Can ML Predict My Period Better Than My Steps? A Dissection of iOS Health Data and Flo App Data

1/ Data Explanation:

For the past 5.5 years, I've been diligently tracking not just my daily steps but also my menstrual cycle. I always felt guilty that this abundance of data was just sitting in my phone when it could be used to help solve my curiosity: Do I walk more or less during my period days? I decided to pull data from my iOS Health app and combine it with my Flo app (menstrual cycle tracker).

Data from iOS Health:

- HKQuantityTypeIdentifierStepCount (steps): A count of the total number of steps taken each day.
- HKQuantityTypeIdentifierBasalEnergyBurned (energy_burned): The amount of energy burned at rest, measured in kilocalories.
- HKQuantityTypeIdentifierDistanceWalkingRunning (distance_walked): The total distance walked or run during the day, measured in kilometers.
- HKQuantityTypeIdentifierWalkingSpeed (walking_speed): The average speed at which I walked during the day, measured in meters per second.
- HKCategoryTypeIdentifierSleepAnalysis (timezone): The sleep data was not included (as I turned on the sleep mode randomly, not only when I sleep), but the timezone was extracted instead.

Source:
https://developer.apple.com/documentation/healthkit/hkquantitytypeidentifiersixminutewalktest

Data from Flo: Emailed the Flo team and recieved for the json file

- is_on_period: A simple yes or no that lets us know whether I was on my period on any given day. This data was logged manually in the Flo app, where I document each cycle with the dedication of a scientist (and someone who just likes to know when cramps are coming).

Merging the Data:

- To create a complete picture, I merged my iOS Health data with the Flo app's period data based on the date index. This way, I can see if my activity levels—steps, speed and

other factors influenced by my menstrual cycle.

Data loading

In [1]:
```
import xml.etree.ElementTree as ET
import pandas as pd

# Open the XML file from iOS
with open('/Users/zeldudu/Downloads/apple_health_export/export.xml', 'r') as f
    lines = file.readlines()

    # Find the starting point of <HealthData> and the ending </HealthData>
    start_index = None
    end_index = None

    # Locate where <HealthData> starts and ends
    for i, line in enumerate(lines):
        if '<HealthData' in line:
            start_index = i
        if '</HealthData>' in line:
            end_index = i

    # Ensure both start and end are found
    if start_index is not None and end_index is not None:
        clean_content = ''.join(lines[start_index:end_index + 1])
    else:
        print("Could not find valid HealthData section.")
        clean_content = None

if clean_content is not None:
    try:
        root = ET.fromstring(clean_content)  # Parse the XML starting from <Hea
    except ET.ParseError as e:
        print(f"XML parsing error: {e}")
        root = None

if root is not None:
    # List to hold aggregated daily data
    daily_data = {}
    time_zone_data = {}

    # Iterate through the XML tree to find relevant records
    for elem in root.findall('.//Record'):
        record_type = elem.attrib.get('type')

        # Extract the main attributes (startDate, endDate, value)
        start_date = pd.to_datetime(elem.attrib['startDate']).date()
        value = pd.to_numeric(elem.attrib['value'], errors='coerce')

        # Process each record type and aggregate by day
        if record_type == 'HKQuantityTypeIdentifierStepCount':
            daily_data.setdefault(start_date, {}).setdefault('steps', 0)
            daily_data[start_date]['steps'] += value
        elif record_type == 'HKQuantityTypeIdentifierBasalEnergyBurned':
            daily_data.setdefault(start_date, {}).setdefault('energy_burned', (
            daily_data[start_date]['energy_burned'] += value
        elif record_type == 'HKQuantityTypeIdentifierDistanceWalkingRunning':
            daily_data.setdefault(start_date, {}).setdefault('distance_walked'
```

```
            daily_data[start_date]['distance_walked'] += value
        elif record_type == 'HKQuantityTypeIdentifierWalkingSpeed':
            daily_data.setdefault(start_date, {}).setdefault('walking_speed',

        # Extract time zone from sleep analysis
        if record_type == 'HKCategoryTypeIdentifierSleepAnalysis':
            for meta in elem.findall('.//MetadataEntry'):
                if meta.attrib['key'] == 'HKTimeZone':
                    time_zone_data[start_date] = meta.attrib['value']  # Extra

# Walking speed (average per day)
for day in daily_data:
    if 'walking_speed' in daily_data[day]:
        daily_data[day]['walking_speed'] = sum(daily_data[day]['walking_speed']

# Convert the aggregated data into a pandas DataFrame
df_daily = pd.DataFrame.from_dict(daily_data, orient='index')

# Add time zone information
df_daily['timeZone'] = df_daily.index.map(lambda x: time_zone_data.get(x, 'Unk

# Fill any missing values with zero
df_daily.fillna(0, inplace=True)

print(df_daily.tail())
```

```
            steps  distance_walked  walking_speed  energy_burned  \
2024-10-03   2137         0.904995       3.019865       1436.081
2024-10-04   6766         2.672039       2.787223       1492.052
2024-10-05   3384         1.486473       2.809593       1445.217
2024-10-06   2259         1.017390       2.710761       1485.527
2024-10-07   1031         0.464186       0.000000       1007.112

                                timeZone
2024-10-03  America/Argentina/Buenos_Aires
2024-10-04  America/Argentina/Buenos_Aires
2024-10-05  America/Argentina/Buenos_Aires
2024-10-06  America/Argentina/Buenos_Aires
2024-10-07  America/Argentina/Buenos_Aires
```

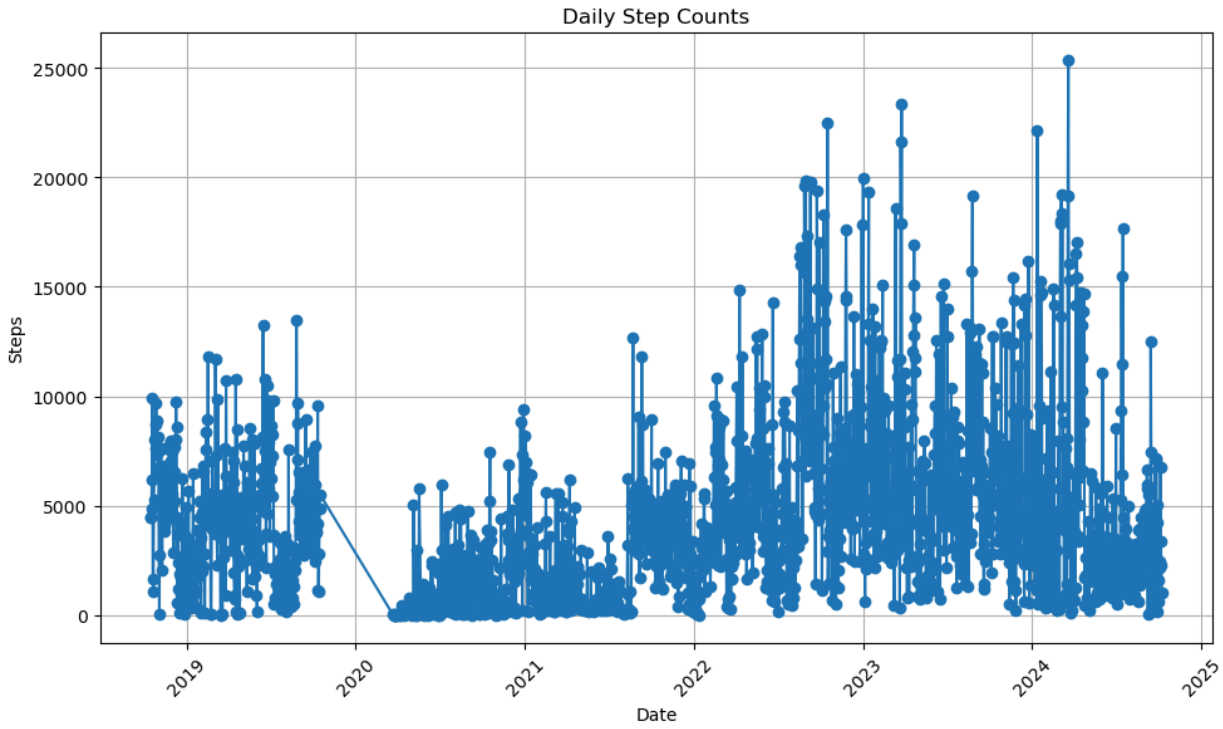
In [48]:
```
# Ensure the index is in datetime format
df_daily.index = pd.to_datetime(df_daily.index)
```

In [49]:
```
df_daily['timeZone'].unique().tolist()
```

Out[49]:
```
['Unknown',
 'Pacific/Auckland',
 'Asia/Ho_Chi_Minh',
 'Asia/Qatar',
 'America/New_York',
 'America/Los_Angeles',
 'Asia/Taipei',
 'Asia/Seoul',
 'Europe/Berlin',
 'Europe/Prague',
 'Europe/Rome',
 'Europe/Vienna',
 'Europe/Amsterdam',
 'Europe/Madrid',
 'Europe/Athens',
 'America/Argentina/Buenos_Aires']
```

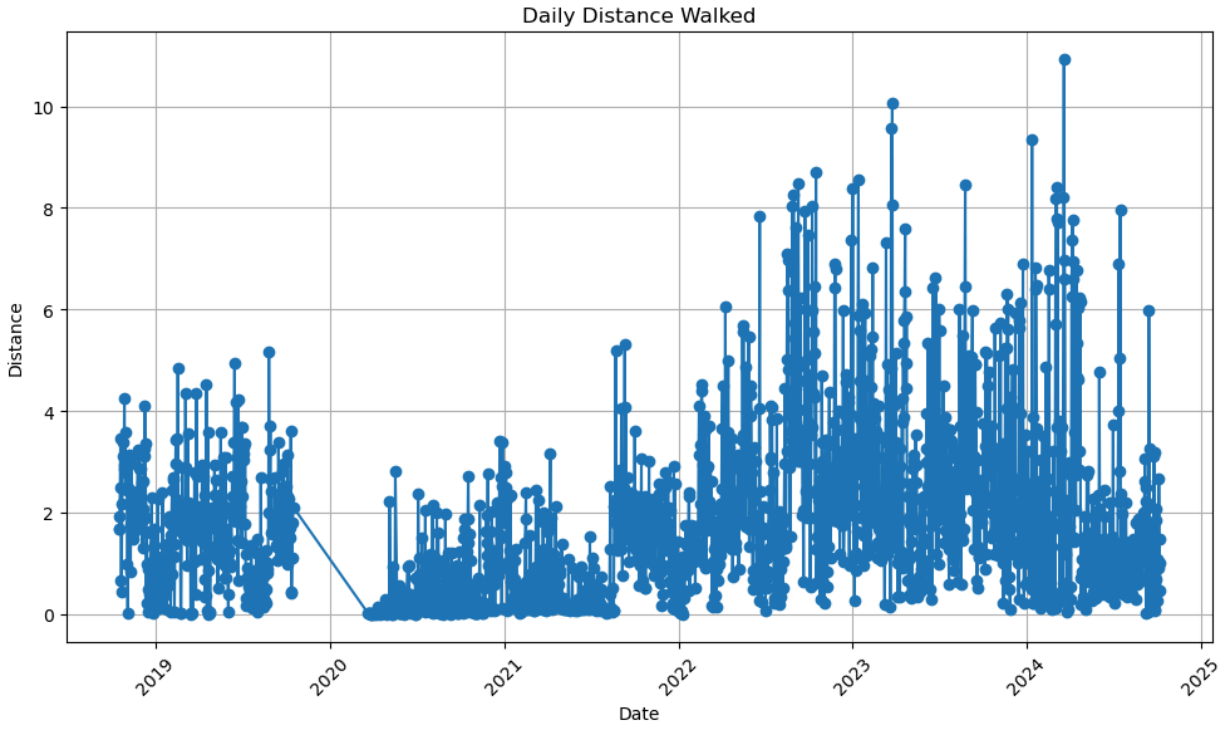
Exploratory Data Analysis

In [50]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Plot the daily step counts
plt.figure(figsize=(10, 6))
plt.plot(df_daily.index, df_daily['steps'], marker='o')
plt.title('Daily Step Counts')
plt.xlabel('Date')
plt.ylabel('Steps')
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



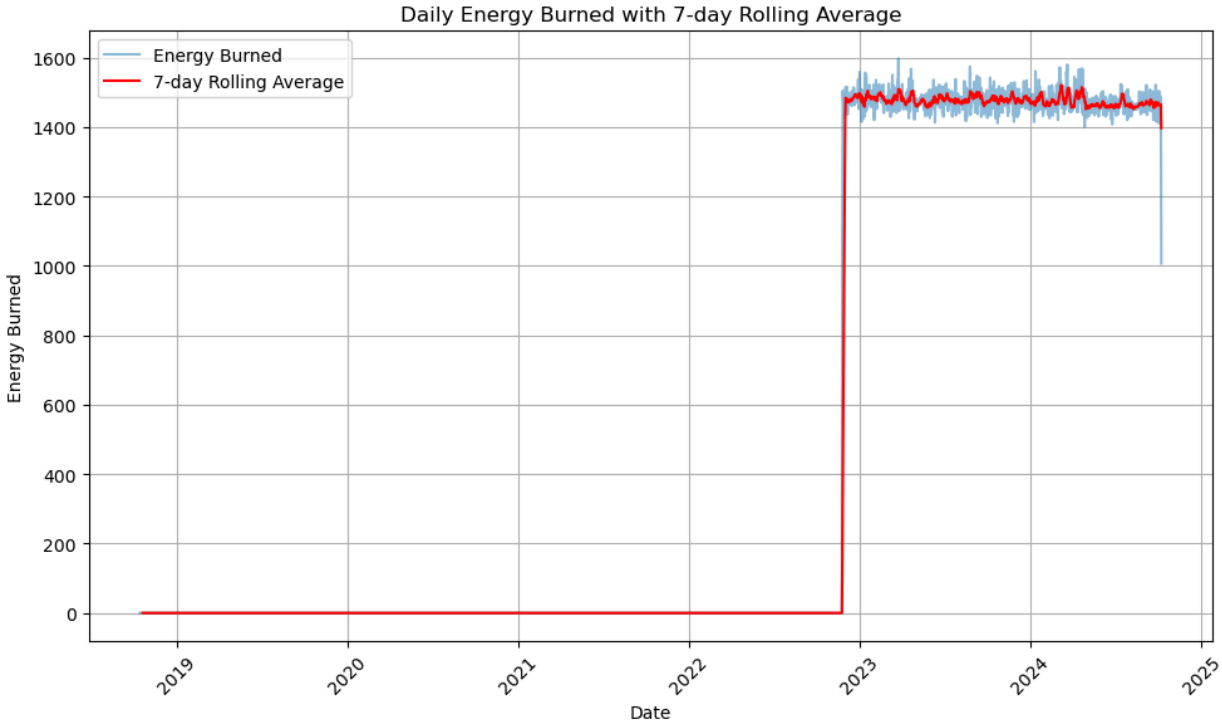
In [51]:
```
# Plot the daily distance walked
plt.figure(figsize=(10, 6))
```

```
plt.plot(df_daily.index, df_daily['distance_walked'], marker='o')
plt.title('Daily Distance Walked')
plt.xlabel('Date')
plt.ylabel('Distance')
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

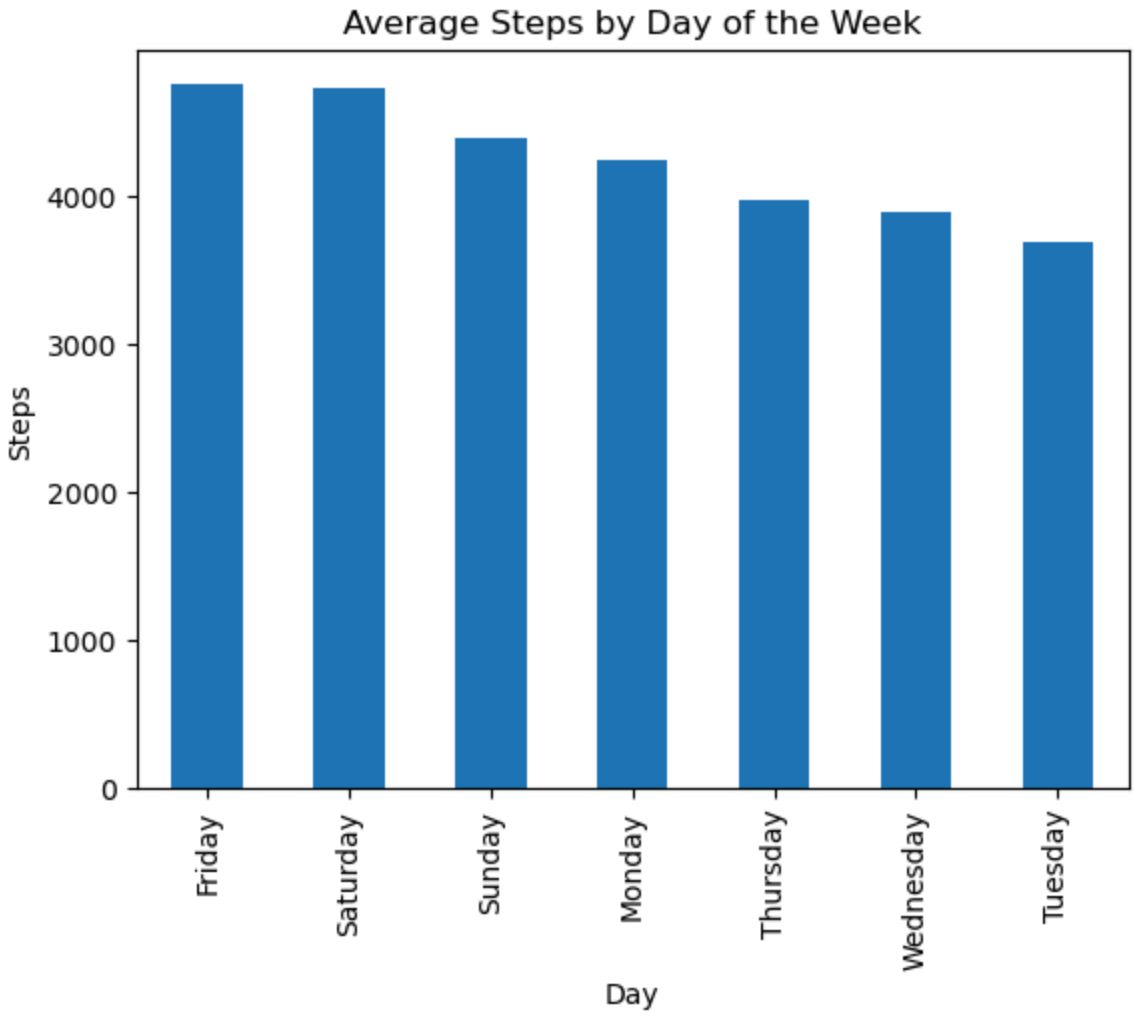


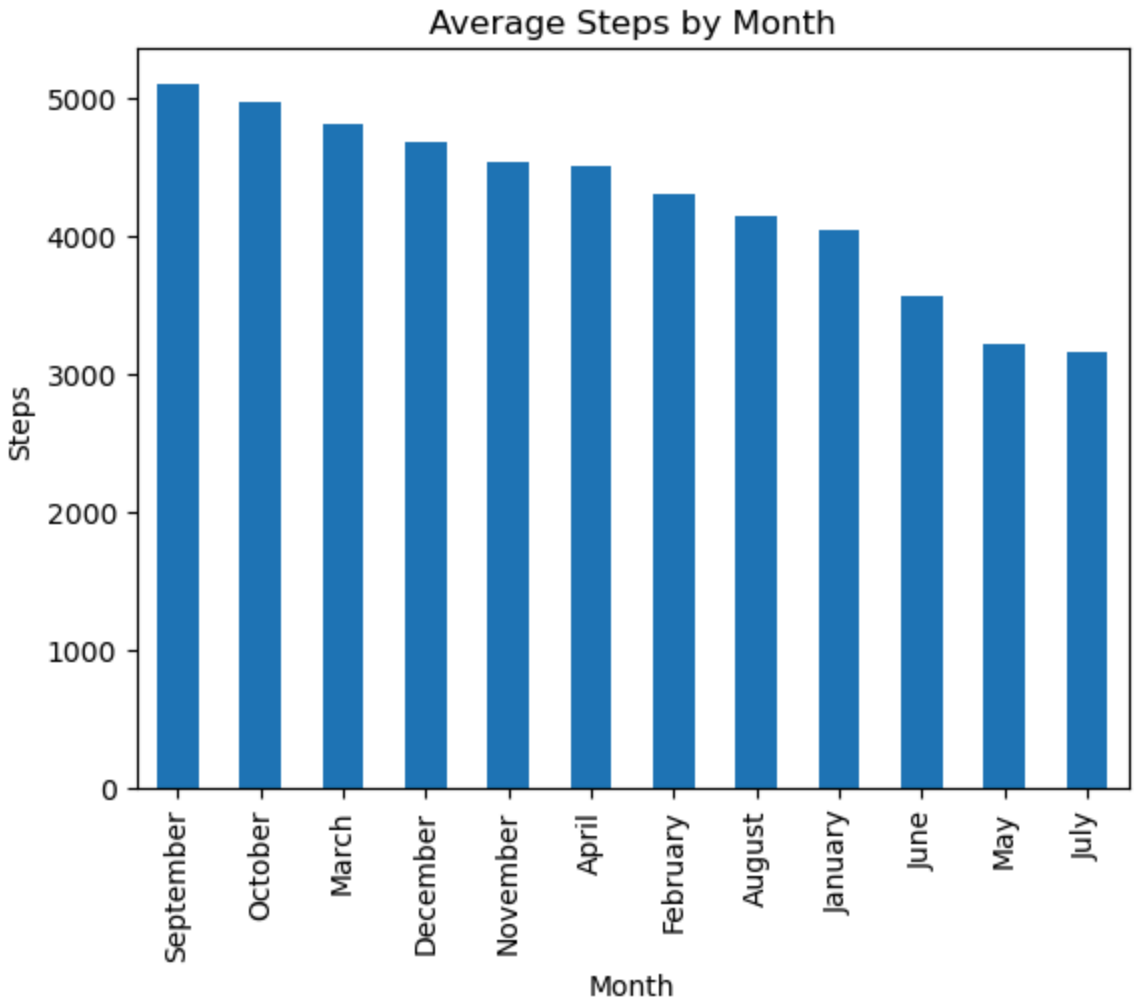
In [52]:

```
# Calculate 7-day rolling average of energy burned
rolling_avg = df_daily['energy_burned'].rolling(window=7).mean()

# Plot rolling average alongside daily steps
plt.figure(figsize=(10, 6))
plt.plot(df_daily.index, df_daily['energy_burned'], label='Energy Burned', alpl
plt.plot(df_daily.index, rolling_avg, label='7-day Rolling Average', color='re
plt.title('Daily Energy Burned with 7-day Rolling Average')
plt.xlabel('Date')
plt.ylabel('Energy Burned')
plt.grid(True)
plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

