PROJECT SUBMISSION EXPLANATION

SMARTWATCH PROJECT

INTRODUCTION OF THE PROJECT:

Brief Overview of the Smartwatch Project:

Purpose and Objectives:

The project aims to analyze data from smartwatch devices to understand the relationship between physical activity, heart rate, and overall fitness. The primary objectives are:

- To identify patterns and correlations between activity levels, heart rate, and fitness metrics like calories burned, distance, and intensity.
- To provide actionable insights and recommendations to optimize user health and fitness.

Problem the Project Aims to Solve:

The project addresses the need to better understand how various factors such as physical activity and heart rate influence fitness outcomes. It seeks to answer questions like:

- What trends exist between activity type, heart rate, and calorie expenditure?
- How can users optimize their fitness routines based on data insights?

Relevance and Importance:

With the increasing use of wearable technology, this project is highly relevant as it leverages data to enhance personal health management. The insights derived can:

- Empower users to make informed fitness decisions.
- Support the development of more effective health and fitness applications.

High-Level Summary of the Approach:

1. **Data Cleaning and Preprocessing**:
    - Handle missing values, duplicates, and normalize data for consistent analysis.
    - Transform categorical variables to numeric formats if necessary.
2. **Exploratory Data Analysis (EDA)**:
    - Visualize metrics like age, gender, heart rate, and steps.
    - Conduct correlation and trend analyses to uncover relationships between variables.
3. **Feature Engineering**:
    - Create new features (e.g., combining steps and distance) to derive additional insights.
4. **Documentation and Reporting**:

- Compile findings into a detailed report, including objectives, methodology, results, and recommendations.
- Share the project on platforms like GitHub and LinkedIn for visibility.

This structured approach ensures a thorough analysis, leading to actionable insights that align with the project's goals.

Part 1: DATA CLEANING AND PREPROCESSING

```
import pandas as pd

# Correct URL to the raw CSV file
url = "https://raw.githubusercontent.com/heera-02/smartwatch-project/main/smartwatch.csv"

# Load the dataset
data = pd.read_csv(url)

# Inspect the dataset
print(data.head())
```

Code Breakdown:

1. **Import Libraries**:
   - `pandas` is imported for data manipulation and analysis.
2. **Dataset Loading**:
   - The `pd.read_csv(url)` function loads the dataset from the specified URL into a Pandas DataFrame.
3. **Inspect the Dataset**:
   - `data.head()` displays the first five rows of the dataset, providing a quick look at its structure, column names, and sample data.

Why It's Important:

1. **Familiarizing with the Data**:
   - Understanding the structure, column names, and data types is crucial for planning the analysis.
   - Identifying potential issues like missing values or inconsistent formats early.
2. **Foundation for Analysis**:
   - This step sets the stage for further preprocessing, cleaning, and exploratory data analysis (EDA).
3. **Efficient Workflow**:
   - Knowing the dataset's structure helps streamline subsequent steps, such as feature engineering or visualization, by aligning with the data's organization.

```
# Check for missing values and duplicates
print("Missing Values:\n", data.isnull().sum())
```

```
print("Duplicate Rows:", data.duplicated().sum())
```

Purpose:

This code checks the dataset for missing values and duplicate rows, which are common issues in raw datasets that can affect analysis accuracy.

Code Breakdown:

1. **Check for Missing Values**:
   - `data.isnull().sum()`:
     - This checks each column for missing (`NaN`) values and provides a count for each column.
2. **Check for Duplicate Rows**:
   - `data.duplicated().sum()`:
     - This identifies duplicate rows in the dataset and returns their count.

Why It's Important:

1. **Data Quality Assessment**:
   - Missing values can distort statistical analyses or cause errors in machine learning models.
   - Duplicate rows can bias results or inflate sample sizes unnecessarily.
2. **Guiding Preprocessing**:
   - Knowing the extent of missing data helps decide on handling strategies (e.g., imputation, removal).
   - Identifying duplicates allows for their removal, ensuring a clean dataset for analysis.

```
# Fill missing values for numerical columns only
numerical_columns = data.select_dtypes(include=['number']).columns
data[numerical_columns] = data[numerical_columns].fillna(data[numerical_columns].median())

# For categorical columns, you can fill missing values with a placeholder (e.g., 'Unknown') or mode
categorical_columns = data.select_dtypes(include=['object']).columns
data[categorical_columns] = data[categorical_columns].fillna('Unknown')

    # Confirm no missing values remain
    print("Missing Values After Handling:\n", data.isnull().sum())
```

Purpose:

This code handles missing values in the dataset by applying specific strategies for numerical and categorical columns to ensure no missing data remains.

