```
#0. Load & Prepare the Titanic Dataset
import pandas as pd
# Example: Loading from a public GitHub source
url = "https://raw.githubusercontent.com/nevendujmovic/titanic/main/titanic.csv"
df = pd.read_csv(url)
# Quick look at the data
print(df.head())
#Basic Preprocessing
#Fill missing ages:
df['Age'] = df['Age'].fillna(df['Age'].median())
#Encode categorical features manually:
df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})
df['Embarked'] = df['Embarked'].fillna('S')
df['Embarked'] = df['Embarked'].map({'S': 0, 'C': 1, 'Q': 2})
#Select features for analysis:
features = df[['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked']]
labels = df['Survived']
# Drop rows with missing 'Fare' (if any)
df = df.dropna(subset=['Fare'])
# View a few values
print(df[['Fare']].head())
```

```
   PassengerId Survived  Pclass                                          Name  \
0          343       No       2                   Collander, Mr. Erik Gustaf
1           76       No       3                      Moen, Mr. Sigurd Hansen
2          641       No       3                        Jensen, Mr. Hans Peder
3          568       No       3  Palsson, Mrs. Nils (Alma Cornelia Berglund)
4          672       No       1                        Davidson, Mr. Thornton

      Sex   Age  SibSp  Parch      Ticket     Fare  Cabin Embarked
0    male  28.0      0      0      248740  13.0000    NaN        S
1    male  25.0      0      0      348123   7.6500  F G73        S
2    male  20.0      0      0      350050   7.8542    NaN        S
3  female  29.0      0      4      349909  21.0750    NaN        S
4    male  31.0      1      0  F.C. 12750  52.0000    B71        S
      Fare
0  13.0000
1   7.6500
2   7.8542
3  21.0750
4  52.0000
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from typing import List, Tuple
```

```
#1. Implement a function that drops highly correlated features from a dataset
# Function Definition and Parameters
def drop_highly_correlated_features(X: pd.DataFrame, threshold: float = 0.9) -> Tuple[pd.DataFrame, List[str]]:
    # Compute the Absolute Correlation Matrix
    corr_matrix = X.corr().abs()

    # Extract the Upper Triangle of the Matrix
    upper_triangle = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))

    # Identify Columns to Drop
    to_drop = [col for col in upper_triangle.columns if any(upper_triangle[col] > threshold)]

    # Drop the Identified Columns and Return
    cleaned = X.drop(columns=to_drop)
    return cleaned, to_drop


cleaned_features, dropped = drop_highly_correlated_features(features, threshold=0.9)
print("Columns dropped due to high correlation:", dropped)
print("Remaining columns:", cleaned_features.columns.tolist())
```

```
Columns dropped due to high correlation: []
Remaining columns: ['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked']
```