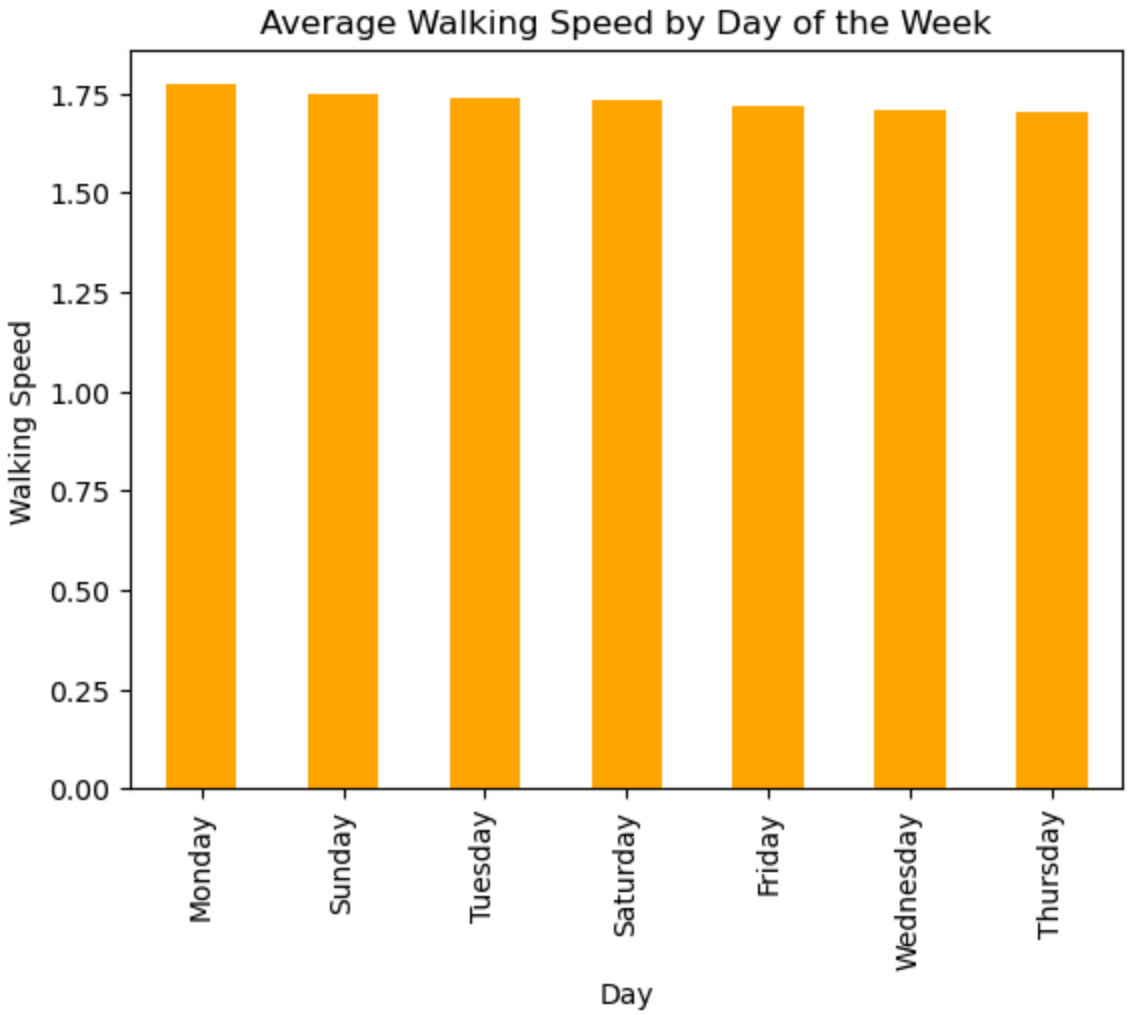


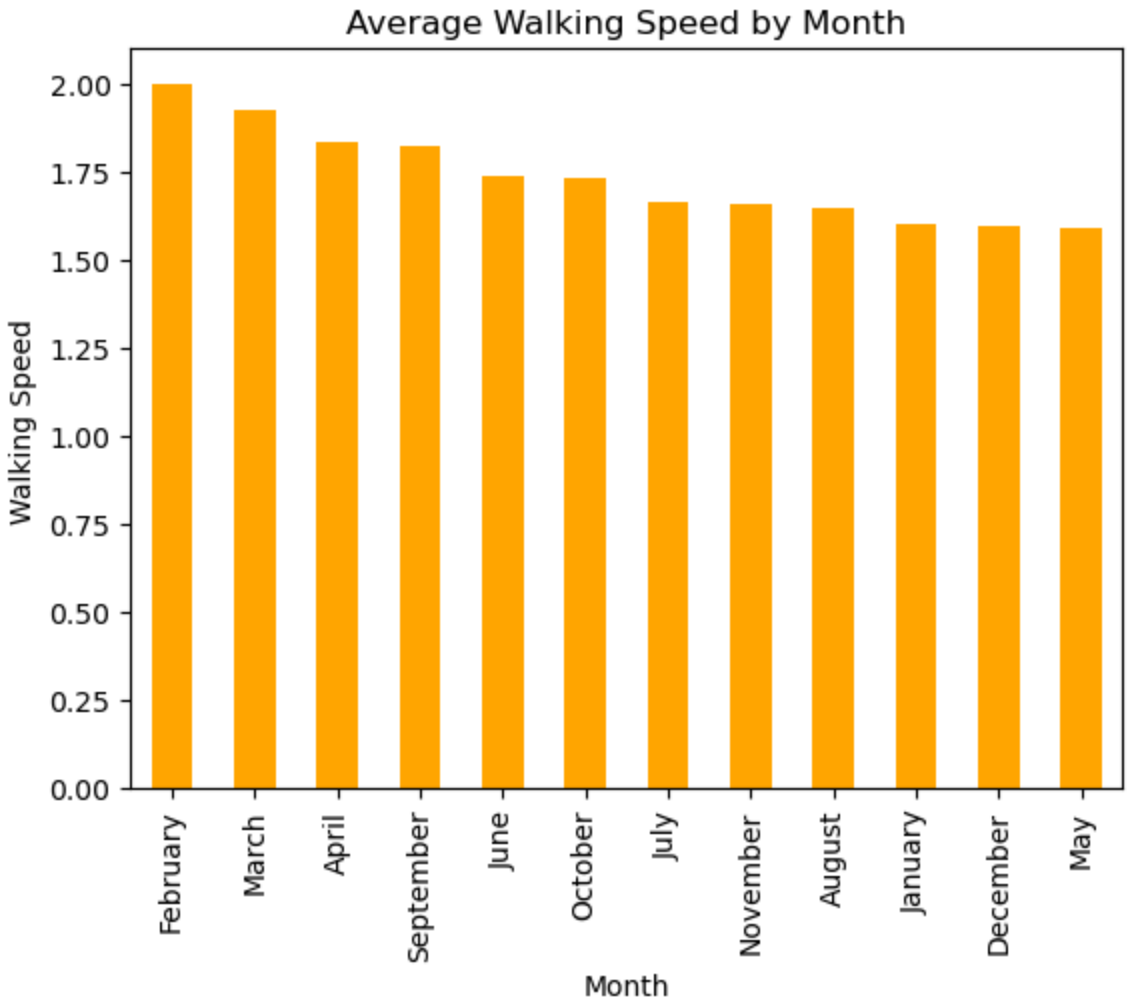
In [53]:
```
# Extract day of the week and month for seasonality analysis
df_daily['day_of_week'] = df_daily.index.day_name()
df_daily['month'] = df_daily.index.month_name()

# Calculate average steps by day of the week
avg_by_day = df_daily.groupby('day_of_week')['steps'].mean().sort_values(ascen

# Calculate average steps by month
avg_by_month = df_daily.groupby('month')['steps'].mean().sort_values(ascending=

# Plot average steps by day of the week
avg_by_day.plot(kind='bar', title='Average Steps by Day of the Week', ylabel='S
plt.show()

# Plot average steps by month
avg_by_month.plot(kind='bar', title='Average Steps by Month', ylabel='Steps', x
plt.show()
```


Average Steps by Day of the Week
Steps
0
1000
2000
3000
4000
Friday
Saturday
Sunday
Monday
Thursday
Wednesday
Tuesday
Day

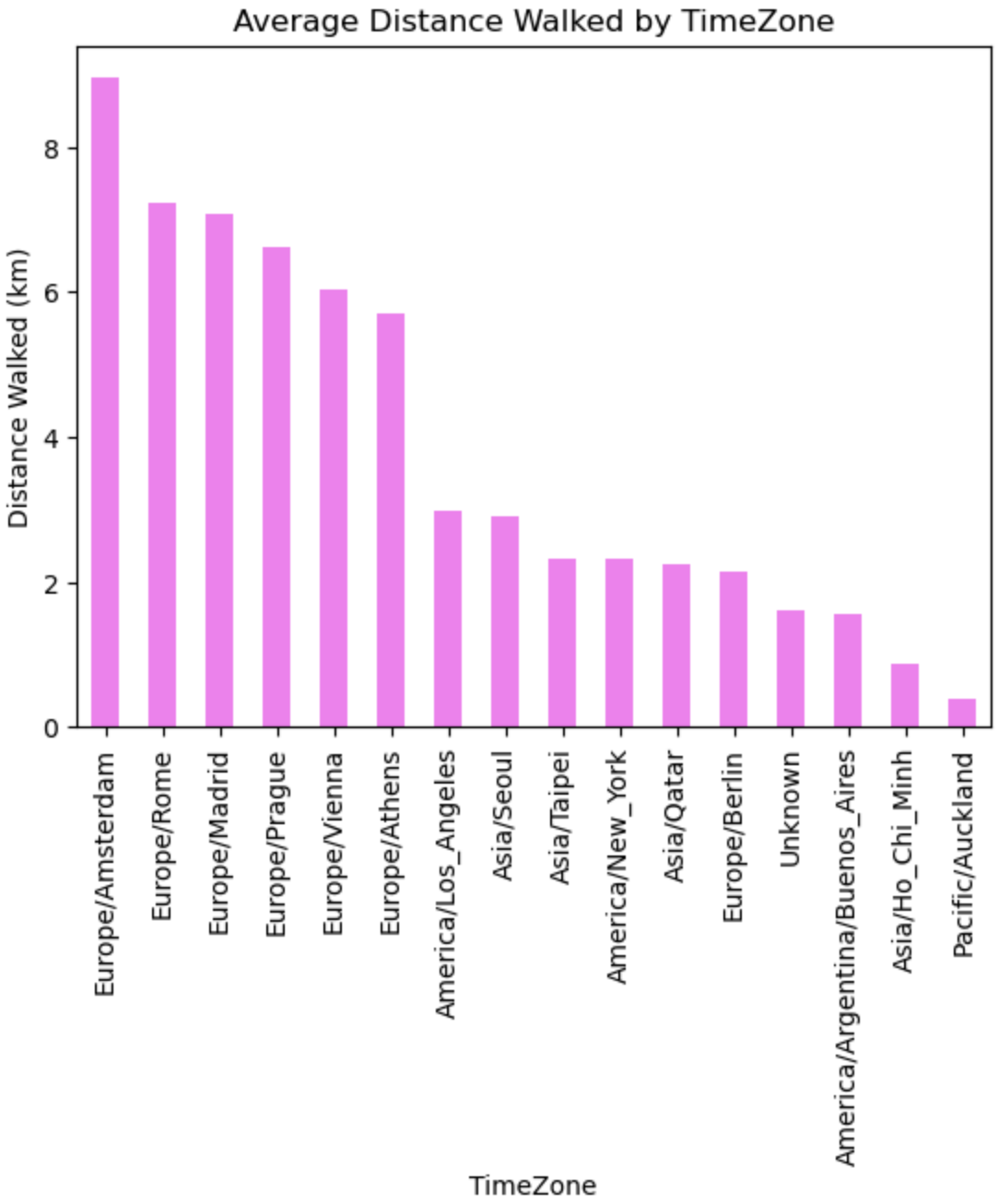


In [54]:
```
# Calculate average walking speed by day of the week
avg_by_day = df_daily.groupby('day_of_week')['walking_speed'].mean().sort_valu

# Calculate average walking speed by month
avg_by_month = df_daily.groupby('month')['walking_speed'].mean().sort_values(a

# Plot average walking speed by day of the week
avg_by_day.plot(kind='bar', title='Average Walking Speed by Day of the Week',
plt.show()

# Plot average walking speed by month
avg_by_month.plot(kind='bar', title='Average Walking Speed by Month', ylabel='
plt.show()
```


Average Walking Speed by Day of the Week
Walking Speed
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Monday
Sunday
Tuesday
Saturday
Friday
Wednesday
Thursday
Day



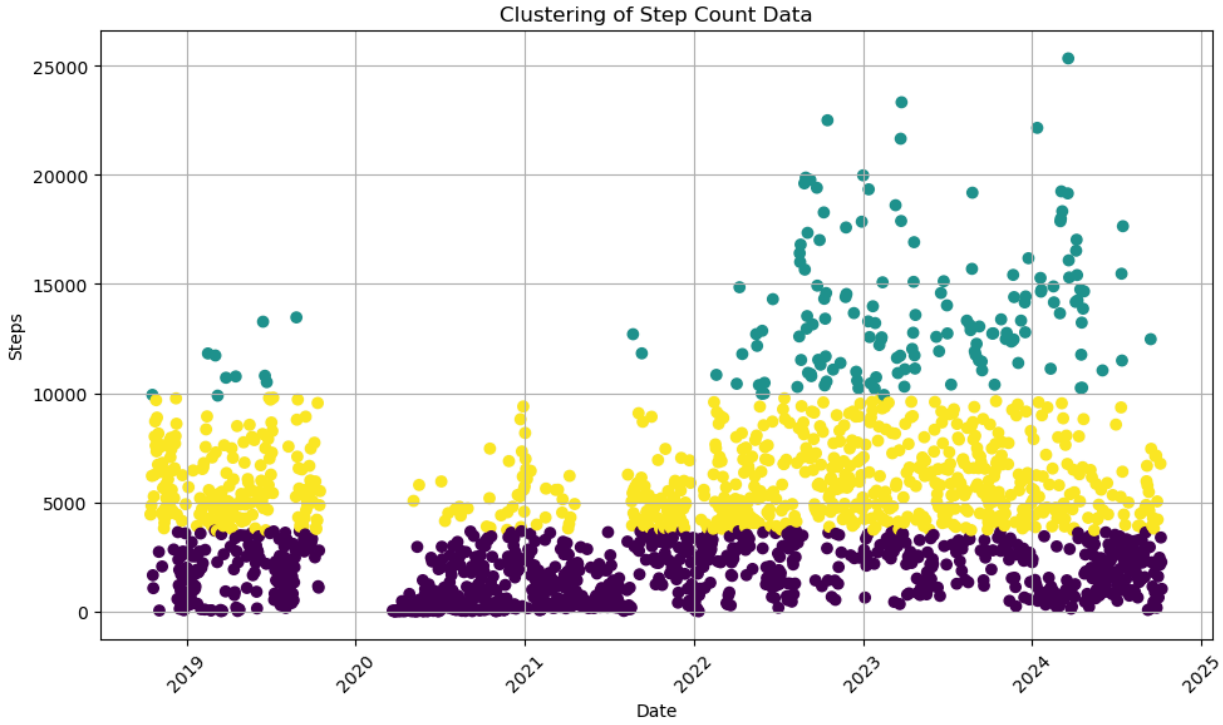
In [55]:
```
# Calculate average distance walked per time zone
avg_by_timezone = df_daily.groupby('timeZone')['distance_walked'].mean().sort_

# Plot average distance walked by time zone
avg_by_timezone.plot(kind='bar', title='Average Distance Walked by TimeZone', 
plt.show()
```



In [56]:
```
from sklearn.cluster import KMeans

# Prepare data for clustering
X = df_daily[['steps']]

# Apply K-Means clustering with 3 clusters (low, medium, high activity)
kmeans = KMeans(n_clusters=3)
df_daily['cluster'] = kmeans.fit_predict(X)

# Plot the results
plt.figure(figsize=(10, 6))
plt.scatter(df_daily.index, df_daily['steps'], c=df_daily['cluster'], cmap='vi
plt.title('Clustering of Step Count Data')
plt.xlabel('Date')
plt.ylabel('Steps')
plt.grid(True)
plt.xticks(rotation=45)
```

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

```
/Users/zeldudu/anaconda3/lib/python3.11/site-packages/sklearn/cluster/_kmeans.
py:1412: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
```

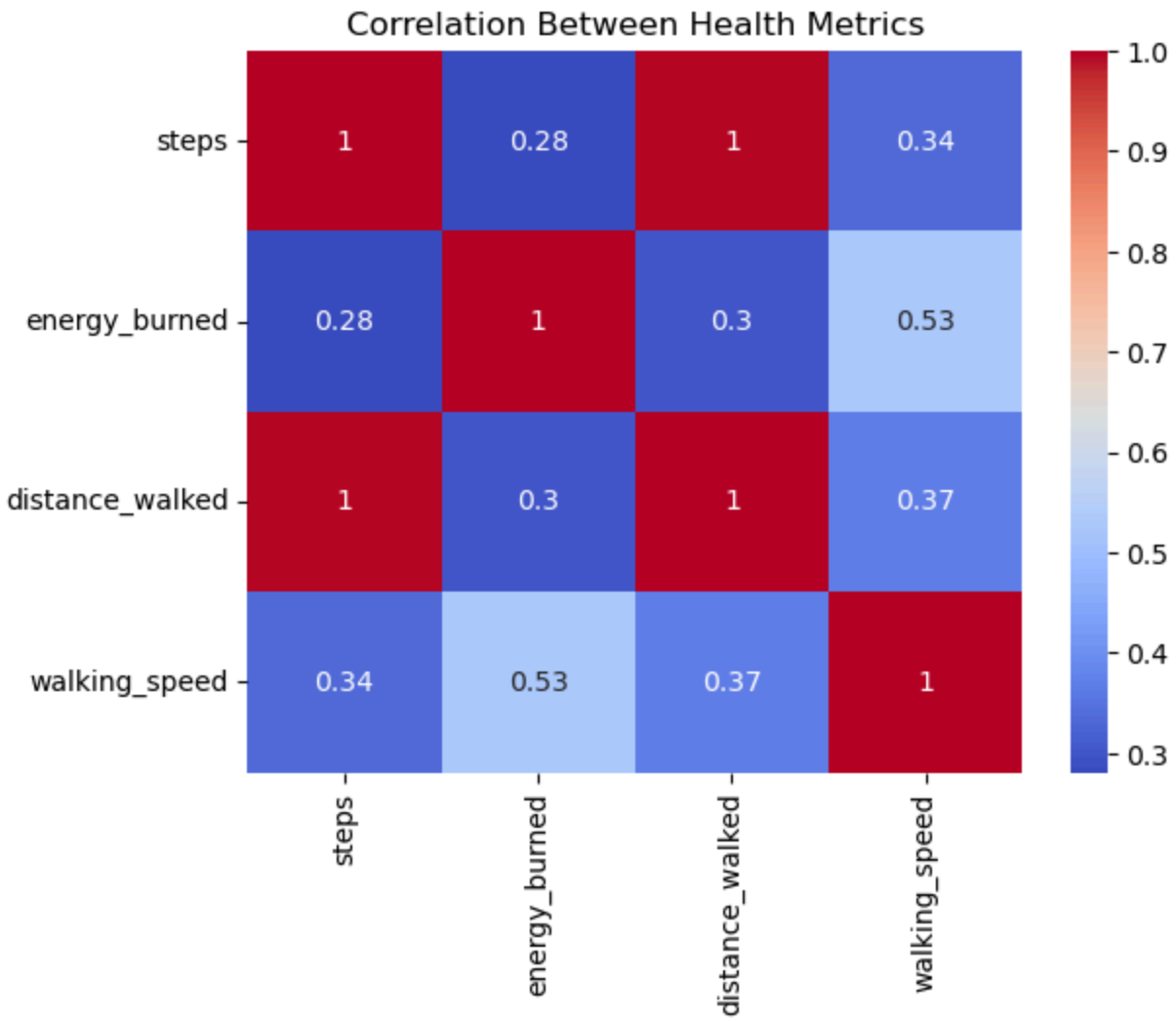


In [58]:
```
# Basic descriptive statistics for step count
print(df_daily['steps'].describe())

# Descriptive statistics for other variables like energy burned, walking speed
print(df_daily[['energy_burned', 'distance_walked', 'walking_speed']].describe
```

```
count     2000.000000
mean      4236.894000
std       3846.731389
min          3.000000
25%       1240.500000
50%       3513.500000
75%       5808.500000
max      25333.000000
Name: steps, dtype: float64
       energy_burned  distance_walked  walking_speed
count    2000.000000      2000.000000    2000.000000
mean      503.811158         1.787419       1.731423
std       700.134238         1.652445       1.385161
min         0.000000         0.000050       0.000000
25%         0.000000         0.515015       0.000000
50%         0.000000         1.467586       2.639584
75%      1455.498250         2.437487       2.867659
max      1597.499000        10.922764       4.194255
```

In [59]:
```
# Correlation matrix for step count, energy burned, distance walked, walking sp
correlation_matrix = df_daily[['steps', 'energy_burned', 'distance_walked', 'wa

# Visualize the correlation matrix
```

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



Loading Data From Flo App

In [60]:
```
import re
import pandas as pd

file_path = '/Users/zeldudu/Downloads/menstrual data/res.txt'

# Define regex patterns to match start and end dates
start_date_pattern = r'Period start date: (\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}'
end_date_pattern = r'Period end date: (\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}\.\d

# Initialize lists to store the extracted dates
start_dates = []
end_dates = []

with open(file_path, 'r') as file:
    text_data = file.read()

    # Find all start and end dates using regex
    start_dates = re.findall(start_date_pattern, text_data)
    end_dates = re.findall(end_date_pattern, text_data)
```

```
# Create a DataFrame from the extracted dates
menstrual_df = pd.DataFrame({
    'period_start_date': pd.to_datetime(start_dates),
    'period_end_date': pd.to_datetime(end_dates)
})

# Remove the time component, keeping only the date
menstrual_df['period_start_date'] = menstrual_df['period_start_date'].dt.date
menstrual_df['period_end_date'] = menstrual_df['period_end_date'].dt.date

print(menstrual_df.head())
```

```
  period_start_date period_end_date
0        2024-09-07      2024-09-12
1        2024-08-04      2024-08-09
2        2024-07-01      2024-07-06
3        2024-05-26      2024-05-31
4        2024-04-22      2024-04-28
```