Code Breakdown:

1. **Identify Column Types**:
   - `numerical_columns`: Selects columns with numerical data types using `select_dtypes(include=['number'])`.
   - `categorical_columns`: Selects columns with categorical (string) data types using `select_dtypes(include=['object'])`.
2. **Handle Missing Values**:
   - **Numerical Columns**:
     - Missing values are replaced with the median of each column using `.fillna(data[numerical_columns].median())`.
     - Median is robust to outliers and ensures the distribution is minimally affected.
   - **Categorical Columns**:
     - Missing values are replaced with `'Unknown'` or another placeholder using `.fillna('Unknown')`.
     - Alternatively, the mode (most frequent value) can be used to fill missing values.
3. **Confirm Handling**:
   - `data.isnull().sum()` ensures no missing values remain in any column.

Why It's Important:

1. **Data Integrity**:
   - Missing values can lead to errors or inaccurate results during analysis or modeling.
   - Filling values ensures the dataset is complete and ready for further processing.
2. **Preserving Data Characteristics**:
   - Median imputation for numerical data retains the central tendency without being skewed by outliers.
   - Using placeholders for categorical data avoids introducing unintended bias.

Part 2: Exploratory Data Analysis (EDA)

```
from sklearn.preprocessing import MinMaxScaler

# Verify and update numerical columns
numerical_columns = [col for col in ['age', 'steps', 'heart_rate',
'calories', 'distance'] if col in data.columns]

# Normalize only the available numerical columns
scaler = MinMaxScaler()
data[numerical_columns] = scaler.fit_transform(data[numerical_columns])
```

```
print(data.head())
```

Steps Taken:

1. **Importing MinMaxScaler**:
    - `from sklearn.preprocessing import MinMaxScaler`: This imports the `MinMaxScaler` from the `sklearn` library, which is used for feature scaling.
2. **Verify and Update Numerical Columns**:
    - `numerical_columns = [col for col in ['age', 'steps', 'heart_rate', 'calories', 'distance'] if col in data.columns]`: Ensures that only the columns present in the dataset are included for scaling.
3. **Normalize the Numerical Data**:
    - `scaler = MinMaxScaler()`: Initializes the MinMaxScaler, which scales the data to a range between 0 and 1.
    - `data[numerical_columns] = scaler.fit_transform(data[numerical_columns])`: Applies the MinMaxScaler to the selected numerical columns, transforming them into a standardized range between 0 and 1.
4. **Display the Updated Data**:
    - `print(data.head())`: Displays the first few rows of the dataset to confirm the transformation has been applied.

Why It's Important:

1. **Scaling for Machine Learning**:
    - **Normalization** ensures that all numerical features are on the same scale, which is crucial for many machine learning algorithms (like k-NN, SVM, and neural networks) that rely on distance metrics or gradient-based optimization.
2. **Improved Model Performance**:
    - Scaling can help improve the convergence speed and stability of algorithms during training.
3. **Data Consistency**:
    - By transforming all numerical features to the same range, it ensures consistency across the dataset, preventing one feature from dominating due to larger values.

```
if 'gender' in data.columns and 'correct_column_name' in data.columns:
    sns.boxplot(x='gender', y='correct_column_name', data=data)
    plt.title('Heart Rate by Gender')
    plt.show()
```

Steps Taken:

1. **Check for Columns**:

- The `if` condition checks if both the `'gender'` and `'correct_column_name'` columns exist in the dataset (`data.columns`). This ensures that the boxplot is created only if the required columns are present.

2. **Create Boxplot**:
   - `sns.boxplot(x='gender', y='correct_column_name', data=data)`:
     - This uses Seaborn (`sns`) to create a boxplot that visualizes the distribution of the `'correct_column_name'` (e.g., `heart_rate`) by gender.
     - The `x` axis represents the gender, and the `y` axis represents the values from `'correct_column_name'`.
3. **Title and Display**:
   - `plt.title('Heart Rate by Gender')`: Adds a title to the plot.
   - `plt.show()`: Displays the plot.

Why It's Important:

1. **Boxplot Utility**:
   - Boxplots are useful for visualizing the distribution of data, highlighting the median, quartiles, and potential outliers.
   - In this case, it helps to compare heart rate (or any other numerical variable) across different genders.
2. **Data Exploration**:
   - This type of plot can reveal patterns, differences, or potential issues in the data, such as outliers or imbalances.

```
if 'gender' in data.columns and 'HeartRate' in data.columns:
    sns.boxplot(x='gender', y='HeartRate', data=data)
    plt.title('Heart Rate by Gender')
    plt.show()
```

Steps Taken:

1. **Check for Columns**:
   - The `if` condition checks if both `'gender'` and `'HeartRate'` columns exist in the dataset (`data.columns`). This ensures that the boxplot is created only if these columns are present.
2. **Create Boxplot**:
   - `sns.boxplot(x='gender', y='HeartRate', data=data)`:
     - This uses Seaborn (`sns`) to create a boxplot that visualizes the distribution of `'HeartRate'` by `'gender'`.
     - The `x` axis represents gender, and the `y` axis represents the heart rate values.
3. **Title and Display**:
   - `plt.title('Heart Rate by Gender')`: Adds a title to the plot for clarity.

- `plt.show()`: Displays the plot.

Why It's Important:

1. **Boxplot Utility**:
   - Boxplots provide a visual summary of the distribution of the data, showing the median, quartiles, and potential outliers.
   - This is useful for comparing heart rate across different genders, identifying any disparities or trends.
2. **Data Exploration**:
   - This visualization can highlight important insights, such as whether there are significant differences in heart rate between genders, or if any outliers need further investigation.

```
if 'gender' in data.columns and 'calories' in data.columns:
    sns.boxplot(x='gender', y='calories', data=data)
    plt.title('Calories by Gender')
    plt.show()
```

Steps Taken:

1. **Check for Columns**:
   - The `if` condition ensures that the `'gender'` and `'calories'` columns exist in the dataset before creating the boxplot.
2. **Create Boxplot**:
   - `sns.boxplot(x='gender', y='calories', data=data)`:
     - This generates a boxplot using Seaborn, with `'gender'` on the x-axis and `'calories'` on the y-axis.
     - The boxplot visualizes the distribution of calories across different genders.
3. **Title and Display**:
   - `plt.title('Calories by Gender')`: Adds a title to the plot for clarity.
   - `plt.show()`: Displays the plot.

Why It's Important:

1. **Boxplot Utility**:
   - Boxplots are effective for summarizing the distribution of a variable and highlighting the median, quartiles, and any outliers.
   - In this case, it will show how calorie expenditure varies by gender, revealing any significant differences.
2. **Data Exploration**:
   - This visualization helps identify trends or patterns, such as whether one gender tends to burn more calories than the other.
   - It can also highlight any outliers or inconsistencies in the data.

```

```

```
# Select only numeric columns
numeric_data = data.select_dtypes(include=['number'])

# Compute the correlation matrix
correlation_matrix = numeric_data.corr()

# Plot the heatmap
import seaborn as sns
import matplotlib.pyplot as plt

sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

Steps Taken:

1. **Select Numeric Columns**:
   - `numeric_data = data.select_dtypes(include=['number'])`: This selects only the numerical columns from the dataset. These columns are typically the ones you want to analyze for correlations.
2. **Compute the Correlation Matrix**:
   - `correlation_matrix = numeric_data.corr()`: This computes the pairwise correlation coefficients between all numeric columns. The correlation values range from -1 (perfect negative correlation) to +1 (perfect positive correlation). A value of 0 indicates no correlation.
3. **Plot the Heatmap**:
   - **Import Libraries**: `import seaborn as sns` and `import matplotlib.pyplot as plt` are used for visualization.
   - `sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')`: This creates a heatmap of the correlation matrix, with the `annot=True` argument displaying the correlation values in each cell, and `cmap='coolwarm'` defining the color scheme.
   - `plt.title('Correlation Matrix')`: Adds a title to the heatmap.
   - `plt.show()`: Displays the heatmap.

Why It's Important:

1. **Correlation Analysis**:
   - The correlation matrix helps identify relationships between variables, such as whether higher physical activity is associated with higher heart rate or calorie burn.
   - This is crucial for understanding the dynamics of your data and identifying potential patterns.
2. **Data Insights**:
   - The heatmap provides a visual representation of correlations, making it easier to spot strong positive or negative relationships between variables.
   - It can guide feature selection for machine learning models, where highly correlated features might be redundant.

```
# Scatterplot for Steps vs Calories
sns.scatterplot(x='steps', y='calories', hue='gender', data=data)
plt.title('Steps vs Calories')
plt.show()
```

Steps Taken:

1. **Create Scatterplot**:
    - `sns.scatterplot(x='steps', y='calories', hue='gender', data=data)`:
        - This creates a scatterplot using Seaborn, where:
            - The `x` axis represents the number of steps.
            - The `y` axis represents the number of calories burned.
            - The `hue='gender'` argument adds color coding based on the gender, allowing you to distinguish between male and female data points visually.
    - `plt.title('Steps vs Calories')`: Adds a title to the plot for clarity.
    - `plt.show()`: Displays the plot.

Why It's Important:

1. **Data Visualization**:
    - Scatterplots are excellent for visualizing the relationship between two continuous variables, in this case, `steps` and `calories`.
    - This plot can help you understand if there is a trend or pattern, such as whether more steps correlate with more calories burned.
2. **Gender-Based Insights**:
    - By using the `hue='gender'` argument, the plot differentiates the data points by gender, helping you explore whether the relationship between steps and calories differs for males and females.
3. **Pattern Identification**:
    - The scatterplot can reveal patterns such as linear or non-linear relationships, clusters, or outliers that could inform further analysis or modeling.

```
# Example: Steps-Distance Ratio
data['steps_distance_ratio'] = data['steps'] / (data['distance'] + 1e-5)  # Avoid division by zero
print(data.head())
```

Steps Taken:

1. **Create a New Feature (Steps-Distance Ratio)**:
    - `data['steps_distance_ratio'] = data['steps'] / (data['distance'] + 1e-5)`:
        - This calculates the ratio of `steps` to `distance` and stores it as a new column `'steps_distance_ratio'`.

- The `+ 1e-5` is added to the denominator to prevent division by zero errors, ensuring the calculation is safe even if `distance` is zero.

2. **Display the Updated Data**:
   - `print(data.head())`: This displays the first few rows of the updated dataset, including the new `steps_distance_ratio` column.

Why It's Important:

1. **Feature Engineering**:
   - Creating new features, like the steps-to-distance ratio, can provide valuable insights. For example, this ratio might indicate how efficient a person is at converting steps into distance.
   - This new feature could be useful in further analysis or predictive modeling.
2. **Avoiding Errors**:
   - Adding a small constant (`1e-5`) ensures that the division operation doesn't result in errors when the `distance` is zero.
3. **Improving Data Insights**:
   - By analyzing the `steps_distance_ratio`, you can better understand the relationship between physical activity (steps) and the distance traveled, which may offer more actionable insights into fitness patterns.

Part 3: Feature Engineering and Model Building (if applicable)

```
# Print available columns
print("Available columns in dataset:", data.columns)

# Dynamically select valid feature columns
feature_columns = [col for col in ['steps', 'heart_rate', 'distance']
if col in data.columns]

# Ensure the target column exists
if 'calories' in data.columns:
    X = data[feature_columns]
    y = data['calories']

    # Proceed with modeling
    from sklearn.model_selection import train_test_split
    from sklearn.linear_model import LinearRegression
    from sklearn.metrics import mean_squared_error
     # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

