Data Cleaning Report: Athlete Performance Analysis

Srijit Kundu RA221100301081

Github Link: https://github.com/SrijitK10/Data-Science

Overview

This report details the data cleaning process applied to an athlete performance dataset for a professional marathon runner. The dataset contains training metrics including speed, distance, heart rate, food intake, hydration levels, and environmental conditions. Several data quality issues were identified and addressed through a systematic cleaning process to prepare the data for meaningful analysis.

Dataset Description

The original dataset contained 110 training session records with 22 variables, including:

- Session identifiers (ID, Date, Time)
- Performance metrics (Distance, Speed, Heart Rate, Calories Burned)
- Physiological factors (Food Item, Calorie Intake, Hydration Level)
- Environmental conditions (Temperature, Humidity)
- Subjective assessments (Training Effort, Session Rating, Recovery Score)
- Miscellaneous information (Shoe Brand, Notes, Social Media Followers, etc.)

Data Quality Issues

Several data quality issues were identified in the original dataset:

1. **Missing Values**: Approximately 10-15% of values were missing in the Heart_Rate (13.64%), Calorie_Intake (9.09%), and Hydration_Level (11.82%) columns.
2. **Duplicate Records**: 20 records (10 unique session duplicates) had identical Session_ID, Date, and Time values.
3. **Inconsistent Formats**:

    - Time recorded in both 24-hour (45%) and 12-hour (55%) formats
    - Distance recorded in both miles (49%) and kilometers (51%)
4. **Outliers**: 2 records had speed values outside the expected range (below 1.56 m/s or above 7.34 m/s)
5. **Incorrect Data Entries**:
    - 16 food items had incorrect calorie values (e.g., "Banana" with 2000 calories)
    - 1 record had a negative hydration level
6. **Irrelevant Columns**: 4 columns (Athlete_Social_Media_Followers, Weather_Forecast_Accuracy, Local_Race_Participants, Training_Plan_Version) were not relevant to the athlete's performance analysis.

Data Cleaning Methodology

1. Handling Missing Data

Approach and Implementation:

- **Heart_Rate**: Missing values were imputed using the median heart rate from similar training sessions based on distance bins. This approach leverages the relationship between exercise intensity (distance) and heart rate.
- **Calorie_Intake**: Missing values were imputed using the median calorie value for the same food item, preserving the relationship between specific foods and their typical calorie content.
- **Hydration_Level**: Missing values were imputed with the median hydration level after excluding negative values, ensuring physiologically plausible values.
- **Notes**: Missing values were left as is since they are descriptive and not critical for analysis.

Justification:
The imputation strategy preserved the logical relationships between variables. Using group-specific medians rather than global values maintains the natural patterns in the data, respecting the athlete's specific physiological responses to different training conditions. Median was chosen over mean to minimize the effect of outliers or extreme values that could skew imputations.

2. Removing Duplicate Records

Approach and Implementation:

- Identified duplicate training sessions based on identical Session_ID, Date, and Time values
- When duplicates were found, kept the record with fewer missing values
- Removed 10 duplicate records, reducing the dataset to 100 records

Justification:
Duplicate records would artificially weight certain training sessions in analysis, potentially biasing results. The strategy of keeping the most complete record ensures maximal data retention while eliminating redundancy. Preserving the record with fewer missing values minimizes the need for imputation in subsequent steps.

3. Standardizing Data Formats

Approach and Implementation:

- **Time Format**: Converted all times to 24-hour format using a custom function that handled special cases like 12 AM/PM
- **Distance Units**: Standardized all distance measurements to kilometers by applying the conversion factor of 1.60934 to values recorded in miles

Justification:
Standardized formats are essential for consistent time-series analysis and accurate comparisons between training sessions. The 24-hour time format was chosen as it aligns with scientific standards and eliminates AM/PM ambiguity. Kilometers were selected as the distance unit to align with the international standard for running metrics, facilitating comparison with benchmark data.

4. Identifying and Handling Outliers

Approach and Implementation:

- Used the Interquartile Range (IQR) method to identify outliers in the Speed column
- Calculated bounds: [Q1 - 1.5 * IQR, Q3 + 1.5 * IQR] = [1.56, 7.34] m/s
- Identified 2 outliers: one very slow (1.03 m/s) and one very fast (9.80 m/s)

- Capped extreme values at the calculated bounds rather than removing them

Justification:
The IQR method is robust for outlier detection as it's not influenced by extreme values. Capping rather than removing outliers preserves the record count and training session continuity while preventing extreme values from skewing statistical analyses. The bounds (1.56 to 7.34 m/s) represent reasonable speed ranges for marathon training, from slow recovery runs to sprint intervals.

