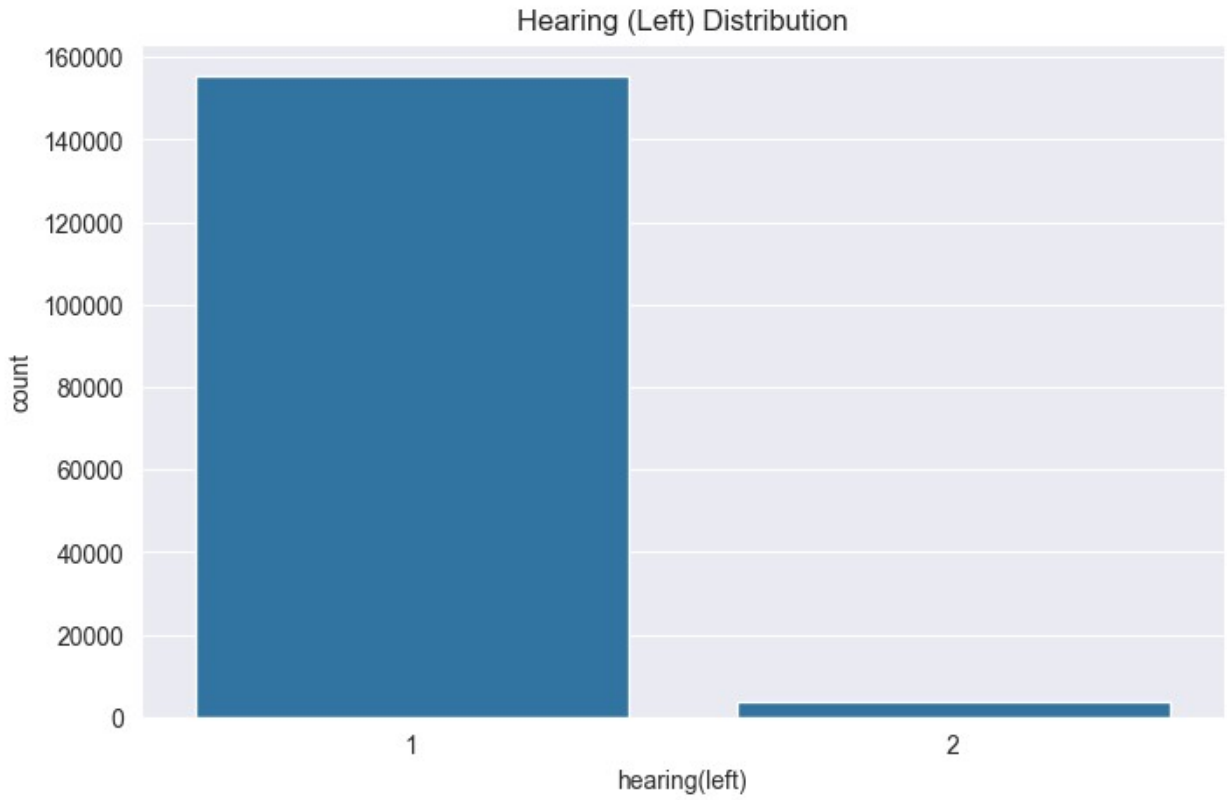
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV


# Function to display comprehensive univariate analysis for
categorical variables
def categorical_univariate_analysis(feature_name, data_series):
    # Frequency Distribution
    frequency_distribution = data_series.value_counts()

    # Display results
    print(f"\n------ Univariate Analysis for {feature_name} ------")
    print(f"Frequency Distribution:\n{frequency_distribution}\n")

    # Visualization
    plt.figure(figsize=(8, 5))
    sns.countplot(x=data_series,
order=data_series.value_counts().index)  # Added order parameter
    plt.title(f'{feature_name} Distribution')
    plt.show()

def outlier_removal(feature_name, data_series):
    # IQR Method for Outlier Detection
    Q1 = data_series.quantile(0.25)
    Q3 = data_series.quantile(0.75)
    IQR = Q3 - Q1

    # Identify and remove outliers for the current feature
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Create a mask for non-outliers for the current feature
    outlier_mask = (df[feature_name] >= lower_bound) &
(df[feature_name] <= upper_bound)

    # Apply the mask to the DataFrame
    global df_no_outliers  # Use global to modify the variable outside
the function
    if df_no_outliers is None:
        df_no_outliers = df[outlier_mask]
    else:
        df_no_outliers = df_no_outliers[outlier_mask]
```

```
    # Display results
    print(f"\n------ Univariate Analysis for {feature_name} ------")
    print(f"IQR Method for Outlier Detection:")
    print(f"Lower Bound: {lower_bound}")
    print(f"Upper Bound: {upper_bound}")
    print(f"Number of Outliers Removed: {len(df) -
len(df_no_outliers)}\n")

    # Visualizations
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    sns.histplot(df_no_outliers[feature_name])
    plt.title(f'{feature_name} Distribution (No Outliers)')

    plt.subplot(1, 2, 2)
    sns.boxplot(x=df_no_outliers[feature_name])
    plt.title(f'{feature_name} Box Plot (No Outliers)')
    plt.show()

# Assuming 'data.csv' is your dataset file
data_path = 'my_data.csv'

# Read the dataset
df = pd.read_csv(data_path)
df.info()

# DataFrame without outliers
df_no_outliers = None  # Initialize to None
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 159256 entries, 0 to 159255
Data columns (total 12 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   id              159256 non-null  int64
 1   ALT             159256 non-null  int64
 2   AST             159256 non-null  int64
 3   hearing(left)   159256 non-null  int64
 4   weight(kg)      159256 non-null  int64
 5   hearing(right)  159256 non-null  int64
 6   relaxation      159256 non-null  int64
 7   waist(cm)       159256 non-null  float64
 8   Cholesterol     159256 non-null  int64
 9   HDL             159256 non-null  int64
 10  systolic        159256 non-null  int64
 11  smoking         159256 non-null  int64
dtypes: float64(1), int64(11)
memory usage: 14.6 MB
```

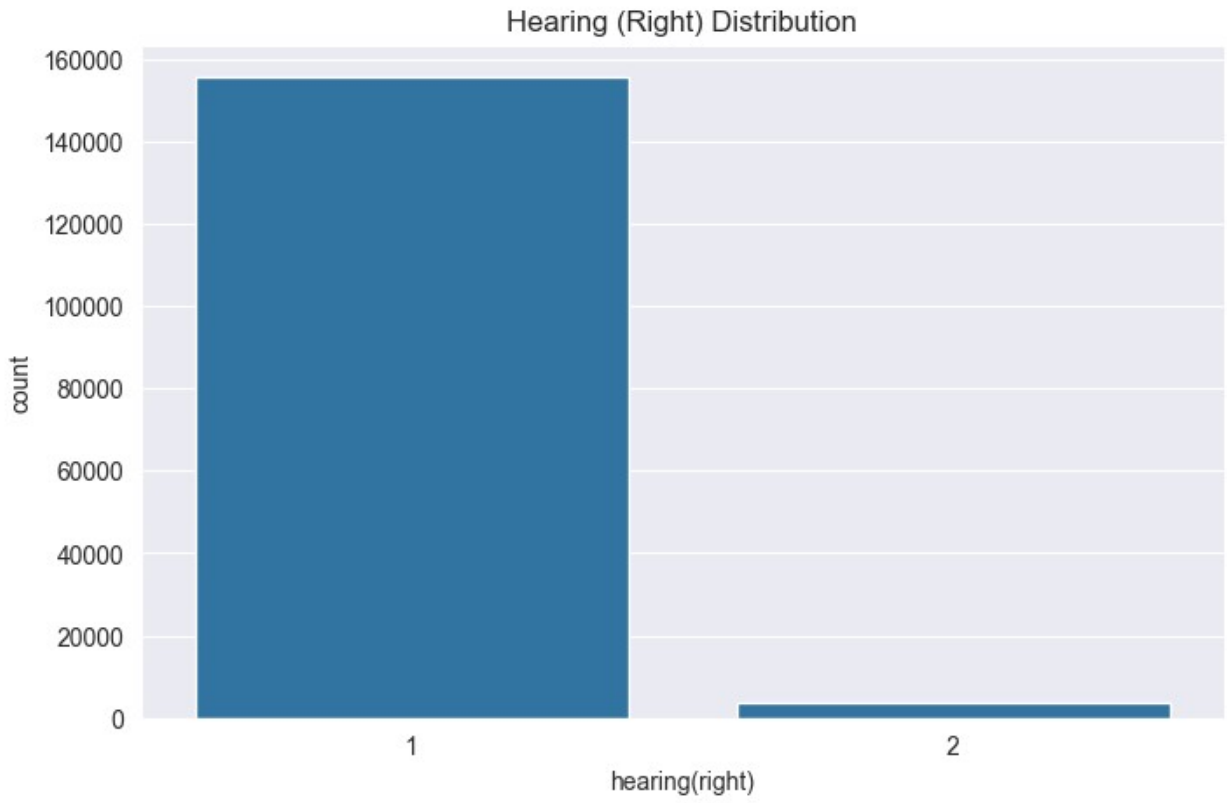
```
# Apply the functions to each feature
categorical_univariate_analysis('Hearing (Left)', df['hearing(left)'])
```