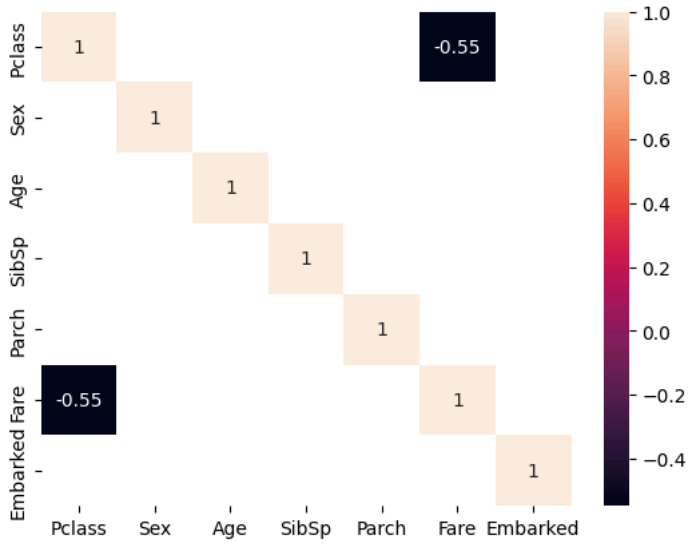
```
#2.Write a function to plot a correlation matrix, highlighting correlations above a given threshold of 0.5
import seaborn as sns
import numpy as np

def plot_correlation_matrix(features, threshold=0.5):

    # Step 1: Compute Correlation Matrix
    corr_matrix = features.corr()
```

```
    # Highlight correlations above the threshold
    mask = np.abs(corr_matrix) < threshold
    corr_matrix = corr_matrix.mask(mask)

    ax = sns.heatmap(corr_matrix, annot=True)
plot_correlation_matrix(features)
```




```
#3 Implement a function to remove features with very low variance
# Function Definition and Parameters
def remove_low_variance_features(X: pd.DataFrame, var_threshold: float = 0.3) -> Tuple[pd.DataFrame, List[str]]:
    # Calculate Variances:
    variances = X.var()

    #Identify Features to Keep:
    keep = variances[variances > var_threshold].index.tolist()

    #Identify Removed Features:
    removed = [c for c in X.columns if c not in keep]

    #Return Filtered Data and Removed List:
    return X[keep].copy(), removed

kept, removed_low_var = remove_low_variance_features(features, var_threshold=0.3)
print("Low-variance removed:", removed_low_var)
print("Kept:", kept.columns.tolist())
```

```
Low-variance removed: ['Sex']
Kept: ['Pclass', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked']
```

```
#4 Write your own PCA function using covariance matrices and eigen decomposition
def manual_pca(X: pd.DataFrame, num_components: int = 2) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:

    # Converts the input pandas DataFrame X into a NumPy array,
    X_np = np.asarray(X, dtype=float)

    X_meaned = X_np - np.mean(X_np, axis=0)

    #Calculates the covariance matrix of the mean-centered data
    cov_matrix = np.cov(X_meaned, rowvar=False)

    #Computes the eigenvalues and eigenvectors of the covariance matrix.
    eigen_values, eigen_vectors = np.linalg.eigh(cov_matrix)

    # Sorts the value of eigen values and eigen vectors
    sorted_index = np.argsort(eigen_values)[::-1]
    eigenvalues_sorted = eigen_values[sorted_index]
    eigenvectors_sorted = eigen_vectors[:, sorted_index]


    eigenvector_subset = eigenvectors_sorted[:, 0:num_components]

    X_reduced = np.dot(X_meaned, eigenvector_subset)
    return X_reduced, eigenvalues_sorted, eigenvectors_sorted

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_numeric = features.select_dtypes(include=[np.number]).copy()
X_scaled = pd.DataFrame(scaler.fit_transform(X_numeric), columns=X_numeric.columns)
```