5. Correcting Data Entry Errors

Approach and Implementation:

- **Food Calorie Corrections**: Identified 16 records with implausibly high calorie values (e.g., 2000 calories for a banana) and replaced them with accurate reference values from a nutrition database
- **Negative Hydration**: Converted 1 negative hydration level to its absolute value, maintaining the magnitude while correcting the sign error

Justification:
Correcting these errors was critical as they would severely impact nutritional and hydration analyses. For food calories, using reference values from established nutrition databases ensures accuracy. Converting negative hydration to positive preserves the recorded value's magnitude while making it physiologically plausible. These corrections maintain data integrity without sacrificing data points.

6. Dropping Irrelevant Columns

Approach and Implementation:

- Identified and removed 4 columns not relevant to performance analysis:
    - Athlete_Social_Media_Followers
    - Weather_Forecast_Accuracy
    - Local_Race_Participants
    - Training_Plan_Version

Justification:
These columns had no clear causal relationship with athletic performance and would introduce noise to analyses. Removing them simplifies the dataset, improves

computational efficiency, and focuses the analysis on performance-relevant variables. This decision was made after careful consideration of which variables could theoretically impact training outcomes.

7. Final Dataset Preparation

Approach and Implementation:

- **Data Type Optimization**: Converted columns to appropriate data types (int, float, object)
- **Derived Metrics Creation**:
  - Calories_Per_Km: Calories burned per kilometer (efficiency metric)
  - Training_Efficiency: Speed relative to heart rate (cardiovascular efficiency)
  - Hydration_Ratio: Hydration level per kilometer (hydration strategy metric)
- **Data Sorting**: Arranged records chronologically by date and time
- **Final Quality Check**: Verified no remaining data issues

Justification:

Proper data types ensure accurate calculations and optimal storage. The derived metrics provide valuable performance insights beyond raw data, enabling deeper analysis of training efficiency and physiological responses. Chronological sorting facilitates time-series analysis and progression tracking.

Results and Impact

The data cleaning process successfully addressed all identified issues, resulting in a high-quality dataset ready for analysis:

- **Completeness**: All critical variables now have 100% completeness
- **Consistency**: All times are in 24-hour format and distances in kilometers
- **Accuracy**: Food calorie values and hydration levels are now physiologically plausible
- **Relevance**: Dataset contains only performance-relevant variables
- **Enhanced Analysis Potential**: New derived metrics enable deeper performance insights

The final cleaned dataset contains 100 records with 21 columns, providing a comprehensive yet focused view of the athlete's training performance. This clean dataset will enable more accurate analysis of training patterns, physiological responses, and performance optimization strategies.

Conclusion

The systematic data cleaning process transformed a problematic dataset with missing values, inconsistencies, and errors into a reliable foundation for athletic performance analysis. The careful consideration of domain knowledge in each cleaning step (e.g., using training intensity to guide heart rate imputation) ensures that the cleaned data maintains its physiological relevance while eliminating noise and errors.

This clean dataset will allow sports scientists to identify optimal training patterns, nutrition strategies, and recovery approaches, ultimately supporting the marathon runner's performance goals.

In [2]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import re
from datetime import datetime
```

In [3]:
```python
def load_data():
    print("Loading the athlete performance dataset...")
    df = pd.read_csv("data/athlete_performance_raw.csv")
    print(f"Original dataset shape: {df.shape}")
    return df
```

In [4]:
```python
# Task 1: Handling Missing Data
def handle_missing_data(df):
    print("\n--- Task 1: Handling Missing Data ---")

    # Check for missing values
    missing_values = df.isnull().sum()
    missing_percent = (missing_values / len(df) * 100).round(2)

    # Display columns with missing values
    print("Columns with missing values:")
    for col in df.columns:
        if missing_values[col] > 0:
            print(f"  {col}: {missing_values[col]} missing values ({missing_percent[col]}%)")

    # Handling strategy for each column with missing values

    # 1. Heart_Rate: Impute with median based on similar training sessions (by distance range)
    if missing_values['Heart_Rate'] > 0:
        print("\nImputing missing Heart_Rate values based on similar training distances...")