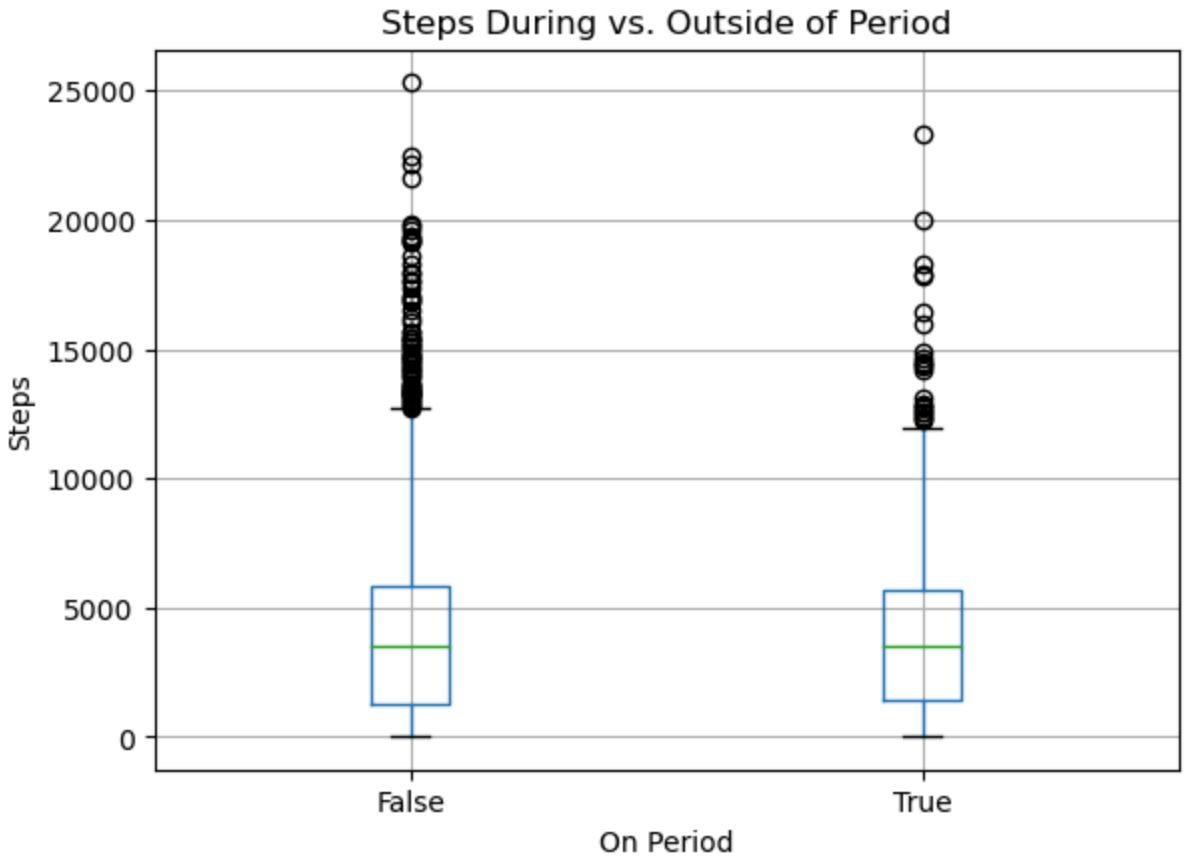
In [61]:
```
# Convert the 'period_start_date' and 'period_end_date' to datetime64[ns]
menstrual_df['period_start_date'] = pd.to_datetime(menstrual_df['period_start_
menstrual_df['period_end_date'] = pd.to_datetime(menstrual_df['period_end_date

# Ensure that the index of df_daily is also in datetime format
df_daily.index = pd.to_datetime(df_daily.index)

# Add a column 'is_on_period' to mark whether a day is within a period
df_daily['is_on_period'] = False

# Loop through the menstrual data and flag the period days in df_daily
for index, row in menstrual_df.iterrows():
    df_daily.loc[(df_daily.index >= row['period_start_date']) & (df_daily.index

print(df_daily[['steps', 'is_on_period']].head())
```

```
            steps  is_on_period
2018-10-15   4436         False
2018-10-16   4473         False
2018-10-17   4846         False
2018-10-18   6205         False
2018-10-19   9919         False
```

In [62]:
```
# Compare average step counts on period days vs. non-period days
steps_on_period = df_daily[df_daily['is_on_period'] == True]['steps'].mean()
steps_off_period = df_daily[df_daily['is_on_period'] == False]['steps'].mean()

print(f"Average steps during period: {steps_on_period}")
print(f"Average steps outside period: {steps_off_period}")
```

```
Average steps during period: 4339.4735294117645
Average steps outside period: 4215.883734939759
```

In [64]:
```
# Boxplot to compare step counts on period vs. non-period days
df_daily.boxplot(column='steps', by='is_on_period')
plt.title('Steps During vs. Outside of Period')
plt.suptitle('')
plt.xlabel('On Period')
plt.ylabel('Steps')
plt.show()
```



In [66]:
```
df_daily.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2000 entries, 2018-10-15 to 2024-10-07
Data columns (total 9 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   steps            2000 non-null   int64
 1   distance_walked  2000 non-null   float64
 2   walking_speed    2000 non-null   float64
 3   energy_burned    2000 non-null   float64
 4   timeZone         2000 non-null   object
 5   day_of_week      2000 non-null   object
 6   month            2000 non-null   object
 7   cluster          2000 non-null   int32
 8   is_on_period     2000 non-null   bool
dtypes: bool(1), float64(3), int32(1), int64(1), object(3)
memory usage: 134.8+ KB
```

In [67]:
```
df_daily.tail(5)
```

Out[67]:

	steps	distance_walked	walking_speed	energy_burned	timeZone
2024-10-03	2137	0.904995	3.019865	1436.081	America/Argentina/Buenos_Aires
2024-10-04	6766	2.672039	2.787223	1492.052	America/Argentina/Buenos_Aires
2024-10-05	3384	1.486473	2.809593	1445.217	America/Argentina/Buenos_Aires
2024-10-06	2259	1.017390	2.710761	1485.527	America/Argentina/Buenos_Aires
2024-10-07	1031	0.464186	0.000000	1007.112	America/Argentina/Buenos_Aires

Data wrangling

In [68]:
```
(df_daily['timeZone'] == "Unknown").sum()
```

Out[68]: 1124

In [69]:
```
(df_daily['walking_speed'] == 0).sum()
```

Out[69]: 772

In [70]:
```
(df_daily['energy_burned'] == 0).sum()
```

Out[70]: 1317

First problem:

- 1124 out of 2000 data points (56%) are missing information on time zones. One option would be to fill in the missing values with the most frequent time zone. However, given that I've studied in 3 different countries in high school and 4 for university (I like to keep things interesting!), this would be an oversimplification that doesn't reflect my lifestyle. So instead, I'll leave the missing time zones as they are and omit them from certain parts of the analysis.

Second problem:

- 772 out of 2000 data points (38%) are missing for walking speed. Given that I very rarely feel the need to sprint (thanks to the lack of wild animals in the cities) and considering that I also can't afford to stroll leisurely in this economy, my walking speed is relatively consistent. Therefore, I'll fill in the missing walking speed data using the median.

Third problem:

- 1317 out of 2000 data points (66%) are missing for energy burned. Since I don't know the exact formula that Apple Health uses for calculating how much energy I burn on a

daily basis, and I'd rather not guess for similar reasons with the first problem, I'll leave this data as missing and omit it from relevant analyses.

In [71]:
```
# Replace 0 walking_speed values with the median walking speed
median_walking_speed = df_daily.loc[df_daily['walking_speed'] > 0, 'walking_spe
df_daily['walking_speed'].replace(0, median_walking_speed, inplace=True)
print((df_daily['walking_speed'] == 0).sum())
df_daily.head()
```

```
0
```

Out[71]:

	steps	distance_walked	walking_speed	energy_burned	timeZone	day_of_week	month
2018-10-15	4436	1.673496	2.827384	0.0	Unknown	Monday	Octobe
2018-10-16	4473	1.679522	2.827384	0.0	Unknown	Tuesday	Octobe
2018-10-17	4846	1.932489	2.827384	0.0	Unknown	Wednesday	Octobe
2018-10-18	6205	2.484088	2.827384	0.0	Unknown	Thursday	Octobe
2018-10-19	9919	3.455977	2.827384	0.0	Unknown	Friday	Octobe

Fourth problem:

- We've stumbled upon a rather glaring 157-day (5-month) gap in the data—no valid records from October 16, 2019, to March 22, 2020. Looking at the plot, this period shows up as a dramatic cliff, connecting the dots between two distant points. This was the period I had no phone.

How I plan to handle it:

- It would be inaccurate to interpolate the data given how long the gap is. Filling in 5 months of data based on trends before or after would not be ideal (especially given that Covid happened right after this period). I think the most reasonable approach is to omit the gap period from analyses where continuous time matters (e.g., trend analysis), but leave it in for any larger overviews where the gap doesn't skew the results significantly. This approach respects the reality of my phone-less life without artificially fabricating data.

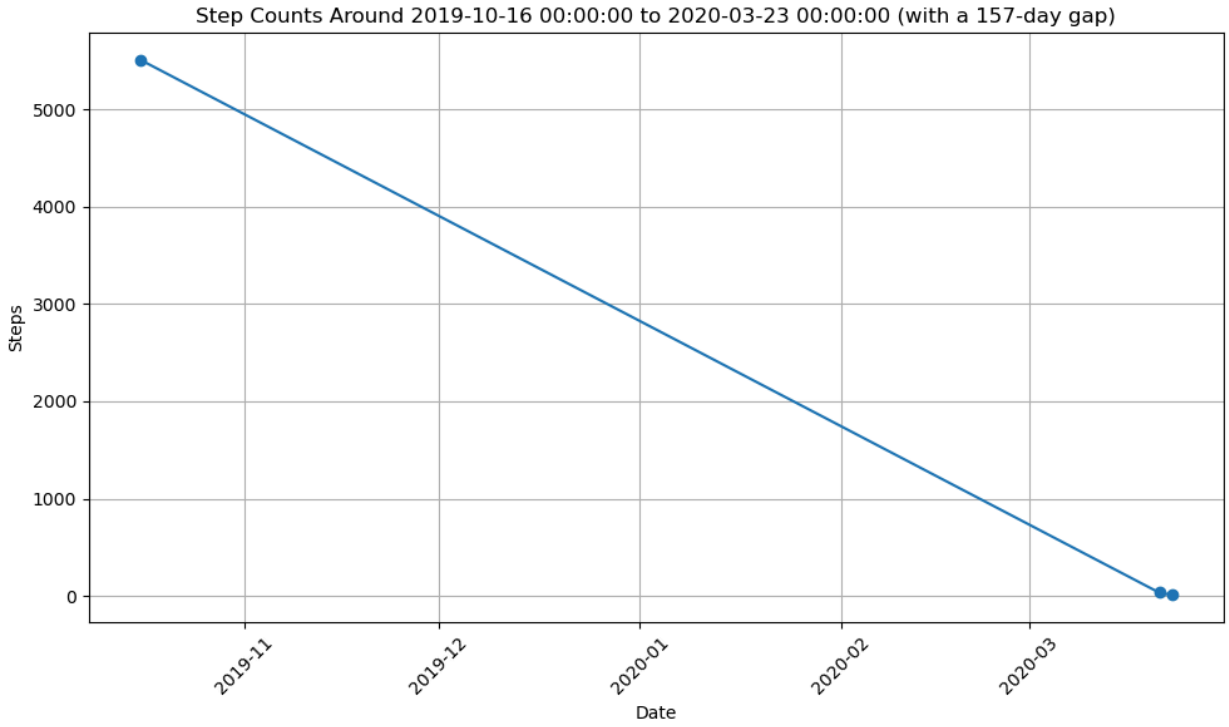
In [72]:
```
# Identify gaps in time
df_daily['date_diff'] = df_daily.index.to_series().diff().dt.days
print(df_daily[df_daily['date_diff'] > 5])  # Days with large gaps
```

```
            steps  distance_walked  walking_speed  energy_burned timeZone  \
2020-03-21     34         0.012235       2.827384            0.0  Unknown

           day_of_week  month  cluster  is_on_period  date_diff
2020-03-21    Saturday  March        0         False      157.0
```

In [73]:
```
# Find the closest date before the gap
prev_date = df_daily[:'2020-03-20'].last_valid_index()
print("Last date before the gap:", prev_date)

# Find the closest date after the gap
next_date = df_daily['2020-03-22':].first_valid_index()
print("First date after the gap:", next_date)
```

```
Last date before the gap: 2019-10-16 00:00:00
First date after the gap: 2020-03-23 00:00:00
```

In [74]:
```
# Filter the data between the found previous and next dates
df_gap_period = df_daily.loc[prev_date:next_date]

# Plot the step counts for this period
plt.figure(figsize=(10, 6))
plt.plot(df_gap_period.index, df_gap_period['steps'], marker='o')
plt.title(f'Step Counts Around {prev_date} to {next_date} (with a 157-day gap)')
plt.xlabel('Date')
plt.ylabel('Steps')
plt.xticks(rotation=45)
plt.grid(True)
plt.tight_layout()
plt.show()
```



Next, some feature engineering:

In [3]:
```
df_daily['day_of_week'] = df_daily.index.day_name()
df_daily['month'] = df_daily.index.month_name()
```

2/ Data Analysis:

Approach justification:

a. Why try multiple (2) prediction tasks?

For this project, the goal is to explore how various physical health metrics—such as steps, walking speed, and energy burned—can help predict two different aspects of daily life:

1. Whether I'm on my period (Classification Task)
2. How many steps I'll take on a given day (Regression Task)

By focusing on these two tasks, we can better understand the relationship between the health data I've been tracking and how it correlates with my menstrual cycle and overall activity levels. This dual approach provides a richer understanding of the dataset and uncovers patterns that a single prediction task might miss.

b. Why try multiple (2) models for each task?

Model comparison in statistics class was my favorite section and I'm curious to see how this process differs in machine learning. For both tasks, we'll use 2 different machine learning models:

1. Tree-based models (Decision Trees/ Random Forests): More flexible models to capture non-linear relationships and handle interactions between features.

2. Neural Networks (Multi-layer perceptrons MLP): The more complex model out of the two to capture the patterns that other models may have missed.

c. What I expect to learn*

- Compare performance: We'll see which models are best suited for each task, given this data set. Some models might work better for classification, while others might be more effective for regression.
- Understand feature importance: Tree-based models will (hopefully) help us explore which features are most predictive of period days or step counts.
- Explore relationships: How health metrics influence my activity.

In [8]:
```
import pandas as pd
df_daily = pd.read_pickle('/Users/zeldudu/Desktop/156-data.pkl')
df_daily.head()
```

Out[8]:

	steps	distance_walked	walking_speed	energy_burned	timeZone	is_on_period	day_of
2018-10-15	4436	1.673496	2.827384	0.0	Unknown	False	M
2018-10-16	4473	1.679522	2.827384	0.0	Unknown	False	Tu
2018-10-17	4846	1.932489	2.827384	0.0	Unknown	False	Wedr
2018-10-18	6205	2.484088	2.827384	0.0	Unknown	False	Th
2018-10-19	9919	3.455977	2.827384	0.0	Unknown	False	

3/ Classification Task

Decision Trees:

1. What are Decision Trees?

a. Definition: A supervised (data with labels) machine learning model used for both classification and regression tasks. It starts with a root node and ends with a decision made by leaves.

b. How it works:

- The algorithm starts at the top, which is the root node. Then a feature is chosen to split the data based on a criteria (such as "Is walking speed > 2m/s?").
- Each split results in a child node, which contrinues to split until a stopping condition is met (like reaching a maxmimum depth or all data points belong to one class)
- The nodes at the end of the tree are called leaf nodes, which are the predicted values that we are looking for.
- For this classificaiton task, the Gini impurity at a node is calculated as:

$$\text{Gini} = 1 - \sum_{i=1}^{C} p_i^2$$

where p_i is the probability of picking a data point with class i, and C is the number of classes.

This formula works because p_i^2 represents the likelihood of randomly choosing two samples of the same class. Thus, subtracting this from 1 gives the probability that the two randomly chosen samples are from different classes, indicating impurity.

When a dataset is split into two subsets, the overall Gini impurity is calculated as a weighted average of the Gini impurities of the two subsets:

$$\text{Gini}_{\text{split}} = \frac{n_1}{n} \cdot \text{Gini}(D_1) + \frac{n_2}{n} \cdot \text{Gini}(D_2)$$

where n is total number of samples. n_1 and n_2 are the number of samples in the subsets D_1 and D_2. $Gini(D_1)$ and $Gini(D_2)$ are the Gini impurities of the subsets after the split.

The decision tree algorithm aims to reduce the Gini impurity as much as possible at each step. By evaluating splits using the weighted average of the Gini impurities, it ensures that splits producing purer subsets are chosen.

2. What are we doing?

One-hot encoding:

- This is a technique to convert categorical data into numerical format, which is necessary for algorithms that only work with numerical data.
- Ie: We have a categorical feature "day of the week" with values: ["Monday", "Tuesday", "Wednesday"]. One-hot encoding converts them into three seperate columns:

$$\[\text{Monday} = [1, 0, 0], \quad \text{Tuesday} = [0, 1, 0], \quad \text{Wednesday} = [0, 0, 1]\]$$

- Each row in the original dataset is now represented as a vector in a matrix, where each column corresponds to one category.
- If we were to use numbers like Monday = 1, Tuesday = 2, Wednesday = 3, it would suggest that there is some kind of ranking/ order between the days, which doesn't make sense because days of the week should be treated aas indepdent feature. By having separate colunmns, we can present each category is as a "yes or no" question. "Is this day monday?" "Is this day Wednesday?" and so on.

StandardScaler:

- This is a preprocessing technique that standardizes features by removing the mean and scaling to unit variance.
- For example, in this datasets like the one being used in this assignment where features have different ranges (e.g., "steps" might be in thousands, "walking speed" in meters/second), algorithms can be biased toward larger values.
- Standardization ensures each feature has a mean of 0 and a standard deviation of 1, making all features equally important.

$$\text{Standardized Value} = \frac{x_i - \text{mean}}{\text{standard deviation}}$$

Train_test_split (80%, 20%)

- The training set is used to train the model, while the testing set is used to evaluate its performance on unseen data.
- I skipped using a separate validation set here because the hyperparameters are being fine-tuned using grid search. This method splits the training set internally to find the

best parameters.

In [2]:
```
# Create a copy of the original DataFrame
df_daily_original = df_daily.copy()

# Apply one-hot encoding on the copy
df_daily_encoded = pd.get_dummies(df_daily_original, columns=['timeZone', 'day_
```

In [3]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Select features
features = ['steps', 'distance_walked', 'walking_speed', 'energy_burned'] + \
           [col for col in df_daily_encoded if col.startswith('timeZone')] + \
           [col for col in df_daily_encoded if col.startswith('day_of_week')] +
           [col for col in df_daily_encoded if col.startswith('month')] + \
           [col for col in df_daily_encoded if col.startswith('cluster')]

# Create the feature matrix (X) and target variable (y)
X = df_daily_encoded[features]
y = df_daily_encoded['is_on_period']

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, randor

# Standardize the numerical features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Print shape to verify
print(f'Training data shape: {X_train_scaled.shape}')
print(f'Test data shape: {X_test_scaled.shape}')
```

```
Training data shape: (1600, 42)
Test data shape: (400, 42)
```

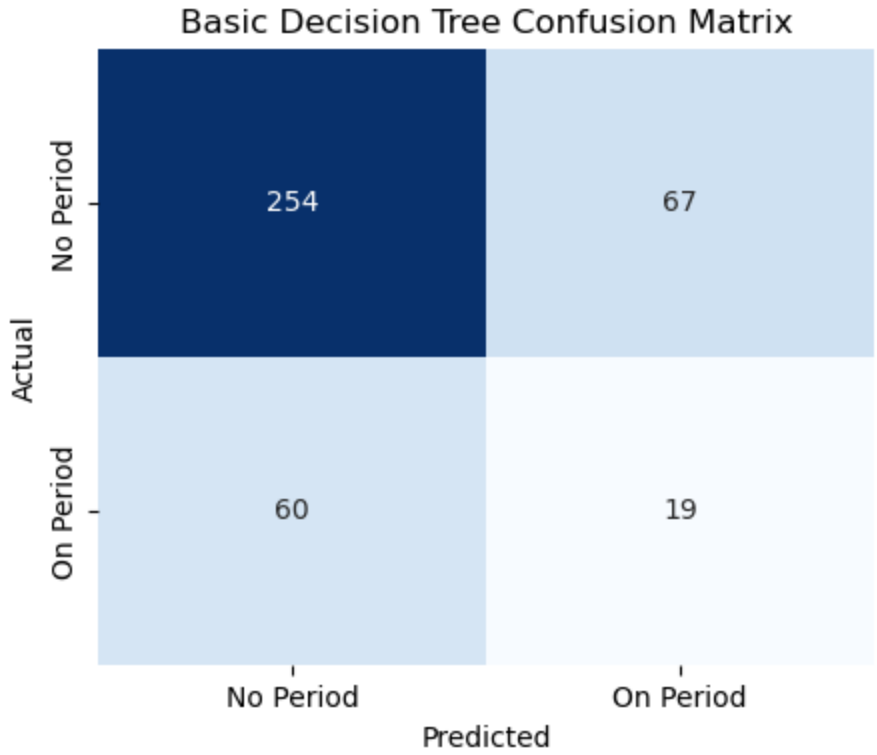
Basic Decision Tree

We start with a basic deicion tree without any tuning:

In [43]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_s
confusion_matrix, roc_auc_score, roc_curve
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV

# Step 1: Decision Tree without any tuning
tree = DecisionTreeClassifier(random_state=42)
tree.fit(X_train, y_train)
y_pred_tree = tree.predict(X_test)

# Evaluate the basic model
accuracy = accuracy_score(y_test, y_pred_tree)
precision = precision_score(y_test, y_pred_tree)
recall = recall_score(y_test, y_pred_tree)
f1 = f1_score(y_test, y_pred_tree)
```

```
roc_auc = roc_auc_score(y_test, y_pred_tree)

print(f'Basic Decision Tree Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
print(f'ROC-AUC: {roc_auc}')

# Confusion matrix for basic model
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_tree), annot=True, fmt='d', cmap='
            xticklabels=['No Period', 'On Period'], yticklabels=['No Period',
plt.title('Basic Decision Tree Confusion Matrix')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

```
Basic Decision Tree Accuracy: 0.6825
Precision: 0.22093023255813954
Recall: 0.24050632911392406
F1 Score: 0.23030303030303031
ROC-AUC: 0.5158917938404511
```



Class-Weighted Decision Tree

Since the basic decision tree model gives us a low F1 score, we can try to improve this by examining and addressing the problem of class imbalance (Class no-period has 83% of the samples and and Class period has only 17%).

This is a problem because since the majority class has more samples, the model might become biased towards predicting that class (false), leading to poor generalization on the minority class. In this case, the minority class is of particular interest because I could better plan my schedules and prepare accessories accordingly.

There are several ways to address this problem such as random oversampling, random undersampling, synthetic over-sampling: SMOTE. For this assignment, I will use the class weights as it's simple and doesn't modify the dataset. It would have been better if I could test multiple methods but there are other sections I want to prioritize in this assignment.

Class weights: assign higher weights to the samples of the minority class and lower weigths to the majority class during the traning process. In Python, we will set the class_weight parameter to 'balanced', which adjusts the weights inversely proportional to the class frequencies, meaning the model pay more attention to the minority class.

$$\text{Weight for class\_i} = \frac{n_{total}}{n_i x C}$$

where n_{total} is the total number of samples, n_i is the number of samples in class_i, C is the number of classes

The smaller the denominator is (samples of the minority class), the more weight it will have. This makes the model pay more attention to the minority class by giving a higher weight.

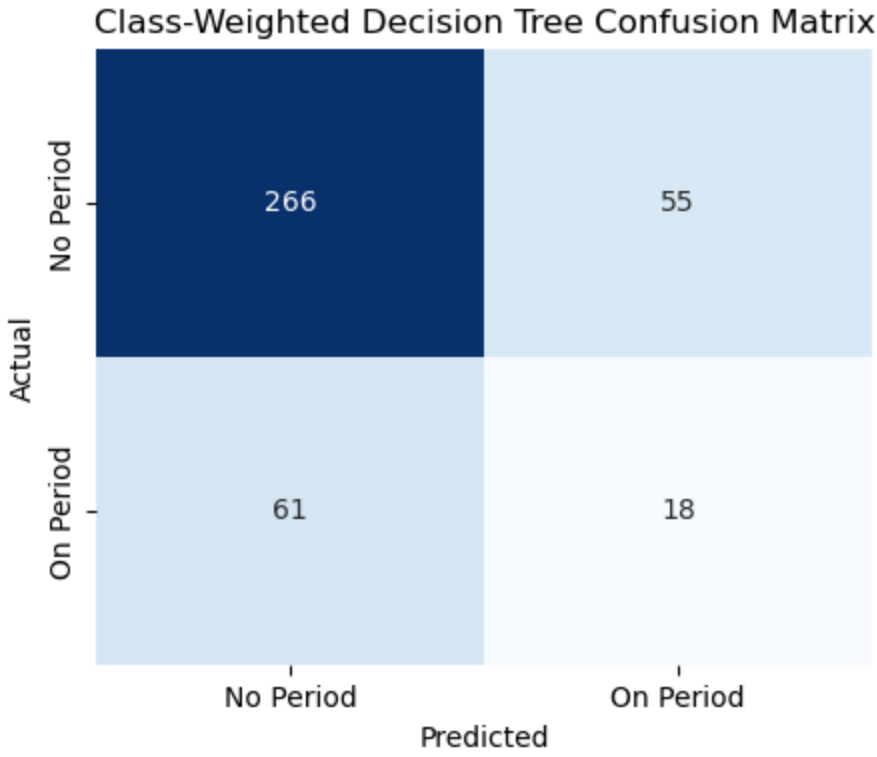
In [4]:
```
# Check for class imbalance
print(df_daily['is_on_period'].value_counts(), df_daily['is_on_period'].value_
```

```
is_on_period
False    1660
True      340
Name: count, dtype: int64 is_on_period
False    0.83
True     0.17
Name: count, dtype: float64
```

In [42]:
```
# Step 2: Decision Tree with class weighting
tree_weighted = DecisionTreeClassifier(class_weight='balanced', random_state=4
tree_weighted.fit(X_train, y_train)
y_pred_weighted = tree_weighted.predict(X_test)

# Evaluate the class-weighted model
accuracy = accuracy_score(y_test, y_pred_weighted)
precision = precision_score(y_test, y_pred_weighted)
recall = recall_score(y_test, y_pred_weighted)
f1 = f1_score(y_test, y_pred_weighted)
roc_auc = roc_auc_score(y_test, y_pred_weighted)

print(f'Class-Weighted Decision Tree Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
print(f'ROC-AUC: {roc_auc}')

# Confusion matrix for class-weighted model
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_weighted), annot=True, fmt='d', cma
            xticklabels=['No Period', 'On Period'], yticklabels=['No Period',
plt.title('Class-Weighted Decision Tree Confusion Matrix')
plt.ylabel('Actual')
```

```
plt.xlabel('Predicted')
plt.show()
```

```
Class-Weighted Decision Tree Accuracy: 0.71
Precision: 0.2465753424657534
Recall: 0.22784810126582278
F1 Score: 0.23684210526315788
ROC-AUC: 0.5282542687014471
```



Hyperparameter Tuning

Since the Class-Weighted Decision Tree improved F1 score only by 0.01, we will try hyperparameter tuning.

Why Hyperparameter Tuning is Done After Class Weighting?

- Class weighting addresses class imbalance, improving the model's performance on the minority class. However, the decision tree's structure (e.g., depth, splitting criteria) is still set to default values, which may not be optimal.
- By performing hyperparameter tuning after class weighting, we can further fine-tune the tree's parameters (e.g., depth, number of samples required to split a node) to achieve better results

What is Grid Search doing?

- A method of cross validation. It exhaustively tries every combination of the provided hyper-parameter values to find the best model.
- For example, if we are tuning 2 parameters:

Max depth: [3, 5, 7]/ Min samples split: [2, 5]

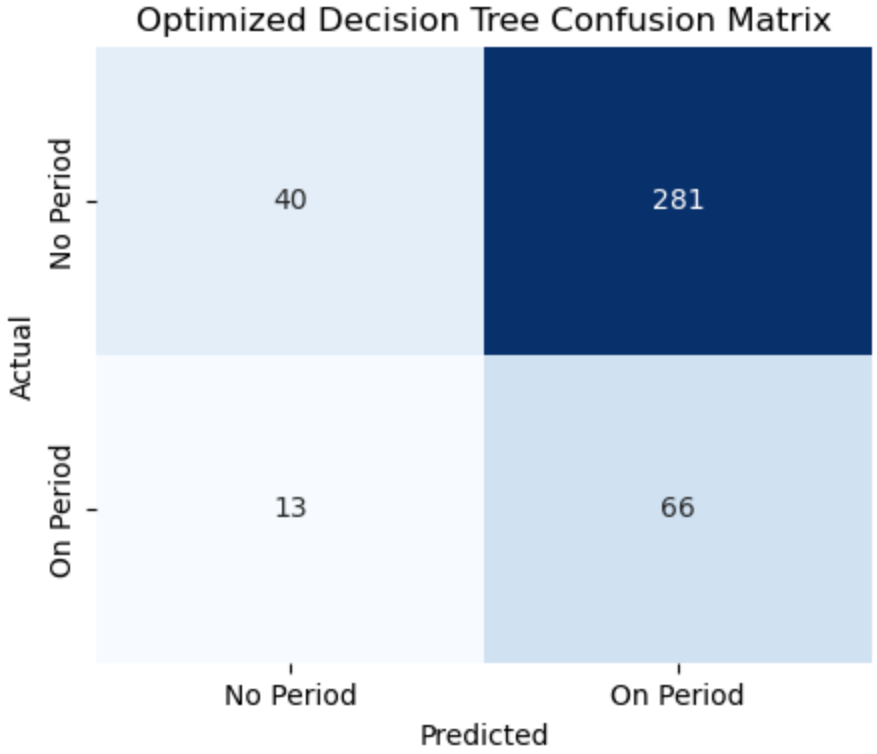
This results in 3×2=6 possible combinations.

- Each combination is evaluated using cross validation, where dataset is splited into multiple subsets (folds), the model is trained on some of them, and evaluated on others. This helps ensure that the model's performance is robust and not just tailored to the training data. The hyperparameters that produce the best average performance across the folds are chosen as the optimal parameters.

- **k-fold cross validation** was used in this assignment (Grid Search). The dataset is divided into k folds (cv=5, 5 folds). For each combination of hyperparameters, the model is trained on k–1 folds and evaluated on the remaining fold. This process is repeated k times, with a different fold used as the validation set each time. This ensures that the model is evaluated on multiple subsets of the data, reducing the risk of overfitting.

In [45]:
```
# Step 3: Hyperparameter Tuning with GridSearchCV on the class-weighted model
param_grid = {
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'criterion': ['gini', 'entropy']
}

grid_search = GridSearchCV(estimator=tree_weighted, param_grid=param_grid, cv=5
grid_search.fit(X_train, y_train)

# Best model after hyperparameter tuning
best_tree = grid_search.best_estimator_
y_pred_best_tree = best_tree.predict(X_test)

# Evaluate the optimized model
accuracy = accuracy_score(y_test, y_pred_best_tree)
precision = precision_score(y_test, y_pred_best_tree)
recall = recall_score(y_test, y_pred_best_tree)
f1 = f1_score(y_test, y_pred_best_tree)
roc_auc = roc_auc_score(y_test, y_pred_best_tree)

print(f'Optimized Decision Tree Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
print(f'ROC-AUC: {roc_auc}')

# Confusion matrix for the optimized model
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_best_tree), annot=True, fmt='d', cm
            xticklabels=['No Period', 'On Period'], yticklabels=['No Period',
plt.title('Optimized Decision Tree Confusion Matrix')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

```
Fitting 5 folds for each of 72 candidates, totalling 360 fits
Optimized Decision Tree Accuracy: 0.265
Precision: 0.19020172910662825
Recall: 0.8354430379746836
F1 Score: 0.30985915492957744
ROC-AUC: 0.48002681493749755
```



Error metrics:

- Accuracy: The proportion of correctly classified instances out of the total instances. Accuracy is most useful when classes are balanced and can be misleading when the majority class dominates (which is the case of this dataset)

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Number of Instances}}$$

- Precision: The proportion of correctly predicted positive instances out of all instances predicted as positive. Precision should be favored when false positives are costly.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Postitives} + \text{False Postitives}}$$

- Recall: The proportion of correctly predicted positive instances out of all actual positive instances. Recall should be favored when false negatives are costly.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Postitives} + \text{False Negatives}}$$

- F1 Score: A balanced mean of precision and recall.

$$\text{F1 Score} = 2 * \frac{\text{Precision x Recall}}{\text{Precision} + \text{Recall}}$$

- ROC-AUC: Area Under the Curve. Measures the model's ability to distinguish between classes, with higher values indicating better performance. An AUC of 0.5 indicates no discriminative power (similar to random guessing). An AUC of 1 indicates perfect classification.

For this classification problem:

- **Recall is crucial** here because it measures the ability of the model to correctly identify period days. Maximizing recall reduces the chance of false negatives, ensuring that the model catches as many actual period days as possible.
- F1 Score is also important because it balances both precision and recall. Since there is some cost associated with false positives (predicting a period when there isn't one), it makes sense to use the F1 Score to maintain a balance.
- Thus, we will focus on both F1 Score and Recall, but give slightly more weight to recall.

In [9]:
```
# Create a DataFrame for the results
results = pd.DataFrame({
    'Model': ['Basic Decision Tree', 'Class-Weighted Decision Tree', 'Optimize
    'Accuracy': [0.68, 0.71, 0.27],
    'Precision': [0.22, 0.25, 0.19],
    'Recall': [0.24, 0.23, 0.84],
    'F1 Score': [0.23, 0.24, 0.31],
    'ROC-AUC': [0.52, 0.53, 0.48]
})=

results.head()
```

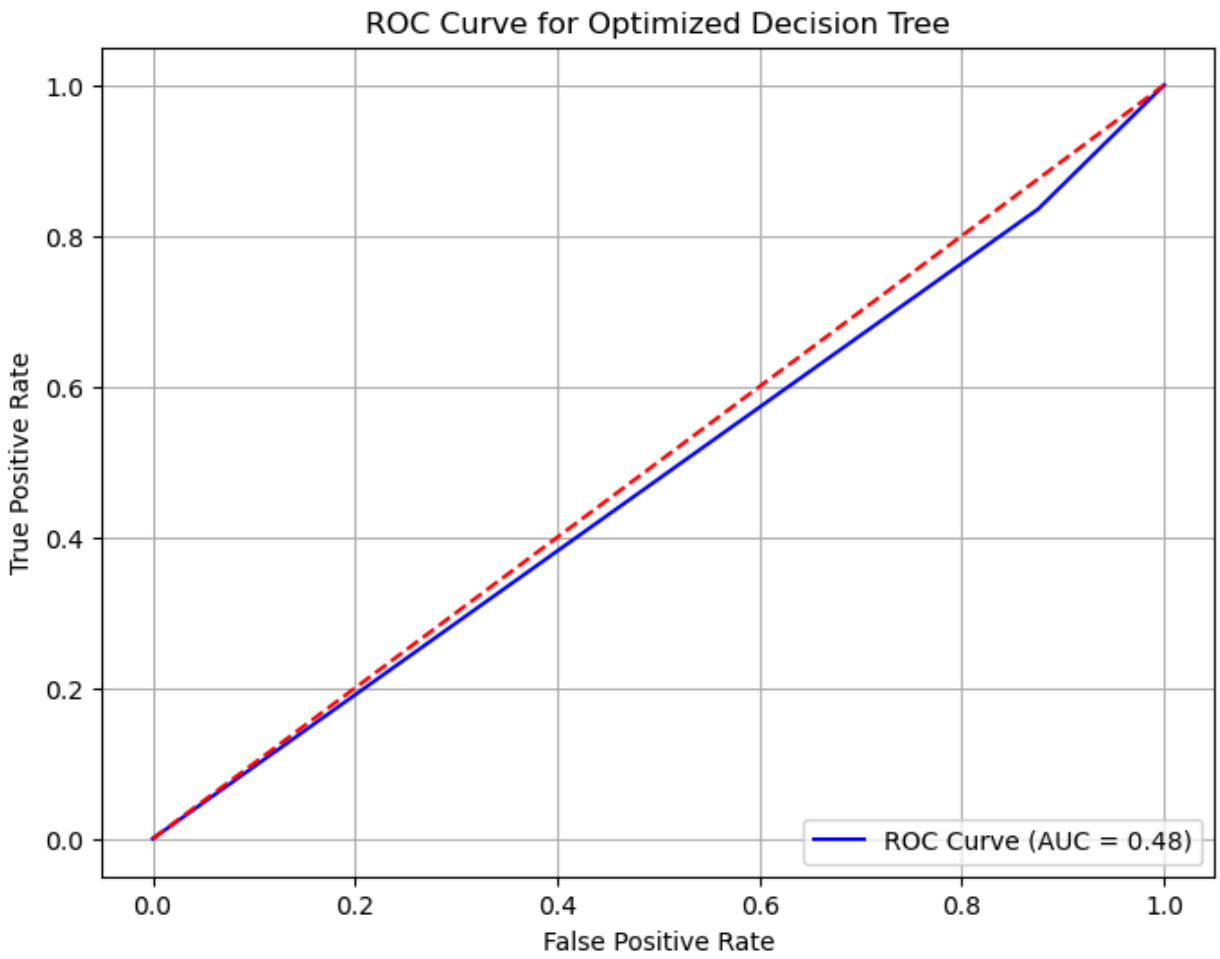
Out[9]:

	Model	Accuracy	Precision	Recall	F1 Score	ROC-AUC
0	Basic Decision Tree	0.68	0.22	0.24	0.23	0.52
1	Class-Weighted Decision Tree	0.71	0.25	0.23	0.24	0.53
2	Optimized Decision Tree	0.27	0.19	0.84	0.31	0.48

The Optimized Decision Tree has the highest recall (0.84) and F1 score (0.31), but at the cost of significantly lower accuracy and a ROC-AUC performing worse than random chance. This suggests that the model may classify many positive instances correctly but at the expense of increased false positives.

In [46]:
```
# Plot ROC Curve for the optimized decision tree
fpr, tpr, thresholds = roc_curve(y_test, y_pred_best_tree)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.title('ROC Curve for Optimized Decision Tree')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

```
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```



This plot further showcases how our 'optimized' model is performing worse than random chance (the red line). This indicates that the model is not effectively learning the patterns in the data, potentially due to issues such as insufficient data quality, inappropriate feature selection, or limitations in model architecture. The results suggest that further investigation is needed to improve the model's performance, such as trying alternative algorithms, adding more features, or collecting more representative data.

Let's try our luck with Neural Networks!

Neural Network

What is Neural Network?

A neural network is composed of layers of interconnected neurons (nodes) that process input data to perform tasks such as classification or regression.

Data first goes through the input layer which receives the input features. Then they are tossed into (one or more) hidden layers where the data is transformed using biases, weights, and activation functions. The output layer produces what we're predicting.

Mathematical Representation

1. Layer Computations:

For each layer l, the output before the applying the activation is given by:

$$Z^{[l]} = A^{[l-1]}W^{[l]} + b^{[l]}$$

- $A[l-1]$ is the activated output from the previous layer (or the input X for the first layer). W^[l] is the weight matrix for layer l. b^[l] is the bias vector layer l. Z^l is the wieghted sum before activation for layer l.

1. Activation Function:

After computing Z^l, an activation fucntion is applied to introduce non-linearity. Without activation functions, the entire neural network would be equivalent to a linear transformation.

$$A^{[l]} = f(Z^{[l]})$$

The sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

maps the output to a range between 0 and 1, making it particularly useful for binary classification problems. It squashes large positive and negative values to a probability-like range, enabling the network to predict class probabilities.

Backpropagation

This is the process opposite of forward propagation (input -> hidden layers -> output), aimed to minimize the error between the predicted output and the actual target values by adjusting the weights.

1. Loss function:

The error for the output layer is computed using a loss function:

$$L = \frac{1}{m}\sum_{i=1}^{m} \text{Loss}(\hat{y}_i, y_i)$$

where m is the number of samples, $\hat{y}_i$ is the predicted output, and y_i is the actual output.

1. Gradient Calculation using the chain rule:

Next, we need to figure out which direction each weight should be adjusted to minimize the loss function above. We do that using the chain rule - the gradients of the loss function with respect to each weight:

$$\frac{\partial L}{\partial W^{[l]}} = \frac{\partial L}{\partial A^{[l]}} \cdot \frac{\partial A^{[l]}}{\partial Z^{[l]}} \cdot \frac{\partial Z^{[l]}}{\partial W^{[l]}}$$

1. Weight Update:

After direction, we iteratively update the weights using gradient descent. Gradient descent in simple terms is an optimization method for finding a local minimum of a differentiable function.

$$W^{[l]} = W^{[l]} - \alpha \frac{\partial L}{\partial W^{[l]}}$$

Here we introduce α, the learning rate, and we want to find a number not too small (steps too tiny = long time to find the optimal solution) but not too big (steps too large we may overshoot the minimum).

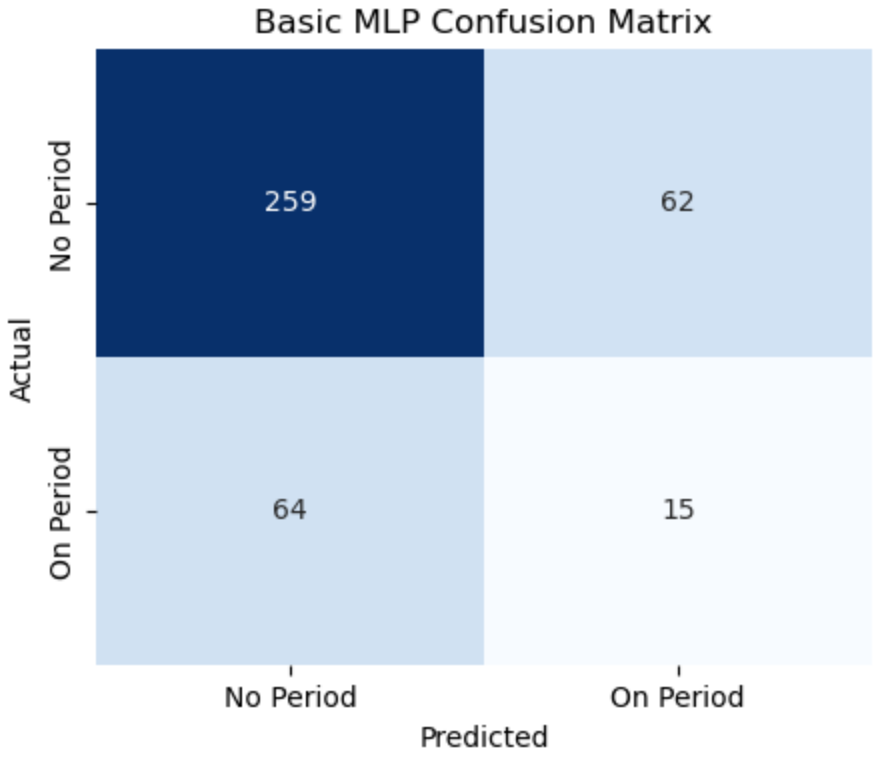
For this project, we used a Multi-Layer Perceptron (MLP), which is a type of neural network composed of fully connected layers.

Basic Neural Network (MLP)

In [19]:
```
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_s
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import RandomizedSearchCV

# Step 1: Basic Neural Network (MLP) Model Training and Evaluation
mlp_basic = MLPClassifier(hidden_layer_sizes=(64, 32), max_iter=2000, solver='

# Train the Neural Network on the scaled training data
mlp_basic.fit(X_train_scaled, y_train)

# Make predictions on the test set
y_pred_basic = mlp_basic.predict(X_test_scaled)

# Evaluate the basic model
accuracy_basic = accuracy_score(y_test, y_pred_basic)
precision_basic = precision_score(y_test, y_pred_basic)
recall_basic = recall_score(y_test, y_pred_basic)
f1_basic = f1_score(y_test, y_pred_basic)
roc_auc_basic = roc_auc_score(y_test, y_pred_basic)

print(f'Basic MLP Accuracy: {accuracy_basic}')
print(f'Precision: {precision_basic}')
print(f'Recall: {recall_basic}')
print(f'F1 Score: {f1_basic}')
print(f'ROC-AUC: {roc_auc_basic}')

# Confusion matrix for the basic model
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_basic), annot=True, fmt='d', cmap=
            xticklabels=['No Period', 'On Period'], yticklabels=['No Period',
plt.title('Basic MLP Confusion Matrix')
plt.ylabel('Actual')
```

```
plt.xlabel('Predicted')
plt.show()
```

```
Basic MLP Accuracy: 0.685
Precision: 0.19480519480519481
Recall: 0.189873417721519
F1 Score: 0.19230769230769232
ROC-AUC: 0.49836350013801806
```



Oversampled MLP

Oversampling is a technique used to address class imbalance by increasing the number of samples in the minority class. For our Multi-Layer Perceptron (MLP), we applied manual oversampling to ensure that the model sees a more balanced set of examples during training.

For Decision Trees, we used class weights to balance the data. However, this method does not work for MLP in scikit-learn as it does not support the class_weight parameter. Thus, we'll manually oversample.

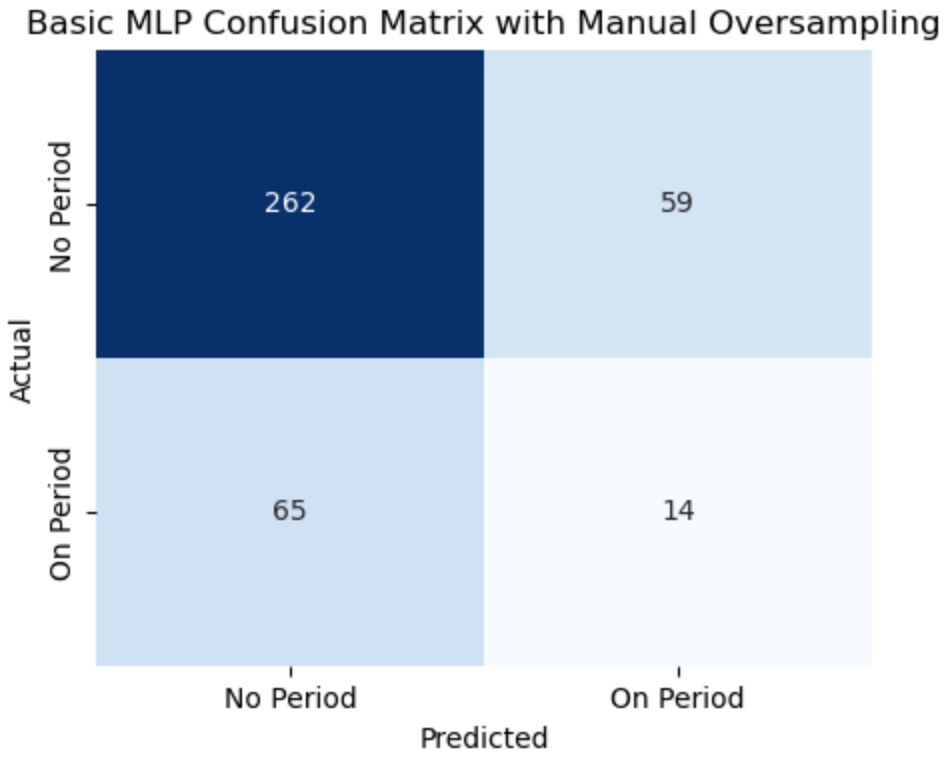
Manual oversampling:

I randomly duplicated instances from the minority class to match the number of instances in the majority class. This created a new training data set where both classes have the same number of samples.

The oversampled dataset was used to train the MLP. By training on this balanced dataset, the model is better able to learn the patterns associated with the minority class (on-period), rather than being biased towards predicting the majority class (no period).

In [13]:
```
import numpy as np
from sklearn.utils import resample

# Combine the training features and target into a single DataFrame
train_data = np.concatenate((X_train_scaled, y_train.values.reshape(-1, 1)), ax

# Separate majority and minority classes
majority_class = train_data[train_data[:, -1] == 0]  # No period
minority_class = train_data[train_data[:, -1] == 1]  # On period

# Upsample the minority class
minority_upsampled = resample(minority_class,
                              replace=True,            # Sample with replacemen
                              n_samples=len(majority_class), # Match majority c
                              random_state=42)

# Combine the majority class with the upsampled minority class
train_data_balanced = np.vstack((majority_class, minority_upsampled))

# Separate features and target again
X_train_balanced = train_data_balanced[:, :-1]
y_train_balanced = train_data_balanced[:, -1]

# Train the Basic MLP Model on the manually resampled data
mlp_basic = MLPClassifier(hidden_layer_sizes=(64, 32), max_iter=500, random_sta

# Train the model on the balanced training data
mlp_basic.fit(X_train_balanced, y_train_balanced)

# Make predictions on the original test set (X_test_scaled)
y_pred_basic = mlp_basic.predict(X_test_scaled)

# Evaluate the model
accuracy_basic = accuracy_score(y_test, y_pred_basic)
precision_basic = precision_score(y_test, y_pred_basic)
recall_basic = recall_score(y_test, y_pred_basic)
f1_basic = f1_score(y_test, y_pred_basic)
roc_auc_basic = roc_auc_score(y_test, y_pred_basic)

print(f'Basic MLP Accuracy: {accuracy_basic}')
print(f'Precision: {precision_basic}')
print(f'Recall: {recall_basic}')
print(f'F1 Score: {f1_basic}')
print(f'ROC-AUC: {roc_auc_basic}')

# Confusion matrix for the manually balanced model
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_basic), annot=True, fmt='d', cmap=
            xticklabels=['No Period', 'On Period'], yticklabels=['No Period',
plt.title('Basic MLP Confusion Matrix with Manual Oversampling')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

```
Basic MLP Accuracy: 0.69
Precision: 0.1917808219178082
Recall: 0.17721518987341772
F1 Score: 0.18421052631578946
ROC-AUC: 0.49670728341022913
```



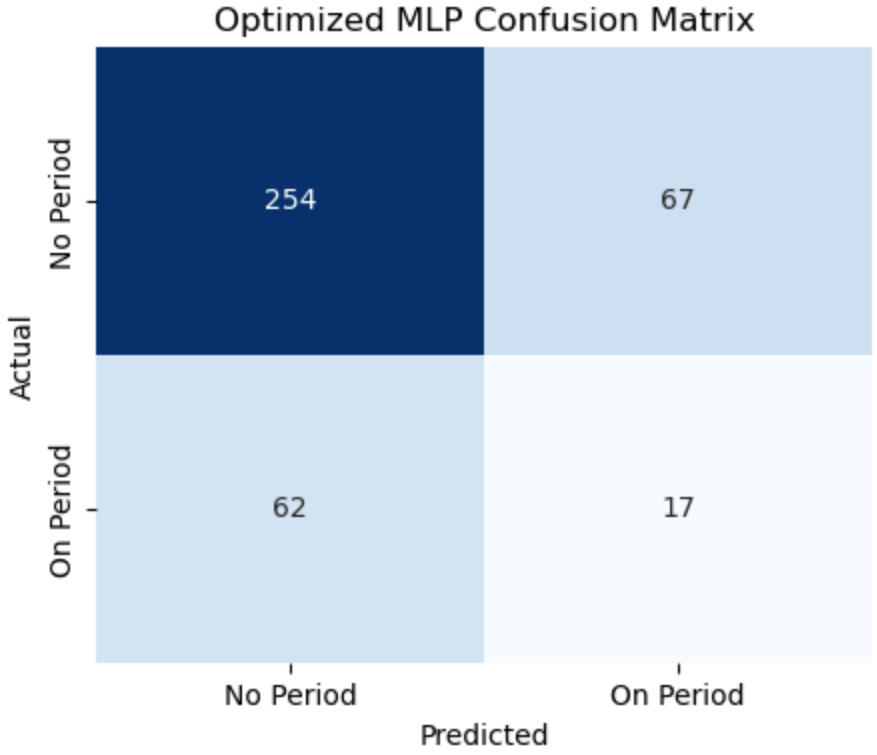
Hyperameter Tuning:

We used RandomizedSearchCV to explore different combinations of hyperparameters, such as the number of hidden layers, activation functions, and learning rates.

In [27]:
```
from sklearn.model_selection import RandomizedSearchCV

# Step 2: Hyperparameter Tuning with RandomizedSearchCV
param_dist = {
    'hidden_layer_sizes': [(64, 32), (128, 64), (64, 32, 16)],
    'activation': ['relu', 'tanh'],
    'solver': ['adam', 'sgd'],
    'alpha': [0.0001, 0.001],
    'learning_rate': ['constant', 'adaptive'],
    'max_iter': [200, 500, 1000]
}

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=MLPClassifier(random_state=42, ea
                                   param_distributions=param_dist,
                                   n_iter=10,
                                   cv=5,
                                   scoring='f1',
                                   n_jobs=-1,
                                   verbose=1)

# Fit RandomizedSearchCV on the scaled balanced training data
random_search.fit(X_train_balanced, y_train_balanced)


# Get the best model from RandomizedSearchCV
best_mlp = random_search.best_estimator_
y_pred_optimized = best_mlp.predict(X_test_scaled)
```

```
accuracy_optimized = accuracy_score(y_test, y_pred_optimized)
precision_optimized = precision_score(y_test, y_pred_optimized)
recall_optimized = recall_score(y_test, y_pred_optimized)
f1_optimized = f1_score(y_test, y_pred_optimized)
roc_auc_optimized = roc_auc_score(y_test, y_pred_optimized)

print(f'Optimized MLP Accuracy: {accuracy_optimized}')
print(f'Optimized MLP Precision: {precision_optimized}')
print(f'Optimized MLP Recall: {recall_optimized}')
print(f'Optimized MLP F1 Score: {f1_optimized}')
print(f'Optimized MLP ROC-AUC: {roc_auc_optimized}')

# Confusion matrix for the optimized model
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_optimized), annot=True, fmt='d', cm
            xticklabels=['No Period', 'On Period'], yticklabels=['No Period',
plt.title('Optimized MLP Confusion Matrix')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()

print(f'Best Hyperparameters: {random_search.best_params_}')
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Optimized MLP Accuracy: 0.6775
Optimized MLP Precision: 0.20238095238095238
Optimized MLP Recall: 0.21518987341772153
Optimized MLP F1 Score: 0.2085889570552147
Optimized MLP ROC-AUC: 0.5032335659923498
```




```
Best Hyperparameters: {'solver': 'adam', 'max_iter': 200, 'learning_rate': 'co
nstant', 'hidden_layer_sizes': (128, 64), 'alpha': 0.001, 'activation': 'rel
u'}
```

In [31]:
```
# Create a DataFrame for all results
import pandas as pd
```

```
results_ = pd.DataFrame({
    'Model': ['Basic MLP', 'Oversampled MLP', 'Optimized MLP'],
    'Accuracy': [0.69, 0.69, 0.68],
    'Precision': [0.19, 0.19, 0.20],
    'Recall': [0.19, 0.18, 0.22],
    'F1 Score': [0.19, 0.18, 0.21],
    'ROC-AUC': [0.50, 0.50, 0.50]
})

results_.head()
```

Out[31]:

	Model	Accuracy	Precision	Recall	F1 Score	ROC-AUC
0	Basic MLP	0.69	0.19	0.19	0.19	0.5
1	Oversampled MLP	0.69	0.19	0.18	0.18	0.5
2	Optimized MLP	0.68	0.20	0.22	0.21	0.5

A constant classifier:

Given the poor results from our machine learning models, we need to determine whether the problem lies in our model or if the data is not good enough to predict period days effectively. To establish a benchmark, we will use a simple baseline model: a constant classifier that always predicts the majority class ("No Period").

Explanation:

The DummyClassifier with strategy='most_frequent' is set to predict the majority class ("No Period") for all instances. This means the baseline model assumes that every day is a "No Period" day, without considering any input features.

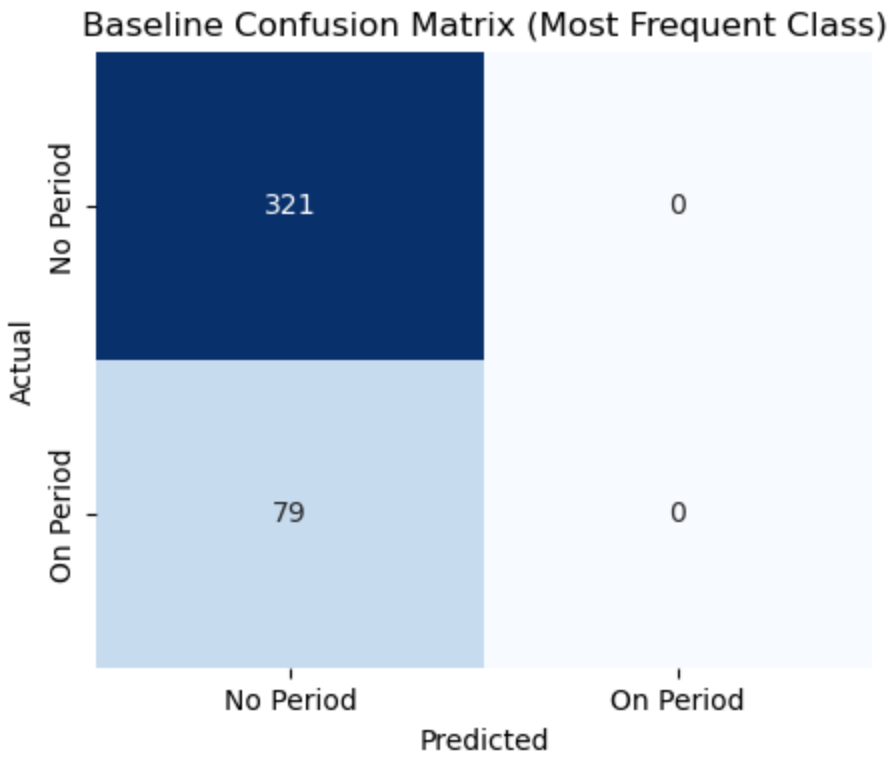
It requires minimal computation and sets a baseline that more complex models should outperform. If machine learning models perform better than this constant baseline, it indicates that they are indeed learning meaningful patterns in the data, even if the results are not ideal.

In [36]:

```
from sklearn.dummy import DummyClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_s
import seaborn as sns
import matplotlib.pyplot as plt

# Initialize the DummyClassifier with the "most_frequent" strategy
baseline_model = DummyClassifier(strategy='most_frequent', random_state=42)

# Train the baseline model on the original training data
baseline_model.fit(X_train_scaled, y_train)

# Make predictions on the original test set
y_pred_baseline = baseline_model.predict(X_test_scaled)

# Evaluate the baseline model
accuracy_baseline = accuracy_score(y_test, y_pred_baseline)
precision_baseline = precision_score(y_test, y_pred_baseline, zero_division=1)
```

```
recall_baseline = recall_score(y_test, y_pred_baseline, zero_division=1)
f1_baseline = f1_score(y_test, y_pred_baseline, zero_division=1)
roc_auc_baseline = roc_auc_score(y_test, y_pred_baseline)

# Print the baseline model results
print(f'Baseline Accuracy: {accuracy_baseline}')
print(f'Baseline Precision: {precision_baseline}')
print(f'Baseline Recall: {recall_baseline}')
print(f'Baseline F1 Score: {f1_baseline}')
print(f'Baseline ROC-AUC: {roc_auc_baseline}')

# Confusion matrix for the baseline model
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_baseline), annot=True, fmt='d', cma
            xticklabels=['No Period', 'On Period'], yticklabels=['No Period',
plt.title('Baseline Confusion Matrix (Most Frequent Class)')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

```
Baseline Accuracy: 0.8025
Baseline Precision: 1.0
Baseline Recall: 0.0
Baseline F1 Score: 0.0
Baseline ROC-AUC: 0.5
```



In [47]:
```
# Create a dictionary for the baseline results
baseline_results = {
    'Model': 'Baseline',
    'Accuracy': 0.8025,
    'Precision': 1.0,
    'Recall': 0.0,
    'F1 Score': 0.0,
    'ROC-AUC': 0.5
}
```

```
# Append the baseline results to the combined DataFrame
final_clas_results = pd.concat([final_clas_results, pd.DataFrame([baseline_resu

# Add a column to rank the models by F1 score
final_clas_results['Rank (by F1 Score)'] = final_clas_results['F1 Score'].rank

# Add a column to rank the models by Recall
final_clas_results['Rank (by Recall)'] = final_clas_results['Recall'].rank(asc

# Sort the DataFrame by F1 Score in descending order
final_clas_results = final_clas_results.sort_values(by='F1 Score', ascending=Fa

# Drop duplicate rows based on the 'Model' column to keep only one baseline ent
final_clas_results = final_clas_results.drop_duplicates(subset=['Model'], keep=

# Display the final results DataFrame
final_clas_results.head(10)
```

Out[47]:

	Model	Accuracy	Precision	Recall	F1 Score	ROC-AUC	Rank (by F1 Score)	Rank (by Recall)
0	Optimized Decision Tree	0.2700	0.19	0.84	0.31	0.48	1.0	1.0
1	Class-Weighted Decision Tree	0.7100	0.25	0.23	0.24	0.53	2.0	3.0
2	Basic Decision Tree	0.6800	0.22	0.24	0.23	0.52	3.0	2.0
3	Optimized MLP	0.6800	0.20	0.22	0.21	0.50	4.0	4.0
4	Basic MLP	0.6900	0.19	0.19	0.19	0.50	5.0	5.0
5	Oversampled MLP	0.6900	0.19	0.18	0.18	0.50	6.0	6.0
6	Baseline	0.8025	1.00	0.00	0.00	0.50	7.0	7.0

Result Discussion

1. Best Model for Period-Days Classfication:

The Optimized Decision Tree achieves the highest F1 score (0.31) and Recall (0.84), indicating that it is the most effective at correctly identifying the "Period" days among the models tested. Despite having a lower accuracy (0.27), this model's performance on the minority class is the strongest, as shown by its higher recall.

1. Baseline Model:

The Baseline model has the highest accuracy (0.8025) because it predicts the majority class ("No Period") for all instances, which would be true ~80% of the time given the frequency of this class. However, its recall and F1 score are zero, showing that it cannot identify any "Period" days. This means that while the accuracy appears high, it is not a meaningful measure in this case, as the model completely ignores the minority class.

The fact that the machine learning models have non-zero recall and F1 scores demonstrates that they are indeed learning from the data, finding patterns that go beyond merely

predicting the majority class, though still performing poorly.

1. Balanced Data vs. Basic Models:

The Class-Weighted Decision Tree performs slightly better than the Basic Decision Tree in terms of F1 score (0.24 vs. 0.23) and precision (0.25 vs. 0.22), showing that class weighting helps handle the imbalance.

For the MLP models, there is minimal improvement with oversampling, suggesting that addressing class imbalance in this way didn't significantly help in learning useful patterns.

1. ROC-AUC Scores:

The ROC-AUC values for all models are close to 0.5, indicating limited discriminative power across the models. This suggests that there is room for further improvement, such as through better feature engineering or trying other algorithms.

4/ Regression

Random Forest:

In the classification task, we have learnt about decision trees and how they work. Like any forests, a random forest is also made up of trees, decision trees in particular. The final prediction is obtained by averaging the predictions from all the individual trees.

Data Sampling:

- For each decision tree in the random forest, a boostrap sample is created by randomly sampling the training data with replacement. This means that each tree is trained on a slightly different dataset, which helps create diverse trees.
- For example, our original training dataset has 1,600 observations; each tree will be trained on a sample of the same size (1,600), but some observations may be repeated, while others may be omitted.

Random Feature Selection:

- At each node in a tree, a subset of the features is randomly selected, such as a subset of variables like walking_speed, energy_burned, timeZone_America/New_York, etc. This ensures that the trees are decorrelated, meaning each tree "sees" different features when learning to make predictions.

Split Criteria:

Suppose a dataset D is split into two subsets, D1 and D2, the variance reduction is calculated as:

$$\text{Variance Reduction} = \text{Var}(D) - \left(\frac{n_1}{n}\text{Var}(D_1) + \frac{n_2}{n}\text{Var}(D_2)\right)$$

The purpose of variance reduction is to **measure how much a split decreases the overall uncertainty in the target variable.** Why the subtraction: The $\text{Var}(D)$ represents the initial level of uncertainty. When we split the data into two subsets, we expected to reduce this uncertainty if the split is good. $\left(\frac{n_1}{n}\text{Var}(D_1) + \frac{n_2}{n}\text{Var}(D_2)\right)$ represent the weighted remaining uncertainty after the split.

A higher variance reduction means the split has effectively grouped similar data points together, resulting in lower variability within each group.

Prediction:

- For a given input X, each tree makes a prediction based on the mean value of the target variable in the leaf mode where X falls.
- The final prediction of RF Regression is the aberage predictions of all individual trees:

$$\hat{y} = \frac{1}{M}\sum_{j=1}^{M}\hat{y}_j$$

where M is the total number of trees in the forest, and $\hat{y}_j$ is the prediction made by the j-th tree.

Pseudocode for Random Forest:

Input: Training data (X, y), number of trees B, number of features to consider at each split m
Output: Trained random forest model

1. For i = 1 to B:

   a. Create a bootstrap sample from the training data (X_bootstrap, y_bootstrap)

   b. Train a decision tree T_i on (X_bootstrap, y_bootstrap):

   i. At each node:

   _ Randomly select m features from the available features

   _ Choose the best split among the m features based on a chosen criterion (mean squared error for regression)

   c. Add the trained tree T_i to the forest

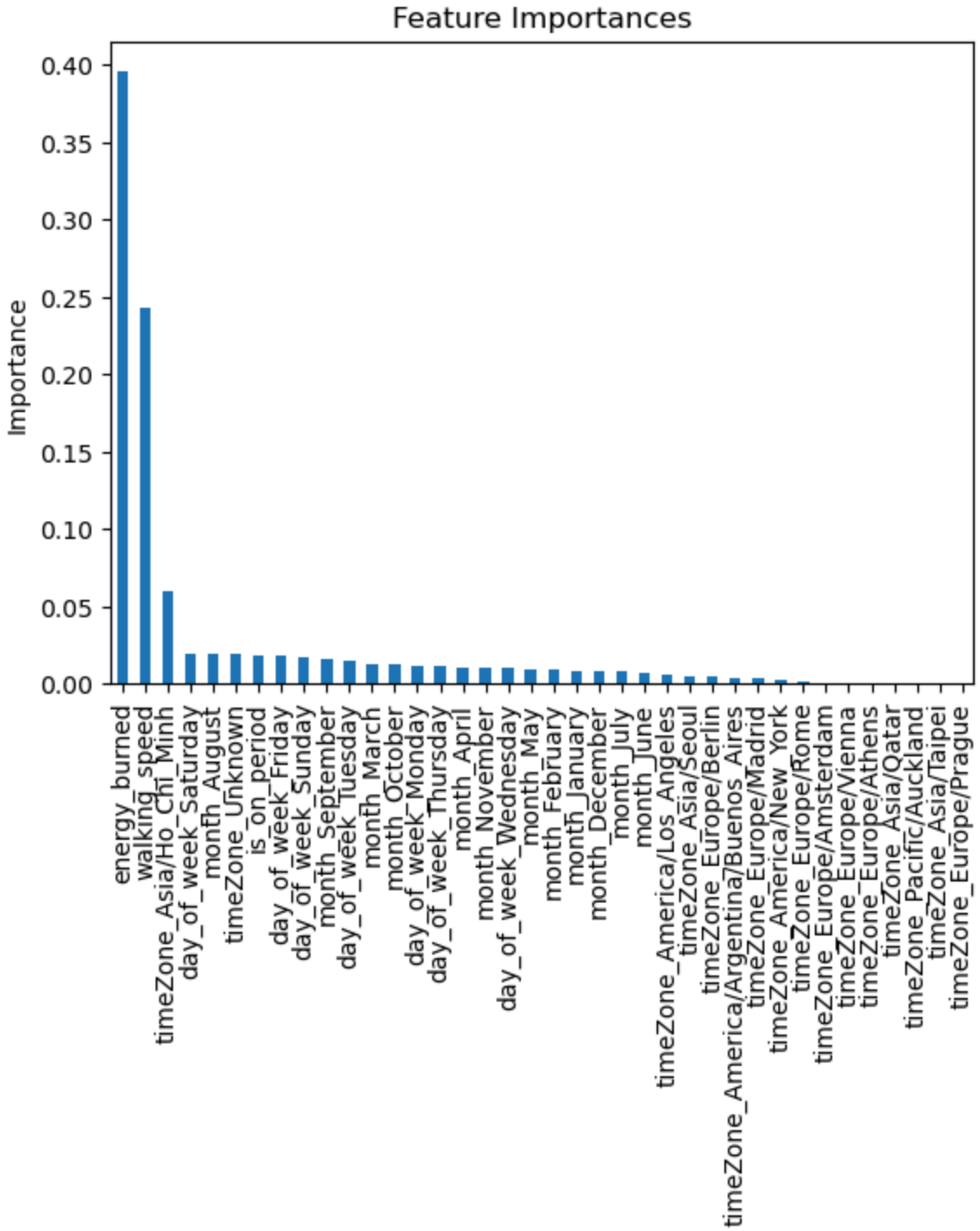
2. Output the ensemble of trees {T_1, T_2, ..., T_B}