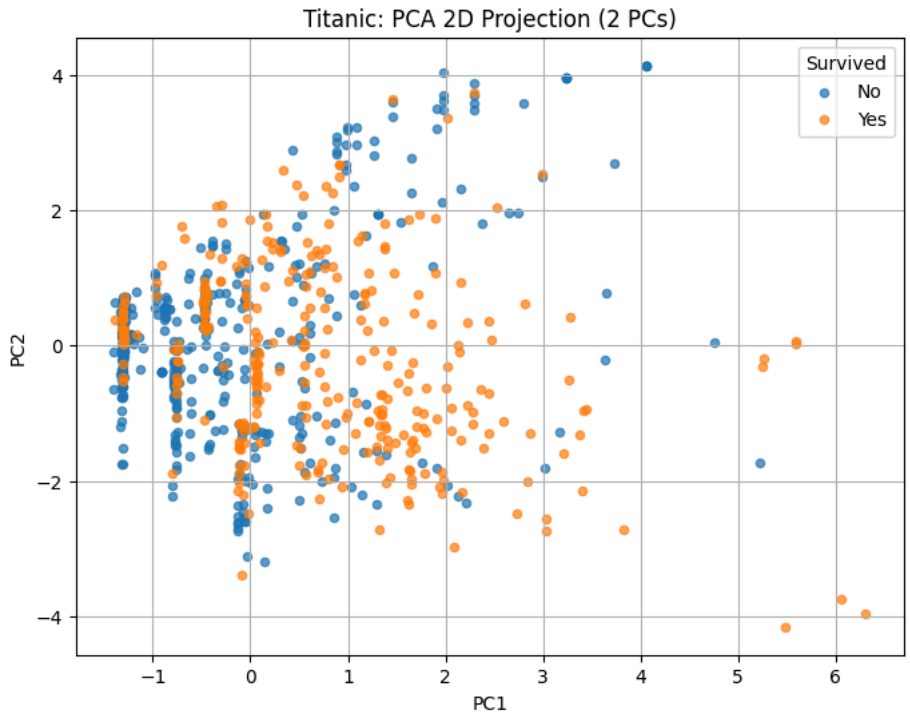
```
#Calls the custom function to perform PCA on the scaled data, reducing it to 2 principal components.
X_pca, eigvals, eigvecs = manual_pca(X_scaled, num_components=2)
print("PCA reduced shape:", X_pca.shape)
```

```
PCA reduced shape: (891, 2)
```

```
#5 Visualize the dataset after projecting it onto the top two principal components

# The function is designed to create a 2-dimensional scatter plot
def plot_pca_2d(X_reduced: np.ndarray, y: pd.Series = None, title="PCA 2D Projection"):
    plt.figure(figsize=(8,6))
    #Handles Class Labels (if y is None vs. else):
    if y is None:
        plt.scatter(X_reduced[:,0], X_reduced[:,1], s=20, alpha=0.7)
    else:
        classes = np.unique(y)
        for cls in classes:
            mask = (y == cls)
            plt.scatter(X_reduced[mask,0], X_reduced[mask,1], label=str(cls), s=20, alpha=0.7)
        plt.legend(title="Survived")

    # The x-axis is labeled "PC1" and the y-axis is labeled "PC2"
    plt.xlabel("PC1")
    plt.ylabel("PC2")
    plt.title(title)
    plt.grid(True)
    # displays the visualization
    plt.show()
plot_pca_2d(X_pca, y=labels, title="Titanic: PCA 2D Projection (2 PCs)")
```




```
#6 Implement a simple Recursive Feature Elimination (RFE) algorithm

# This function calculates the total variance across all features and compute the sum
def accuracy_like_score(X_df: pd.DataFrame, y_series: pd.Series) -> float:
    return X_df.var().sum()
# This function performs Recursive Feature Elimination (RFE) using variance as the criterion for removing features.
def simple_rfe(X: pd.DataFrame, y: pd.Series, num_features_to_keep: int = 3) -> List[str]:
    features_left = list(X.columns)

    # Loop until the number of remaining features is less than or equal to num_features_to_keep
    while len(features_left) > num_features_to_keep:
        scores = {}
        for feature in features_left:
            # Create a reduced dataset by dropping the current feature
            reduced = X[features_left].drop(columns=[feature])

            # calculate the accuracy
            scores[feature] = accuracy_like_score(reduced, y)
        worst_feature = min(scores, key=scores.get)
        features_left.remove(worst_feature)
```

```
            print(f"Removed (worst) feature: {worst_feature}, new set size: {len(features_left)}")

        # Return the remaining features after elimination
        return features_left

print("Score of all features:", accuracy_like_score(features, labels))
reduced_set = simple_rfe(features, labels, num_features_to_keep=3)
print("Selected features by simple RFE:", reduced_set)
```

```
Score of all features: 2642.1466857292476
Removed (worst) feature: Fare, new set size: 6
Removed (worst) feature: Age, new set size: 5
Removed (worst) feature: SibSp, new set size: 4
Removed (worst) feature: Pclass, new set size: 3
Selected features by simple RFE: ['Sex', 'Parch', 'Embarked']
```

```
#7 Manually compute chi-squared statistics for categorical feature selection
def chi_squared(observed: np.ndarray) -> float:
    total = observed.sum()
    rows_sum = observed.sum(axis=1).reshape(-1, 1)
    cols_sum = observed.sum(axis=0).reshape(1, -1)
    expected = (rows_sum @ cols_sum) / total
    expected = np.where(expected == 0, 1e-10, expected)
    chi2 = ((observed - expected) ** 2 / expected).sum()
    return chi2

def compute_chi2_for_features(X: pd.DataFrame, y: pd.Series) -> List[Tuple[str, float]]:
    Xs = X.astype(str)
    ys = y.astype(str)
    results = {}
    for col in Xs.columns:
        contingency = pd.crosstab(Xs[col], ys)
        observed = contingency.values
        results[col] = chi_squared(observed)
    return sorted(results.items(), key=lambda x: x[1], reverse=True)

chi2_results = compute_chi2_for_features(features[['Sex','Embarked']], labels)
print("Chi-squared scores (higher = more predictive):", chi2_results)
```

```
Chi-squared scores (higher = more predictive): [('Sex', np.float64(263.05057407065567)), ('Embarked', np.float64(25.964452888
```

Double-click (or enter) to edit

Q1: Explain what each of the following lines does: rows_sum = observed.sum(axis=1).reshape(-1, 1) cols_sum = observed.sum(axis=0).reshape(1, -1) expected = (rows_sum @ cols_sum) / total

Ans:

rows_sum = observed.sum(axis=1).reshape(-1, 1): Sums each row of the observed matrix and reshapes it into a column vector.

cols_sum = observed.sum(axis=0).reshape(1, -1): Sums each column of the observed matrix and reshapes it into a row vector.

expected = (rows_sum @ cols_sum) / total: Computes the expected frequencies for each cell in the contingency table using matrix multiplication of row and column sums, normalized by the total count.