        # Create distance bins
        df['Distance_Bin'] = pd.cut(df['Distance'], bins=5)

        # Fill missing heart rates with the median of the same distance bin
        df['Heart_Rate'] = df.groupby('Distance_Bin')['Heart_Rate'].transform(
            lambda x: x.fillna(x.median())
        )

        # Remove the temporary bin column
        df = df.drop(columns=['Distance_Bin'])

    # 2. Calorie_Intake: Impute with median value for the same food item
    if missing_values['Calorie_Intake'] > 0:
        print("Imputing missing Calorie_Intake values based on the food item...")

        # Group by food item to get the median calorie intake
        food_median_calories = df.groupby('Food_Item')['Calorie_Intake'].median()

        # For each food item with missing calories, fill with the median for that food
        for food in food_median_calories.index:
            mask = (df['Food_Item'] == food) & (df['Calorie_Intake'].isna())
            df.loc[mask, 'Calorie_Intake'] = food_median_calories[food]

    # 3. Hydration_Level: Impute with median based on training effort and distance
    if missing_values['Hydration_Level'] > 0:
        print("Imputing missing Hydration_Level values based on training effort and distance...")

        # First, make sure there are no negative hydration levels affecting the median
        hydration_for_median = df[df['Hydration_Level'] > 0]['Hydration_Level']
        median_hydration = hydration_for_median.median()

        # Fill missing values with overall median
        df['Hydration_Level'] = df['Hydration_Level'].fillna(median_hydration)

    # Check if any missing values remain
    missing_after = df.isnull().sum()
    if missing_after.sum() > 0:
        print("\nRemaining missing values after imputation:")
        for col in df.columns:
            if missing_after[col] > 0:
                print(f"  {col}: {missing_after[col]} missing values")
    else:
        print("\nAll missing values have been handled successfully.")

    return df
```

In [5]:
```python
# Task 2: Removing Duplicate Records
def remove_duplicates(df):
    print("\n--- Task 2: Removing Duplicate Records ---")

    # Count initial records
    initial_count = len(df)
    print(f"Initial record count: {initial_count}")

    # Check for exact duplicates first
    exact_duplicates = df.duplicated().sum()
    print(f"Exact duplicate records: {exact_duplicates}")

    # Define which columns should be considered for identifying training session duplicates
    session_key_columns = ['Session_ID', 'Date', 'Time']

    # Find duplicates based on session identifiers
    session_duplicates = df.duplicated(subset=session_key_columns, keep=False)

    if session_duplicates.sum() > 0:
        print(f"\nFound {session_duplicates.sum()} records with duplicate session identifiers")

        # Display some examples of duplicates
        print("\nSample session duplicates:")
        duplicate_examples = df[session_duplicates].sort_values(by=session_key_columns).head(10)
        print(duplicate_examples[session_key_columns + ['Distance', 'Speed', 'Heart_Rate']].to_string())

        # Keep the record with fewer missing values when there are duplicates
        df['missing_count'] = df.isnull().sum(axis=1)

        # Sort by session keys and missing count, then remove duplicates
```

```
        df = df.sort_values(by=session_key_columns + ['missing_count'])
        df = df.drop_duplicates(subset=session_key_columns, keep='first')
        df = df.drop(columns=['missing_count'])

        # Count records after removing duplicates
        final_count = len(df)
        print(f"\nRemoved {initial_count - final_count} duplicate records")
        print(f"Record count after removing duplicates: {final_count}")
    else:
        print("No session duplicates found.")

    return df
```

In [6]:

```
# Task 3: Standardizing Data Formats
def standardize_formats(df):
    print("\n--- Task 3: Standardizing Data Formats ---")

    # 1. Standardize Time format to 24-hour
    print("Standardizing time format to 24-hour...")

    # Check current time formats
    time_patterns = df['Time'].str.contains('AM|PM', case=False).sum()
    print(f"Records with 12-hour format (AM/PM): {time_patterns}")
    print(f"Records with 24-hour format: {len(df) - time_patterns}")