```
------ Univariate Analysis for Hearing (Left) ------
Frequency Distribution:
hearing(left)
1    155438
2      3818
Name: count, dtype: int64
```

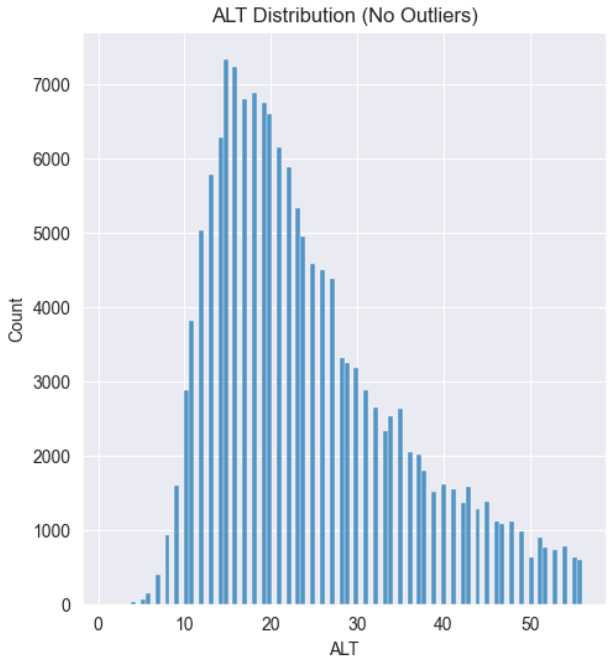


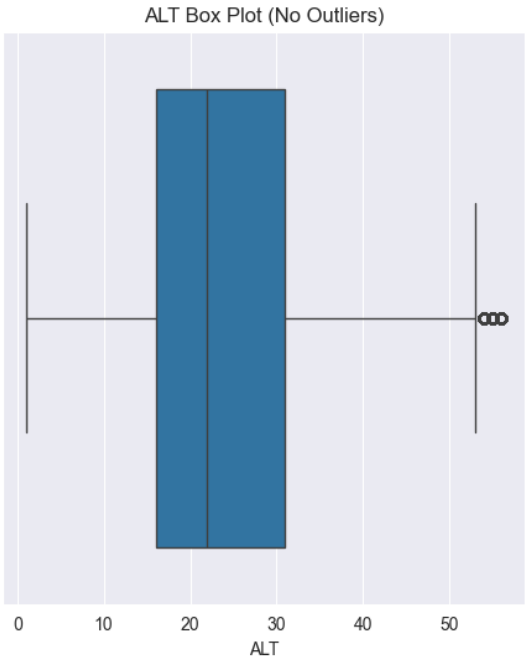

```
categorical_univariate_analysis('Hearing (Right)',
df['hearing(right)'])
```

```
------ Univariate Analysis for Hearing (Right) ------
Frequency Distribution:
hearing(right)
1    155526
2      3730
Name: count, dtype: int64
```



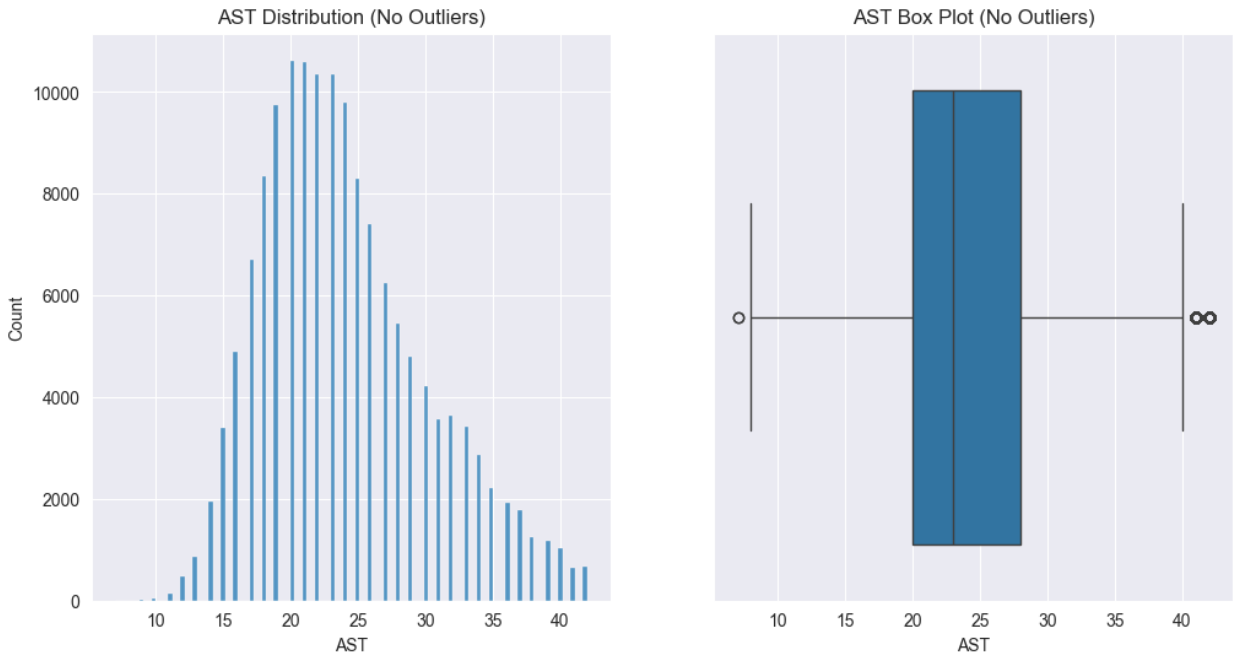

```
outlier_removal('ALT', df['ALT'])


------ Univariate Analysis for ALT ------
IQR Method for Outlier Detection:
Lower Bound: -8.0
Upper Bound: 56.0
Number of Outliers Removed: 6746
```





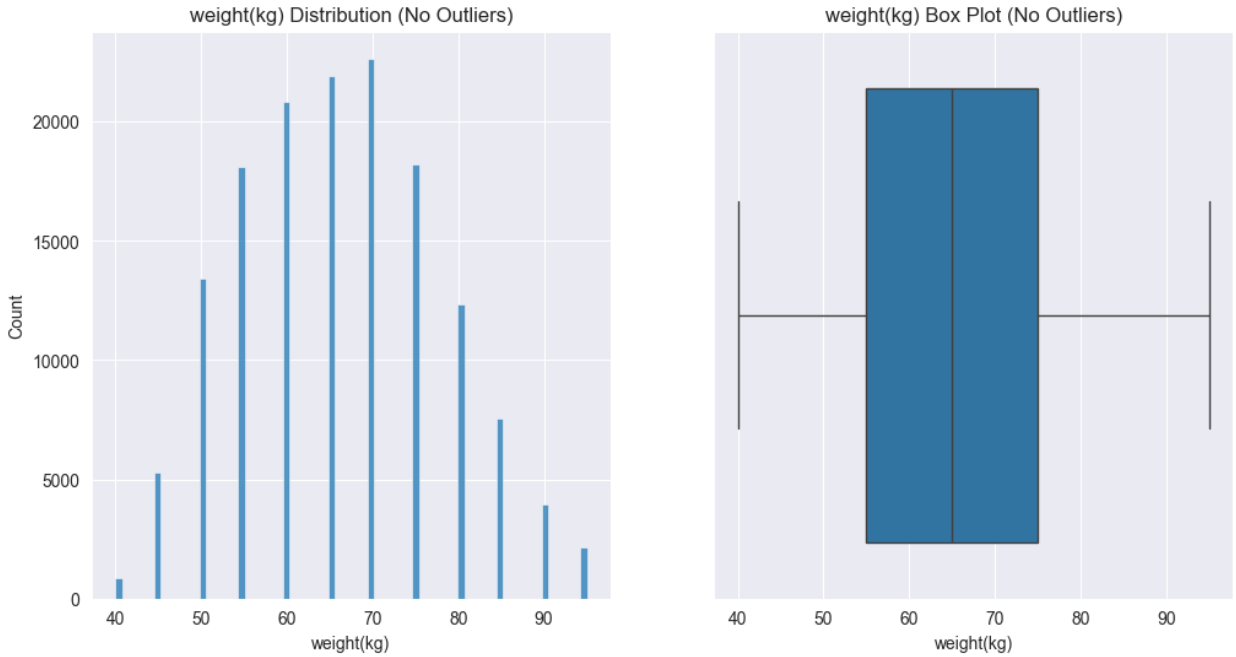

