Computational Narrative Reflection: Rule et al.'s Ten Simple Rules

This notebook adheres to key principles (Rule 1, Rule 3, Rule 5 and Rule 9) from Rule et al.'s *Ten Simple Rules for Writing and Sharing Computational Analyses*. The objective of this notebook was to build visualizations and present them as a cohesive, understandable, and reproducible story.

📌 Rule 1: Tell a Story for an audience

This notebook tells a story through its visualizations, starting with data cleaning and preparation and going through exploratory questions (e.g., "What does cadence tell us about activity type?") to advanced analyses like GPS Animation Path to create the visualizations in the notebook. Each visualization includes commentary that clarifies its significance, aiding the reader in deriving meaningful insights from the data. The structured approach with labeled sections and interactive elements guides the reader in exploring athletic performance and movement efficiency, leading to a coherent understanding of the data.

📌 Rule 2: Document the Process, not just the results

The notebook outlines the process of analyzing the data and details several key steps, including loading the Strava.csv file, cleaning the data, conducting exploratory analysis, and generating visualizations. Each code block is accompanied by a Markdown cell that explains its purpose. Important data cleaning steps, such as removing null GPS points and converting timestamps, are clearly presented and justified through Markdown cells to help the reader better understand the data. In addition, the notebook incorporates visual checks and printed outputs to ensure that the data analysis and transformations are valid.

📌 Rule 5: Record Dependencies

All the code in this notebook can be run end-to-end using a single dataset and standard Python packages like `pandas`, `altair`, `plotly.express`, and `ipywidgets`. The interactivity in the notebook is implemented in a way that's reproducible across any Jupyter environment. To note, here are no proprietary tools or hidden dependencies.

📌 Rule 9: Design your notebooks to be read, run, and explored

The layout of the notebook itself supports reproducibility by separating the Code from the Markdown cells. In addition, the parameters are adjustable and data transformations are all performed step-by-step. This makes it easy for readers to not only reproduce the exact output in this notebook but also modify the analysis for new questions or conduct additional exploratory analysis.

1. Import Libraries

The libraries listed below will be utilized for data cleaning and visualizations.

In [4]:
```
import altair as alt

import numpy as np
import pandas as pd
from scipy.stats import linregress
```

2. Dataset Cleaning and Preprocessing

2.1 Overview

The dataset used in this analysis comes from the `Strava.csv` Dataset.

- This dataset contains GPS and physiological data such as altitude, cadence, heart rate, power, and timestamps.
- Each row represents a time-stamped reading from an activity.

2.3 Data Cleaning Overview

Before building visualizations, we must prepare the dataset:

- Convert the `timestamp` column from object to `datetime` format.
- Drop rows with missing GPS or biomechanical values for certain charts.
- Suppress FutureWarning messages in Python, preventing them from being displayed. This ensures that the dashboard doesn't become cluttered.
- Suppress SettingWithCopyWarning messages in Python, preventing them from being displayed. This ensures that the dashboard doesn't become cluttered.

This line suppresses FutureWarning messages in Python, preventing them from being displayed. This ensures that the dashboard doesn't become cluttered.

In [7]:
```
import warnings

warnings.simplefilter(action='ignore', category=FutureWarning)
```

This line suppresses SettingWithCopyWarning messages in Python, preventing them from being displayed. This ensures that the dashboard doesn't become cluttered.

In [9]:
```
pd.options.mode.chained_assignment = None
```

3. Data Cleaning Steps

Load the Strava.csv Dataset

In [12]:
```
df = pd.read_csv('strava.csv')
```

Display the first few rows of the Strava.csv Dataset.