    # Function to convert 12-hour time to 24-hour format
    def convert_to_24hr(time_str):
        if isinstance(time_str, str) and ('AM' in time_str.upper() or 'PM' in time_str.upper()):
            # Parse 12-hour format
            try:
                if 'AM' in time_str.upper():
                    # Remove AM and handle 12 AM special case
                    time_str = time_str.upper().replace('AM', '').strip()
                    hour, minute = map(int, time_str.split(':'))
                    hour = 0 if hour == 12 else hour
                    return f"{hour:02d}:{minute:02d}"
                else:  # PM
                    # Remove PM and handle 12 PM special case
                    time_str = time_str.upper().replace('PM', '').strip()
                    hour, minute = map(int, time_str.split(':'))
                    hour = hour if hour == 12 else hour + 12
                    return f"{hour:02d}:{minute:02d}"
            except Exception as e:
                print(f"Error converting time: {time_str}, Error: {e}")
                return time_str
        return time_str  # Already in 24-hour format

    # Apply the conversion function
    df['Time'] = df['Time'].apply(convert_to_24hr)

    # 2. Standardize Distance to kilometers
    print("\nStandardizing distance to kilometers...")

    # Check current distance units
    miles_count = (df['Distance_Unit'] == 'miles').sum()
    km_count = (df['Distance_Unit'] == 'km').sum()
    print(f"Records with distance in miles: {miles_count}")
    print(f"Records with distance in kilometers: {km_count}")

    # Convert miles to kilometers where needed
    miles_mask = df['Distance_Unit'] == 'miles'
    df.loc[miles_mask, 'Distance'] = df.loc[miles_mask, 'Distance'] * 1.60934
    df.loc[miles_mask, 'Distance_Unit'] = 'km'

    # Round distance to 2 decimal places for consistency
    df['Distance'] = df['Distance'].round(2)

    # Verify standardization
    print(f"\nAfter standardization, records with distance in km: {(df['Distance_Unit'] == 'km').sum()}")

    return df
```

In [7]:

```
# Task 4: Identifying and Handling Outliers
def handle_outliers(df):
    print("\n--- Task 4: Identifying and Handling Outliers ---")

    # Focus on detecting outliers in the Speed column
    print("Detecting outliers in the Speed column...")

    # Plot speed distribution
    plt.figure(figsize=(10, 6))
    sns.boxplot(x=df['Speed'])
    plt.title('Speed Distribution (Before Outlier Handling)')
    plt.savefig('data/speed_distribution_before.png')
    plt.close()

    # Calculate IQR for Speed
    Q1 = df['Speed'].quantile(0.25)
    Q3 = df['Speed'].quantile(0.75)
    IQR = Q3 - Q1

    # Define bounds for outliers
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Identify outliers
    speed_outliers = df[(df['Speed'] < lower_bound) | (df['Speed'] > upper_bound)]
    outlier_count = len(speed_outliers)

    print(f"Speed quartiles - Q1: {Q1:.2f}, Median: {df['Speed'].median():.2f}, Q3: {Q3:.2f}")
    print(f"Speed IQR: {IQR:.2f}")
    print(f"Outlier bounds: [{lower_bound:.2f}, {upper_bound:.2f}]")
    print(f"Detected {outlier_count} outliers in Speed column")

    if outlier_count > 0:
        # Display outliers
        print("\nSample speed outliers:")
        print(speed_outliers[['Session_ID', 'Date', 'Speed', 'Distance', 'Heart_Rate']].head().to_string())

        # Method: Cap outliers at the bounds
        print("\nCapping extreme speed values at the calculated bounds...")
```

```
        df['Speed'] = df['Speed'].clip(lower=lower_bound, upper=upper_bound)

        # Plot speed distribution after handling outliers
        plt.figure(figsize=(10, 6))
        sns.boxplot(x=df['Speed'])
        plt.title('Speed Distribution (After Outlier Handling)')
        plt.savefig('data/speed_distribution_after.png')
        plt.close()

        # Verify outliers were handled
        remaining_outliers = df[(df['Speed'] < lower_bound) | (df['Speed'] > upper_bound)]
        print(f"Remaining outliers after capping: {len(remaining_outliers)}")

    return df
```

In [8]:

```
# Task 5: Correcting Data Entry Errors
def correct_data_errors(df):
    print("\n--- Task 5: Correcting Data Entry Errors ---")

