w) $ are introduced. Both functions include a hyperparameter $ a $ that controls the influence of outliers.

Behavior of Loss Functions Compared to Standard Square Loss

The **Standard Square Loss** (also known as Mean Squared Error or MSE) is given by the following equation:

$$\mathrm{MSE}(w)=\frac{1}{N}\sum_{i=1}^{N}(r_i)^2$$

Where:

- $ N $ is the number of data points,
- $ r_i $ is the residual or error between the predicted and actual values.

1. $L1(w)$ **Function:**

$$L1(w)=\frac{1}{N}\sum_{i=1}^{N}\frac{r_i^2}{a^2+r_i^2}$$

- For small residuals (r_i), $L1(w)$ behaves similarly to the standard square loss.
- For large residuals, $L1(w)$ reduces the influence more gradually, as r_i^2 becomes relatively smaller compared to a^2, effectively limiting the impact of large outliers.

1. $L2(w)$ **Function:**

$$L2(w)=\frac{1}{N}\sum_{i=1}^{N}\left(1-\exp\left(-2\frac{|r_i|}{a}\right)\right)$$

- For small residuals, $L2(w)$ mimics the behavior of the standard square loss.
- For large residuals, $L2(w)$ quickly diminishes their impact by exponentially reducing the influence, making it highly robust to outliers compared to the standard square loss.

Influence of Parameter $ a $

2] What happens when a → 0

Considering $ L_1(w) $, we see that as $ a \to 0 $, the equation for the loss:

$$L_1(w) = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{r_i^2}{a^2 + r_i^2}\right)$$

The limit of $ L_1(w) $ as $ a \to 0 $ is:

$$\lim_{a \to 0} L_1(w) = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{r_i^2}{r_i^2}\right) = \frac{1}{N}\sum_{i=1}^{N} 1 = 1$$

When the residual $ r_i $ is large relative to $ a $, as $ a \to 0 $, more residuals exhibit this behavior, and the loss for most residuals approaches 1.

Similarly, considering $ L_2(w) $, the equation for the loss is:

$$L_2(w) = \frac{1}{N}\sum_{i=1}^{N}\left(1 - \exp\left(-\frac{2|r_i|}{a}\right)\right)$$

The limit of $ L_2(w) $ as $ a \to 0 $ is:

$$\lim_{a \to 0} L_2(w) = \frac{1}{N}\sum_{i=1}^{N}\left(1 - \exp\left(-\frac{2|r_i|}{a}\right)\right) = \frac{1}{N}\sum_{i=1}^{N} 1 = 1$$

Conclusion

Thus, as $ a \to 0 $, both $ L_1(w) $ and $ L_2(w) $ approach 1, indicating that the loss becomes constant and insensitive to residuals, in contrast to standard loss functions, which increase indefinitely with residual size.

3] **Minimizing influence for $ |r_i| \geq 40 $**:

To minimize the impact of residuals greater than or equal to 40, we can select $5 \le a \le 30$ with $L_1(w)$. This is because, for non outlier larger residuals, $L_1(w)$ produces a lower loss compared to $L_2(w)$. When the residual $|r_i|$ reaches 40, we want the loss to remain at 1 and not increase further for larger residuals. For a values smaller than 5, it considers small residuals, even those smaller than 20 — as outliers, with the maximum loss value of 1. When a exceeds 30, even outliers greater than 40 receive low loss values below 0.6.

Here according to the given graph L1(w) with a=25 performs better for this.

Test:

```
import numpy as np

# Example data
residuals = np.array([12, -2.3, 27, 5.5, 70])  # Example residuals
a = 25  # Parameter 'a' for L1 and L2

# Standard Square Loss (Mean Squared Error)
def mse_loss(residuals):
```

```
    return np.mean(residuals ** 2)

# L1(w) Loss Function
def l1_loss(residuals, a):
    return np.mean((residuals ** 2) / (a ** 2 + residuals ** 2))

# L2(w) Loss Function
def l2_loss(residuals, a):
    return np.mean(1 - np.exp(-2 * np.abs(residuals) / a))

# Calculate each loss
mse = mse_loss(residuals)
l1 = l1_loss(residuals, a)
l2 = l2_loss(residuals, a)

print(f"Standard Square Loss (MSE): {mse}")
print(f"L1(w) Loss: {l1}")
print(f"L2(w) Loss: {l2}")
```

```
Standard Square Loss (MSE): 1161.708
L1(w) Loss: 0.33341945585343397
L2(w) Loss: 0.6044223808721059
```

This implementation demonstrates L1(w) with a=25 performs better than MSE and L2(w) with residuals set [12, -2.3, 27, 5.5, 70] where I added outlier 70.

Python Code Implementations for Each Loss Function

Here is a code snippet that implements the **Standard Square Loss**, $L1(w)$, and $L2(w)$ functions.

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

# Define r values from -60 to 60
r_values = np.linspace(-60, 60, 500)

# Standard Square Loss (MSE)
def mse_loss(r_values):
    return r_values ** 2

# L1(w) Loss Function
def l1_loss(r_values, a):
    return (r_values ** 2) / (a ** 2 + r_values ** 2)

# L2(w) Loss Function
def l2_loss(r_values, a):
    return 1 - np.exp(-2 * np.abs(r_values) / a)

# Values of 'a' for L1 and L2 losses
a_values = [2, 5, 10, 20, 30, 50]
```

```python
# Create subplots
fig, axs = plt.subplots(1, 3, figsize=(18, 6))

# Plot Standard Square Loss (MSE) on the first subplot
axs[0].plot(r_values, mse_loss(r_values), label="MSE", color="blue")
axs[0].set_title("Standard Square Loss (MSE)")
axs[0].set_xlabel("Residual r")
axs[0].set_ylabel("L0(r)")
axs[0].grid(True)

# Plot L1(w) Loss for different values of 'a' on the second subplot
for a in a_values:
    axs[1].plot(r_values, l1_loss(r_values, a), label=f"a = {a}")
axs[1].set_title("L1(w) Loss")
axs[1].set_xlabel("Residual r")
axs[1].set_ylabel("L1(r)")
axs[1].grid(True)
axs[1].legend()

# Plot L2(w) Loss for different values of 'a' on the third subplot
for a in a_values:
    axs[2].plot(r_values, l2_loss(r_values, a), label=f"a = {a}")
axs[2].set_title("L2(w) Loss")
axs[2].set_xlabel("Residual r")
axs[2].set_ylabel("L2(r)")
axs[2].grid(True)
axs[2].legend()

# Display the plots
plt.tight_layout()
plt.show()
```