```
outlier_removal('AST', df['AST'])
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_18984\232537705.py:19:
UserWarning: Boolean Series key will be reindexed to match DataFrame
index.
  df_no_outliers = df_no_outliers[outlier_mask]


------ Univariate Analysis for AST ------
IQR Method for Outlier Detection:
Lower Bound: 6.5
Upper Bound: 42.5
Number of Outliers Removed: 10420
```



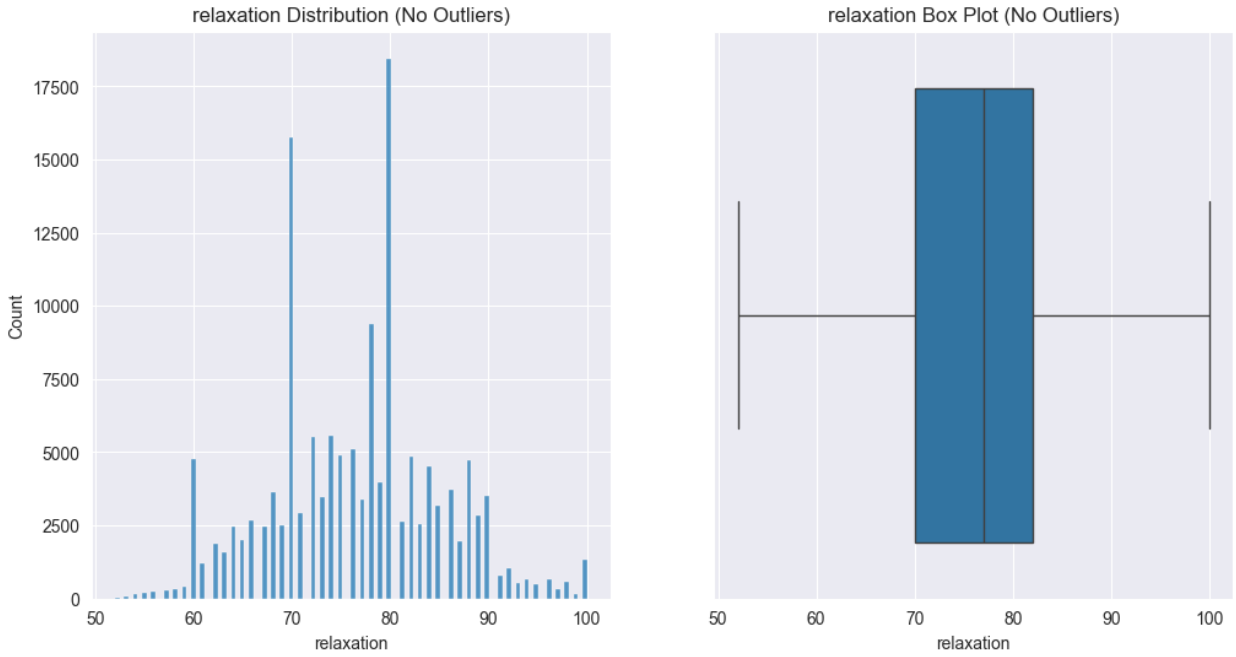

```
outlier_removal('weight(kg)', df['weight(kg)'])
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_18984\232537705.py:19:
UserWarning: Boolean Series key will be reindexed to match DataFrame
index.
  df_no_outliers = df_no_outliers[outlier_mask]


------ Univariate Analysis for weight(kg) ------
IQR Method for Outlier Detection:
Lower Bound: 37.5
Upper Bound: 97.5
Number of Outliers Removed: 12046
```



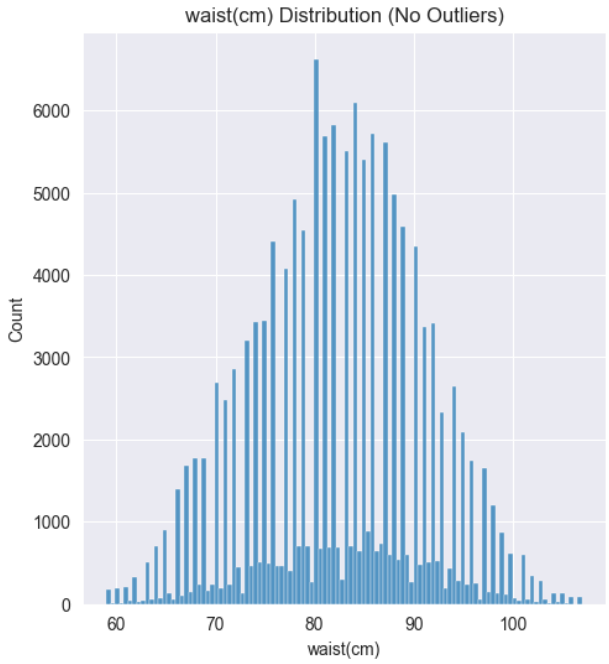

```
outlier_removal('relaxation', df['relaxation'])
```

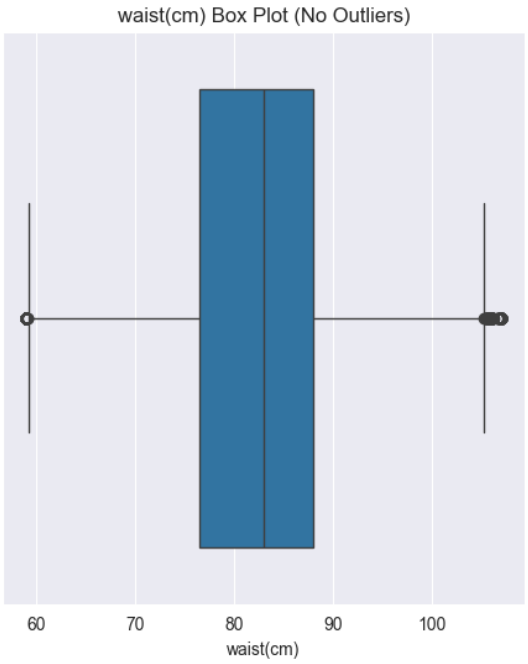
```
C:\Users\DELL\AppData\Local\Temp\ipykernel_18984\232537705.py:19:
UserWarning: Boolean Series key will be reindexed to match DataFrame
index.
  df_no_outliers = df_no_outliers[outlier_mask]


------ Univariate Analysis for relaxation ------
IQR Method for Outlier Detection:
Lower Bound: 52.0
Upper Bound: 100.0
Number of Outliers Removed: 12906
```




```
outlier_removal('waist(cm)', df['waist(cm)'])
```

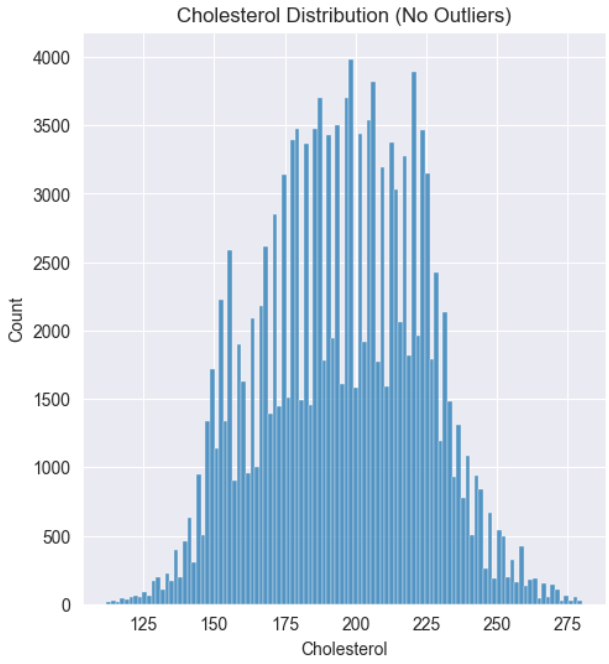
```
C:\Users\DELL\AppData\Local\Temp\ipykernel_18984\232537705.py:19:
UserWarning: Boolean Series key will be reindexed to match DataFrame
index.
  df_no_outliers = df_no_outliers[outlier_mask]