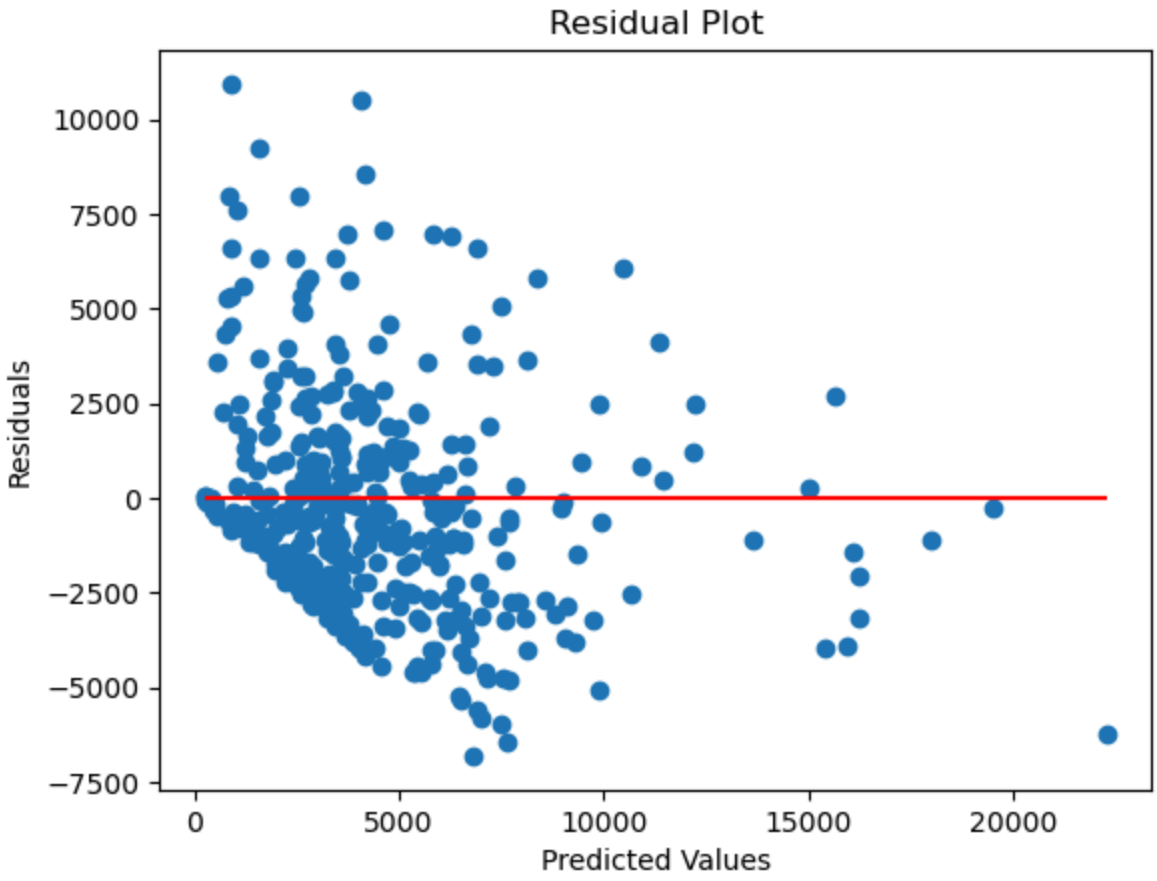
In [9]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df_daily_2 = df_daily.copy()

# One-hot encode categorical variables on the copy
df_daily_encoded = pd.get_dummies(df_daily_2, columns=['timeZone', 'day_of_weel

# Excluding 'distance_walked' due to perfect correlation
features = ['walking_speed', 'energy_burned', 'is_on_period'] + \
           [col for col in df_daily_encoded if col.startswith('timeZone')] + \
           [col for col in df_daily_encoded if col.startswith('day_of_week')] +
           [col for col in df_daily_encoded if col.startswith('month')]

# Define features (X) and target (y)
X = df_daily_encoded[features]
y = df_daily_encoded['steps']

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, randor

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Train a Random Forest Regressor
rf = RandomForestRegressor(random_state=42, n_estimators=100)

rf.fit(X_train_scaled, y_train)

# Make predictions on the test set
y_pred = rf.predict(X_test_scaled)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Random Forest Regression Results:")
print(f"Mean Squared Error (MSE): {mse}")
print(f"Mean Absolute Error (MAE): {mae}")
print(f"R-squared (R2): {r2}")

# Feature importance analysis
importances = rf.feature_importances_
features_importance = pd.Series(importances, index=features).sort_values(ascen

# Plot the feature importances
features_importance.plot(kind='bar', title='Feature Importances')
plt.ylabel('Importance')
plt.show()

# Residual analysis
residuals = y_test - y_pred
plt.scatter(y_pred, residuals)
plt.hlines(y=0, xmin=min(y_pred), xmax=max(y_pred), color='red')
plt.title('Residual Plot')
plt.xlabel('Predicted Values')
```

```
plt.ylabel('Residuals')
plt.show()
```

```
Random Forest Regression Results:
Mean Squared Error (MSE): 8373162.787122573
Mean Absolute Error (MAE): 2200.3405196816725
R-squared (R2): 0.39573406641305764
```





Plot interpretation

Feature importances:

- The importance values are calculated from how much each feature contributes to reducing impurity (like variance) across the trees in the forest.
- Energy burned and walking speed are the most influential features, which means that these features have the highest predictive power for step counts.
- Categorical variables like timezone and day of the week also play a role but to a much lesser extent.

Residual plot:

- The plot shows a pattern where residuals are more spread out at lower step counts, indicating that the model performs better with higher step counts but has more error when predicting lower step counts.
- Ideally, the residuals should be randomly scattered around the red line (zero residual), indicating no pattern. However, here, the fan shape suggests heteroscedasticity, where the variance of the error increases with the predicted values.

Error metrics:

1. Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

MSE measures the average squared difference between the actual and predicted values. It penalizes larger errors more than smaller ones due to the squaring of the differences. A lower MSE indicates better model performance.

Evaluation: For the Random Forest model, the MSE is approximately 8,373,162. This large value suggests that the predicted step counts deviate significantly from the actual values. It could be due to high variability in the target variable or insufficient model tuning.

1. Mean Absolute Error (MAE):

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

MAE measures the average absolute difference between actual and predicted values. Unlike MSE, it does not square the errors, making it more interpretable as the average magnitude of errors.

Evaluation: The MAE for the model is 2,200. This means that **on average the predictions differ from the actual step counts by 2,200 steps**. While it indicates the average error size, it doesn't show how well the model performs in cases with larger errors.

1. R-squared (R^2):

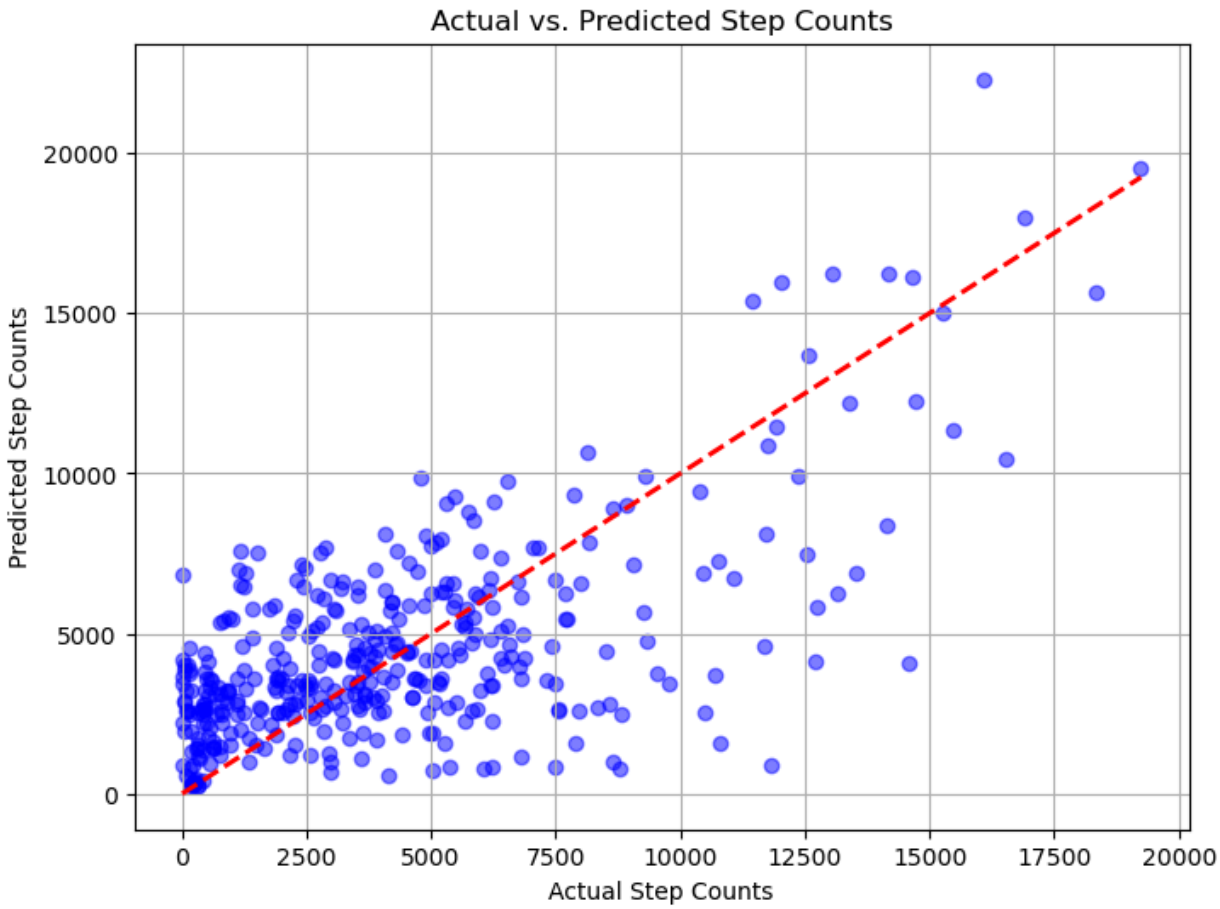
$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}$$

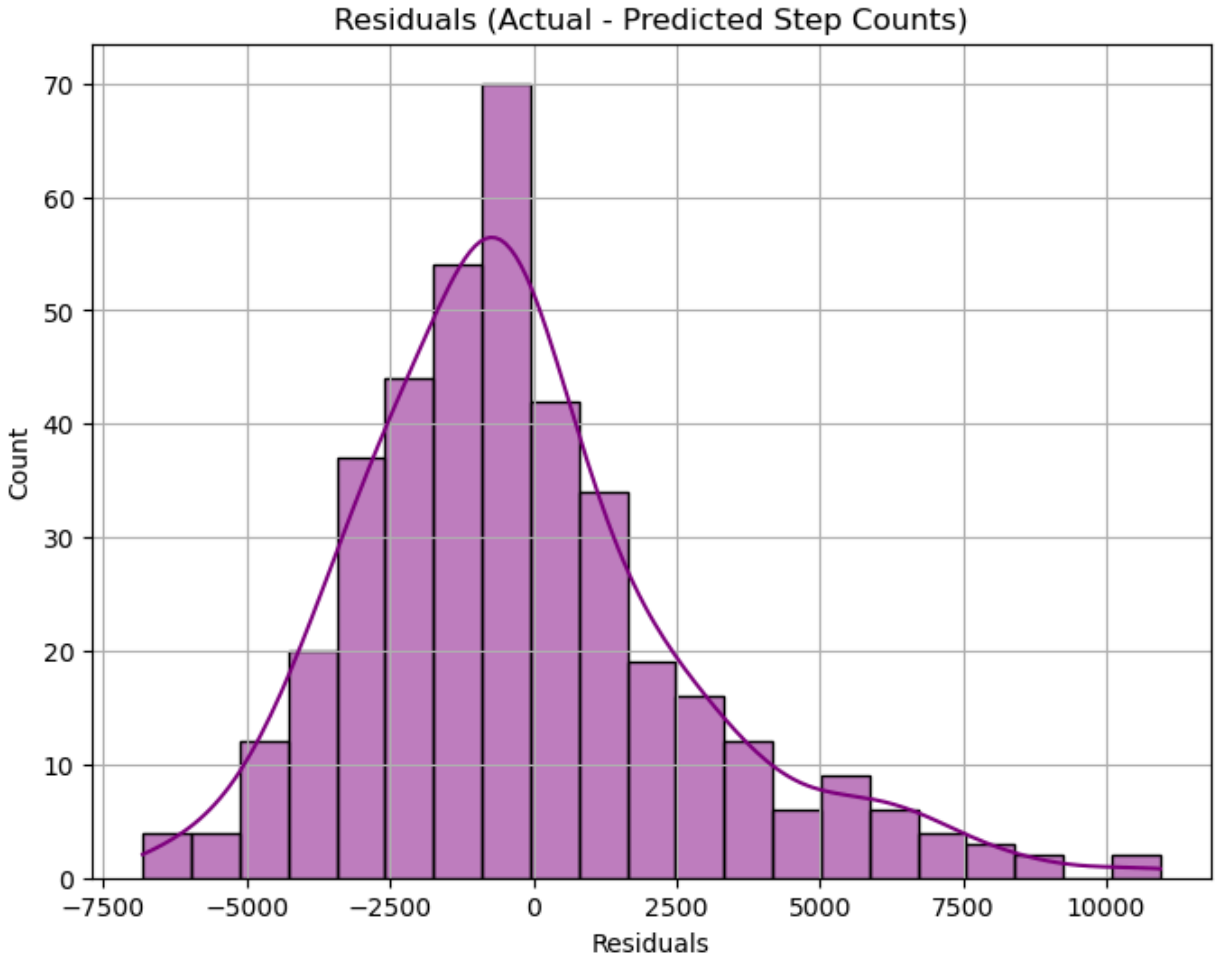
R^2 represents the proportion of variance in the dependent variable (steps) that is predictable from the independent variables. It ranges from 0 to 1, where 1 indicates perfect prediction and 0 indicates no predictive power. Negative values can occur if the model performs worse than a horizontal line representing the mean of the target values.

Evaluation: The R^2 value is approximately 0.396, suggesting that **around 39.6% of the variance in step counts is explained by the model**. This relatively low value indicates that the model is not capturing much of the variability in the data.

In [16]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Create a scatter plot of actual vs. predicted step counts
plt.figure(figsize=(8,6))
plt.scatter(y_test, y_pred, alpha=0.5, color='blue')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=
plt.title('Actual vs. Predicted Step Counts')
plt.xlabel('Actual Step Counts')
plt.ylabel('Predicted Step Counts')
plt.grid(True)
```

```
plt.show()

# Plot the residuals (difference between actual and predicted)
residuals = y_test - y_pred
plt.figure(figsize=(8,6))
sns.histplot(residuals, kde=True, color='purple')
plt.title('Residuals (Actual - Predicted Step Counts)')
plt.xlabel('Residuals')
plt.grid(True)
plt.show()
```





Plot Interpretation:

Scatter plot of Actual vs. Predicted Step Counts:

- The plot shows a reasonable trend where predicted step counts generally increase with actual step counts. However, there is a noticeable spread, especially at mid-higher step counts, indicating the model struggles more with larger values.

Residual Plot (Histogram):

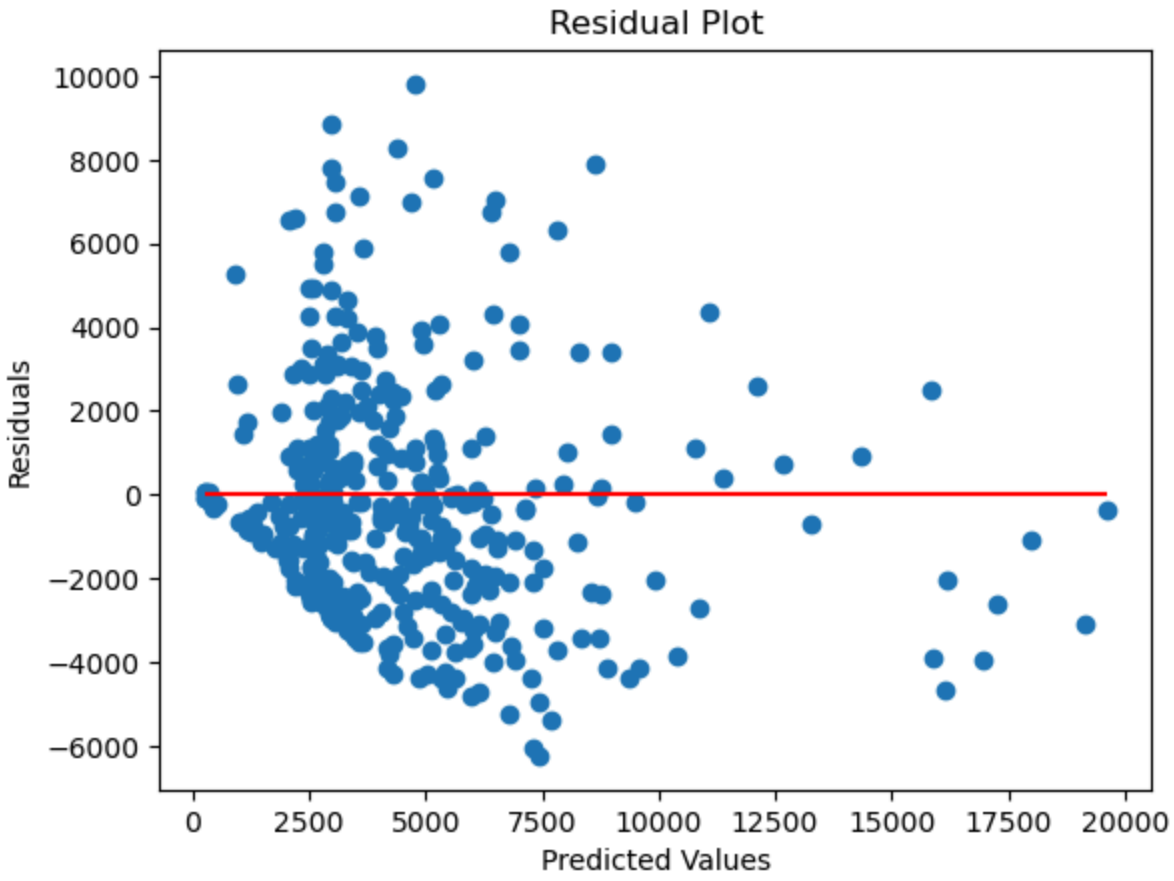
- Ideally, residuals should be centered around zero with a roughly normal distribution if the model has no systematic errors in prediction.
- The histogram shows a slight skew to the right, suggesting that the model tends to underestimate the step counts more often than it overestimates them.
- There are also some large residuals, particularly on the positive side, indicating some significant underestimations where actual step counts are much higher than predicted.

Hyperrameter Tuning for Random Forest

In [38]:
```
from sklearn.model_selection import train_test_split, GridSearchCV

# Step 6: Hyperparameter tuning using GridSearchCV
param_grid = {
```

```
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

rf = RandomForestRegressor(random_state=42)

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid,
                           cv=5, n_jobs=-1, verbose=0, scoring='neg_mean_squar

# Fit GridSearchCV on the training data
grid_search.fit(X_train_scaled, y_train)

# Get the best parameters
best_params = grid_search.best_params_
print(f"Best Hyperparameters: {best_params}")

# Step 7: Train the Random Forest Regressor with the best parameters
best_rf = grid_search.best_estimator_
best_rf.fit(X_train_scaled, y_train)

# Make predictions on the test set
y_pred = best_rf.predict(X_test_scaled)

# Step 8: Evaluate the model
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Optimized Random Forest Regression Results:")
print(f"Mean Squared Error (MSE): {mse}")
print(f"Mean Absolute Error (MAE): {mae}")
print(f"R-squared (R2): {r2}")

# Residual analysis
residuals = y_test - y_pred
plt.scatter(y_pred, residuals)
plt.hlines(y=0, xmin=min(y_pred), xmax=max(y_pred), color='red')
plt.title('Residual Plot')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.show()
```

```
Best Hyperparameters: {'bootstrap': True, 'max_depth': 10, 'min_samples_leaf':
4, 'min_samples_split': 2, 'n_estimators': 300}
Optimized Random Forest Regression Results:
Mean Squared Error (MSE): 7570066.967421255
Mean Absolute Error (MAE): 2121.250831880645
R-squared (R2): 0.45369107233654504
```



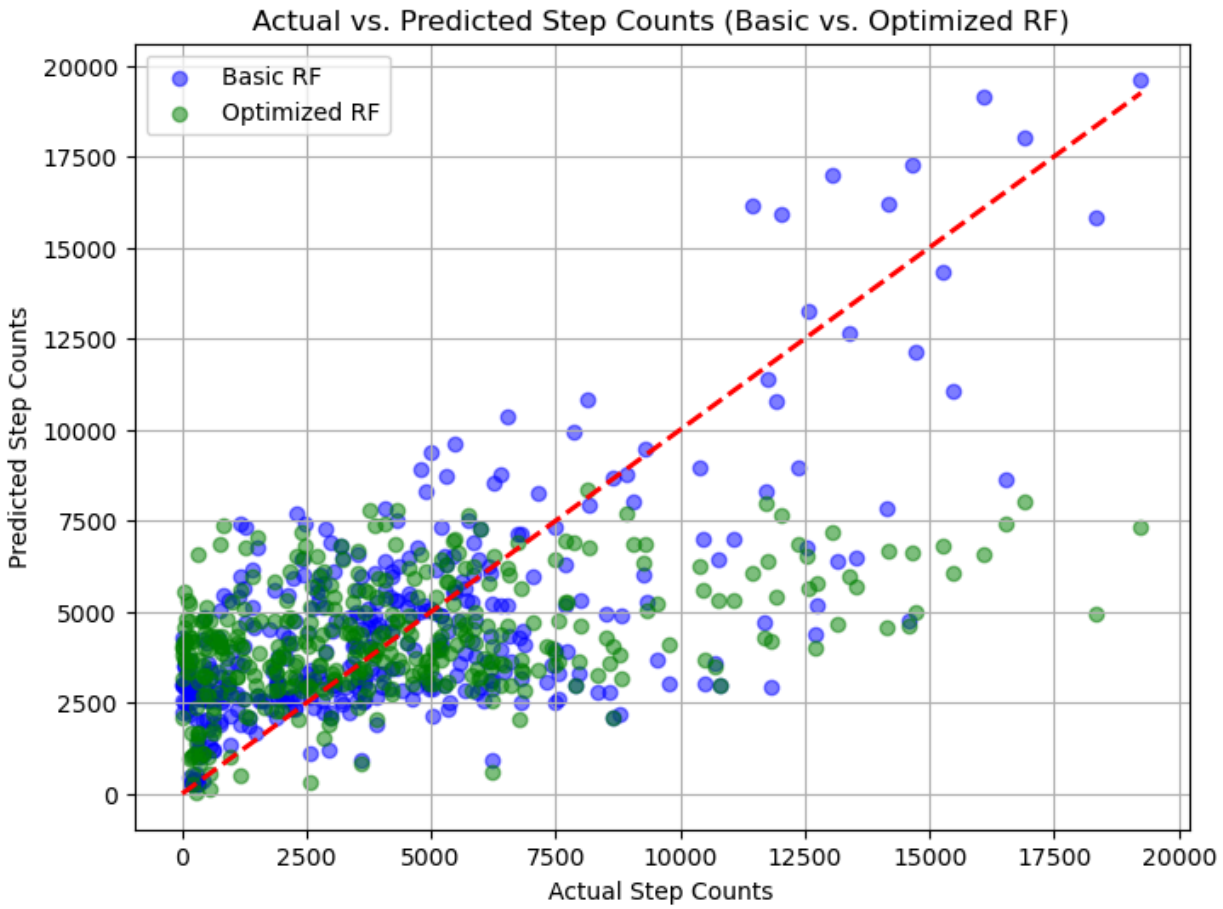
- Heteroscedasticity: The plot shows that the residuals are not uniformly spread around the horizontal line at zero. There is a funnel shape, where the spread of residuals increases as the predicted values get larger. This indicates heteroscedasticity, meaning the variance of the residuals is not constant across the range of predicted values.
- There is a cluster of residuals around zero for lower predicted values (around 0 to 5,000), indicating that the model performs reasonably well for these cases.

Comparison:

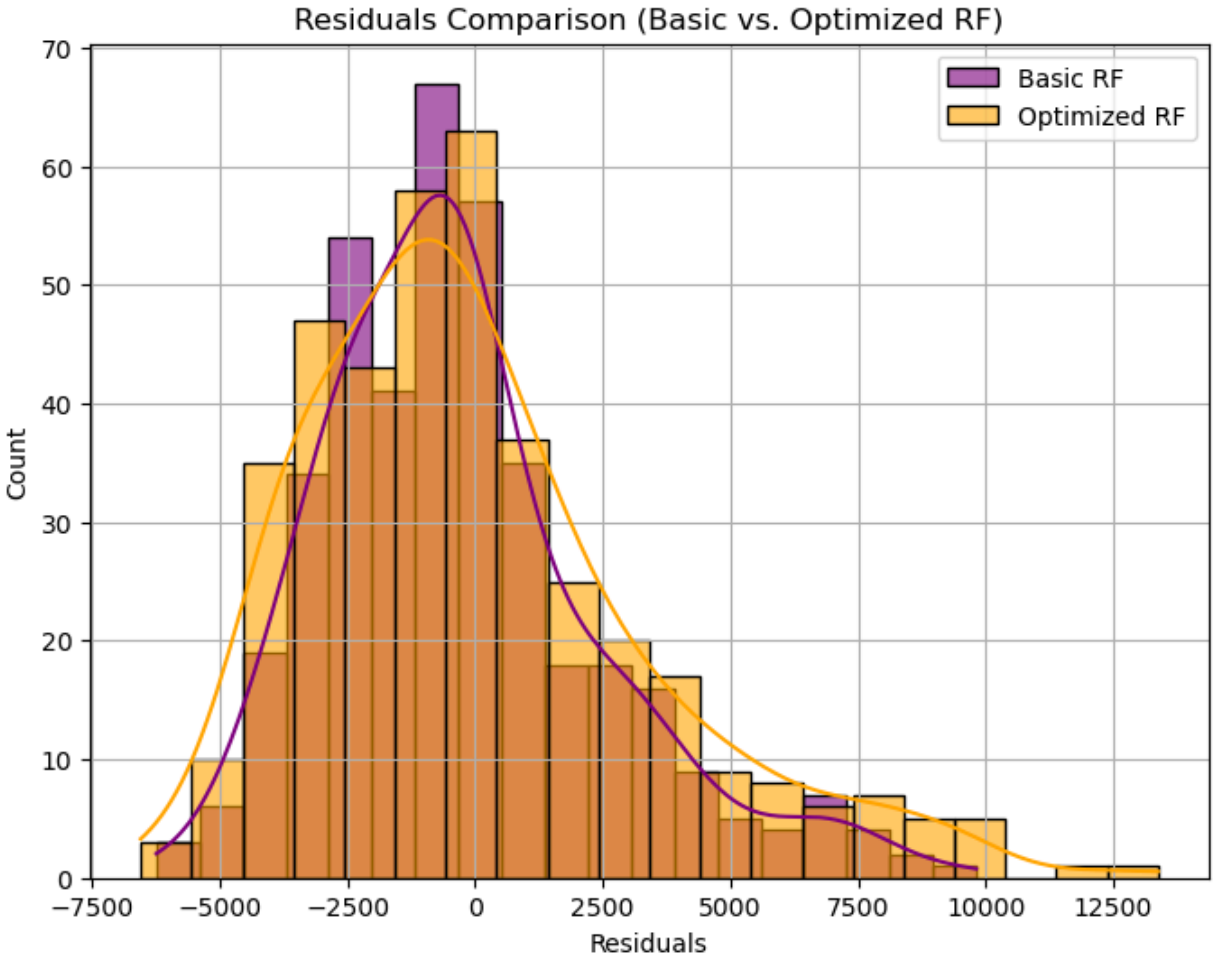
I first tried overlaying the results for both the basic and optimized Random Forest models on the same plot, for both the scatter plot (Actual vs. Predicted) and the histogram (Residuals). However, the visualizations are too muddling, making them diffcult to interpret and gain insights from.

In [40]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Combined Scatter Plot: Actual vs. Predicted for both Basic and Optimized RF
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, alpha=0.5, color='blue', label='Basic RF')
plt.scatter(y_test, y_pred_optimized, alpha=0.5, color='green', label='Optimiz
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw
plt.title('Actual vs. Predicted Step Counts (Basic vs. Optimized RF)')
plt.xlabel('Actual Step Counts')
plt.ylabel('Predicted Step Counts')
```

```
plt.legend()
plt.grid(True)
plt.show()
```



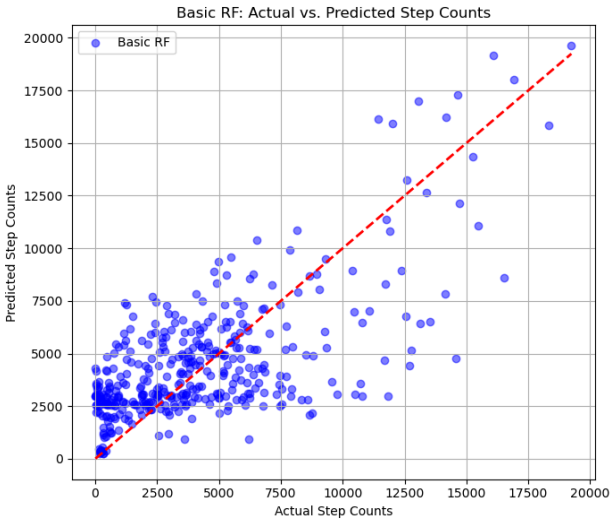
In [41]:
```
# Combined Histogram of Residuals for both Basic and Optimized RF
plt.figure(figsize=(8, 6))
sns.histplot(residuals, kde=True, color='purple', label='Basic RF', alpha=0.6)
sns.histplot(residuals_2, kde=True, color='orange', label='Optimized RF', alpha
plt.title('Residuals Comparison (Basic vs. Optimized RF)')
plt.xlabel('Residuals')
plt.legend()
plt.grid(True)
plt.show()
```

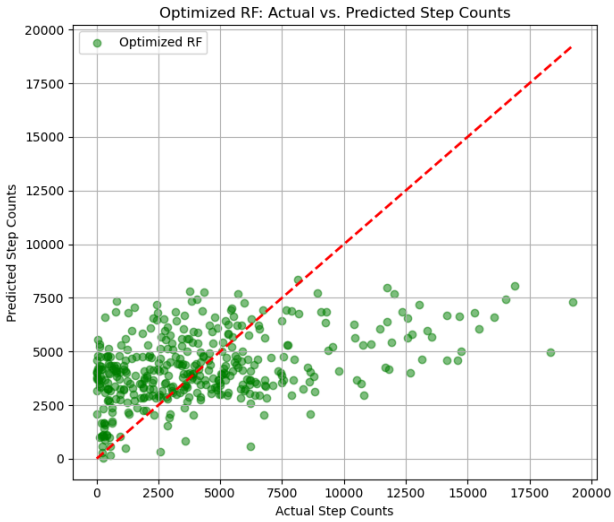


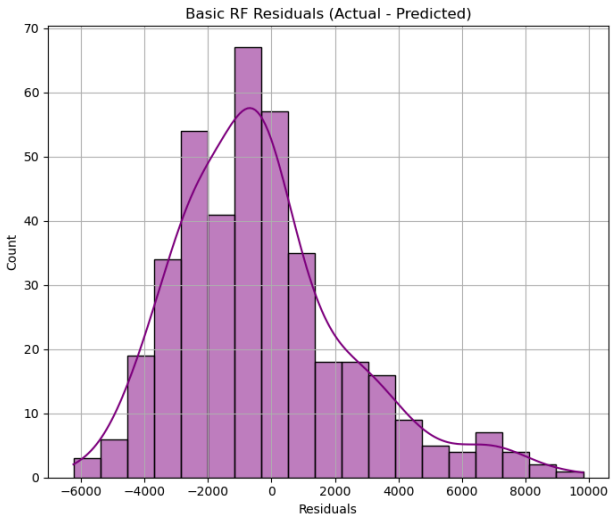
Thus, I decided to plot both side-by-side for comparisons, instead of overlapping them into one plot.

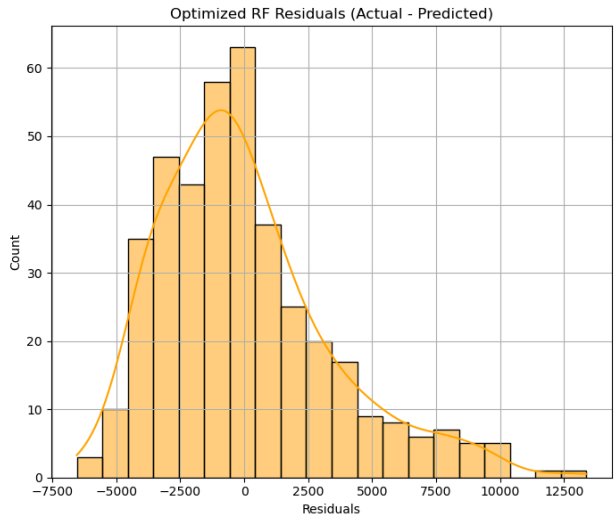
In [39]:
```
# Scatter Plot: Actual vs. Predicted for Basic and Optimized Random Forest
plt.figure(figsize=(14, 6))

# Basic Random Forest
plt.subplot(1, 2, 1)
plt.scatter(y_test, y_pred, alpha=0.5, color='blue', label='Basic RF')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=
plt.title('Basic RF: Actual vs. Predicted Step Counts')
plt.xlabel('Actual Step Counts')
plt.ylabel('Predicted Step Counts')
plt.legend()
plt.grid(True)

# Optimized Random Forest
plt.subplot(1, 2, 2)
plt.scatter(y_test, y_pred_optimized, alpha=0.5, color='green', label='Optimiz
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=
plt.title('Optimized RF: Actual vs. Predicted Step Counts')
plt.xlabel('Actual Step Counts')
plt.ylabel('Predicted Step Counts')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
# Histogram of Residuals for Basic and Optimized Random Forest
plt.figure(figsize=(14, 6))

# Basic Random Forest Residuals
plt.subplot(1, 2, 1)
residuals = y_test - y_pred
sns.histplot(residuals, kde=True, color='purple')
plt.title('Basic RF Residuals (Actual - Predicted)')
plt.xlabel('Residuals')
plt.grid(True)

# Optimized Random Forest Residuals
plt.subplot(1, 2, 2)
residuals_2 = y_test - y_pred_optimized
sns.histplot(residuals_2, kde=True, color='orange')
plt.title('Optimized RF Residuals (Actual - Predicted)')
plt.xlabel('Residuals')
plt.grid(True)

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









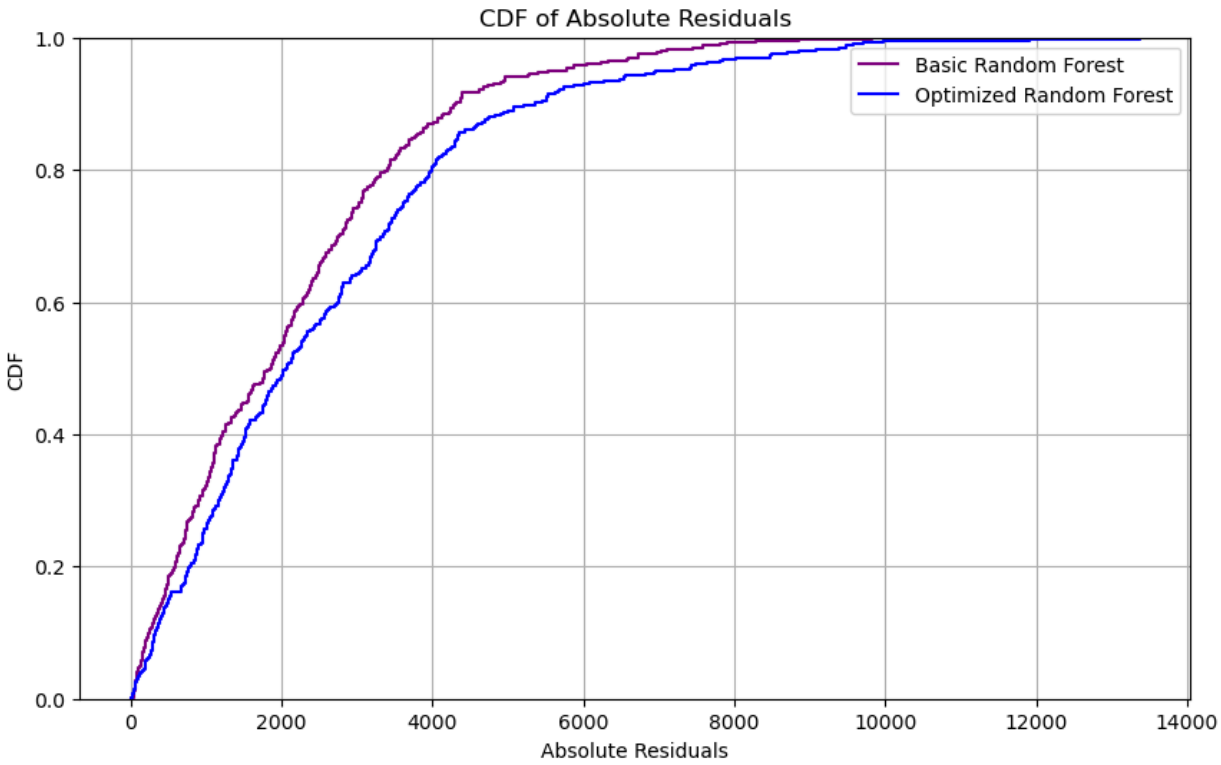
Plot Intepretation

- The optimized model seems to have more points clustered closer to the diagonal line (0 to 7500), especially in the lower range of step counts, suggesting better prediction

accuracy in that range.

- The optimized model does perform slightly better by clustering predictions closer to the ideal diagonal line with a more balanced distribution.
- However, both models struggle with predictions in the higher range (above 7,500 steps), where there is still noticeable scatter around the diagonal line.

In [44]:
```
plt.figure(figsize=(10, 6))

# Plot CDF of absolute residuals for the basic model
sns.ecdfplot(np.abs(residuals), label='Basic Random Forest', color='purple')

# Plot CDF of absolute residuals for the optimized model
sns.ecdfplot(np.abs(residuals_2), label='Optimized Random Forest', color='blue

plt.title('CDF of Absolute Residuals')
plt.xlabel('Absolute Residuals')
plt.ylabel('CDF')
plt.legend()
plt.grid(True)
plt.show()
```



The Basic Random Forest's CDF (purple) is consistently higher than the optimized Random Forest's CDF (blue) for most of the range. This means that, at any given residual value, a higher proportion of the basic model's errors are smaller than those of the optimized model. This suggests that the basic model has more residuals concentrated near zero (indicating smaller errors on average) compared to the optimized model.

Despite the optimized model having a higher R^2 value (0.453 compared to 0.395 for the basic model), it does not necessarily mean that the optimized model has smaller residuals

across the board. R^2 measures the proportion of variance in the dependent variable explained by the model, while the CDF of residuals focuses on the distribution of errors.

Thus, the CDF suggests that, **in terms of minimizing residuals, the basic model may actually be performing better than the optimized model. However, the optimized model's higher R^2 indicates that it is capturing more of the overall variance in the data.**

Neural Networks:

In the previous episode of classification, our friend Neural Networks has made her entrance (forward pass, backpropagation, weight updates, etc.)

The backpropagation process and the mathematical principles of neural networks remain fundamentally the same for both classification and regression tasks. In this episode of regression, she has made a slight chance - the choice of the loss function.

Loss Function:

The loss function used here is Mean Squared Error (MSE)

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where N is the total number of samples, y_i is the true value, and $\hat{y}_i$ is the predicted value.

Activation Function in the Output Layer:

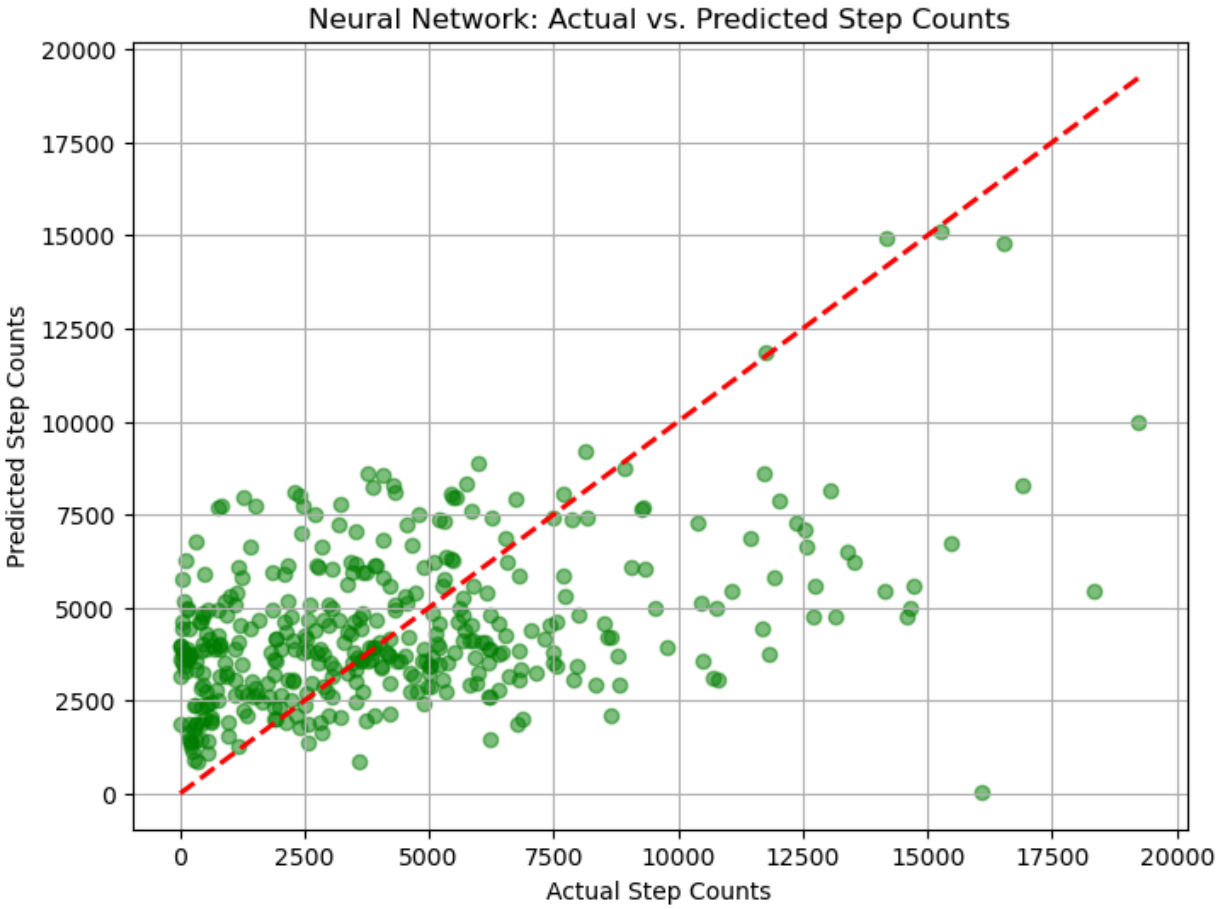
In regression tasks, where the goal is to predict a continuous number (like step counts in this case), we don't apply an activation function at the output layer of a neural network. Activation functions are typically used to introduce non-linearity, but in this case, we want the output to be a real number without any transformation. So, the output neuron directly gives a continuous value, which is our prediction.

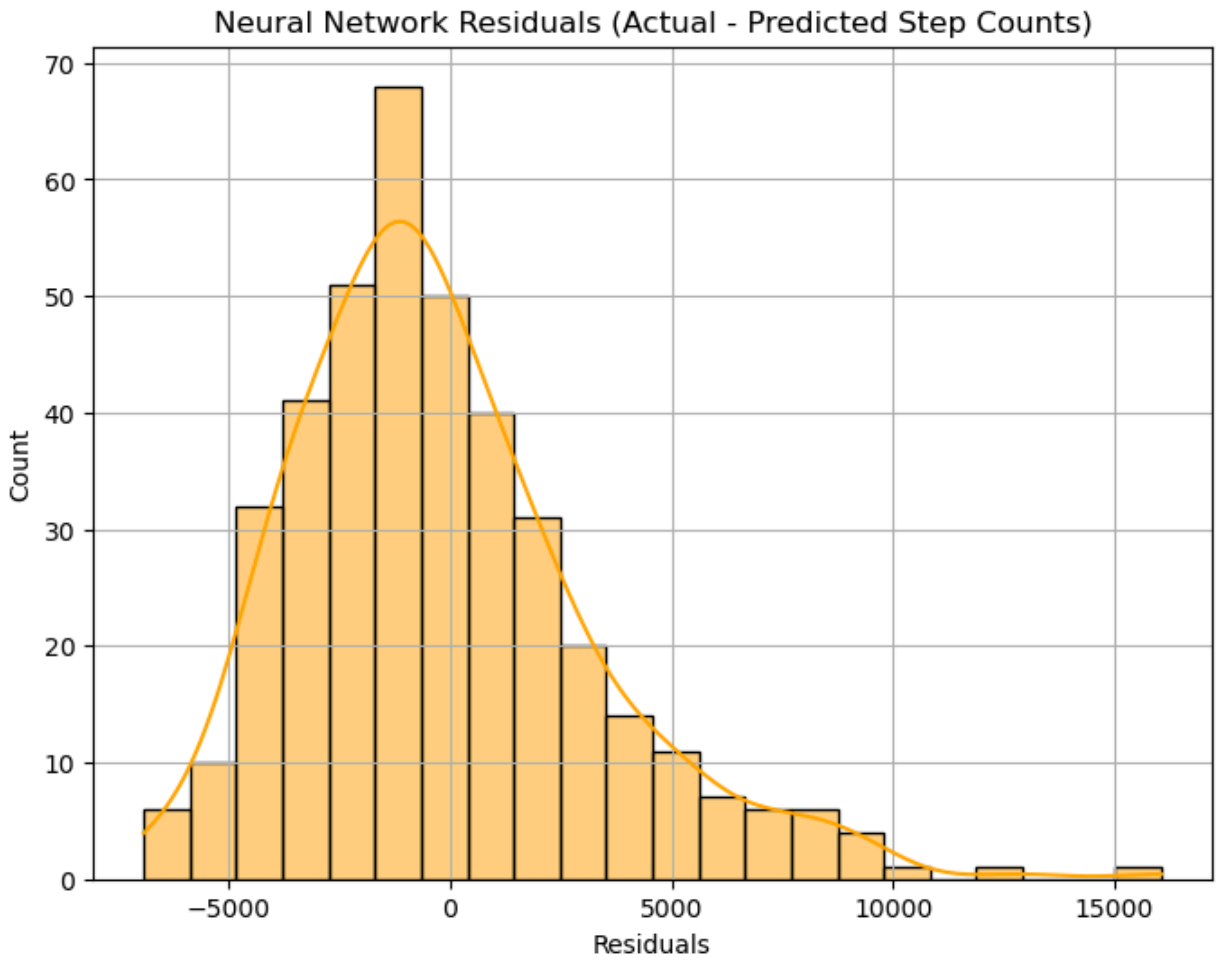
In contrast, for classification tasks, activation functions like the sigmoid or softmax are used in the output layer to convert the output into probabilities. But in regression, using a linear activation (or no activation) allows the model to predict any real number, which is essential to predict step counts.