In [14]:
```
df.head()
```

Out[14]:

	Air Power	Cadence	Form Power	Ground Time	Leg Spring Stiffness	Power	Vertical Oscillation	altitude	cadence	datafile	...	enhanced_speed	fractional_cadence	heart_rate	position_lat
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	activities/2675855419.fit.gz	...	0.000	0.0	68.0	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	activities/2675855419.fit.gz	...	0.000	0.0	68.0	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	54.0	activities/2675855419.fit.gz	...	1.316	0.0	71.0	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	3747.0	77.0	activities/2675855419.fit.gz	...	1.866	0.0	77.0	504432050.0
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	3798.0	77.0	activities/2675855419.fit.gz	...	1.894	0.0	80.0	504432492.0

5 rows × 22 columns

Let's check the data types in the Dataset.

In [16]:
```
df.dtypes
```

Out[16]:
```
Air Power               float64
Cadence                 float64
Form Power              float64
Ground Time             float64
Leg Spring Stiffness    float64
Power                   float64
Vertical Oscillation    float64
altitude                float64
cadence                 float64
datafile                 object
distance                float64
enhanced_altitude       float64
enhanced_speed          float64
fractional_cadence      float64
heart_rate              float64
position_lat            float64
position_long           float64
speed                   float64
timestamp                object
unknown_87              float64
unknown_88              float64
unknown_90              float64
dtype: object
```

Let's find how many rows of data there are.

In [18]:
```
print(f"There are {len(df)} rows of data.")
```

```
There are 40649 rows of data.
```

The timestamp is stored as an object. We need to convert it to a date time value for analysis.

In [20]:
```
df['timestamp'] = pd.to_datetime(df['timestamp'], errors='coerce')
```

In [21]:
```
print(f" The earliest date is: {df['timestamp'].min()}.")
print(f" The latest date is: {df['timestamp'].max()}.")
```

```
The earliest date is: 2019-07-08 21:04:03.
The latest date is: 2019-10-03 23:05:05.
```

4. Creating a Correlation Heatmap

We will create a correlation table and a heatmap to identify which fields are best for generating charts.

In [24]:
```
unknown_corr = df.drop(columns=['datafile', 'timestamp']).corr()
unknown_corr
```

Out[24]:

	Air Power	Cadence	Form Power	Ground Time	Leg Spring Stiffness	Power	Vertical Oscillation	altitude	cadence	distance	enhanced_altitude	enhanced_speed	fractional_ca
Air Power	1.000000	0.146446	0.150502	-0.101692	0.055933	0.308071	0.137140	NaN	0.146685	0.007198	0.051763	0.338407	
Cadence	0.146446	1.000000	0.831033	0.276398	0.782156	0.796973	0.668764	NaN	0.933191	0.048127	0.175427	0.760382	
Form Power	0.150502	0.831033	1.000000	0.023542	0.802700	0.768840	0.846785	NaN	0.774036	-0.068124	0.235795	0.752237	
Ground Time	-0.101692	0.276398	0.023542	1.000000	-0.089329	0.008799	-0.133232	NaN	0.260521	-0.017146	-0.113758	-0.014013	
Leg Spring Stiffness	0.055933	0.782156	0.802700	-0.089329	1.000000	0.642273	0.753608	NaN	0.732922	-0.032569	0.063097	0.505980	
Power	0.308071	0.796973	0.768840	0.008799	0.642273	1.000000	0.637812	NaN	0.759318	-0.068990	0.328629	0.812346	
Vertical Oscillation	0.137140	0.668764	0.846785	-0.133232	0.753608	0.637812	1.000000	NaN	0.604854	-0.139752	0.189731	0.657616	
altitude	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1.000000	-0.083869	-0.036738	1.000000	-0.036491	-0.0
cadence	0.146685	0.933191	0.774036	0.260521	0.732922	0.759318	0.604854	-0.083869	1.000000	-0.116760	0.095441	-0.097341	0.0
distance	0.007198	0.048127	-0.068124	-0.017146	-0.032569	-0.068990	-0.139752	-0.036738	-0.116760	1.000000	-0.193163	0.513813	-0.0
enhanced_altitude	0.051763	0.175427	0.235795	-0.113758	0.063097	0.328629	0.189731	1.000000	0.095441	-0.193163	1.000000	-0.181469	-0.0
enhanced_speed	0.338407	0.760382	0.752237	-0.014013	0.505980	0.812346	0.657616	-0.036491	-0.097341	0.513813	-0.181469	1.000000	-0.2
fractional_cadence	NaN	NaN	NaN	NaN	NaN	NaN	NaN	-0.007626	0.048313	-0.091675	-0.038209	-0.218453	1.0
heart_rate	0.109088	0.268843	0.251545	-0.210593	0.198915	0.421617	0.129861	-0.166704	0.268619	0.075619	0.206821	0.133977	-0.3
position_lat	0.067768	0.186116	0.228414	-0.132884	0.140141	0.307375	0.167242	0.218518	0.049820	0.506477	0.100644	0.478564	-0.1
position_long	0.125027	-0.095324	-0.156978	0.109225	-0.011202	-0.281674	-0.124514	-0.340737	-0.086096	-0.048292	-0.359750	-0.079477	0.1
speed	NaN	NaN	NaN	NaN	NaN	NaN	NaN	-0.036491	0.620992	-0.165152	-0.036491	1.000000	-0.1
unknown_87	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
unknown_88	-0.021287	0.061667	0.115594	-0.091543	0.120377	0.070942	0.091307	-0.080790	0.057885	0.047011	-0.045494	0.047277	-0.0
unknown_90	0.138214	-0.013463	0.047080	-0.104774	-0.054814	0.095937	0.072702	0.144160	-0.034305	0.068507	-0.036498	0.080342	0.0

Now that we have created a correlation table, our next step is to create a heatmap. We also want to categorize the correlations, ranging from Very Low to Very Strong. This will help us analyze the data more effectively.

In [26]:
```
def categorize_correlation(value):
    if abs(value) < 0.2:
        return 'Very Low'
    elif 0.2 <= abs(value) < 0.4:
        return 'Low'
    elif 0.4 <= abs(value) < 0.6:
        return 'Standard (Strong enough)'
    elif 0.6 <= abs(value) < 0.8:
        return 'Strong'
```

```
    else:
        return 'Very Strong'

correlation_categories = unknown_corr.map(categorize_correlation)

correlation_categories
```

Out[26]:

	Air Power	Cadence	Form Power	Ground Time	Leg Spring Stiffness	Power	Vertical Oscillation	altitude	cadence	distance	enhanced_altitude	enhanced_speed	fractional_cadence	heart_r
Air Power	Very Strong	Very Low	Very Low	Very Low	Very Low	Low	Very Low	Very Strong	Very Low	Very Low	Very Low	Low	Very Strong	Very l
Cadence	Very Low	Very Strong	Very Strong	Low	Strong	Strong	Strong	Very Strong	Very Strong	Very Low	Very Low	Strong	Very Strong	l
Form Power	Very Low	Very Strong	Very Strong	Very Low	Very Strong	Strong	Very Strong	Very Strong	Strong	Very Low	Low	Strong	Very Strong	l
Ground Time	Very Low	Low	Very Low	Very Strong	Very Low	Very Low	Very Low	Very Strong	Low	Very Low	Very Low	Very Low	Very Strong	l
Leg Spring Stiffness	Very Low	Strong	Very Strong	Very Low	Very Strong	Strong	Strong	Very Strong	Strong	Very Low	Very Low	Standard (Strong enough)	Very Strong	Very l
Power	Low	Strong	Strong	Very Low	Strong	Very Strong	Strong	Very Strong	Strong	Very Low	Low	Very Strong	Very Strong	Stand (Str enou
Vertical Oscillation	Very Low	Strong	Very Strong	Very Low	Strong	Strong	Very Strong	Very Strong	Strong	Very Low	Very Low	Strong	Very Strong	Very l
altitude	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Low	Very Low	Very Strong	Very Low	Very Low	Very l
cadence	Very Low	Very Strong	Strong	Low	Strong	Strong	Strong	Very Low	Very Strong	Very Low	Very Low	Very Low	Very Low	l
distance	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Strong	Very Low	Standard (Strong enough)	Very Low	Very l
enhanced_altitude	Very Low	Very Low	Low	Very Low	Very Low	Low	Very Low	Very Strong	Very Low	Very Low	Very Strong	Very Low	Very Low	l
enhanced_speed	Low	Strong	Strong	Very Low	Standard (Strong enough)	Very Strong	Strong	Very Low	Very Low	Standard (Strong enough)	Very Low	Very Strong	Low	Very l
fractional_cadence	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Low	Very Low	Very Low	Very Low	Low	Very Strong	l
heart_rate	Very Low	Low	Low	Low	Very Low	Standard (Strong enough)	Very Low	Very Low	Low	Very Low	Low	Very Low	Low	V Str
position_lat	Very Low	Very Low	Low	Very Low	Very Low	Low	Very Low	Low	Very Low	Standard (Strong enough)	Very Low	Standard (Strong enough)	Very Low	l
position_long	Very Low	Very Low	Very Low	Very Low	Very Low	Low	Very Low	Low	Very Low	Very Low	Low	Very Low	Very Low	l