    # 1. Correct incorrect calorie values for food items
    print("Correcting invalid calorie values for food items...")

    # Define correct calorie values for common food items
    correct_calories = {
        "Banana": 105,
        "Energy Bar": 250,
        "Oatmeal": 150,
        "Protein Shake": 300,
        "Yogurt": 120,
        "Pasta": 400,
        "Chicken Breast": 165,
        "Salmon": 230,
        "Rice": 210,
        "Sweet Potato": 115
    }

    # Identify anomalous calorie values (much higher than expected)
    anomalies = 0
    for food, correct_value in correct_calories.items():
        # Consider values significantly higher than correct as errors
        mask = (df['Food_Item'] == food) & (df['Calorie_Intake'] > correct_value * 5)
        anomaly_count = mask.sum()

        if anomaly_count > 0:
            anomalies += anomaly_count
            print(f"  Found {anomaly_count} incorrect calorie values for {food}")
            df.loc[mask, 'Calorie_Intake'] = correct_value

    print(f"Total incorrect calorie values corrected: {anomalies}")

    # 2. Fix negative hydration levels
    negative_hydration = (df['Hydration_Level'] < 0).sum()
    if negative_hydration > 0:
        print(f"\nFound {negative_hydration} records with negative hydration levels")

        # Fix by taking the absolute value
        df.loc[df['Hydration_Level'] < 0, 'Hydration_Level'] = df.loc[df['Hydration_Level'] < 0, 'Hydration_Level'].abs()

        print("Corrected negative hydration levels by converting to positive values")
    else:
        print("\nNo negative hydration levels found")

    return df
```

In [9]:

```
# Task 6: Dropping Irrelevant Columns
def drop_irrelevant_columns(df):
    print("\n--- Task 6: Dropping Irrelevant Columns ---")

    # Identify columns that don't contribute to performance analysis
    irrelevant_columns = [
        'Athlete_Social_Media_Followers',
        'Weather_Forecast_Accuracy',
        'Local_Race_Participants',
        'Training_Plan_Version'
    ]

    # Check if these columns exist in the dataset
    existing_irrelevant = [col for col in irrelevant_columns if col in df.columns]

    if existing_irrelevant:
        print(f"Dropping {len(existing_irrelevant)} irrelevant columns:")
        for col in existing_irrelevant:
            print(f"  - {col}")

        # Drop the irrelevant columns
        df = df.drop(columns=existing_irrelevant)

        print(f"\nRemaining columns after dropping irrelevant ones: {df.shape[1]}")
    else:
        print("No irrelevant columns found")

    return df
```

In [10]:

```
# Task 7: Final Dataset Preparation
def prepare_final_dataset(df):
    print("\n--- Task 7: Final Dataset Preparation ---")

    # 1. Ensure proper data types
    print("Ensuring proper data types...")

    # Convert numeric columns to appropriate types
    numeric_columns = ['Distance', 'Speed', 'Heart_Rate', 'Calories_Burned',
                       'Temperature', 'Humidity', 'Calorie_Intake',
                       'Hydration_Level', 'Training_Effort', 'Session_Rating',
                       'Post_Recovery_Score']

    for col in numeric_columns:
        if col in df.columns:
            if col in ['Heart_Rate', 'Calories_Burned', 'Calorie_Intake',
                       'Training_Effort', 'Session_Rating', 'Post_Recovery_Score']:
                df[col] = df[col].astype(int)
```

```
            else:
                df[col] = df[col].astype(float)

    # 2. Create derived metrics for analysis
    print("\nCreating derived metrics for analysis...")

    # Calculate calories burned per kilometer
    df['Calories_Per_Km'] = (df['Calories_Burned'] / df['Distance']).round(1)

    # Calculate training efficiency (speed relative to heart rate)
    df['Training_Efficiency'] = (df['Speed'] / df['Heart_Rate'] * 100).round(2)

    # Calculate hydration ratio (hydration level per kilometer)
    df['Hydration_Ratio'] = (df['Hydration_Level'] / df['Distance']).round(2)

    # 3. Sort dataset by date and time for better analysis
    df = df.sort_values(by=['Date', 'Time'])

    # 4. Generate a final dataset summary
    print("\nFinal dataset info:")
    print(df.info())

    print("\nFinal dataset descriptive statistics:")
    print(df.describe().round(2))

    # 5. Save the cleaned dataset
    df.to_csv('data/athlete_performance_cleaned.csv', index=False)
    print("\nFinal cleaned dataset saved to: data/athlete_performance_cleaned.csv")

    return df
```

In [11]:
```
# Main function to run all cleaning tasks
def main():
    # Load the data
    df = load_data()

    # Apply each cleaning task
    df = handle_missing_data(df)
    df = remove_duplicates(df)
    df = standardize_formats(df)
    df = handle_outliers(df)
    df = correct_data_errors(df)
    df = drop_irrelevant_columns(df)
    df = prepare_final_dataset(df)

    print("\nData cleaning complete!")