Q2. Why do we use @ instead of *?

Ans: 1] @ is for matrix multiplication, while * is for element-wise multiplication. 2] In the case of chi-squared, we need matrix multiplication (outer product of row and column sums), not element-wise multiplication

Q3: What is a contingency table and how does pd.crosstab() create it?

Ans: Contingency Table: A contingency table (also called a cross-tabulation or cross-tab) is a table that displays the frequency distribution of two (or more) categorical variables. It shows how the values of one categorical variable relate to the values of another categorical variable. Each cell in the table represents the count of occurrences where the corresponding row and column variables meet.

pd.crosstab() is a pandas function that computes a contingency table (cross-tabulation) between two (or more) categorical variables. It takes in two (or more) Series or arrays and returns a DataFrame with the frequencies of the combinations of the values in the input.

Q4. Why Do We Convert All Feature Values to Strings Before Analysis?

Ans: We convert all feature values to strings before analysis to ensure consistency, especially when working with categorical data. This helps avoid issues with mixed data types, ensures proper handling of categories (like "Male" and "Female"), and allows functions like pd.crosstab() and chi-squared tests to treat all values as discrete categories, leading to correct analysis.

```
#8 Identify and remove categorical features that provide duplicate or redundan information.
```

```
# Function to identifie and removes duplicate columns
def drop_duplicate_categorical_columns(df: pd.DataFrame) -> pd.DataFrame:
    # Identify Potential Categorical Columns
    cat_cols = df.select_dtypes(include=['object', 'category']).columns.tolist()
    # Identify Potential Low-Cardinality Columns:
    small_unique_cols = [c for c in df.columns if df[c].nunique() <= 10 and c not in cat_cols]
    # Create a Combined List of Candidates
    candidates = list(set(cat_cols + small_unique_cols))
    # Find Duplicates Among Candidates
    duplicates = set()
    for i in range(len(candidates)):
        for j in range(i+1, len(candidates)):
            c1, c2 = candidates[i], candidates[j]
            if df[c1].equals(df[c2]):
                duplicates.add(c2)
    # Return the Result
    return df.drop(columns=list(duplicates)), list(duplicates)

df_no_dup, dup_list = drop_duplicate_categorical_columns(df)
print("Dropped categorical duplicates:", dup_list)
```

```
Dropped categorical duplicates: []
```

```
#9 Rank features based on their variance
def rank_features_by_variance(X: pd.DataFrame) -> List[Tuple[str, float]]:
    # Calculate the variance for each feature
    variances = X.var().sort_values(ascending=False)

    # Create a list of tuples
    return list(zip(variances.index.tolist(), variances.values.tolist()))

# Call the function to rank features by their variance
variance_ranking = rank_features_by_variance(features)
print("Top variance features:", variance_ranking[:5])
```

```
Top variance features: [('Fare', 2469.436845743119), ('Age', 169.5124982794234), ('SibSp', 1.216043077466295), ('Pclass', 0.
```

```
#10 Build a complete pipeline combining correlation filtering, variance filtering.

# Function to Perform feature selection and dimensionality reduction
def feature_selection_pipeline(df: pd.DataFrame, variance_thresh: float = 0.5, corr_thresh: float = 0.8, pca_components: in
    # Select only numeric columns from the input DataFrame
    X = df.select_dtypes(include=[np.number]).copy()
    # Compute variance for each numeric feature
    variances = X.var()
    # Keep only features with variance greater than the specified threshold
    keep_vars = variances[variances > variance_thresh].index.tolist()
    # Filter DataFrame to keep only high-variance features
    X_var_filtered = X[keep_vars]
    # Drop highly correlated features
    X_corr_filtered, dropped_corr = drop_highly_correlated_features(X_var_filtered, threshold=corr_thresh)
    # Standardize data
    scaler = StandardScaler()
    X_std = pd.DataFrame(scaler.fit_transform(X_corr_filtered), columns=X_corr_filtered.columns)
    # Apply PCA (Principal Component Analysis)
    X_pca_reduced, eigvals, eigvecs = manual_pca(X_std, num_components=pca_components)
    return {
        "X_initial": X,
        "X_var_filtered": X_var_filtered,
        "X_corr_filtered": X_corr_filtered,
        "pca_projection": X_pca_reduced,
        "dropped_corr": dropped_corr,
        "kept_vars": keep_vars
    }

# Running the pipeline on the given dataset 'features'
pipeline_res = feature_selection_pipeline(features, variance_thresh=0.5, corr_thresh=0.8, pca_components=2)
print("Pipeline kept numeric vars:", pipeline_res['kept_vars'])
print("Pipeline dropped corr columns:", pipeline_res['dropped_corr'])
print("PCA projection shape:", pipeline_res['pca_projection'].shape)

fare = df['Fare'].copy()
```

```
Pipeline kept numeric vars: ['Pclass', 'Age', 'SibSp', 'Parch', 'Fare']
Pipeline dropped corr columns: []
PCA projection shape: (891, 2)
```

```
#11 Identify outliers using IQR method.Use boxplt and swarm plot for the same
fare = df['Fare'].copy()
```