------ Univariate Analysis for waist(cm) ------
IQR Method for Outlier Detection:
Lower Bound: 59.0
Upper Bound: 107.0
Number of Outliers Removed: 13151
```

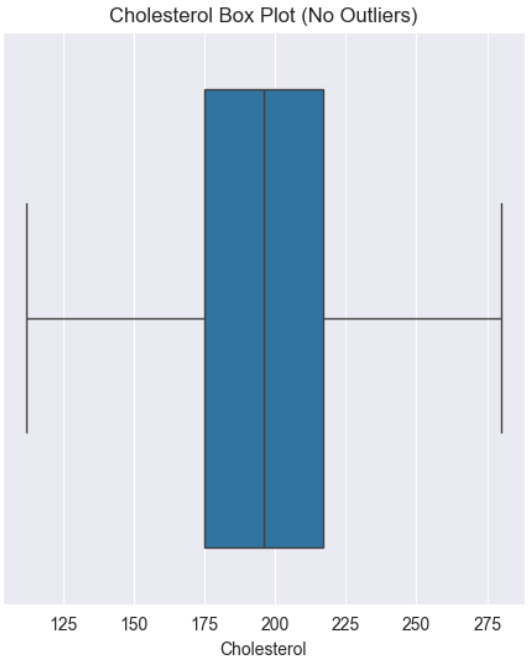





```
outlier_removal('Cholesterol', df['Cholesterol'])
```

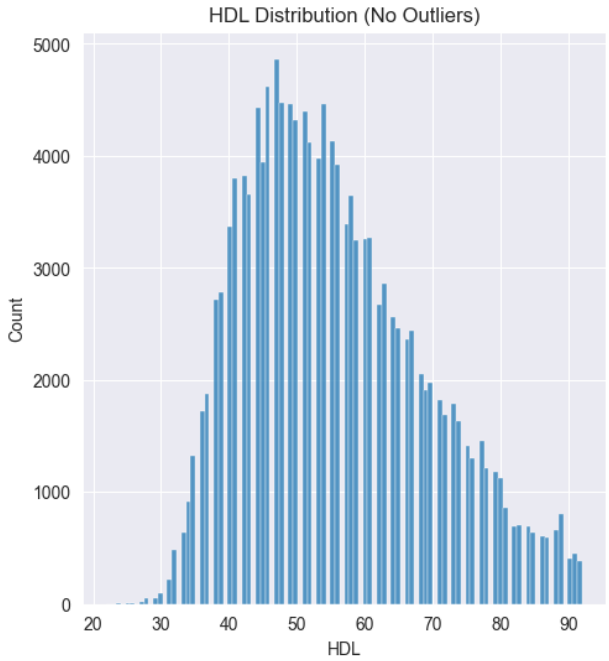
```
C:\Users\DELL\AppData\Local\Temp\ipykernel_18984\232537705.py:19:
UserWarning: Boolean Series key will be reindexed to match DataFrame
index.
  df_no_outliers = df_no_outliers[outlier_mask]


------ Univariate Analysis for Cholesterol ------
IQR Method for Outlier Detection:
Lower Bound: 112.0
Upper Bound: 280.0
Number of Outliers Removed: 13518
```

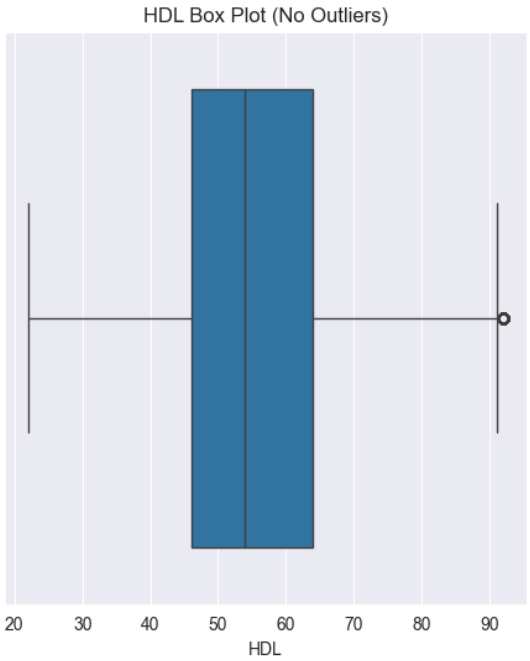





```
outlier_removal('HDL', df['HDL'])
```

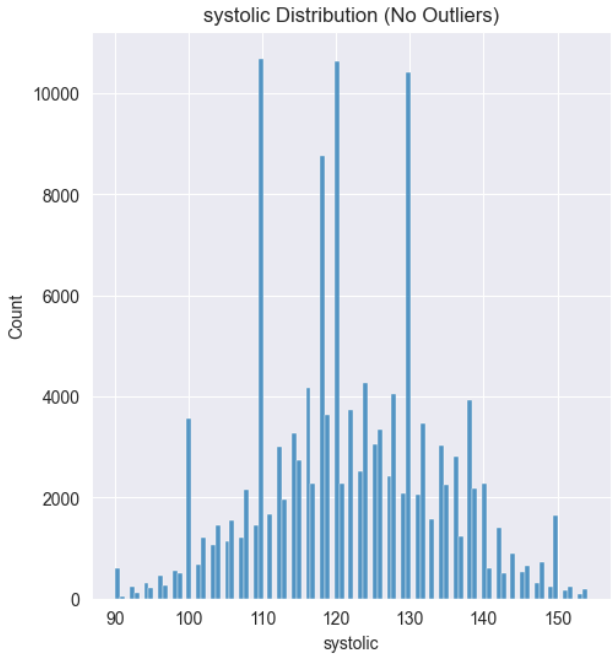
```
C:\Users\DELL\AppData\Local\Temp\ipykernel_18984\232537705.py:19:
UserWarning: Boolean Series key will be reindexed to match DataFrame
index.
  df_no_outliers = df_no_outliers[outlier_mask]