In [18]:
```
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import matplotlib.pyplot as plt
import seaborn as sns

mlp_regressor = MLPRegressor(
    hidden_layer_sizes=(64, 32),
    max_iter=1000,
    random_state=42,
```

```
    solver='adam',
    learning_rate_init=0.001,
    early_stopping=True,
    validation_fraction=0.1,
    n_iter_no_change=10
)

# Train the model on the scaled training data
mlp_regressor.fit(X_train_scaled, y_train)

# Make predictions on the test set
y_pred_mlp = mlp_regressor.predict(X_test_scaled)

# Evaluate the model
mse_mlp = mean_squared_error(y_test, y_pred_mlp)
mae_mlp = mean_absolute_error(y_test, y_pred_mlp)
r2_mlp = r2_score(y_test, y_pred_mlp)

print(f"Neural Network Regression Results:")
print(f"Mean Squared Error (MSE): {mse_mlp}")
print(f"Mean Absolute Error (MAE): {mae_mlp}")
print(f"R-squared (R2): {r2_mlp}")

# Create a scatter plot of actual vs. predicted step counts for Neural Network
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred_mlp, alpha=0.5, color='green')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=
plt.title('Neural Network: Actual vs. Predicted Step Counts')
plt.xlabel('Actual Step Counts')
plt.ylabel('Predicted Step Counts')
plt.grid(True)
plt.show()

# Residual plot for Neural Network
residuals_mlp = y_test - y_pred_mlp
plt.figure(figsize=(8, 6))
sns.histplot(residuals_mlp, kde=True, color='orange')
plt.title('Neural Network Residuals (Actual - Predicted Step Counts)')
plt.xlabel('Residuals')
plt.grid(True)
plt.show()
```

```
Neural Network Regression Results:
Mean Squared Error (MSE): 11397405.630206497
Mean Absolute Error (MAE): 2596.478604300608
R-squared (R2): 0.1774835711784205
```


Neural Network: Actual vs. Predicted Step Counts
Predicted Step Counts
Actual Step Counts
0
2500
5000
7500
10000
12500
15000
17500
20000


Neural Network Residuals (Actual - Predicted Step Counts)
Count
Residuals
0
10
20
30
40
50
60
70
−5000
0
5000
10000
15000

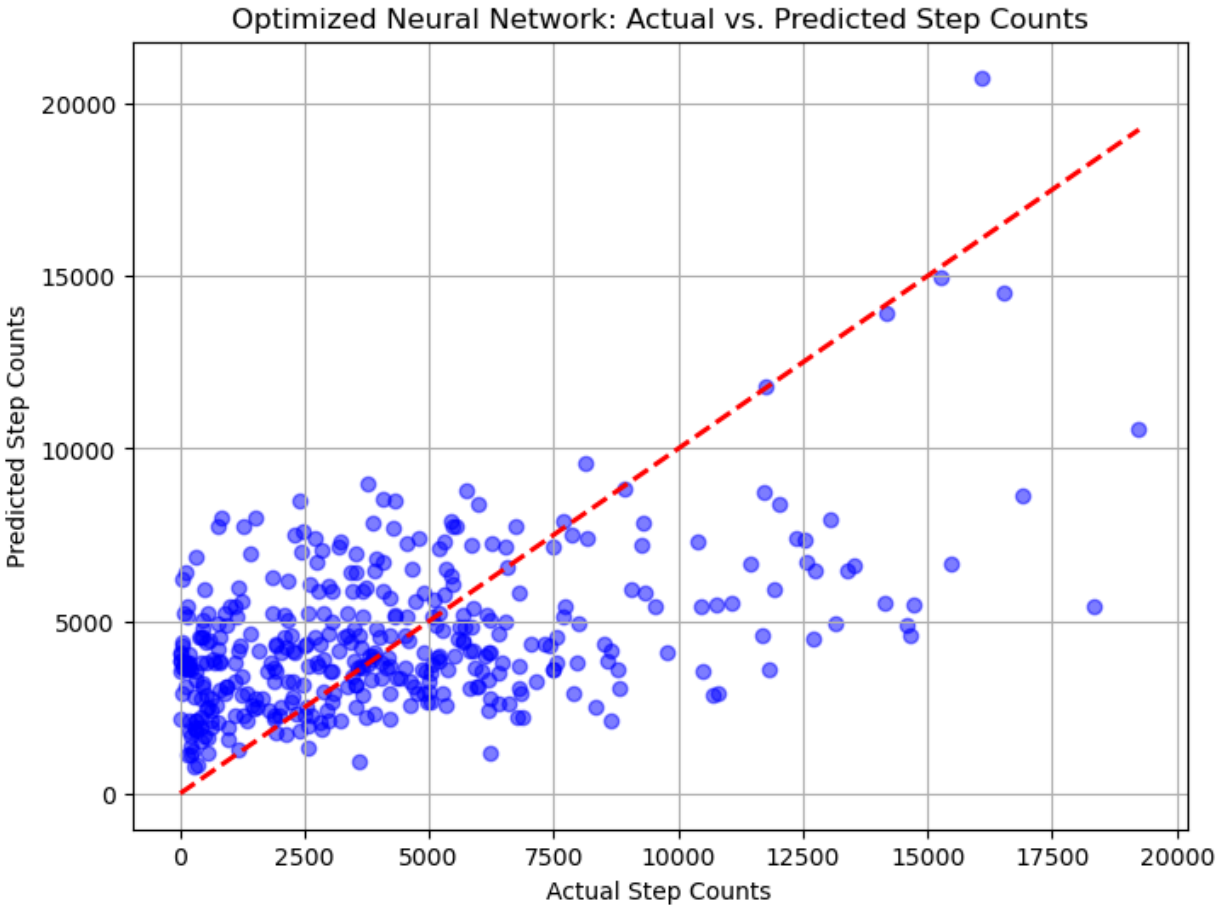
MLP with hyperparameter tuning and cross validation

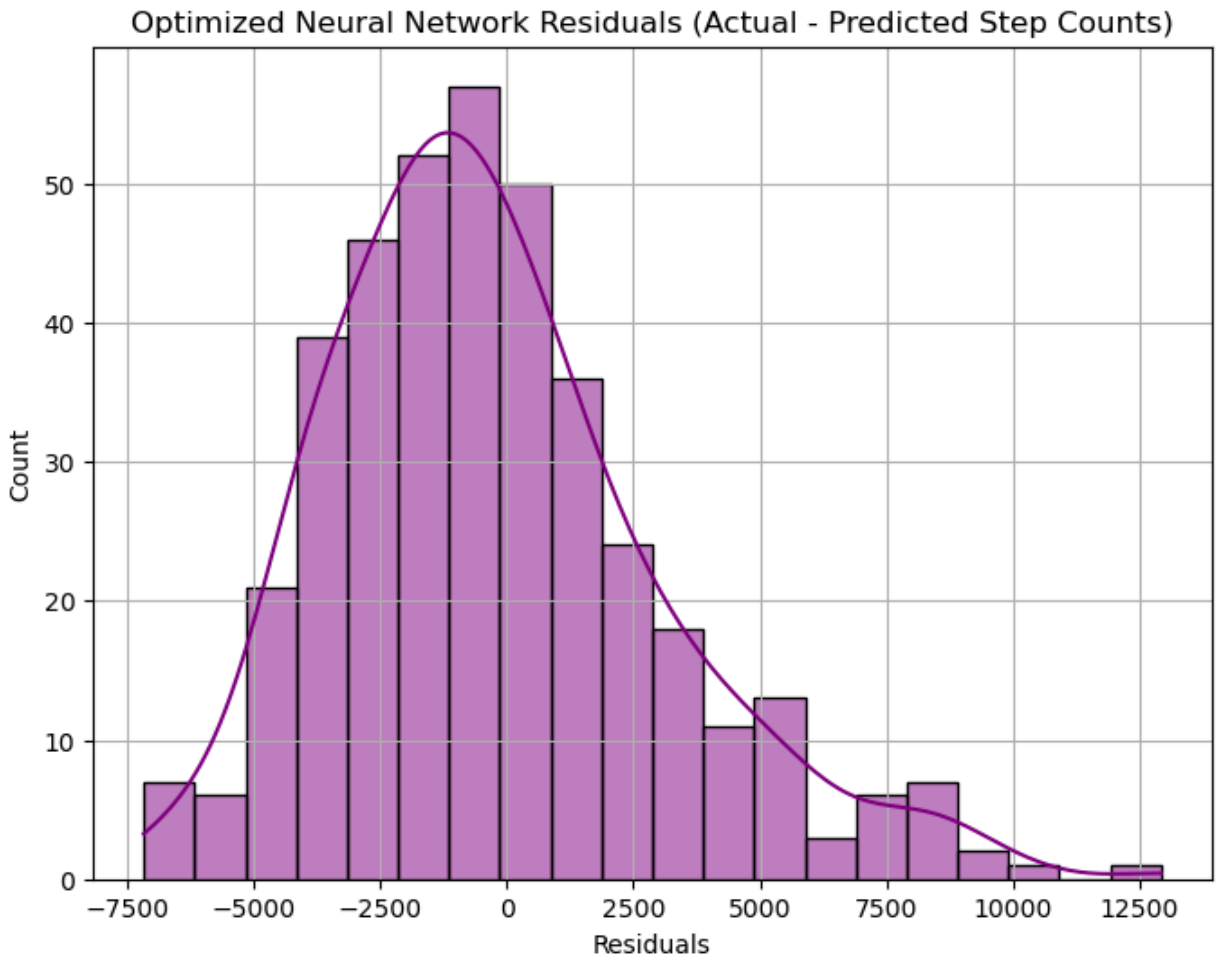
In [15]:
```
# Define the hyperparameter grid
param_grid = {
    'hidden_layer_sizes': [(64, 32), (128, 64)],
    'activation': ['tanh', 'relu'],  # Consider using 'tanh' to avoid overflow
    'solver': ['adam'],  # Use 'adam' solver for better stability
    'alpha': [0.01, 0.0],  # Higher regularization to control weight size
    'learning_rate': ['constant', 'adaptive'],
    'learning_rate_init': [0.0001, 0.001],  # Lower learning rate
    'max_iter': [5000, 10000]  # Increase the maximum iterations
}

# Initialize the MLP Regressor with early stopping and suppress verbose output
mlp = MLPRegressor(
    random_state=42,
    early_stopping=True,
    validation_fraction=0.1,
    n_iter_no_change=5,
    verbose=False  # Suppress verbose output
)

# Set up the GridSearchCV
grid_search = GridSearchCV(
    estimator=mlp,
    param_grid=param_grid,
    cv=5,  # 5-fold cross-validation
    scoring='neg_mean_squared_error',
    n_jobs=-1,
    verbose=0
)

# Fit GridSearchCV on the scaled training data
grid_search.fit(X_train_scaled, y_train)

# Evaluate the best model
best_mlp = grid_search.best_estimator_
y_pred_best_mlp = best_mlp.predict(X_test_scaled)

mse_mlp = mean_squared_error(y_test, y_pred_best_mlp)
mae_mlp = mean_absolute_error(y_test, y_pred_best_mlp)
r2_mlp = r2_score(y_test, y_pred_best_mlp)

print(f"Optimized Neural Network Regression Results:")
print(f"Mean Squared Error (MSE): {mse_mlp}")
print(f"Mean Absolute Error (MAE): {mae_mlp}")
print(f"R-squared (R2): {r2_mlp}")
print(f"Best Hyperparameters: {grid_search.best_params_}")

# Plot the optimized model's predictions
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred_best_mlp, alpha=0.5, color='blue')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=
plt.title('Optimized Neural Network: Actual vs. Predicted Step Counts')
plt.xlabel('Actual Step Counts')
plt.ylabel('Predicted Step Counts')
plt.grid(True)
plt.show()
```

```
# Residual plot for the optimized Neural Network
residuals_mlp_optimized = y_test - y_pred_best_mlp
plt.figure(figsize=(8, 6))
sns.histplot(residuals_mlp_optimized, kde=True, color='purple')
plt.title('Optimized Neural Network Residuals (Actual - Predicted Step Counts)
plt.xlabel('Residuals')
plt.grid(True)
plt.show()
```

```
/Users/zeldudu/anaconda3/lib/python3.11/site-packages/sklearn/neural_network/_
multilayer_perceptron.py:690: ConvergenceWarning: Stochastic Optimizer: Maximu
m iterations (5000) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/zeldudu/anaconda3/lib/python3.11/site-packages/sklearn/neural_network/_
multilayer_perceptron.py:690: ConvergenceWarning: Stochastic Optimizer: Maximu
m iterations (5000) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/zeldudu/anaconda3/lib/python3.11/site-packages/sklearn/neural_network/_
multilayer_perceptron.py:690: ConvergenceWarning: Stochastic Optimizer: Maximu
m iterations (5000) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/zeldudu/anaconda3/lib/python3.11/site-packages/sklearn/neural_network/_
multilayer_perceptron.py:690: ConvergenceWarning: Stochastic Optimizer: Maximu
m iterations (5000) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
Optimized Neural Network Regression Results:
Mean Squared Error (MSE): 10843998.540226206
Mean Absolute Error (MAE): 2576.3739512710063
R-squared (R2): 0.2174212936833344
Best Hyperparameters: {'activation': 'relu', 'alpha': 0.001, 'hidden_layer_siz
es': (128, 64), 'learning_rate': 'constant', 'learning_rate_init': 0.001, 'max
_iter': 5000, 'solver': 'adam'}
```


Optimized Neural Network: Actual vs. Predicted Step Counts
Predicted Step Counts
Actual Step Counts
0
5000
10000
15000
20000
0
2500
5000
7500
10000
12500
15000
17500
20000


Optimized Neural Network Residuals (Actual - Predicted Step Counts)
Count
Residuals
0
10
20
30
40
50
−7500
−5000
−2500
0
2500
5000
7500
10000
12500

In [46]:
```
results_ = pd.DataFrame({
    'Model': ['Basic RF', 'Optimized RF', 'Basic MLP', 'Optimized MLP'],
    'MSE': [8373162, 7570066, 11397405, 10843998],
    'MAE': [2200, 2121, 2596, 2576],
    'R2': [0.395, 0.454, 0.177, 0.217],
})

# Sort the DataFrame by R2 in descending order
results_ = results_.sort_values(by='R2', ascending=False).reset_index(drop=Tru

results_.head()
```

Out[46]:

	Model	MSE	MAE	R2
0	Optimized RF	7570066	2121	0.454
1	Basic RF	8373162	2200	0.395
2	Optimized MLP	10843998	2576	0.217
3	Basic MLP	11397405	2596	0.177

Result Interpretation:

The **optimized RF model** performs the best among the four models, with the lowest MSE and MAE, indicating that it makes smaller errors on average compared to the others. An R^2 of 0.454 means that approximately 45.4% of the variance in the target variable (step counts) is explained by the model, suggesting a moderate level of predictive power.

The optimized MLP has a higher MSE and MAE compared to both RF models, indicating less accurate predictions. The R^2 score of 0.217 suggests that only 21.7% of the variance in the step counts is explained by the model, implying weaker predictive power despite optimization, but still better than the Basic MLP.

Overall, Neural networks (MLP) struggled to capture the patterns in the data effectively, possibly due to issues like numerical instability, inadequate feature engineering, or the model being less suited to this particular problem.

Random Forest likely works better than MLP here because of its structure, which makes it more robust to noisy data and less sensitive to feature scaling (exactly what this dataset is). It combines multiple decision trees to capture complex patterns and interactions in the data, handling irregular feature relationships (like health data) more effectively. In contrast, MLPs require more data, proper scaling, and careful tuning to perform well, which may not have been achieved with the given dataset.

5/ Executive Summary:

First of all, I have no idea whether this is an appropriate length for this assignment :) but my thinking process was 1) I'm curious about both classification and regression tasks so I wanted to explore both 2) For each task, I wanted to compare two models against each

other with multiple metrics 3) For each model, I want to experiment how they progress (i.e. is a non-tuning random forest model better than an optimized MLP?).

Thus, here's the summary of all of the work above:

1/ Data Preparation (load data from iOS Health and Flo)

2/ Exploratory Data Analysis (feature engineering, data visualization, descriptive stats, strategies for dealing with missing data were established to mitigate potential biases.)

3/ Classification (whether I am on my period)

- Decision Trees: definition, how it works
    - Basic model
    - Model with balanced data
    - Model with hyperparameter tuning
    - Confusion matrix, accuracy, recall, precision, F1 score, ROC-AUC
- Neural Networks: definition, backpropagation, layers
    - Basic model
    - Model with balanced data
    - Model with hyperparameter tuning
    - Confusion matrix, accuracy, recall, precision, F1 score, ROC-AUC

4/ Regression (predict my daily steps)

- Random Forest Regression: definition, how it develops from decision trees
    - Basic model
    - Model with hyperparameter tuning + cross validation
    - MAE, MSE, R^2, graphs (histogram, scatter)
- Neural Networks: definition, backpropagation, layers
    - Basic model
    - Model with hyperparameter tuning + cross validation
    - MAE, MSE, R^2, graphs (histogram, scatter)

Key insights

Classification:

- The tuned decision tree model outperformed the basic model in terms of precision and recall, while the neural network showed significant improvements in F1 score and ROC-AUC after optimization. This suggests that more complex models benefit more from hyperparameter tuning in this context.
- Tuned decision tree with balanced data is currently the best model to predict whether I am on my period on a particular day (Recall=0.84, F1 score=0.31). But let's be honest, a

recall of 0.84 still means it misses quite a bit, and the F1 score isn't exactly a bragging point :(

Regression:

- While the basic random forest achieved decent results, tuning improved its performance significantly. The neural network, however, struggled with numerical stability, indicating a need for further adjustments to learning rates and activation functions.
- The best model to predict my steps was the Optimized Random Forest (R^2=0.454), meaning that ~45% of my steps can be explained by the model.

Potential Reasons Why My Models Underperform

Overall, my models perform quite poorly. They could use a lot more than just tweaking hyperparameters; I may need to dive deeper into the data itself. Some ideas why they did not do well:

Data:

- My Health data was noisy and inconsistent, making it hard to learn meaningful patterns. Moreover, handling missing data with basic methods could have reduced variability, and class imbalance could lead to biased predictions.
- The target variable characteristics may make it hard to predict based on our current data. Daily step counts can be influenced by various unrecorded factors (unusual physical activity, mood). For period prediction, since physiological indicators or hormone-related data were not included, it made sense that the classification task would inherently be more challenging.

Model Complexity and Hyperparameters:

- Although I did some hyperparameter tuning, techniques like grid search may not have explored the parameter space exhaustively enough, particularly for neural networks where the search space is vast. MLP also experienced many instances of failure to converge, suggesting that there need to be adjustments in how I tune my parameters next time.

Data Granularity:

- For both predicting steps and period tracking, there may be temporal patterns (e.g., certain days of the week or monthly cycles) that are not being captured if the models were trained without considering temporal relationships. Time-series models or lagged features could help improve performance.

How I want to further develop this project

Improve the current models:

- Automated Hyperparameter Tuning: Techniques like Bayesian optimization could be used to refine models more efficiently. I want to explore tools like Optuna or even do something custom with genetic algorithms.
- Besides hyperparameter tuning, I want to investigate my data more thoroughly (deal with missing data more creatively, perhaps using predictive imputation or something more dynamic than just filling with the median). I think a deeper dive into feature engineering could also uncover hidden signals.

Explore other models:

- Time Series Analysis: Since the data spans multiple years, exploring time-series models like ARIMA or LSTM could be beneficial to understand trends over time. I could also incorporate seasonality and moving averages to get a better sense of the patterns in my data.
- Predict Differences Across Period Days: It could be interesting to compare or predict my step count across different days of my period (e.g., day 1 vs. day 5). A day-wise classification or regression might provide more actionable insights.
- Hybrid Approaches: Combine the strengths of different models, like using a decision tree to preprocess and select features before passing them to a neural network for the final prediction.

6/ References:

Game of Bits. (n.d.). How to deal with imbalanced data in classification. Medium. Retrieved October 19, 2024, from https://medium.com/game-of-bits/how-to-deal-with-imbalanced-data-in-classification-bd03cfc66066

Abhinav, R. (n.d.). Improving class imbalance with class weights in machine learning. Medium. Retrieved October 19, 2024, from https://medium.com/@ravi.abhinav4/improving-class-imbalance-with-class-weights-in-machine-learning-af072fdd4aa4#:~:text=Using%20Class%20Weights%20to%20Address%20Class%20Imbalance
Class%20weights%20offer&text=The%20idea%20is%20to%20assign,make%20better%20pre

Ghosh, R. (2023, February 27). Simple guide to hyperparameter tuning in neural networks. Towards Data Science. https://towardsdatascience.com/simple-guide-to-hyperparameter-tuning-in-neural-networks-3fe03dad8594

GeeksforGeeks. (2022, September 19). Difference between random forest and decision tree. https://www.geeksforgeeks.org/difference-between-random-forest-and-decision-tree/

AI Statement: In this assignment, I used ChatGPT to assist with specific tasks while I led the modeling process. I used it for debugging issues related to neural network training, understanding convergence warnings, potential graphs for EDA, strategies to improve model performance, such as adjusting hyperparameters, regularization, and learning rates. I also used ChatGPT to clarify certain machine learning concepts and get insights on

interpreting model evaluation metrics (such as which graphs should be used). While ChatGPT provided suggestions, I tuned the models and wrote/ evaluated the analysis on my own.