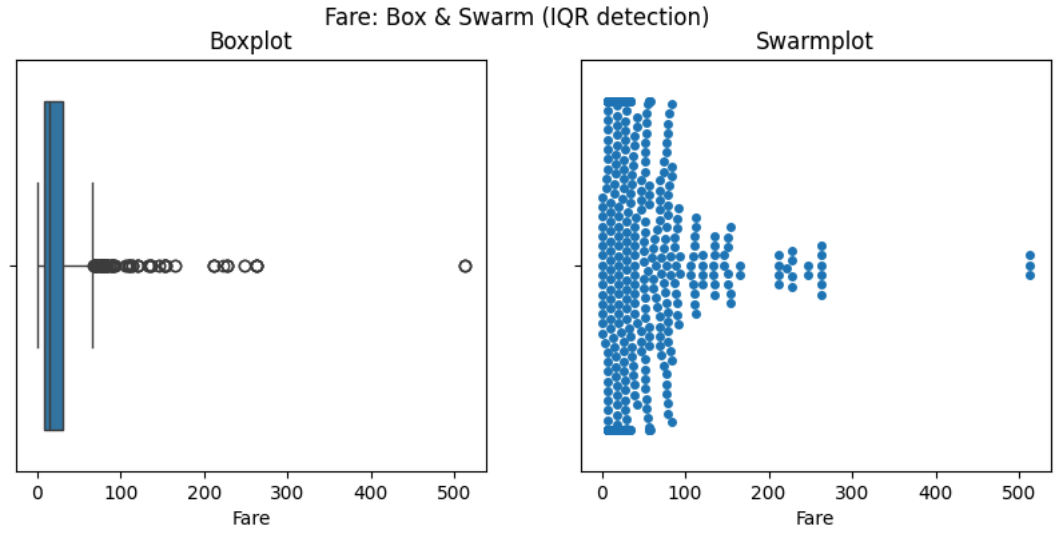
```
# This function implements the standard IQR method for outlier detection.
def detect_outliers_iqr(series: pd.Series, factor: float = 1.5) -> pd.Series:
    # It calculates the first quartile and third quartile of the data and calculates Interquartile Range.
    Q1 = series.quantile(0.25)
    Q3 = series.quantile(0.75)
    IQR = Q3 - Q1
    # It determines the boundaries for "non-outlier" data points
    lower = Q1 - factor * IQR
    upper = Q3 + factor * IQR
    return (series < lower) | (series > upper)

# This function generates two visualizations side-by-side
def plot_box_swarm(series: pd.Series, title="Boxplot & Swarmplot"):
    # Boxplot visualization
    plt.figure(figsize=(10,4))
    plt.subplot(1,2,1)
    sns.boxplot(x=series)
    plt.title("Boxplot")

    # Swarmplot visualization
    plt.subplot(1,2,2)
    sns.swarmplot(x=series)
    plt.title("Swarmplot")
    plt.suptitle(title)
    plt.show()

# The outlier detection function is applied to the fare data, creating a mask of outliers.
outliers_mask = detect_outliers_iqr(fare)
print("IQR outliers count (Fare):", outliers_mask.sum())
plot_box_swarm(fare, title="Fare: Box & Swarm (IQR detection)")
```

```
IQR outliers count (Fare): 116
/usr/local/lib/python3.12/dist-packages/seaborn/categorical.py:3399: UserWarning: 18.6% of the points cannot be placed; you
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.12/dist-packages/seaborn/categorical.py:3399: UserWarning: 65.3% of the points cannot be placed; you
  warnings.warn(msg, UserWarning)
```




```
#12 Remove rows with outliers beyond 1.5*IQR.
# Store original dataset shape
shape_before = df.shape
# Remove outlier rows from 'Fare' using 1.5*IQR method
df_iqr_cleaned = df.loc[~detect_outliers_iqr(df['Fare'])].copy()
# Store dataset shape after outlier removal
shape_after = df_iqr_cleaned.shape
# Print shapes before and after IQR cleaning
print("Shape before:", shape_before, "after IQR clean:", shape_after)
```

```
Shape before: (891, 12) after IQR clean: (775, 12)
```

```
#13 Detect outliers using Z-score method.
# Define a function to detect outliers in a series using the Z-score method
def detect_outliers_zscore(series: pd.Series, threshold: float = 3.0) -> pd.Series:
    # Calculate Z-scores for each value in the series
    zs = (series - series.mean()) / series.std(ddof=0)
    # Return a boolean mask where True indicates an outlier
    return zs.abs() > threshold
# Apply the Z-score outlier detection on the 'Fare' column
z_outliers_mask = detect_outliers_zscore(fare, threshold=3.0)
# Print the total number of outliers detected in 'Fare'
```

```
print("Number of Z-score outliers (Fare):", z_outliers_mask.sum())
```

```
Number of Z-score outliers (Fare): 20
```

```
#14 Replace outliers with median values.
# Define a function to replace outliers in a given column with its median
def replace_outliers_with_median(df: pd.DataFrame, column: str, method: str = 'iqr', factor: float = 1.5) -> pd.Series:
    s = df[column].copy()
    # Detect outliers using the IQR method
    if method == 'iqr':
        mask = detect_outliers_iqr(s, factor=factor)
    # Detect outliers using the Z-score method
    elif method == 'zscore':
        mask = detect_outliers_zscore(s)
    # Raise error if invalid method is provided
    else:
        raise ValueError("method must be 'iqr' or 'zscore'")

    # Calculate the median of the column
    median_val = s.median()
    # Replace all outlier values with the median
    s[mask] = median_val
    return s

fare_replaced_median = replace_outliers_with_median(df, 'Fare', method='iqr')
print("Replaced outliers with median - sample:")
print(fare_replaced_median.head())
```