	Air Power	Cadence	Form Power	Ground Time	Leg Spring Stiffness	Power	Vertical Oscillation	altitude	cadence	distance	enhanced_altitude	enhanced_speed	fractional_cadence	heart_r
speed	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Low	Strong	Very Low	Very Low	Very Strong	Very Low	Stand (Str enou
unknown_87	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Very Strong	Str
unknown_88	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very
unknown_90	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very Low	Very

Vega-Lite enforces a row limit of 5000 rows in Jupyter Notebooks. This can be bypassed by aggregating data before visualization.

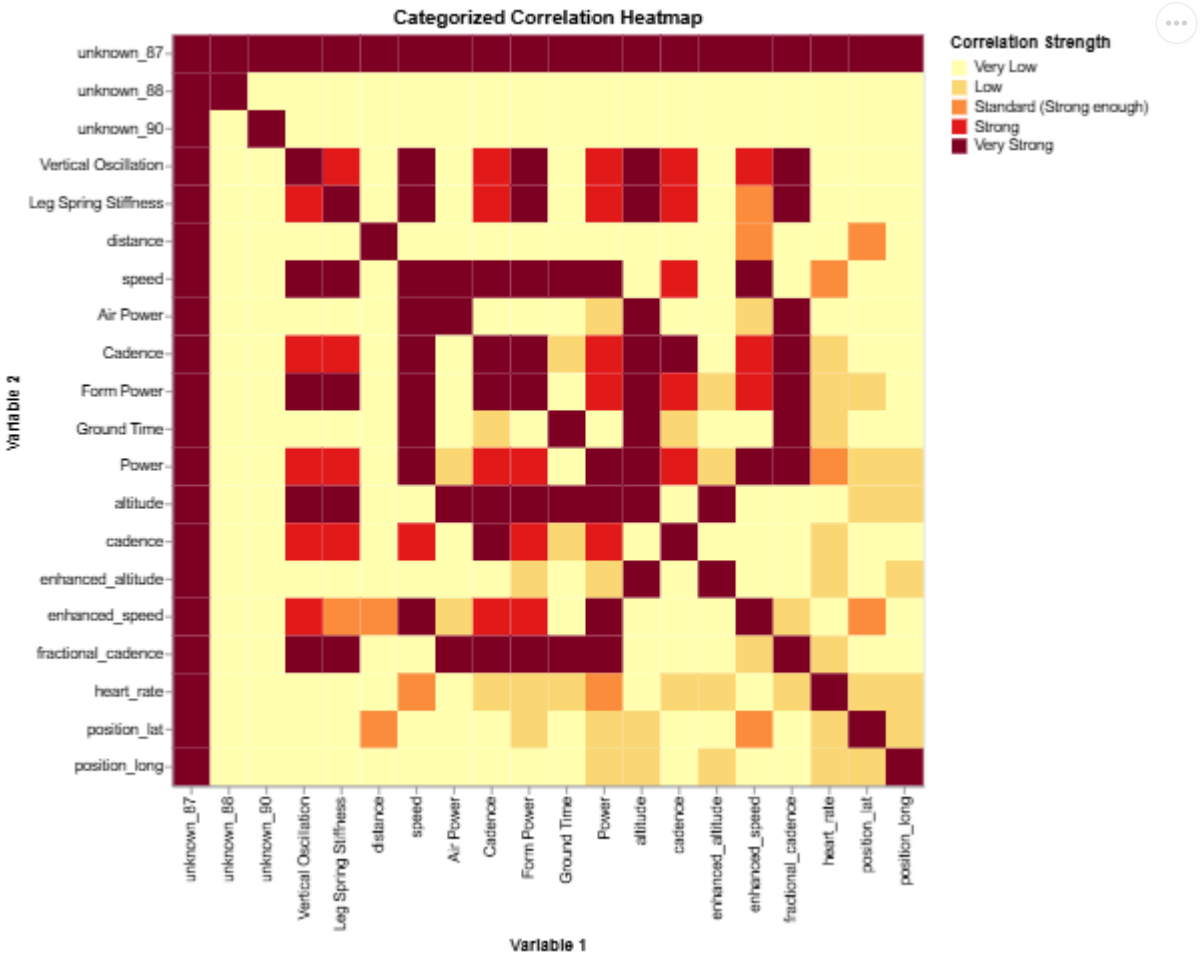
In [28]:
```
alt.data_transformers.disable_max_rows()
```