------ Univariate Analysis for HDL ------
IQR Method for Outlier Detection:
Lower Bound: 16.5
Upper Bound: 92.5
Number of Outliers Removed: 15362
```

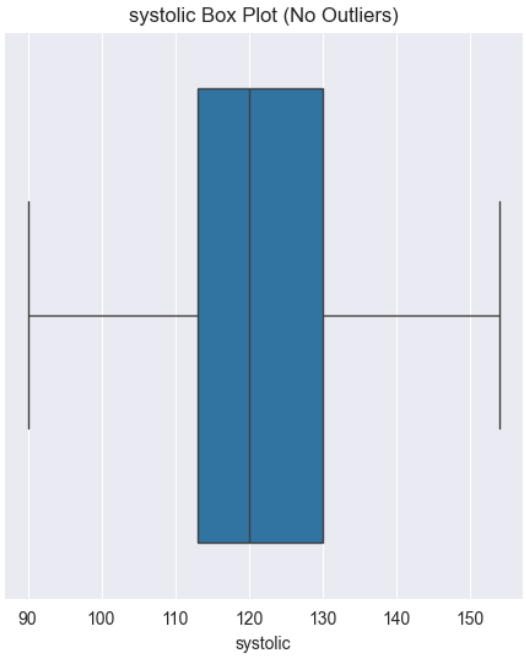




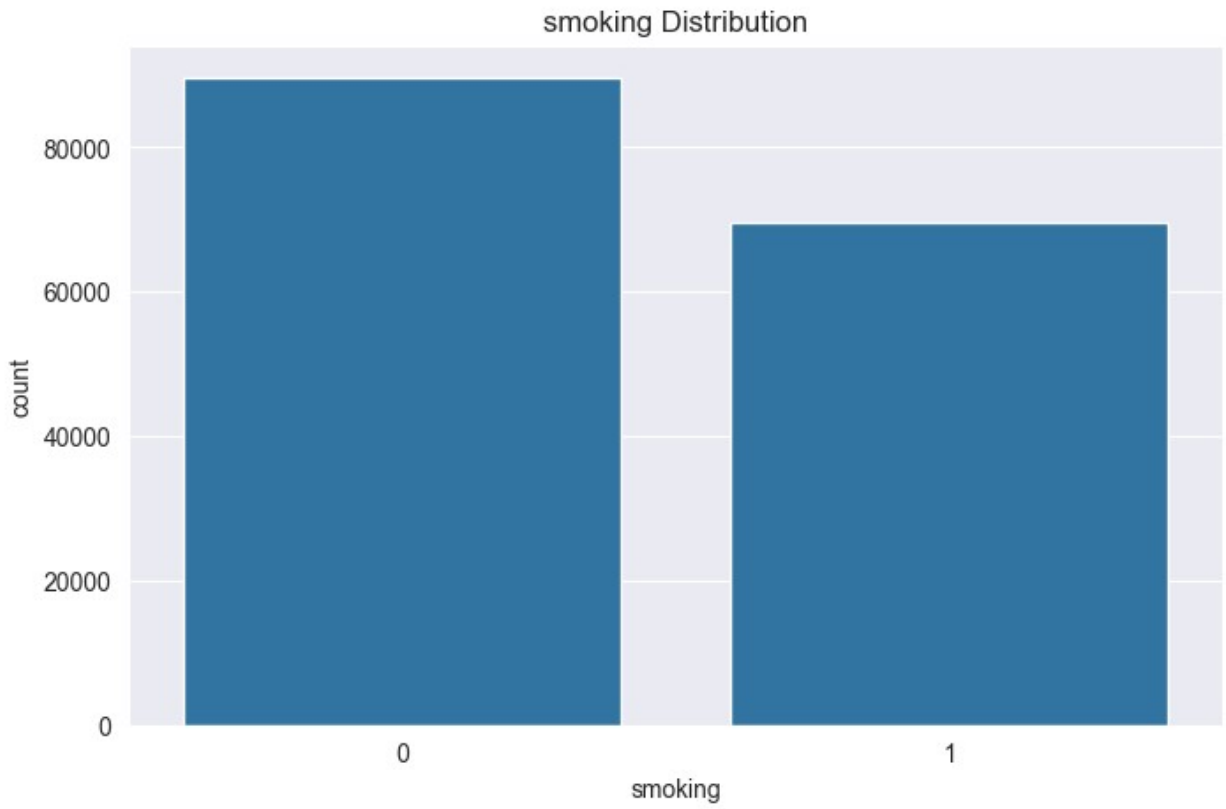

```
outlier_removal('systolic', df['systolic'])
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_18984\232537705.py:19:
UserWarning: Boolean Series key will be reindexed to match DataFrame
index.
  df_no_outliers = df_no_outliers[outlier_mask]


------ Univariate Analysis for systolic ------
IQR Method for Outlier Detection:
Lower Bound: 90.0
Upper Bound: 154.0
Number of Outliers Removed: 16543
```






```
categorical_univariate_analysis('smoking', df['smoking'])


------ Univariate Analysis for smoking ------
Frequency Distribution:
smoking
0    89603
1    69653
Name: count, dtype: int64
```




```
df_no_outliers.describe()
```

	id	ALT	AST	hearing(left)	\
count	142713.000000	142713.000000	142713.000000	142713.000000	
mean	79659.734208	23.992713	24.071290	1.024959	
std	45975.327633	10.632297	6.154968	0.156001	
min	0.000000	1.000000	7.000000	1.000000	
25%	39882.000000	16.000000	20.000000	1.000000	
50%	79641.000000	21.000000	23.000000	1.000000	
75%	119472.000000	30.000000	28.000000	1.000000	
max	159255.000000	56.000000	42.000000	2.000000	

	weight(kg)	hearing(right)	relaxation	waist(cm)	\
count	142713.000000	142713.000000	142713.000000	142713.000000	
mean	66.363898	1.024539	76.493059	82.493755	
std	11.663062	0.154715	8.651413	8.443666	
min	40.000000	1.000000	52.000000	59.000000	
25%	60.000000	1.000000	70.000000	77.000000	
50%	65.000000	1.000000	77.000000	83.000000	
75%	75.000000	1.000000	82.000000	88.000000	
max	95.000000	2.000000	100.000000	107.000000	

	Cholesterol	HDL	systolic	smoking
count	142713.000000	142713.00000	142713.000000	142713.000000
mean	195.355469	55.64711	121.938303	0.430991

```
std        28.124415      13.30671      12.149263       0.495217
min       112.000000      22.00000      90.000000       0.000000
25%       175.000000      45.00000     113.000000       0.000000
50%       196.000000      54.00000     120.000000       0.000000
75%       217.000000      64.00000     130.000000       1.000000
max       280.000000      92.00000     154.000000       1.000000
```

```
numeric_columns_to_normalize = ['ALT', 'AST', 'weight(kg)',
'waist(cm)', 'Cholesterol', 'HDL', 'relaxation', 'systolic']

# Create a copy of the DataFrame to avoid modifying the original data
df_normalized_zscore = df_no_outliers.copy()

# Apply Z-Score Normalization to selected columns
scaler_zscore = StandardScaler()
df_normalized_zscore[numeric_columns_to_normalize] =
scaler_zscore.fit_transform(
    df_no_outliers[numeric_columns_to_normalize])
df_normalized_zscore.reset_index(drop=True, inplace=True)

# Display the first few rows of the normalized DataFrame
print("Z-Score Normalized DataFrame:")
print(df_normalized_zscore)
```

```
Z-Score Normalized DataFrame:
            id       ALT       AST  hearing(left)  weight(kg)
hearing(right)  \
0            0  0.094739 -0.336524              1   -0.545647
1
1            1 -0.093368  0.475830              2   -0.116942
2
2            2  0.659059  0.475830              1    0.740469
1
3            3  0.282846 -0.661466              1    2.455290
1
4            4 -1.033902 -0.823937              1   -0.545647
1
...        ...       ...       ...            ...         ...
...
142708  159251  0.188792  0.150888              1   -1.831764
1
142709  159252 -0.375528 -0.498995              1    0.740469
1
142710  159253 -1.410115 -1.473821              1   -1.403058
1
142711  159254 -0.657688 -0.336524              1    0.740469
1
142712  159255 -0.751742 -0.498995              1   -1.831764
1
```

```
        relaxation  waist(cm)  Cholesterol       HDL  systolic
smoking
0         1.214481  -0.176909    -0.830437 -1.175885  1.075106
1
1         0.752127   0.770550    -0.048196  0.101670  1.980514
0
2        -0.172580  -0.176909    -0.617098 -0.800134 -0.324161
1
3         1.330070   2.665468    -0.545985 -1.326186  0.745867
0
4        -0.056992  -0.236125    -1.434896 -0.875284 -0.077232
1
...            ...        ...          ...       ...       ...       ..
.
142708    0.405362  -1.598098     1.516287  1.228925  0.416627
0
142709    0.405362  -0.058477     0.627376  0.627722 -0.159541
0
142710   -0.750523  -1.953395    -0.225978  2.356180 -0.653400
0
142711    1.561246   1.125847    -1.079332 -0.048631 -0.077232
1
142712    1.214481  -0.721698    -0.332647  2.356180  0.252008
0