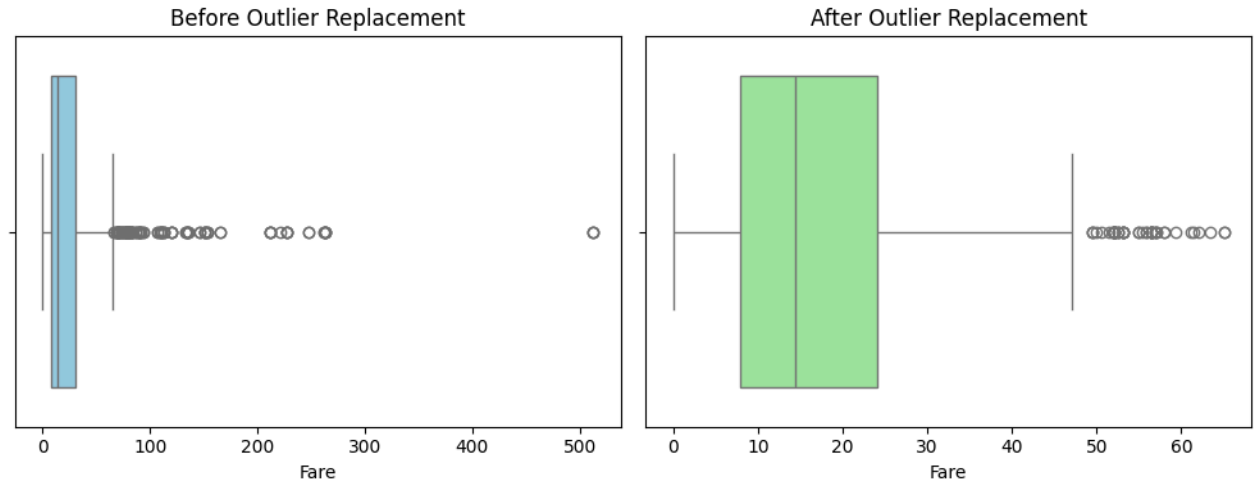
```
Replaced outliers with median - sample:
0    13.0000
1     7.6500
2     7.8542
3    21.0750
4    52.0000
Name: Fare, dtype: float64
```

```
#15 Visualize outliers with boxplots
plt.figure(figsize=(10, 4))

# Boxplot before outlier replacement
plt.subplot(1, 2, 1)
sns.boxplot(x=df['Fare'], color='skyblue')
plt.title("Before Outlier Replacement")

# Boxplot after outlier replacement
plt.subplot(1, 2, 2)
sns.boxplot(x=fare_replaced_median, color='lightgreen')
plt.title("After Outlier Replacement")

# Adjust layout for neat display
plt.tight_layout()
plt.show()
```