    # Generate a simple summary of changes made
    print("\nSummary of Cleaning Actions:")
    print("1. Handled missing values in Heart_Rate, Calorie_Intake, and Hydration_Level columns")
    print("2. Removed duplicate training sessions")
    print("3. Standardized time format to 24-hour and distance to kilometers")
    print("4. Identified and handled outliers in the Speed column")
    print("5. Corrected erroneous calorie values and negative hydration levels")
    print("6. Removed irrelevant columns not related to performance analysis")
    print("7. Created derived metrics for deeper analysis")

if __name__ == "__main__":
    main()
```

```
Loading the athlete performance dataset...
Original dataset shape: (110, 22)

--- Task 1: Handling Missing Data ---
Columns with missing values:
  Heart_Rate: 15 missing values (13.64%)
  Calorie_Intake: 10 missing values (9.09%)
  Hydration_Level: 13 missing values (11.82%)
  Notes: 17 missing values (15.45%)

Imputing missing Heart_Rate values based on similar training distances...
Imputing missing Calorie_Intake values based on the food item...
Imputing missing Hydration_Level values based on training effort and distance...

Remaining missing values after imputation:
  Notes: 17 missing values

--- Task 2: Removing Duplicate Records ---
Initial record count: 110
Exact duplicate records: 0

Found 20 records with duplicate session identifiers

Sample session duplicates:
     Session_ID        Date     Time  Distance  Speed  Heart_Rate
17           18  2023-01-18    19:15     11.92   3.07       141.0
105          18  2023-01-18    19:15     11.92   3.07       141.0
19           20  2023-01-20    07:45     16.73   3.94       152.0
107          20  2023-01-20    07:45     16.73   3.94       152.0
28           29  2023-01-29    09:00      5.49   4.42       180.0
104          29  2023-01-29    09:00      5.49   4.42       180.0
37           38  2023-01-08    10:30      6.15   3.70       141.0
101          38  2023-01-08    10:30      6.15   3.70       141.0
47           48  2023-01-18  7:45 AM     15.91   3.47       142.0
102          48  2023-01-18  7:45 AM     15.91   3.47       142.0
Removed 10 duplicate records
Record count after removing duplicates: 100

--- Task 3: Standardizing Data Formats ---
Standardizing time format to 24-hour...
Records with 12-hour format (AM/PM): 55
Records with 24-hour format: 45

Standardizing distance to kilometers...
Records with distance in miles: 49
Records with distance in kilometers: 51

After standardization, records with distance in km: 100

--- Task 4: Identifying and Handling Outliers ---
Detecting outliers in the Speed column...
Speed quartiles - Q1: 3.73, Median: 4.36, Q3: 5.17
Speed IQR: 1.45
Outlier bounds: [1.56, 7.34]
Detected 2 outliers in Speed column

Sample speed outliers:
    Session_ID        Date  Speed  Distance  Heart_Rate
34          35  2023-01-05   1.03      8.66       155.0
80          81  2023-01-21   9.80     10.83       143.0

Capping extreme speed values at the calculated bounds...
Remaining outliers after capping: 0

--- Task 5: Correcting Data Entry Errors ---
Correcting invalid calorie values for food items...
  Found 3 incorrect calorie values for Banana
  Found 1 incorrect calorie values for Oatmeal
  Found 2 incorrect calorie values for Protein Shake
  Found 2 incorrect calorie values for Yogurt
  Found 1 incorrect calorie values for Pasta
  Found 1 incorrect calorie values for Chicken Breast
  Found 1 incorrect calorie values for Salmon
  Found 3 incorrect calorie values for Rice
  Found 2 incorrect calorie values for Sweet Potato
Total incorrect calorie values corrected: 16

Found 1 records with negative hydration levels
Corrected negative hydration levels by converting to positive values

--- Task 6: Dropping Irrelevant Columns ---
Dropping 4 irrelevant columns:
  - Athlete_Social_Media_Followers
  - Weather_Forecast_Accuracy
  - Local_Race_Participants
  - Training_Plan_Version

Remaining columns after dropping irrelevant ones: 18

--- Task 7: Final Dataset Preparation ---
Ensuring proper data types...