[142713 rows x 12 columns]
```

```
# Dropping hearing and systolic and waist due to high co relation with
hearing and relaxation and weight
df_drop = df_normalized_zscore.drop(columns=['hearing(left)',
'systolic', 'waist(cm)', 'ALT'])
```

We either drop these features due to strong Co-Relation with their other features or use the dataframe from PCA both produced dimensionality reduction

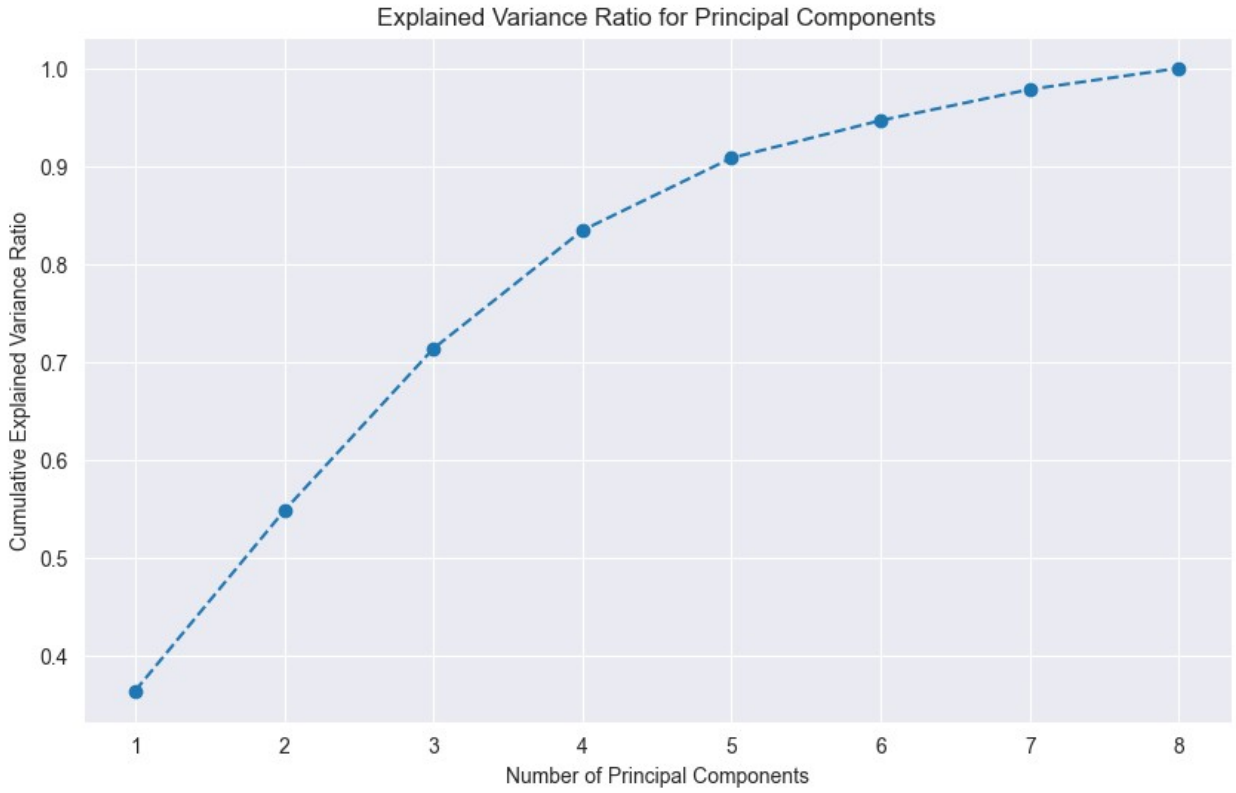
```
# Assuming 'df_normalized_zscore' is your DataFrame with outliers
removed and normalized
numeric_columns_for_pca = ['ALT', 'AST', 'weight(kg)', 'relaxation',
'waist(cm)', 'Cholesterol', 'HDL', 'systolic']

# Standardize the data (important for PCA)
features_standardized_for_pca =
scaler_zscore.fit_transform(df_normalized_zscore[numeric_columns_for_p
ca])

# Apply PCA for dimensionality reduction
pca_for_replacement = PCA()
principal_components_for_replacement =
pca_for_replacement.fit_transform(features_standardized_for_pca)
```

```python
# Variance explained by each principal component
explained_variance_ratio =
pca_for_replacement.explained_variance_ratio_

# The variable 'principal_components_for_replacement' contains the
transformed data
print("Principal Components:")
print(pd.DataFrame(principal_components_for_replacement,
                   columns=[f'PC{i + 1}' for i in
range(len(numeric_columns_for_pca))]))
```

```
Principal Components:
             PC1       PC2       PC3       PC4       PC5       PC6
PC7  \
0       0.681363  1.156628 -0.712029  1.305247 -0.986761 -0.277206 -
0.070044
1       1.263529  1.473325  0.292802  0.499670  0.211427  0.764212 -
0.730433
2       0.697273 -1.001847 -0.274295  0.625430 -0.200735 -0.260789
0.167742
3       3.462270  0.289643 -2.209468 -0.574935  0.709304  0.002516
0.412910
4      -0.849491  0.219042 -1.696362  1.002570 -0.408563  0.246772
0.066247
...          ...       ...       ...       ...       ...       ...
...
142708 -1.548713  1.315786  2.246320 -0.237603 -0.745740 -0.489491 -
0.264905
142709 -0.034539  0.527167 -0.058236 -0.942857  0.515105 -0.272927
0.452416
142710 -3.765110  0.961048  0.218436 -0.184108  0.988913 -0.551913 -
0.168472
142711  0.984868  0.800524 -1.291138  0.399962  0.869098  0.102761
1.230434
142712 -1.921453  2.090212  1.020938  0.597234  1.062709 -0.366772
0.427768

             PC8
0      -0.293692
1      -0.280173
2       0.550928
3      -0.275432
4      -0.225743
...          ...
142708 -0.129784
142709  0.450742
142710  0.213407
142711 -0.504714
142712 -0.992068
```

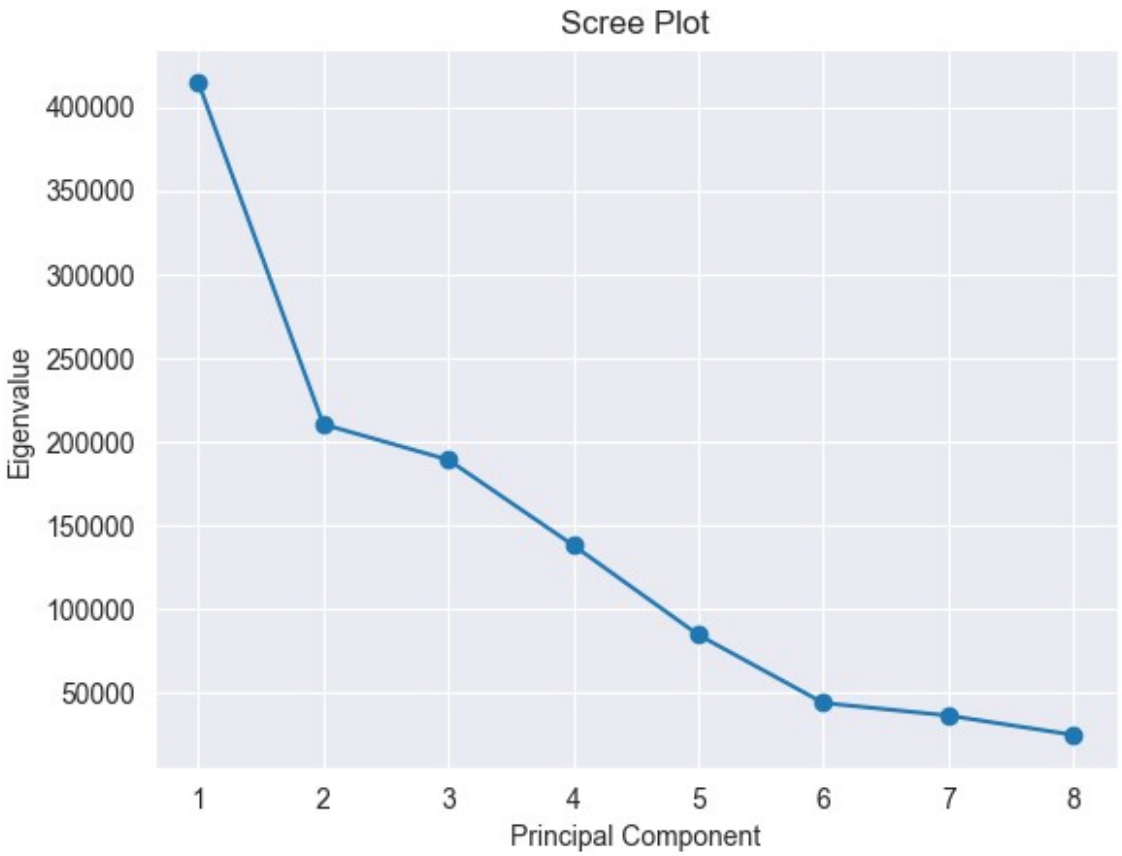
```
[142713 rows x 8 columns]
```