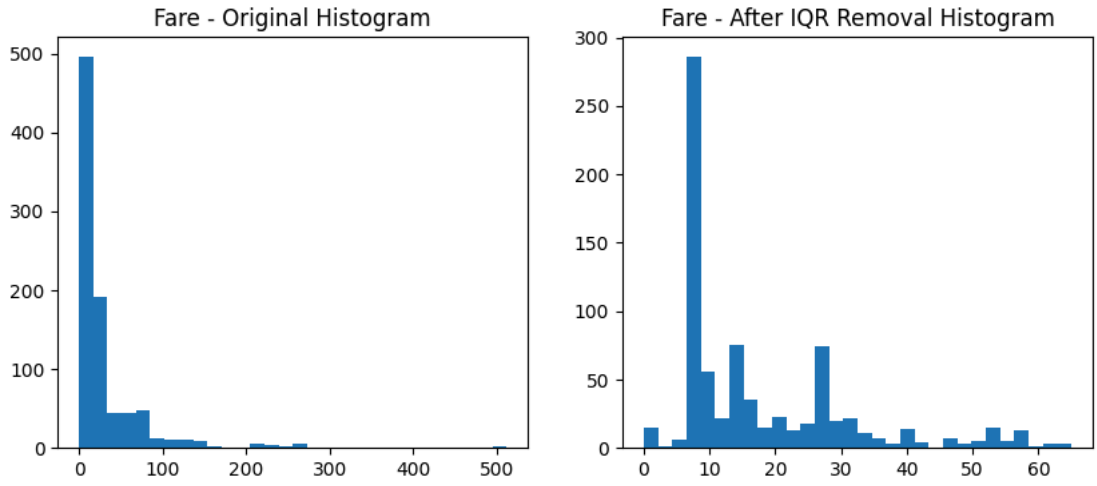



```
#16 Use sns.swarmplot() to identify dense outlier clusters
# Set the figure size (width=8, height=3)
plt.figure(figsize=(8, 3))
# Create a swarmplot of Fare values
sns.swarmplot(x=df['Fare'], color='teal', size=3)
```

```
plt.title("Fare Swarmplot")
plt.xlabel("Fare")
# Display the plot
plt.show()
```

```
/usr/local/lib/python3.12/dist-packages/seaborn/categorical.py:3399: UserWarning: 44.1% of the points cannot be placed; you
  warnings.warn(msg, UserWarning)
```


```
#17 Plot histogram before and after outlier removal
# Create a figure with specified size for side-by-side histograms
plt.figure(figsize=(10, 4))
# Plot histogram of 'Fare' before removing outliers
plt.subplot(1, 2, 1)
plt.hist(df['Fare'].dropna(), bins=30)
plt.title("Fare - Original Histogram")
# Plot histogram of 'Fare' after removing outliers using IQR method
plt.subplot(1, 2, 2)
plt.hist(df_iqr_cleaned['Fare'].dropna(), bins=30)
plt.title("Fare - After IQR Removal Histogram")
# Display both histograms
plt.show()
```




```
#18 Normalize a column using min-max scaling.
# Normalize the 'Fare' column using Min-Max scaling
fare_normalized = (df['Fare'] - df['Fare'].min()) / (df['Fare'].max() - df['Fare'].min())

# Rename the Series for clarity
fare_normalized.name = 'Fare_normalized'

# Display the normalized column details
print(fare_normalized)
```

```
0      0.025374
1      0.014932
2      0.015330
3      0.041136
4      0.101497
         ...
886    0.058694
887    0.014110
888    0.016908
```

```
889    0.015412
890    0.050749
Name: Fare_normalized, Length: 891, dtype: float64
```

```
#19 Standardize a column using z-score normalization.
# Standardize the 'Fare' column using Z-score normalization
fare_standardized = (df['Fare'] - df['Fare'].mean()) / df['Fare'].std()

# Rename the Series for clarity
fare_standardized.name = 'Fare_standardized'

# Display the standardized column details
print(fare_standardized)
```