Creating derived metrics for analysis...

Final dataset info:
<class 'pandas.core.frame.DataFrame'>
Index: 100 entries, 0 to 29
Data columns (total 21 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Session_ID           100 non-null    int64
 1   Date                 100 non-null    object
 2   Time                 100 non-null    object
 3   Distance             100 non-null    float64
 4   Distance_Unit        100 non-null    object
 5   Speed                100 non-null    float64
 6   Heart_Rate           100 non-null    int64
 7   Calories_Burned      100 non-null    int64
 8   Temperature          100 non-null    float64
 9   Humidity             100 non-null    float64
 10  Food_Item            100 non-null    object
```

```
 11  Calorie_Intake       100 non-null    int64
 12  Hydration_Level      100 non-null    float64
 13  Training_Effort      100 non-null    int64
 14  Session_Rating       100 non-null    int64
 15  Shoe_Brand           100 non-null    object
 16  Post_Recovery_Score  100 non-null    int64
 17  Notes                85 non-null     object
 18  Calories_Per_Km      100 non-null    float64
 19  Training_Efficiency  100 non-null    float64
 20  Hydration_Ratio      100 non-null    float64
dtypes: float64(8), int64(7), object(6)
memory usage: 17.2+ KB
None

Final dataset descriptive statistics:
       Session_ID  Distance   Speed  Heart_Rate  Calories_Burned  Temperature  \
count      100.00    100.00  100.00      100.00           100.00       100.00
mean        50.50     12.58    4.41      148.12           761.26        22.37
std         29.01      4.44    0.91       16.47           273.37         7.22
min          1.00      5.15    1.56      120.00           295.00        10.20
25%         25.75      8.65    3.73      135.75           518.25        17.20
50%         50.50     12.18    4.36      144.00           741.50        21.75
75%         75.25     16.76    5.17      162.25          1009.75        28.18
max        100.00     20.74    7.34      180.00          1238.00        34.70

       Humidity  Calorie_Intake  Hydration_Level  Training_Effort  \
count    100.00          100.00           100.00           100.00
mean      60.47          204.25             1.74             5.86
std       18.56           85.12             0.71             3.15
min       30.00          105.00             0.50             1.00
25%       42.75          120.00             1.18             3.00
50%       61.00          210.00             1.80             6.00
75%       77.00          250.00             2.30             9.00
max       90.00          400.00             3.00            10.00

       Session_Rating  Post_Recovery_Score  Calories_Per_Km  \
count          100.00               100.00           100.00
mean             3.15                 5.85            60.45
std              1.30                 2.98             3.34
min              1.00                 1.00            54.20
25%              2.00                 3.00            57.65
50%              3.00                 6.00            60.90
75%              4.00                 8.00            62.82
max              5.00                10.00            66.00

       Training_Efficiency  Hydration_Ratio
count               100.00           100.00
mean                  3.01             0.16
std                   0.71             0.11
min                   1.01             0.03
25%                   2.48             0.09
50%                   2.98             0.13
75%                   3.46             0.20
max                   5.13             0.56

Final cleaned dataset saved to: data/athlete_performance_cleaned.csv

Data cleaning complete!

Summary of Cleaning Actions:
1. Handled missing values in Heart_Rate, Calorie_Intake, and Hydration_Level columns
2. Removed duplicate training sessions
3. Standardized time format to 24-hour and distance to kilometers
4. Identified and handled outliers in the Speed column
5. Corrected erroneous calorie values and negative hydration levels
6. Removed irrelevant columns not related to performance analysis
7. Created derived metrics for deeper analysis
/var/folders/rw/g4p48pvn7w946c84lc3b8sdc0000gn/T/ipykernel_66011/3899851911.py:25: FutureWarning: The default of observed=False is deprecated and will be changed to True in a future version of pandas. Pass observed=False to retain current behavior or observed=True to adopt the future default and silence this warning.
  df['Heart_Rate'] = df.groupby('Distance_Bin')['Heart_Rate'].transform(
```

In [ ]:

In [ ]: