Task 2. Feature engineering + Clustering + Outlier detection

Yevgen Petronyuk Pidhaynyy

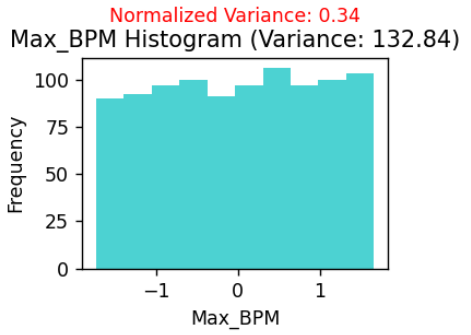
---

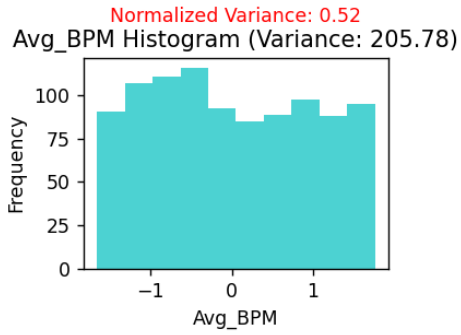
1. Dimension Reduction

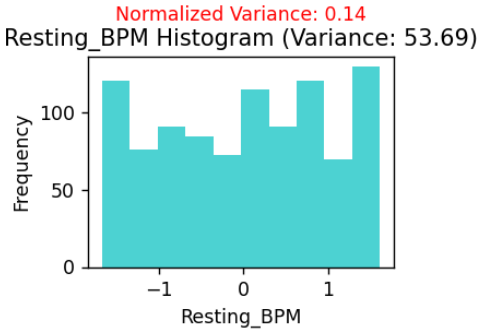
1.1 PCA

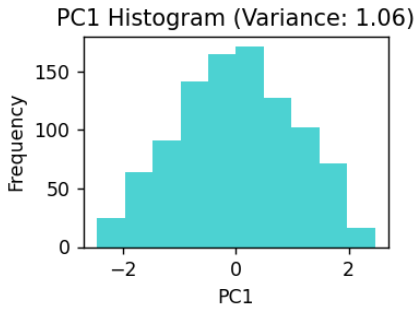
The new feature (PC1) was created combining features Max_BPM,Avg_BPM,Resting_BPM. These 3 features are highly correlated. We can see that the normalized variances for these features are 0.34, 0.52 and 0.14. The variance for PC1 is 1.06

The higher variance of PC1 indicates that it effectively captures the main patterns of variability in the dataset, which is what we expect when the features are correlated, and PC1 explains the majority of that variability. Therefore, it's safe to say that PC1 is a good representation of the three features.









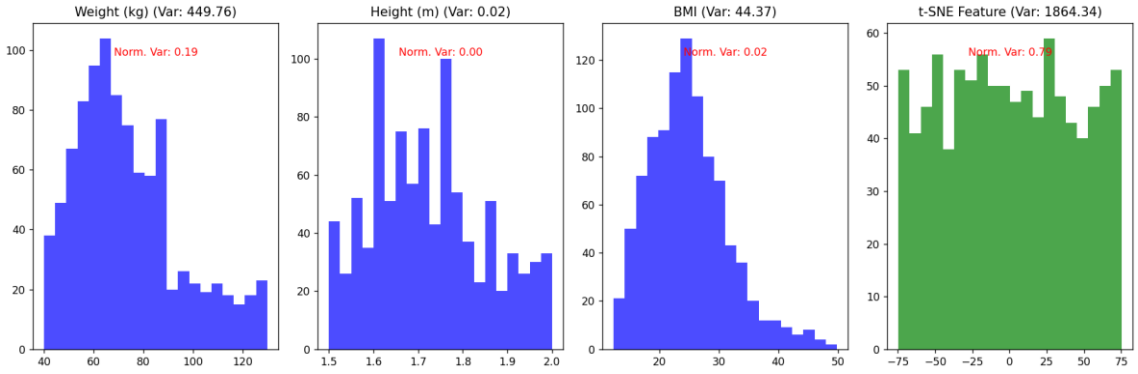
Here is the code to calculate the new PC1 feature and the plots for the Max_BPM, Avg_BPM, Resting_BPM, PC1 features.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

data = pd.read_csv('gym_members_exercise_tracking.csv')

selected_features = ['Max_BPM', 'Avg_BPM', 'Resting_BPM']
data_selected = data[selected_features]

# Standardize the selected features manually
means = data_selected.mean()
stds = data_selected.std()
standardized_data = (data_selected - means) / stds  # Standardize the
data

cov_matrix = np.cov(standardized_data, rowvar=False)
eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

sorted_indices = np.argsort(eigenvalues)[::-1]
sorted_eigenvalues = eigenvalues[sorted_indices]
sorted_eigenvectors = eigenvectors[:, sorted_indices]

principal_component = sorted_eigenvectors[:, 0]

pc1 = standardized_data.dot(principal_component)
data['PC1'] = pc1  # Add the new PCA feature to the dataset

variances = data[selected_features + ['PC1']].var()

total_variance = variances.sum()
normalized_variances = variances[selected_features] / total_variance

fig, axes = plt.subplots(2, 2, figsize=(12, 6))  # Smaller window size
axes = axes.ravel()

# Adjusted the number of bins in the histograms to make them smaller
for i, col in enumerate(selected_features + ['PC1']):
    axes[i].hist(standardized_data[col] if col != 'PC1' else data['PC1'],
bins=10, alpha=0.7, color='c')  # Standardized data for comparison
    axes[i].set_title(f'{col} Histogram (Variance:
{variances[col]:.2f})')
    axes[i].set_xlabel(col)
    axes[i].set_ylabel('Frequency')

    if col in selected_features:
        axes[i].annotate(f'Normalized Variance:
{normalized_variances[col]:.2f}',
                         xy=(0.5, 1.15), xycoords='axes fraction',
```

```
                        fontsize=10, color='red', ha='center',
va='bottom')

plt.subplots_adjust(hspace=1, wspace=1)

plt.show()
```

1.2 t-SNE

Here we can see a new feature created based on Weight, Height and BMI (body mass index). These features were chosen because they are highly correlated.



Weight has a normalized variance of 0.19; Height has 0 variance and BMI has 0,02 normalized variance. As we can see, our t-SNE feature has a high variance (it means that this feature encapsulates a significant amount of the variability present in the original dataset)

Here is the code for the programme that has calculated the variance for each of the features and created the plots:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE


data = pd.read_csv('gym_members_exercise_tracking.csv')



selected_features = ['Weight (kg)', 'Height (m)', 'BMI']
data_selected = data[selected_features]
```

```
data_standardized = (data_selected - data_selected.mean()) /
data_selected.std()

tsne = TSNE(n_components=1, perplexity=30, random_state=42)
new_feature = tsne.fit_transform(data_standardized)
data['t-SNE_Feature'] = new_feature

variances = data[selected_features].var()  # Variances of original
features
new_feature_variance = data['t-SNE_Feature'].var()  # Variance of the new
feature

# Normalized
total_variance = variances.sum() + new_feature_variance
normalized_variances = variances / total_variance
normalized_new_feature_variance = new_feature_variance / total_variance



fig, axes = plt.subplots(1, len(selected_features) + 1, figsize=(15,
5))  # Create subplots
axes = axes.ravel()
for i, col in enumerate(selected_features):
    axes[i].hist(data[col], bins=20, alpha=0.7, color='blue')
    axes[i].set_title(f'{col} (Var: {variances[col]:.2f})')
    axes[i].annotate(f'Norm. Var: {normalized_variances[col]:.2f}',
                     xy=(0.5, 0.9), xycoords='axes fraction',
fontsize=10, color='red', ha='center')

# Plot histogram for new feature
axes[-1].hist(data['t-SNE_Feature'], bins=20, alpha=0.7, color='green')
axes[-1].set_title(f't-SNE Feature (Var: {new_feature_variance:.2f})')
axes[-1].annotate(f'Norm. Var: {normalized_new_feature_variance:.2f}',
                  xy=(0.5, 0.9), xycoords='axes fraction', fontsize=10,
color='red', ha='center')


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

2. Feature Selection

2.1 Variance Threshold

Applying the variance threshold the following features were obtained:

['Age', 'Weight (kg)', 'Max_BPM', 'Avg_BPM', 'Resting_BPM', 'Calories_Burned', 'Fat_Percentage', 'BMI']

At first, a 0,1 threshold was applied, but there were a lot of features. Therefore, it was better to use a threshold value of 1. Depending on this value, there will be more or less excluded features from our data set.

Here is the code for the programme:

```
import pandas as pd
from sklearn.feature_selection import VarianceThreshold

data = pd.read_csv('gym_members_exercise_tracking.csv')

# only numerical columns
numeric_data = data.select_dtypes(include=['number'])

# threshold can be changed
threshold = 1
selector = VarianceThreshold(threshold=threshold)
selector.fit(numeric_data)
selected_features_mask = selector.get_support()

selected_features = numeric_data.columns[selected_features_mask]

print("Selected Features Based on Variance Threshold:")
print(selected_features.tolist())

# reduced data set
reduced_data = numeric_data[selected_features]
print("\nReduced Dataset (First 5 Rows):")
print(reduced_data.head())
```

And the output:

```
['Age', 'Weight (kg)', 'Max_BPM', 'Avg_BPM', 'Resting_BPM', 'Calories_Burned', 'Fat_Percentage', 'BMI']

Reduced Dataset (First 5 Rows):
   Age  Weight (kg)  Max_BPM  Avg_BPM  Resting_BPM  Calories_Burned  Fat_Percentage    BMI
0   56         88.3      180      157           60           1313.0            12.6  30.20
1   46         74.9      179      151           66            883.0            33.9  32.00
2   32         68.1      167      122           54            677.0            33.4  24.71
3   25         53.2      190      164           56            532.0            28.8  18.41
4   38         46.1      188      158           68            556.0            29.2  14.39
```

2.2 SelectKBest (Chi2 and R-regression)

Here we use SelectKBest selection technique with Chi2 test and R-regression test.

```
Selected Features Using Chi2:
['Age', 'Avg_BPM', 'Resting_BPM', 'Workout_Frequency (days/week)', 'Experience_Level']

Selected Features Using R Regression:
['Session_Duration (hours)', 'Fat_Percentage', 'Water_Intake (liters)', 'Workout_Frequency (days/week)', 'Experience_Level']
```

These are the features selected by both tests. We can see that Workout frequency and experience level are selected by both tests. This means that these features are the most relevant.

Here is the code of the programme that calculated the above mentioned features:

```python
import pandas as pd
from sklearn.feature_selection import SelectKBest, chi2, f_regression
from sklearn.preprocessing import MinMaxScaler


data = pd.read_csv('gym_members_exercise_tracking.csv')
# only numerical columns
numeric_data = data.select_dtypes(include=['number'])

X = numeric_data.drop('Calories_Burned', axis=1)
y = numeric_data['Calories_Burned']

# Normalize numeric data to make it suitable for chi2
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)


k = 5  # Select top 5 features
chi2_selector = SelectKBest(chi2, k=k)
X_new_chi2 = chi2_selector.fit_transform(X_scaled, y)

chi2_features = X.columns[chi2_selector.get_support()]

# r-regression
r_selector = SelectKBest(f_regression, k=k)
X_new_r = r_selector.fit_transform(X, y)

r_features = X.columns[r_selector.get_support()]

print("Selected Features Using Chi2:")
print(chi2_features.tolist())

print("\nSelected Features Using R Regression:")
print(r_features.tolist())
```

2.5 SelectFromModel

In this feature selection, we use a model (in this case decision tree) and train it with the data from the csv file. After the training, we can get the most important features. (Also we can get predictions).

Each time the model can give us different features because not every prediction is going to be identical (there is a random component in the training of a model). Here we have a programme that trains a decision tree model and predicts what are the most important features.

```
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeRegressor
from sklearn.feature_selection import SelectFromModel


df = pd.read_csv('gym_members_exercise_tracking.csv')
df = df.select_dtypes(include=['number'])

y_synthetic = np.random.rand(len(df))  # Generate random values as
synthetic target

X = df  # All columns are features

dt_model = DecisionTreeRegressor(random_state=67)
dt_model.fit(X, y_synthetic) # model is being trained (we use y_synthetic
because we dont need prediction)

selector = SelectFromModel(dt_model, threshold="mean")
X_selected = selector.transform(X)
selected_features = X.columns[selector.get_support()]

print("Top features selected by Decision Tree model:")
print(selected_features)

importances = dt_model.feature_importances_
importance_df = pd.DataFrame({
    'Feature': X.columns,
    'Importance': importances
})
importance_df = importance_df.sort_values(by='Importance',
ascending=False)

print("\nFeature importance (sorted):")
print(importance_df)
```

Here we can see that every prediction is different:

Prediction 1:

```
Feature importance (sorted):
                        Feature  Importance
12                          BMI    0.108481
8               Fat_Percentage    0.106222
3                      Max_BPM    0.104385
1                 Weight (kg)    0.100547
0                          Age    0.095484
2                   Height (m)    0.086363
```

Prediction 2:

```
Feature importance (sorted):
                        Feature  Importance
8               Fat_Percentage    0.139479
3                      Max_BPM    0.124205
4                      Avg_BPM    0.096647
2                   Height (m)    0.095563
5                   Resting_BPM    0.082044
6       Session_Duration (hours)    0.079803
```

3. Feature Preprocessing

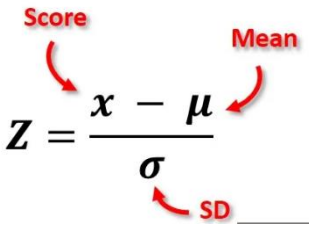
3.1 Normalization
3.2 Standardization

Normalization rescales data to a fixed range and ensures that no features can dominate simply because of its scale.
Standartization transforms data to have a mean of 0 and a standard deviation of 1.
Standartization can be seen as a particular case of normalization

Both preprocessing methods are important for algorithms sensitive to scale, as it ensures features contribute equally. The normalization process centers the data and scales it, removing effects of the original range while maintaining the distribution's shape.

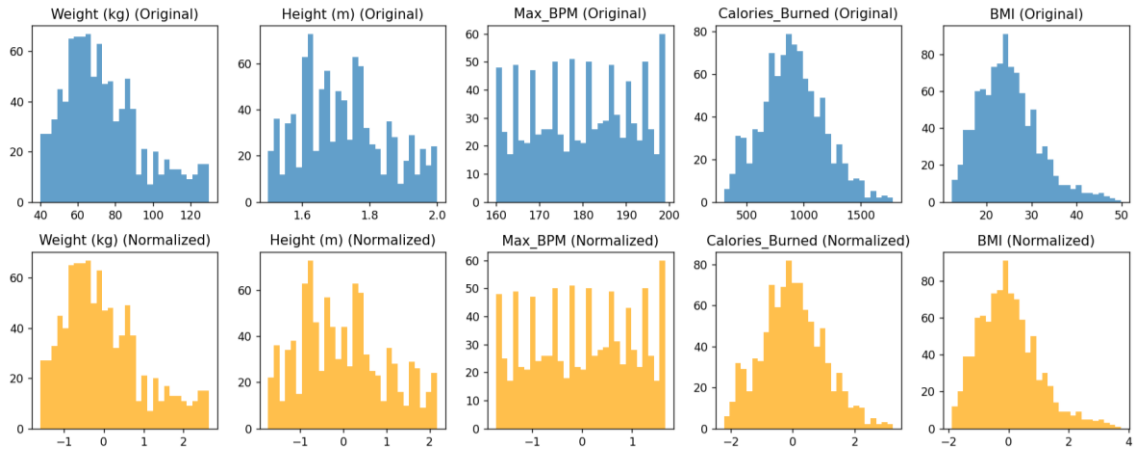
There are different types of normalization such as Min-Max normalization, Z-Score Normalization, Logarithmic Normalization… In this case we are going to use Z-Score normalization as it it the most used one in this type of calculations. We are going to calculate it with mean value 0 and standart deviation value 1 (standartization).

Here is the formula for Z-Score normalization:



First, we calculate the mean of our feature, and then the standart deviation (SD). Then we apply the formula for each of the values.

Here we have plots for some of the features of our csv file and also data statistics (original stats and normalized).



As we can see, the shape remains the same. The only thing that changes is the scale.

```
Original Data Statistics:
       Weight (kg)  Height (m)     Max_BPM  Calories_Burned         BMI
count   973.000000   973.00000  973.000000       973.000000  973.000000
mean     73.854676     1.72258  179.883864       905.422405   24.912127
std      21.207500     0.12772   11.525686       272.641516    6.660879
min      40.000000     1.50000  160.000000       303.000000   12.320000
25%      58.100000     1.62000  170.000000       720.000000   20.110000
50%      70.000000     1.71000  180.000000       893.000000   24.160000
75%      86.000000     1.80000  190.000000      1076.000000   28.560000
max     129.900000     2.00000  199.000000      1783.000000   49.840000
```

```
Normalized Data Statistics:
         Weight (kg)    Height (m)       Max_BPM  Calories_Burned           BMI
count  9.730000e+02  9.730000e+02  9.730000e+02     9.730000e+02  9.730000e+02
mean  -2.345959e-16 -8.324961e-16 -7.266085e-16     8.489270e-17 -1.533545e-16
std    1.000514e+00  1.000514e+00  1.000514e+00     1.000514e+00  1.000514e+00
min   -1.597175e+00 -1.743613e+00 -1.726066e+00    -2.210713e+00 -1.891432e+00
25%   -7.432643e-01 -8.035741e-01 -8.579921e-01    -6.804458e-01 -7.213157e-01
50%   -1.818535e-01 -9.854471e-02  1.008143e-02    -4.558658e-02 -1.129752e-01
75%    5.729845e-01  6.064847e-01  8.781549e-01     6.259697e-01  5.479379e-01
max    2.644071e+00  2.173217e+00  1.659421e+00     3.220452e+00  3.744354e+00
```

As we can see, every feature has approximately a mean value o 0, and a standart deviation value of 1.

Here is the code of the programme that made the plots and the data statistics:

```python
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

file_path = "gym_members_exercise_tracking.csv"
data = pd.read_csv(file_path)

columns = ['Weight (kg)', 'Height (m)', 'Max_BPM', 'Calories_Burned',
'BMI']

original_stats = data[columns].describe()

# normalization (z score)
scaler = StandardScaler()
data_normalized = data.copy()
data_normalized[columns] = scaler.fit_transform(data[columns])

normalized_stats = data_normalized[columns].describe()

# Visualization: Original vs Normalized
def plot_feature_distributions(original, normalized, title):
    plt.figure(figsize=(14, 6))
    for i, col in enumerate(columns):
        plt.subplot(2, len(columns), i + 1)
        plt.hist(original[col], bins=30, alpha=0.7, label='Original')
        plt.title(f'{col} (Original)')
```

```
        plt.subplot(2, len(columns), i + len(columns) + 1)
        plt.hist(normalized[col], bins=30, alpha=0.7, color='orange',
label='Normalized')
        plt.title(f'{col} (Normalized)')

    plt.suptitle(title)
    plt.tight_layout()
    plt.show()

plot_feature_distributions(data, data_normalized, "Z-score
Normalization")

# statistics for each of the data (original and nroamlized)
print("Original Data Statistics:\n", original_stats, "\n")
print("\nNormalized Data Statistics:\n", normalized_stats)
```

3.3 Scaling

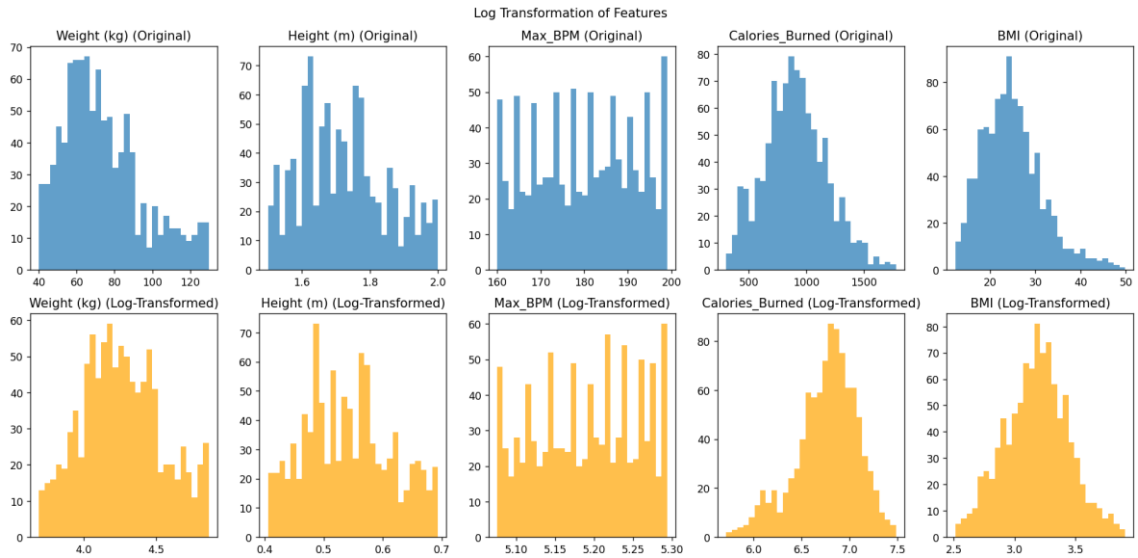
Scaling is used to adjust the range and distribution of our data set values. It helps us interpret and compare data features better. Also, some algorithms like gradient descent converge faster when features are scaled properly.

Normalization and Standartization are both scaling methods.

We also have other scaling methods like logarithmic scaling method. It applies a logarithmic transformation to compress large values. This is the formula for the logarithmic transformation:

$$x' = \log(x)$$

Here are the plots for the features we applied standardization previosuly, this time with the logarithmic scaling method:



As we can see, our data features have been scaled-down. This means that the mean decreased and is now less sensitive to large values. Variance, minimum and maximum have also decreased.

Another interesting phenomenom is that charts have changed their shape. This is because the logarithmic method “compresses” our data and reduces skewness. Skewness is a measure of asymmetry. When skewness is reduced, our data becomes more centered and symmetric, as we can see.

*Logarithmic method needs positive data values to be useful.

Here is the code written to draw the plots:

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

file_path = "gym_members_exercise_tracking.csv"
data = pd.read_csv(file_path)
columns = ['Weight (kg)', 'Height (m)', 'Max_BPM', 'Calories_Burned',
'BMI']

# Handle negative values
data_log_transformed = data.copy()
for col in columns:
    # If there are non-positive values, add a small constant (e.g., 1)
    if (data[col] <= 0).any():
        data_log_transformed[col] = np.log(data[col] + 1)
    else:
        data_log_transformed[col] = np.log(data[col])

def plot_feature_distributions(original, transformed, title):
    plt.figure(figsize=(14, 8))
    for i, col in enumerate(columns):

        plt.subplot(2, len(columns), i + 1)
        plt.hist(original[col], bins=30, alpha=0.7, label='Original')
        plt.title(f'{col} (Original)')

        plt.subplot(2, len(columns), i + len(columns) + 1)
        plt.hist(transformed[col], bins=30, alpha=0.7, color='orange',
label='Log-Transformed')
        plt.title(f'{col} (Log-Transformed)')

    plt.suptitle(title)
    plt.tight_layout()
    plt.show()

plot_feature_distributions(data, data_log_transformed, "Log
Transformation of Features")
```

3.4 Encoding Categorical Features

Encoding categorical features is a process used to transform non-numerical features into a numerical format so that machine learning algorithms can process them and make calculations.

One of the most used encoding mechanism is called one-hot.

It works in the following way:

Categorical feature	One-Hot Encoding
Type1	0001
Type2	0010
Type3	0100
Type4	1000

For each Category in our feature we asign a binary code (all zeros, except for one 1 value)

Here we have encoded two of our features (Gender and Workout Type):

```
=== One-Hot Encoding for 'Gender' ===
  Original Category Encoded Columns
0              Male   Gender_Female
1            Female     Gender_Male

=== One-Hot Encoding for 'Workout_Type' ===
  Original Category        Encoded Columns
0              Yoga    Workout_Type_Cardio
1              HIIT      Workout_Type_HIIT
2           Cardio  Workout_Type_Strength
3          Strength      Workout_Type_Yoga
```

The gender encoding has turned Male category into 0 value and Female category into 1 value.

Regarding the Workout Type, we have 4 different categories:

0001 -> Yoga
0010 -> HIIT
0100 -> Cardio
1000 -> Strength

Here is the code for the encoder programme:

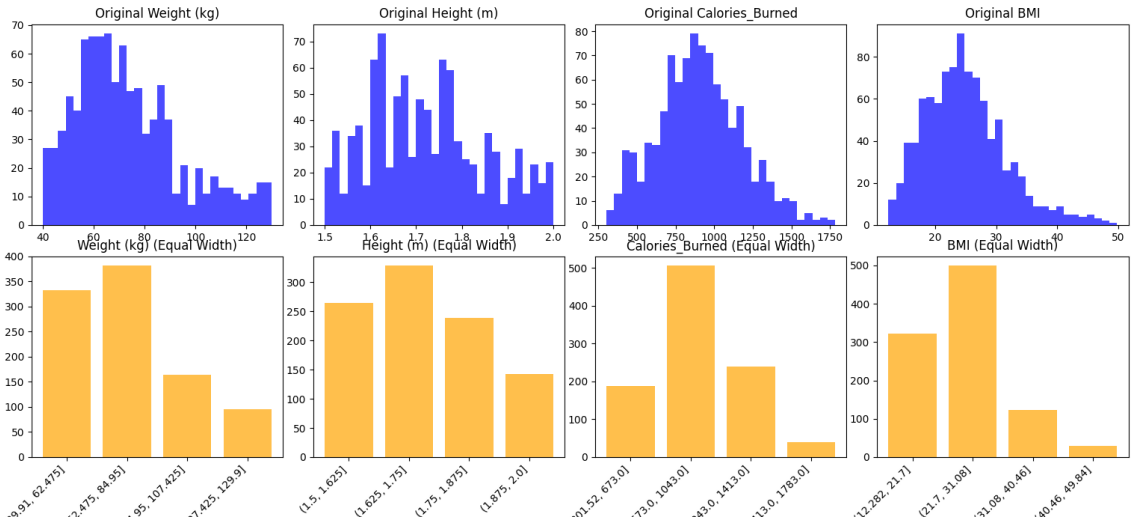
```python
import pandas as pd

file_path = "gym_members_exercise_tracking.csv"
data = pd.read_csv(file_path)

# Identify categorical columns
categorical_columns =
data.select_dtypes(include=['object']).columns.tolist()
one_hot_encoded_data = pd.get_dummies(data, columns=categorical_columns)
```

```
def generate_encoding_chart(data, one_hot_encoded_data,
categorical_columns):
    for col in categorical_columns:
        print(f"=== One-Hot Encoding for '{col}' ===")
        unique_values = data[col].unique()

        encoded_columns = [c for c in one_hot_encoded_data.columns if
c.startswith(col)]

        chart = pd.DataFrame({'Original Category': unique_values,
'Encoded Columns': encoded_columns})
        print(chart, "\n")

generate_encoding_chart(data, one_hot_encoded_data, categorical_columns)
```

3.5 Discretization

It is a process where continous data is split into categories. This method simplifies data analysis and is useful for models that need categorical data input.
In the other hand, discretization can lead to loss of information and reduction in precision. Therefore, statistical measures like mean often shifts and variance can decrease because you're grouping continuous values into bins. The values within each bin are now treated as equivalent, which reduces the spread within each bin (same with SD).
Min and Max remain the same.

Here are features of our csv file being spread into bins:



Here is the code for the programme that split the data into categories:

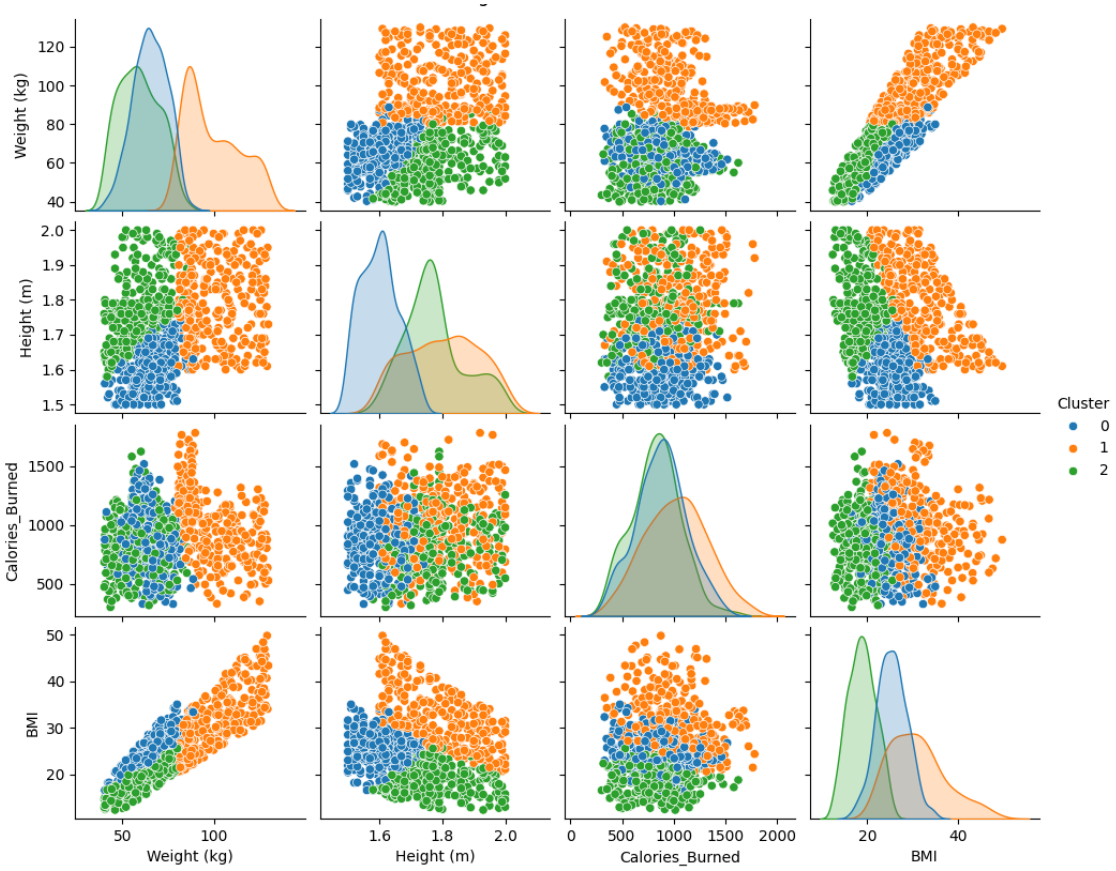
```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

file_path = "gym_members_exercise_tracking.csv"
data = pd.read_csv(file_path)

columns_to_discretize = ['Weight (kg)', 'Height (m)', 'Calories_Burned',
'BMI']

def equal_width_discretization(data, columns, num_bins):
    for col in columns:
        data[f'{col}_EW'] = pd.cut(data[col], bins=num_bins)
    return data

num_bins = 4
data = equal_width_discretization(data, columns_to_discretize, num_bins)

def plot_distributions(data, columns):
    fig, axes = plt.subplots(2, len(columns), figsize=(len(columns) * 5,
10))

    for i, col in enumerate(columns):
        axes[0, i].hist(data[col], bins=30, color='blue', alpha=0.7)
        axes[0, i].set_title(f'Original {col}')

        bin_counts = data[f'{col}_EW'].value_counts().sort_index()  #
Ensure bins are ordered sequentially
        axes[1, i].bar(
            x=bin_counts.index.astype(str),
            height=bin_counts.values,
            color='orange',
            alpha=0.7
        )
        axes[1, i].set_title(f'{col} (Equal Width)')
        axes[1, i].set_xticklabels(bin_counts.index.astype(str),
rotation=45, ha="right")

    plt.tight_layout()
    plt.show()

plot_distributions(data, columns_to_discretize)
print(data[[f'{col}_EW' for col in columns_to_discretize]].head())
```

4. Clustering

Clustering is a technique used to group data points into clusters, where points that belong to the same cluster tend to be more similar. It is used to discover hidden patterns in unlabeled data.

4.1 KMeans

KMeans algorithm splits data into clusters classifing data using a distance criteria. This means that points that are closer to each other will probably belong to the same cluster. KMeans algorithm is useful if you know in advance how many clusters there are supposed to be (also when clusters have circular form)
Here we have some plots for some of our features where we can see them being separated into clusters using Kmeans algorithm:



We can see that cluster 1 (orange) represents higher values for each feature (higher weight, taller height, bigger amount of calories burned, higher BMI), cluster 2 (green) represents medium values, and cluster 0 (blue) represents lower values.
Some correlations that we can observe are:

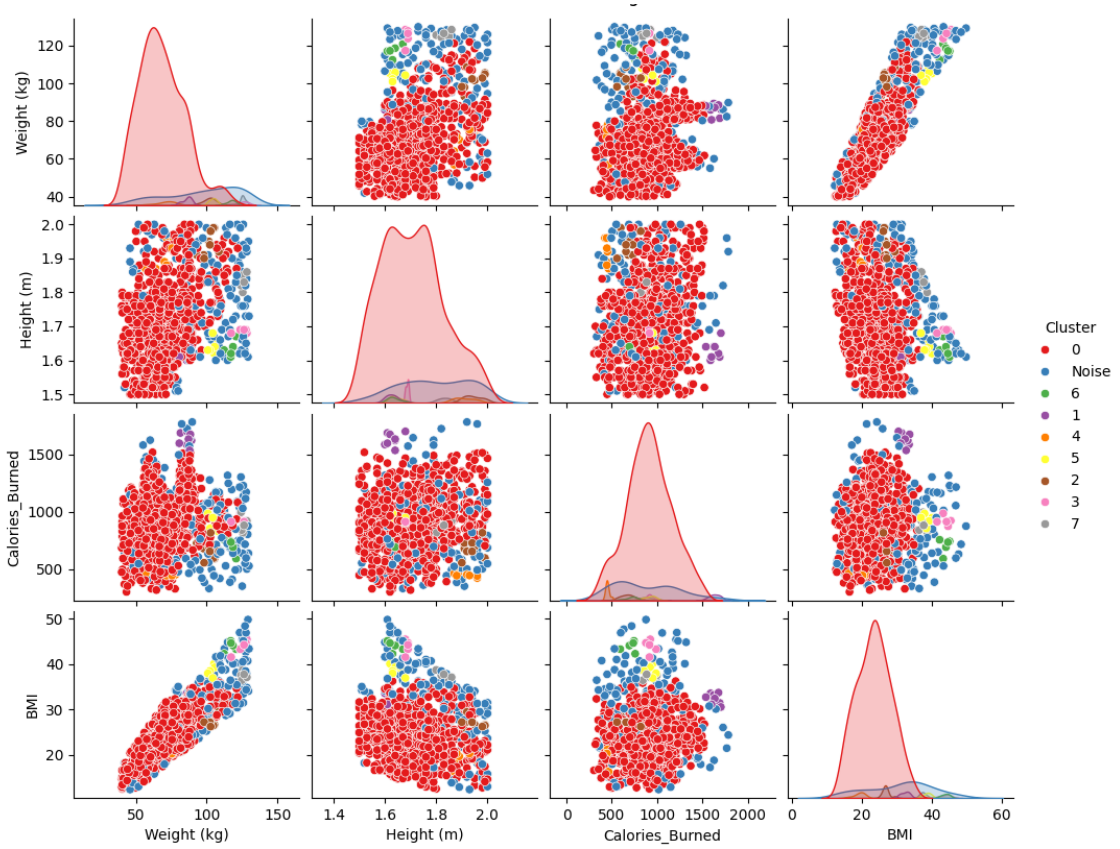
- Weight and BMI: A strong positive correlation between weight and BMI, where people with larger weight tend to have a higher Body Mass Index.
- Weight and Height: Higher people tend to weight more.

Here is the code for the Python programme that calculated the clusters using KMeans algorithm:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import seaborn as sns

file_path = "gym_members_exercise_tracking.csv"  # Replace with the
actual file path
data = pd.read_csv(file_path)

selected_features = ['Weight (kg)', 'Height (m)', 'Calories_Burned',
'BMI']  # Customize these features
selected_data = data[selected_features]

scaler = StandardScaler()
scaled_data = scaler.fit_transform(selected_data)

num_clusters = 3
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
data['Cluster'] = kmeans.fit_predict(scaled_data)

def plot_clusters(data, cluster_col, selected_features):
    sns.pairplot(data[selected_features + [cluster_col]],
hue=cluster_col, palette='tab10')
    plt.suptitle("KMeans Clustering Visualization (Selected Features)",
y=1.02)
    plt.show()

plot_clusters(data, 'Cluster', selected_features)

cluster_centers = scaler.inverse_transform(kmeans.cluster_centers_)
cluster_centers_df = pd.DataFrame(cluster_centers,
columns=selected_features)
print("Cluster Centers:\n", cluster_centers_df)

print("\nCluster Counts:\n", data['Cluster'].value_counts())
```

4.2 DBSCAN

Unlike KMeans algorithm, DBSCAN classifies data into clusters based on density (it means that data doesn't have to be grupped in spherical forms to form cluster). Also, we don't need to specify the number of clusters.
Here are the plots of our selected features being grupped into clusters using DBSCAN algorithm:



We can see that this algorithm didn't work as well as KMeans because our data values tend to form a dense area. This makes our programme classify most of our data into one cluster and the rest into small clusters or noise.

Here is the code for the programme that calculated the clusters and the plots:

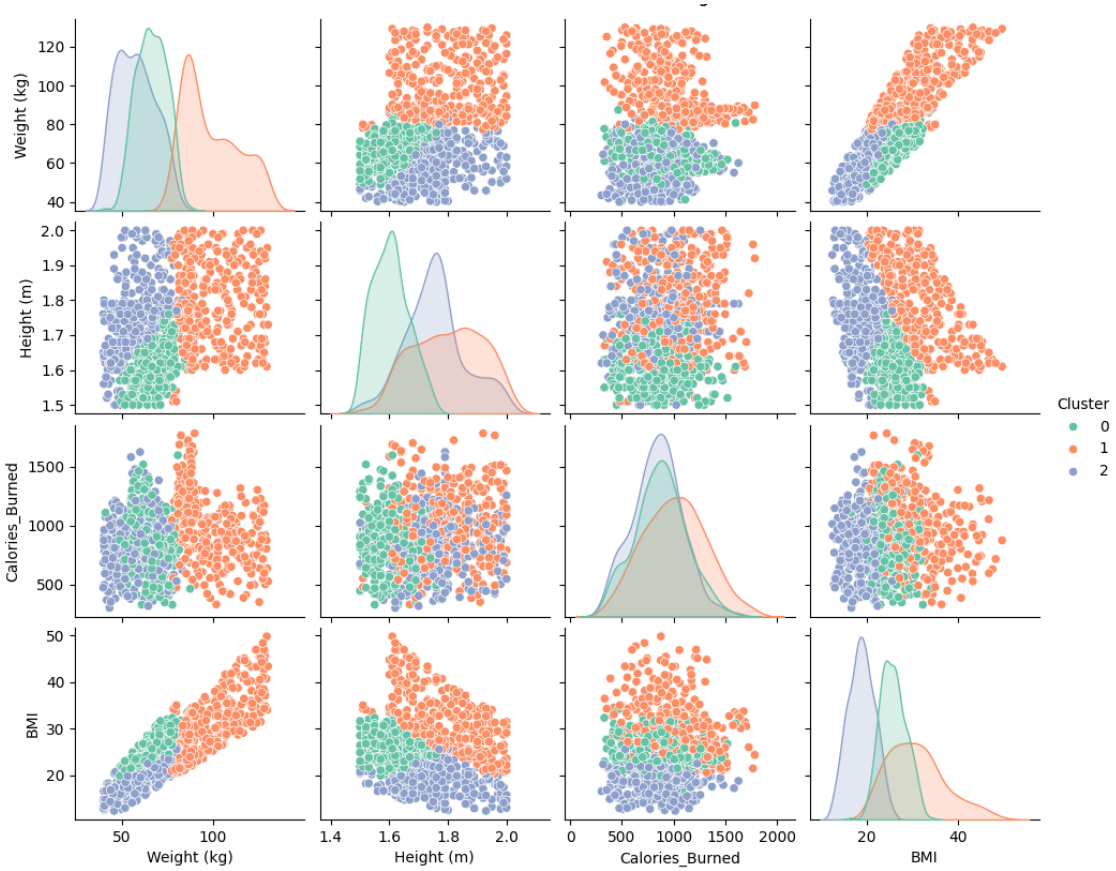
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN

file_path = "gym_members_exercise_tracking.csv"
data = pd.read_csv(file_path)

selected_features = ['Weight (kg)', 'Height (m)', 'Calories_Burned',
'BMI']
```

```
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data[selected_features])

dbscan = DBSCAN(eps=0.5, min_samples=5)
data['Cluster'] = dbscan.fit_predict(data_scaled)

data['Cluster'] = data['Cluster'].apply(lambda x: 'Noise' if x == -1 else
x)

sns.pairplot(
    data=data,
    vars=selected_features,
    hue='Cluster',
    palette='Set1',
    diag_kind='kde',
    height=2.5
)

plt.suptitle('DBSCAN Clustering', y=1.02)
plt.show()
```

4.3 Gaussian Mixture

Gaussian Mixture Model uses Gaussian distribution to create the clusters. Unlike KMeans, clusters can have different size and shapes (not only spherical), which can be more useful for real life scenarios.