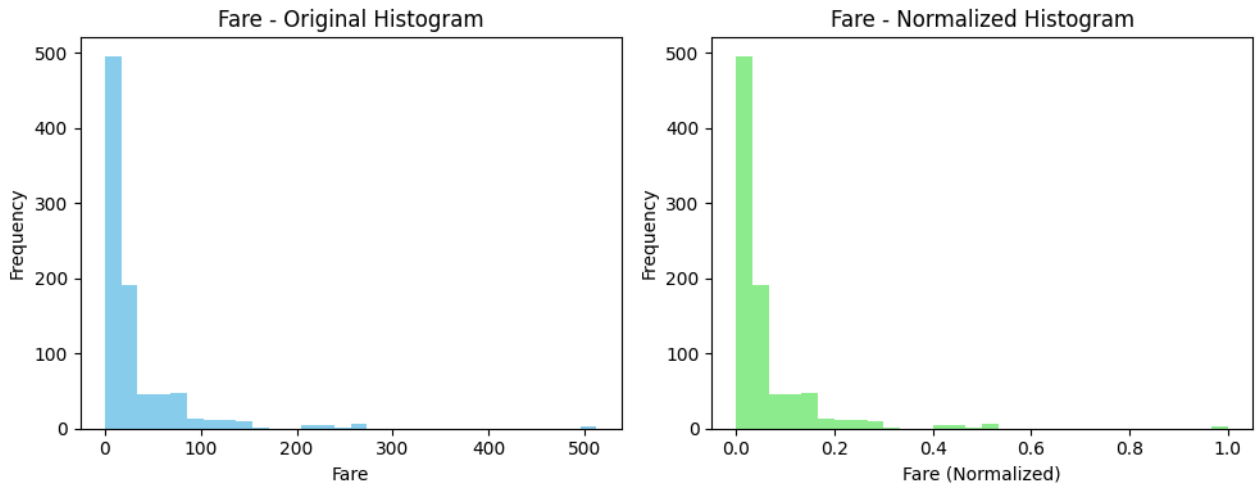
```
0     -0.386454
1     -0.494114
2     -0.490005
3     -0.223957
4      0.398358
         ...
886   -0.042931
887   -0.502582
888   -0.473739
889   -0.489167
890   -0.124850
Name: Fare_standardized, Length: 891, dtype: float64
```

```
#20 Compare histograms before and after normalization
plt.figure(figsize=(10, 4))

# Histogram before normalization
plt.subplot(1, 2, 1)
plt.hist(df['Fare'].dropna(), bins=30, color='skyblue')
plt.title("Fare - Original Histogram")
plt.xlabel("Fare")
plt.ylabel("Frequency")

# Histogram after Min-Max normalization
plt.subplot(1, 2, 2)
plt.hist(fare_normalized.dropna(), bins=30, color='lightgreen')
plt.title("Fare - Normalized Histogram")
plt.xlabel("Fare (Normalized)")
plt.ylabel("Frequency")

# Adjust layout and show both histograms
plt.tight_layout()
plt.show()
```




```
#21 Normalize all numeric columns in a DataFrame.
# Define a function to normalize all numeric columns using Min-Max scaling
def normalize_numeric_columns(df: pd.DataFrame) -> pd.DataFrame:
    # Select only numeric columns
    numeric_df = df.select_dtypes(include=['number'])

    # Apply Min-Max normalization
    normalized_df = (numeric_df - numeric_df.min()) / (numeric_df.max() - numeric_df.min())

    # Rename columns for clarity
    normalized_df = normalized_df.add_suffix('_normalized')

    # Return the normalized DataFrame
```

```
    # Return the normalized DataFrame
    return normalized_df

# Call the function on your DataFrame
df_normalized = normalize_numeric_columns(df)

# Display a sample of the normalized result
print(df_normalized.head())
```

```
   PassengerId_normalized  Pclass_normalized  Sex_normalized  Age_normalized  \
0                0.384270                0.5             0.0        0.346569
1                0.084270                1.0             0.0        0.308872
2                0.719101                1.0             0.0        0.246042
3                0.637079                1.0             1.0        0.359135
4                0.753933                0.0             0.0        0.384267

   SibSp_normalized  Parch_normalized  Fare_normalized  Embarked_normalized
0             0.000          0.000000         0.025374                  0.0
1             0.000          0.000000         0.014932                  0.0
2             0.000          0.000000         0.015330                  0.0
3             0.000          0.666667         0.041136                  0.0
4             0.125          0.000000         0.101497                  0.0
```