Out[28]:
```
DataTransformerRegistry.enable('default')
```

In [29]:
```
def categorize_correlation(value):
    if abs(value) < 0.2:
        return 'Very Low'
    elif 0.2 <= abs(value) < 0.4:
        return 'Low'
    elif 0.4 <= abs(value) < 0.6:
        return 'Standard (Strong enough)'
    elif 0.6 <= abs(value) < 0.8:
        return 'Strong'
    else:
        return 'Very Strong'

correlation_categories = unknown_corr.map(categorize_correlation)

corr_long = correlation_categories.reset_index().melt(id_vars='index')
corr_long.columns = ['Variable 1', 'Variable 2', 'Correlation Category']

color_scale = alt.Scale(
    domain=['Very Low', 'Low', 'Standard (Strong enough)', 'Strong', 'Very Strong'],
    range=['#ffffb2', '#fed976', '#fd8d3c', '#e31a1c', '#800026']
)

heatmap = alt.Chart(corr_long).mark_rect().encode(
    x=alt.X(
        'Variable 1:N',
        title='Variable 1',
        sort=alt.EncodingSortField(field="Correlation Category", order='descending')
    ),
    y=alt.Y(
        'Variable 2:N',
        title='Variable 2',
```

```
        sort=alt.EncodingSortField(field="Correlation Category", order='descending')
    ),
    color=alt.Color(
        'Correlation Category:N',
        scale=color_scale,
        legend=alt.Legend(title="Correlation Strength")
    ),
    tooltip=['Variable 1', 'Variable 2', 'Correlation Category']
)

heatmap = heatmap.properties(title="Categorized Correlation Heatmap", width=500, height=500)

heatmap
```

Out[29]:



4.1 Key Insights from the Correlation Table

🔹 **Cadence** has a very strong correlation with both **Form Power** (correlation = 0.774036) and **Power** (correlation = 0.759318), suggesting that mechanical energy output increases as cadence increases.