Here we can see the same features divided in clusters, this time using GMM:



We can barely notice any difference between GMM claustering and KMeans claustering in this features.
This means that in this case, GMM's flexibility is not really needed, as the clusters in the data are more or less well defined and there is no significant noise.

This is the code of the programme that calculated the clusters and drew the plots:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler

file_path = "gym_members_exercise_tracking.csv"
data = pd.read_csv(file_path)
```

```
selected_features = ['Weight (kg)', 'Height (m)', 'Calories_Burned',
'BMI']

scaler = StandardScaler()
data_scaled = scaler.fit_transform(data[selected_features])

n_components = 3
gmm = GaussianMixture(n_components=n_components, random_state=42)
data['Cluster'] = gmm.fit_predict(data_scaled)

sns.pairplot(
    data=data,
    vars=selected_features,
    hue='Cluster',
    palette='Set2',
    diag_kind='kde',
    height=2.5
)

plt.suptitle('Gaussian Mixture Model Clustering', y=1.02)
plt.show()
```

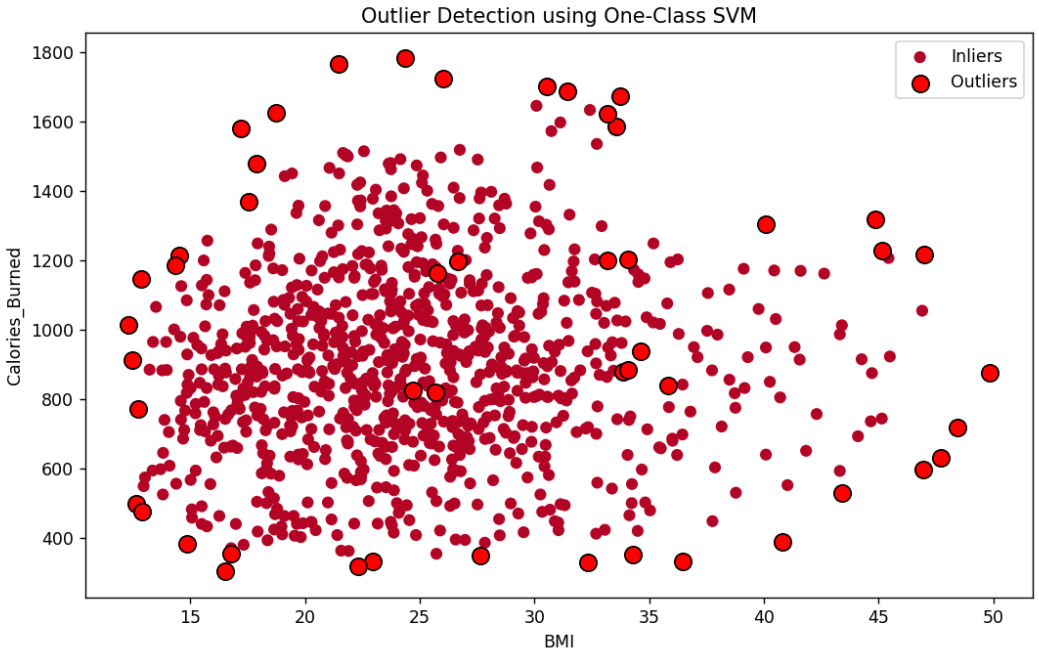
5. Outlier Detection

Outliner detection is the process of identifying points in out dataset that are distanced from from the rest of the data. These points are called outliners.

5.1 OneClass SVM

Support Vector Machine is a popular supervised machine learning algorithm. it is used for both classifications and regression. Its primary aim is to locate instances that notably deviate from the standard. Unlike conventional Machine Learning models focused on binary or multiclass classification, the one-class SVM specializes in outlier or novelty detection within datasets.

In the claustering charts above we can notice that Calories_Burned and BMI have noticeable outliners:



We can see that we have plenty of outliners that differ from the majority on the outside of the main dataset shape. Also, we see that the OneClass SVM algorithm has pointed out some outliers that are not really distant from the mean (the algorithm is not perfect).

Here is the code for the programme that calculated the outliers with OneClass SVM algorithm:

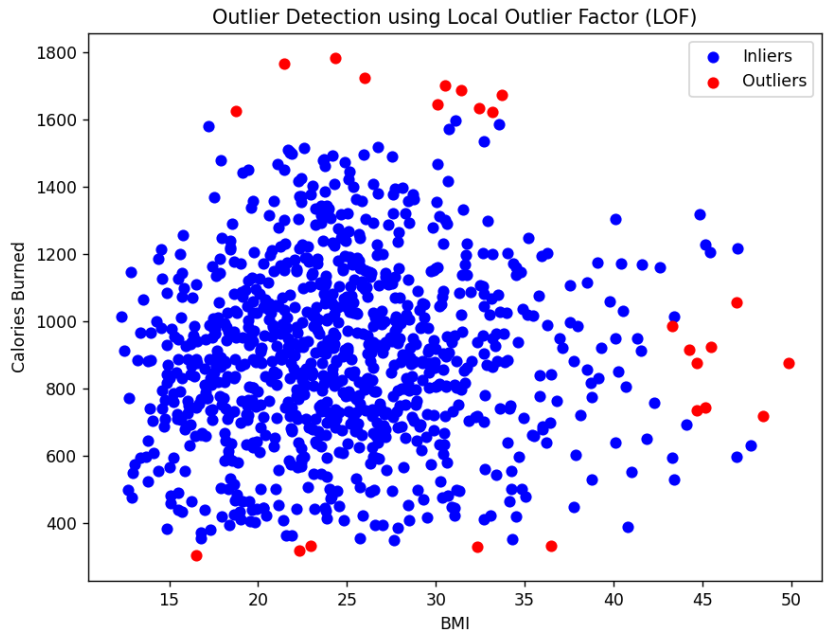
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import OneClassSVM
from sklearn.preprocessing import StandardScaler

file_path = "gym_members_exercise_tracking.csv"
data = pd.read_csv(file_path)

features = ['BMI', 'Calories_Burned']
scaler = StandardScaler()
```

```
data_scaled = scaler.fit_transform(data[features])

ocsvm = OneClassSVM(kernel='rbf', nu=0.05, gamma='auto')  # 'nu' controls
the proportion of outliers

ocsvm.fit(data_scaled)

outliers = ocsvm.predict(data_scaled)

data['Outlier'] = outliers
plt.figure(figsize=(10, 6))
plt.scatter(data[features[0]], data[features[1]], c=data['Outlier'],
cmap='coolwarm', label='Inliers')

plt.scatter(data.loc[data['Outlier'] == -1, features[0]],
            data.loc[data['Outlier'] == -1, features[1]],
            color='red', label='Outliers', s=100, edgecolors='black')

plt.xlabel(features[0])
plt.ylabel(features[1])
plt.title("Outlier Detection using One-Class SVM")
plt.legend()
plt.show()
```

5.2 Local Outlier Factor

LOF is an outlier detection algorithm that detects anomalies by comparing local density of a point with its neighbors. Also, LOF produces a LOF score for each point, unlike One Class SVM, that produces binary predictions. In this case we are going to round the scale to 1 or 0.