```
# Plotting the explained variance ratio
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(explained_variance_ratio) + 1),
explained_variance_ratio.cumsum(), marker='o', linestyle='--')
plt.title('Explained Variance Ratio for Principal Components')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance Ratio')
plt.grid(True)
plt.show()
```



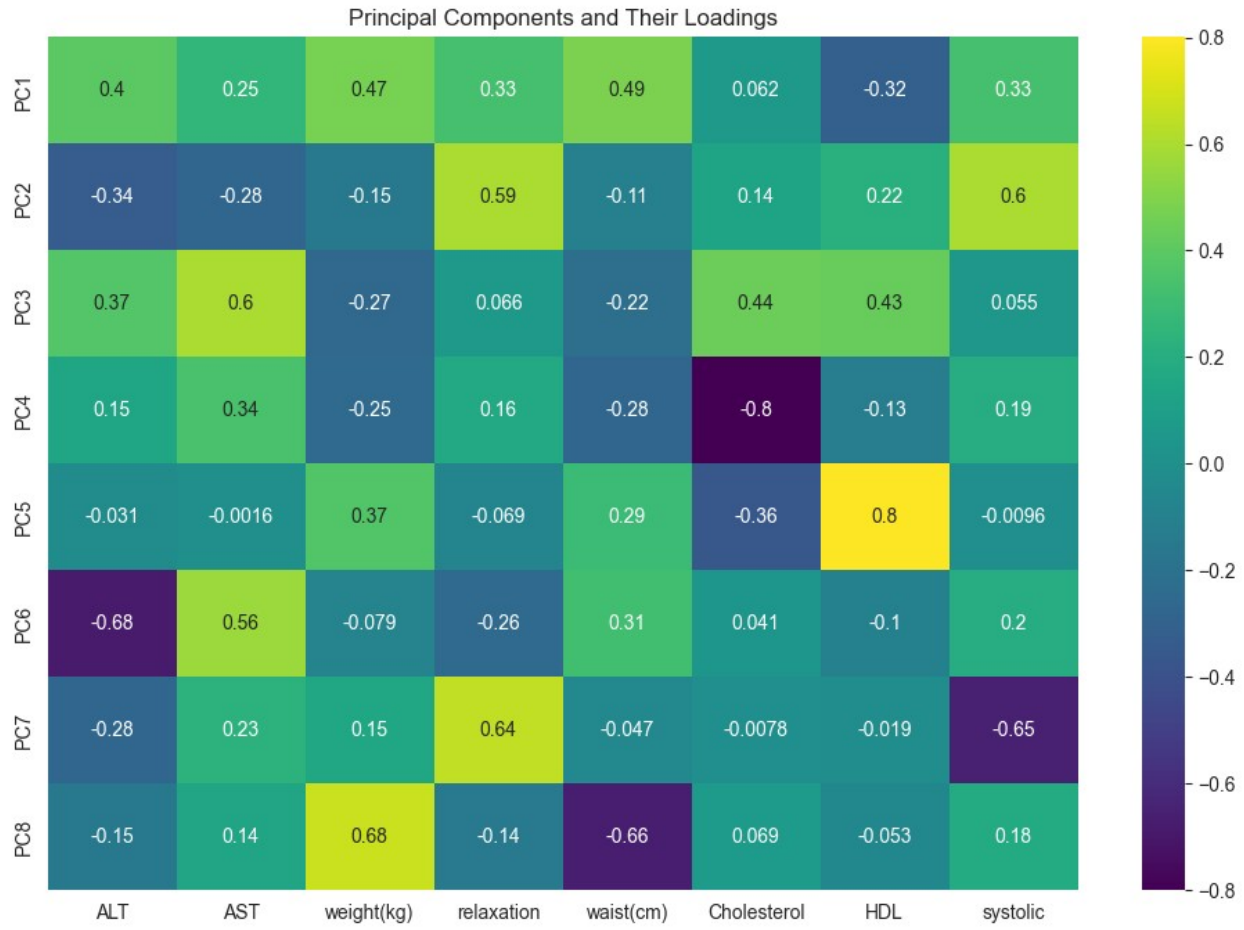
We deduced from the variance ratio that after the pc6 no huge variance occurs

```
# Scree Plot
eigenvalues = pca_for_replacement.singular_values_ ** 2
plt.plot(range(1, len(eigenvalues) + 1), eigenvalues, marker='o')
plt.title('Scree Plot')
plt.xlabel('Principal Component')
plt.ylabel('Eigenvalue')
plt.show()
```



Here the scree plot confirmed that we should take the first 6 principal components

```
# Heatmap for loadings
plt.figure(figsize=(12, 8))
sns.heatmap(pca_for_replacement.components_, cmap='viridis',
annot=True, xticklabels=numeric_columns_for_pca,
            yticklabels=[f'PC{i + 1}' for i in
range(len(numeric_columns_for_pca))])
plt.title('Principal Components and Their Loadings')
plt.show()
```



Now we will produce the final dataframe to start the ensemble methods with 6 principal components

```
# Assuming 'df_normalized_zscore' is your DataFrame with outliers
removed and normalized
numeric_columns_for_pca = ['ALT', 'AST', 'weight(kg)', 'relaxation',
'waist(cm)', 'Cholesterol', 'HDL', 'systolic']

# Standardize the data (important for PCA)
scaler_for_pca = StandardScaler()
features_standardized_for_pca =
scaler_for_pca.fit_transform(df_normalized_zscore[numeric_columns_for_
pca])

# Apply PCA for dimensionality reduction
pca_for_replacement = PCA(n_components=6)  # Set the number of
components to 6
principal_components_for_replacement =
pca_for_replacement.fit_transform(features_standardized_for_pca)

# Create a DataFrame with the first two columns of 'hearing' from
df_normalized_zscore
```

```
final_data = df_normalized_zscore[['hearing(left)',
'hearing(right)']].copy()

# Add the 6 principal components to final_data
final_data = pd.concat(
    [final_data, pd.DataFrame(principal_components_for_replacement,
columns=[f'PC{i + 1}' for i in range(6)])], axis=1)

# Add the 'smoking' column from df_normalized_zscore
final_data['smoking'] = df_normalized_zscore['smoking']
# Display the final DataFrame
print("Final DataFrame:")
print(final_data)
```

```
Final DataFrame:
        hearing(left)  hearing(right)       PC1       PC2       PC3
PC4  \
0                   1               1  0.681363  1.156628 -0.712029
1.305247
1                   2               2  1.263529  1.473325  0.292802
0.499670
2                   1               1  0.697273 -1.001847 -0.274295
0.625430
3                   1               1  3.462270  0.289643 -2.209468 -
0.574935
4                   1               1 -0.849491  0.219042 -1.696362
1.002570
...               ...             ...       ...       ...       ...
...
142708              1               1 -1.548713  1.315786  2.246320 -
0.237603
142709              1               1 -0.034539  0.527167 -0.058236 -
0.942857
142710              1               1 -3.765110  0.961048  0.218436 -
0.184108
142711              1               1  0.984868  0.800524 -1.291138
0.399962
142712              1               1 -1.921453  2.090212  1.020938
0.597234

              PC5       PC6  smoking
0       -0.986761 -0.277206        1
1        0.211427  0.764212        0
2       -0.200735 -0.260789        1
3        0.709304  0.002516        0
4       -0.408563  0.246772        1
...           ...       ...      ...
142708  -0.745740 -0.489491        0
142709   0.515105 -0.272927        0
142710   0.988913 -0.551913        0
```

```
142711  0.869098  0.102761         1
142712  1.062709 -0.366772         0

[142713 rows x 9 columns]
```

```
# Assuming 'final_data' is your DataFrame
X = final_data.drop(columns=['hearing(left)', 'hearing(right)',
'smoking'])
y = final_data['smoking']

# Split the data into training (70%), validation (15%), and testing
(15%) sets
X_train, X_temp, y_train, y_temp = train_test_split(X, y,
test_size=0.3, random_state=42)
X_valid, X_test, y_valid, y_test = train_test_split(X_temp, y_temp,
test_size=0.5, random_state=42)
y_train = np.ravel(y_train)
y_val = np.ravel(y_valid)
y_test = np.ravel(y_test)
```

Now we splitted the data to 70% training 15% validation 15% testing