🔹 **Leg Spring Stiffness** is strongly correlated with **Vertical Oscillation** (correlation = 0.753608) and **Form Power** (correlation = 0.802700), indicating a biomechanical link between efficient leg rebound and vertical motion during running.

🔹 **Enhanced Speed** is very strongly correlated with both **Power** (correlation = 0.812346) and **Cadence** (correlation = 0.760382), reinforcing that higher speeds are associated with stronger and faster strides.

5. Plotting the Data

📌 Dashboard Narrative Overview & Flow

This section outlines how the aforementioned visualizations in this notebook form a cohesive analysis that helps to guide the reader through the different fields of Professor Brooks' Strava.csv exercise data. Each visualization builds on the previous one to support a compelling narrative.

1. **Cadence Histogram by Activity Type**

The histogram effectively displays the distribution of cadence values. Segmenting by activity type breaks down cadence values between activity types: walking, cycling, and running. The dropdown allows users to focus on specific activity types. This chart is effective for revealing how mechanical patterns shift based on different activity types.

2. **GPS Path by Date with Zoom + Filter**

The line chart using GPS coordinates effectively visualizes spatial routes of recorded workouts. The dropdown by date enables filtering to individual sessions, while the zoom interactivity allows users to inspect path detail. This design is effective for showing route coverage, repeat patterns, or geographic clusters of activity. Although it's not meant for quantitative comparison, it serves as an effective tool that connects movement to time and place.

3. **Activity Segmentation Dashboard**

The dashboard displays a bar chart summarizing the total distance covered for each activity, along with a detailed chart of the average heart rate and cadence. This combination enables users to explore performance patterns on a session-by-session basis. It is effective for identifying high-effort outliers, and or sessions that may indicate fatigue, recovery, or overtraining.

4. **Leg Spring Stiffness vs. Ground Time (Efficiency Analysis)**

The scatterplot is ideal for exploring the relationship between leg stiffness and ground contact time. The regression trendline helps identify overall correlation. In addition, the efficiency slider allows for filtering high-performing data points. The color encoding provides useful visual feedback about which data points reflect efficient movement.

5. **Animated GPS Path by Time (Plotly)**

The animation visualizes the GPS route of an activity over time. It enables users to watch the progression of a workout step-by-step and observe patterns like acceleration, pauses, or route loops. Plotly's animation features make it possible to track movement dynamically in a way that static charts cannot through Altair.

Together, these visualizations create an interactive dashboard that allows for exploratory insights and data-driven storytelling, enabling Professor Brooks to gain a personalized understanding of his training data.

Chart 1: Cadence Histogram by Activity Type (with Dropdown Filter)

Overview:

This histogram categorizes cadence into three activity types and allows users to filter by activity using a dropdown.

- Cycling - average cadence of 70-90 rpm
- Running - average cadence of 170-180 spm (85-90 rpm)
- Waling - average cadence of 94-105 spm (47-52.5 rpm)

Find the number of unique values for the Cadence Column.

In [35]:
```
cadence = df['cadence'].value_counts().reset_index()
len(cadence)
print(f"There are {len(cadence) - 1} unique values for the cadence column.")
```

```
There are 111 unique values for the cadence column.
```

Step 1: Categorize cadence into activity types

In [37]:
```
df["activity_type"] = df["cadence"].apply(lambda c: "Walking" if c < 60 else "Running" if c > 85 else "Cycling")
```

Step 2: Create dropdown menu with activity types

In [39]:
```
activity_options = ["All Activities"] + df["activity_type"].unique().tolist()
dropdown = alt.binding_select(options=activity_options, name="Select Activity: ")
selection = alt.param(name="Activity", bind=dropdown, value="All Activities")
```

Step 3: Define the filtering logic

In [41]:
```
activity_filter = alt.expr.if_(selection == "All Activities", True, alt.datum.activity_type == selection)
```