Also, LOF is unsupervised, meaning it doesn't need training like OneClass SVM.
Here we have the outlier points detected by LOF:



We can see that LOF algorithm has detected outliers in a much better way than One Class SVM.

This is because it is density based, and it detects outliers if they far away from others. As we can see, LOF hasn't pointed out outliers in the center of our dataset, as it is a very dense region.

This is the code of the programme that calculated outliers using LOF:

```
import pandas as pd
from sklearn.neighbors import LocalOutlierFactor
import matplotlib.pyplot as plt

file_path = "gym_members_exercise_tracking.csv"
data = pd.read_csv(file_path)

features = ['BMI', 'Calories_Burned']
X = data[features]

lof = LocalOutlierFactor(n_neighbors=20)

outliers = lof.fit_predict(X)

data['LOF_Outlier'] = outliers
data['LOF_Score'] = -lof.negative_outlier_factor_

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

# innliers
plt.scatter(data['BMI'][data['LOF_Outlier'] == 1],
            data['Calories_Burned'][data['LOF_Outlier'] == 1],
            color='blue', label='Inliers')

#outliers
plt.scatter(data['BMI'][data['LOF_Outlier'] == -1],
            data['Calories_Burned'][data['LOF_Outlier'] == -1],
            color='red', label='Outliers')

plt.xlabel('BMI')
plt.ylabel('Calories Burned')
plt.title('Outlier Detection using Local Outlier Factor (LOF)')
plt.legend()
plt.show()
```

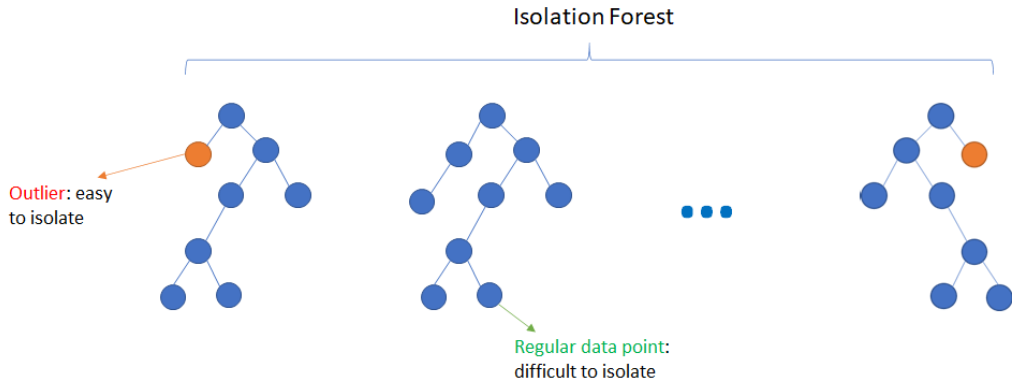
5.3 Isolation Forest

This method uses binary trees to split data and detect outliers.

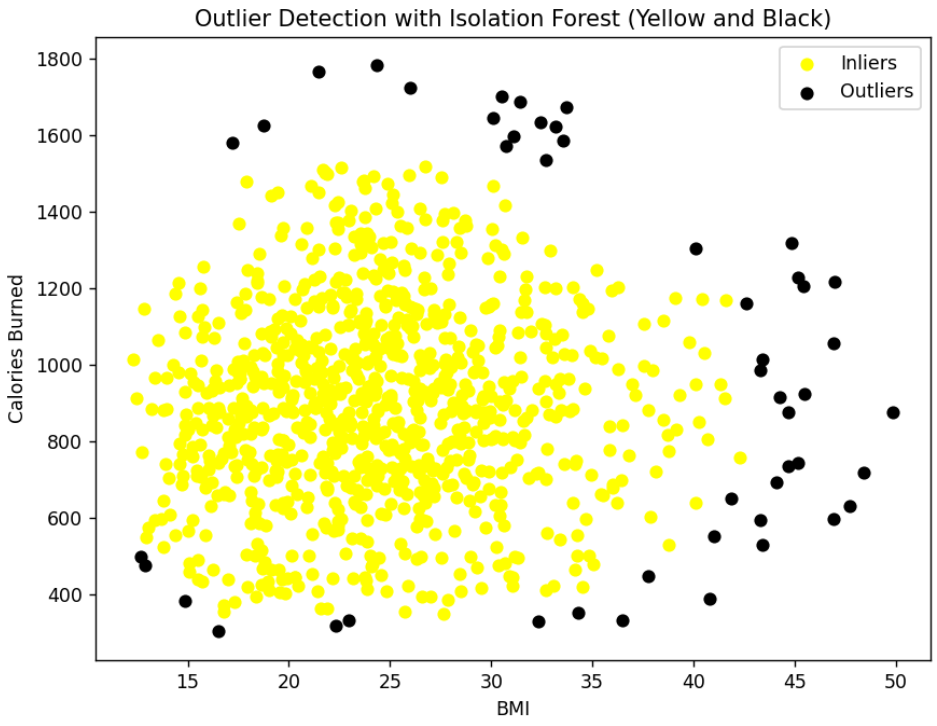
First, it takes all data points (their coordinates) and saves minimum and maximum values of all the dataset. After that, it chooses a random split value in that range.
The split value is used to split the dataset in two child nodes. This process is repeated until all points are split (no more child nodes, only leaves).

Outliers are usually far away from dense regions. Therefore, they will get isolated in this process earlier than data-values that are in dense regions (it takes more splits to isolate each point in a dense area).

Taking this into account, outliers will have a shorter path length (number of splits required to isolate a data point). Therefore, our algorithm will detect outliers by looking into the first splits of the tree:



Here we have a plot that shows us outliers (being detected by isolation forest algorithm):



As we can see, the Isolation Forest algorithm has detected outliers very efficiently. It is a great algorithm because it has a time complexity of O(nlogn) which is linear in relation to the number of samples and logarithmic in relation to the number of features.
Also, it can handle large datasets and it works well for data that does not follow a normal distribution, or when the data structure is complex. Best algorithm ever.

Here is the code for the programme that used Isolation Forest algorithm to detect outliers:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest

file_path = "gym_members_exercise_tracking.csv"
data = pd.read_csv(file_path)

features = ['BMI', 'Calories_Burned']
X = data[features]

iso_forest = IsolationForest(n_estimators=100, contamination=0.05,
random_state=42)  # contamination=percentage of expected outliers
outliers = iso_forest.fit_predict(X)

data['Isolation_Outlier'] = outliers  # -1 for outliers, 1 for inliers
data['Isolation_Score'] = iso_forest.decision_function(X)  # Lower score
= higher chance of being an outlier

plt.figure(figsize=(8, 6))
plt.scatter(data['BMI'][data['Isolation_Outlier'] == 1],
            data['Calories_Burned'][data['Isolation_Outlier'] == 1],
            color='yellow', label='Inliers')
plt.scatter(data['BMI'][data['Isolation_Outlier'] == -1],
            data['Calories_Burned'][data['Isolation_Outlier'] == -1],
            color='black', label='Outliers')
plt.xlabel('BMI')
plt.ylabel('Calories Burned')
plt.title('Outlier Detection with Isolation Forest (Yellow and Black)')
plt.legend()
plt.show()
```