```
class BaggingClassifier:
    def __init__(self, n_estimators=50, max_features=0.7,
max_depth=None):
        self.n_estimators = n_estimators
        self.max_features = max_features
        self.max_depth = max_depth
        self.estimators = []

    def fit(self, X, y):
        for _ in range(self.n_estimators):
            indices = np.random.choice(len(X), len(X), replace=True)
            X_bootstrap = X.iloc[indices]
            y_bootstrap = y[indices]
            estimator =
DecisionTreeClassifier(max_features=self.max_features,
max_depth=self.max_depth).fit(
                X_bootstrap, y_bootstrap)
            self.estimators.append(estimator)

    def predict(self, X):
        # Make predictions using all the base classifiers
        predictions = [estimator.predict(X) for estimator in
self.estimators]
        # Aggregate predictions using majority voting
        majority_votes = np.apply_along_axis(lambda x:
np.bincount(x).argmax(), axis=0, arr=predictions)

        return majority_votes
```

```
    def get_params(self, deep=True):
        return {
            'n_estimators': self.n_estimators,
            'max_features': self.max_features,
            'max_depth': self.max_depth
        }

    def set_params(self, **params):
        for param, value in params.items():
            setattr(self, param, value)
        return self

bagging_classifier_basic = BaggingClassifier()
# Fit BaggingClassifier on training data
bagging_classifier_basic.fit(X_train, y_train)

# Predict on validation set
y_pred = bagging_classifier_basic.predict(X_valid)
# Evaluate accuracy
accuracy = accuracy_score(y_valid, y_pred)
print(f"Accuracy on the validation set (Bagging): {accuracy:.2%}")
```

```
Accuracy on the validation set (Bagging): 64.87%
```

```
class AdaBoostClassifier:
    def __init__(self, n_estimators=50, learning_rate=1.0):
        self.n_estimators = n_estimators
        self.learning_rate = learning_rate
        self.estimators = []
        self.weights = []

    def fit(self, X, y):
        # Initialize Equal Weights
        self.weights = np.ones(len(X)) / len(X)
        for _ in range(self.n_estimators):
            # Train a decision stump
            estimator = DecisionTreeClassifier(max_depth=1).fit(X, y,
sample_weight=self.weights)

            # Compute Error rate
            predictions = estimator.predict(X)
            incorrect = (predictions != y)
            error_rate = np.dot(self.weights, incorrect)

            # Compute Alpha_t
            alpha = self.learning_rate * np.log((1 - error_rate) /
error_rate)
            self.estimators.append((estimator, alpha))

            # Reweighting
```

```
            self.weights *= np.exp(-y * alpha * predictions)

            # Normalization >> SUMMATION = 1
            self.weights /= np.sum(self.weights)

    def predict(self, X):
        predictions = np.empty(len(X))
        for estimator, alpha in self.estimators:
            predictions += alpha * estimator.predict(X)
        return np.sign(predictions)

    def get_params(self, deep=True):
        return {'n_estimators': self.n_estimators, 'learning_rate':
self.learning_rate}

    def set_params(self, **params):
        if 'n_estimators' in params:
            self.n_estimators = params['n_estimators']
        if 'learning_rate' in params:
            self.learning_rate = params['learning_rate']
        return self

boosting_classifier_basic = AdaBoostClassifier()

# Fit BaggingClassifier on training data
boosting_classifier_basic.fit(X_train, y_train)

# Predict on validation set
y_pred = boosting_classifier_basic.predict(X_valid)
# Evaluate accuracy
accuracy = accuracy_score(y_valid, y_pred)
print(f"Accuracy on the validation set (Boosting): {accuracy:.2%}")
```

```
Accuracy on the validation set (Boosting): 53.17%
```

```
class RandomForestClassifier:
    def __init__(self, n_estimators=500, max_features='sqrt',
max_depth=20, min_samples_split=10, min_samples_leaf=1):
        self.n_estimators = n_estimators
        self.max_features = max_features
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf
        self.max_depth = max_depth
        self.estimators = []

    def fit(self, X, y):
        for _ in range(self.n_estimators):
            indices = np.random.choice(len(X), len(X), replace=True)
            X_bootstrap = X.iloc[indices]
            y_bootstrap = y[indices]
            estimator =
```

```
DecisionTreeClassifier(max_features=self.max_features,
max_depth=self.max_depth, min_samples_split = self.min_samples_split,
min_samples_leaf = self.min_samples_leaf).fit(X_bootstrap,
y_bootstrap)
            self.estimators.append(estimator)

    def predict(self, X):
        # Make predictions using all the base classifiers
        predictions = [estimator.predict(X) for estimator in
self.estimators]
        # Aggregate predictions using majority voting
        majority_votes = np.apply_along_axis(lambda x:
np.bincount(x).argmax(), axis=0, arr=predictions)
        return majority_votes

    def get_params(self, deep=True):
        return {
            'n_estimators': self.n_estimators,
            'max_features': self.max_features,
            'min_samples_split': self.min_samples_split,
            'min_samples_leaf': self.min_samples_leaf,
            'max_depth': self.max_depth
        }

    def set_params(self, **params):
        for param, value in params.items():
            setattr(self, param, value)
        return self

random_forest_classifier_basic = RandomForestClassifier()

# Fit BaggingClassifier on training data
random_forest_classifier_basic.fit(X_train, y_train)

# Predict on validation set
y_pred = random_forest_classifier_basic.predict(X_valid)
# Evaluate accuracy
accuracy = accuracy_score(y_valid, y_pred)
print(f"Accuracy on the validation set (Random Forest):
{accuracy:.2%}")
```

```
Accuracy on the validation set (Random Forest): 66.89%
```

```
from sklearn.metrics import make_scorer
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV

scoring = make_scorer(accuracy_score)  # Use accuracy as the scoring
metric
bagging_classifier = BaggingClassifier()
param_distributions = {
```

```
    'n_estimators': [50, 100, 200],
    'max_features': [0.5, 0.7, 0.9],
    'max_depth': [None, 5, 10]
}
randomized_search = RandomizedSearchCV(bagging_classifier,
param_distributions, n_iter=1, cv=5, n_jobs=-1,
                                       random_state=42,
scoring=scoring)
randomized_search.fit(X_train, y_train)
best_bagging_model = randomized_search.best_estimator_

y_pred = best_bagging_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print("\nBest hyperparameters for Bagging Classifier (Randomized
Search):")
print(best_bagging_model.get_params())
print(f"Bagging Classifier Randomized Search Accuracy:
{accuracy:.2%}")
```

```
Best hyperparameters for Bagging Classifier (Randomized Search):
{'n_estimators': 200, 'max_features': 0.9, 'max_depth': None}
Bagging Classifier Randomized Search Accuracy: 65.71%
```

```
param_distributions = {
    'n_estimators': [20, 100, 200, 300],
    'learning_rate': [0.5,1.0]
}
adaBoost_classifier = AdaBoostClassifier()
randomized_search = RandomizedSearchCV(adaBoost_classifier,
param_distributions, n_iter=1, cv=5, n_jobs=-1,
                                       random_state=42,
scoring=scoring)
randomized_search.fit(X_train, y_train)
best_boosting_model = randomized_search.best_estimator_

y_pred = best_boosting_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print("\nBest hyperparameters for AdaBoost Classifier (Randomized
Search):")
print(best_boosting_model.get_params())
print(f"Boosting Classifier Randomized Search Accuracy:
{accuracy:.2%}")
```

```
Best hyperparameters for AdaBoost Classifier (Randomized Search):
{'n_estimators': 100, 'learning_rate': 0.5}
Boosting Classifier Randomized Search Accuracy: 43.62%
```

```
param_grid = {
    'n_estimators': [150, 250, 350, 450],
    'max_depth': [15,25,35,45],
    'min_samples_split': [2, 4, 8],
    'min_samples_leaf': [1, 2, 4]
}
randomForest_classifier = RandomForestClassifier()
randomized_search = RandomizedSearchCV(randomForest_classifier,
param_distributions, n_iter=1, cv=5, n_jobs=-1,
                                       random_state=42,
scoring=scoring)
randomized_search.fit(X_train, y_train)
best_randomForest_model = randomized_search.best_estimator_

y_pred = best_randomForest_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print("\nBest hyperparameters for Random Forest Classifier (Randomized
Search):")
print(best_randomForest_model.get_params())
print(f"Random Forest Classifier Randomized Search Accuracy:
{accuracy:.2%}")
```

```
Best hyperparameters for Random Forest Classifier (Randomized Search):
{'n_estimators': 100, 'max_features': 'sqrt', 'min_samples_split': 10,
'min_samples_leaf': 1, 'max_depth': 20}
Random Forest Classifier Randomized Search Accuracy: 66.82%
```