    # Train regression model
    model = LinearRegression()
    model.fit(X_train, y_train)

    # Evaluate the model
    predictions = model.predict(X_test)
```

```
    mse = mean_squared_error(y_test, predictions)
    print(f"Mean Squared Error: {mse}")
else:
    print("Target column 'calories' not found in the dataset.")
```

Steps Taken:

1. **Print Available Columns**:
   - `print("Available columns in dataset:", data.columns)`:
     - This prints the list of all columns in the dataset, helping you confirm that the necessary columns are present for the analysis.
2. **Dynamically Select Feature Columns**:
   - `feature_columns = [col for col in ['steps', 'heart_rate', 'distance'] if col in data.columns]`:
     - This selects the columns that are available in the dataset from the list of potential feature columns (`'steps'`, `'heart_rate'`, `'distance'`).
3. **Ensure Target Column Exists**:
   - The `if 'calories' in data.columns:` condition checks if the target column `'calories'` exists in the dataset. If it does, the modeling proceeds; otherwise, a message is printed.
4. **Train-Test Split**:
   - `train_test_split(X, y, test_size=0.2, random_state=42)`:
     - This splits the dataset into training and testing sets. 80% of the data is used for training, and 20% is used for testing.
5. **Train Linear Regression Model**:
   - `model = LinearRegression()`:
     - Initializes the linear regression model.
   - `model.fit(X_train, y_train)`:
     - Trains the model using the training data.
6. **Evaluate the Model**:
   - `predictions = model.predict(X_test)`:
     - The trained model is used to predict `calories` on the test set.
   - `mse = mean_squared_error(y_test, predictions)`:
     - The mean squared error (MSE) is calculated to evaluate the model's performance. A lower MSE indicates a better fit.
7. **Handle Missing Target Column**:
   - If the `'calories'` column is missing, the code prints `"Target column 'calories' not found in the dataset."`.

Why It's Important:

1. **Feature Selection**:
   - Dynamically selecting valid feature columns ensures that only available columns are used for modeling, preventing errors due to missing columns.
2. **Model Training**:
   - Linear regression is a simple and interpretable model used to predict continuous variables like `calories`. Training the model on the selected

features helps understand how steps, heart rate, and distance affect calorie expenditure.

3. **Model Evaluation**:
    - Calculating the mean squared error (MSE) helps assess the model's accuracy. A lower MSE indicates better performance, meaning the model's predictions are closer to the actual values.

Conclusion:

Summary of Results and Key Takeaways:

- The project aimed to analyze smartwatch data, focusing on understanding the relationships between physical activity (steps), heart rate, distance, and calories burned.
- Through various data preprocessing steps, such as handling missing values, normalizing numerical columns, and creating new features (like the steps-to-distance ratio), the dataset was prepared for analysis.
- Exploratory data analysis (EDA) included visualizations such as boxplots and scatterplots, which revealed patterns in how gender and activity levels influenced heart rate and calorie expenditure.
- The linear regression model was trained to predict calories based on features like steps, heart rate, and distance. The model was evaluated using mean squared error (MSE), which provided a measure of prediction accuracy.
- Key takeaways include the identification of significant relationships between activity metrics and calorie expenditure, as well as insights into how gender may influence these factors.

How the Project Met Its Objectives:

- **Data Analysis**: The project successfully explored and visualized relationships between physical activity and fitness outcomes, such as calories burned.
- **Feature Engineering**: New features like the steps-to-distance ratio were created, providing deeper insights into the data.
- **Modeling**: A linear regression model was built and evaluated, providing a quantitative understanding of how different factors influence calorie expenditure.
- **Visualization**: Various visualizations, including heatmaps and scatterplots, helped reveal correlations and trends in the data, enhancing the understanding of key relationships.

Limitations and Areas for Future Work:

1. **Data Quality**:
    - Missing values were handled through imputation, but the approach could be refined for more complex datasets. Future work could explore advanced imputation methods or use domain knowledge for more accurate filling of missing values.
2. **Model Performance**:

- While the linear regression model provided useful insights, its performance could be improved. Future work could involve exploring more complex models (e.g., Random Forest, Gradient Boosting) or adding more features for better predictive power.

3. **Feature Engineering**:
   - Additional features, such as time-of-day or activity type, could be included to improve the model's accuracy. Exploring interactions between features might also yield better results.
4. **Data Size and Diversity**:
   - The analysis was based on a limited dataset, which may not represent the broader population. Future work could involve collecting a larger, more diverse dataset to enhance the generalizability of the findings.
5. **Real-Time Predictions**:
   - Integrating the model with real-time data from smartwatches could provide dynamic, personalized fitness recommendations. This would involve developing a more sophisticated system capable of processing and analyzing data in real time.