Step 4: Build the histogram with filtering and interactive dropdown

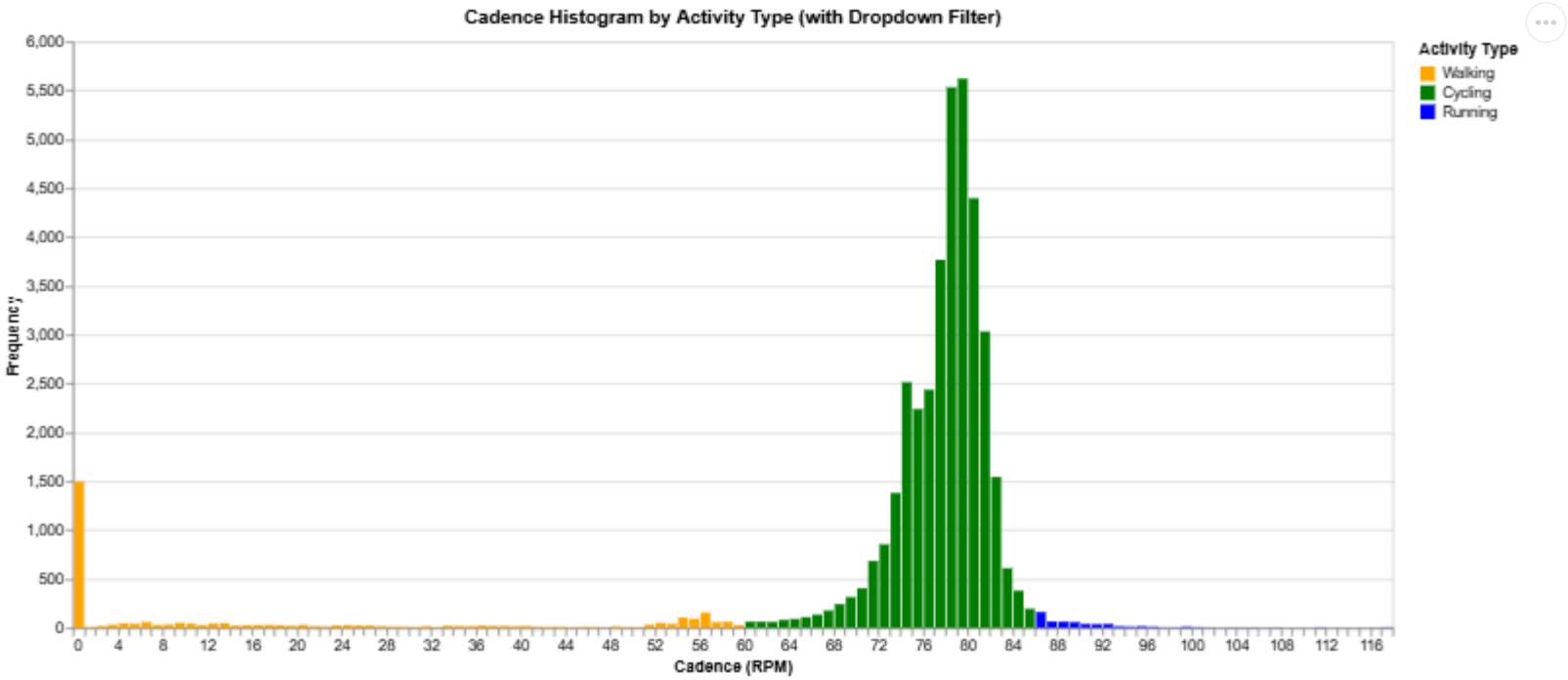
In [43]:
```
histogram = alt.Chart(df).mark_bar().encode(
    x=alt.X('cadence:Q', bin=alt.Bin(maxbins=120), title='Cadence (RPM)'),
    y=alt.Y('count()', title='Frequency'),
    color=alt.Color('activity_type:N',
                    scale=alt.Scale(domain=['Walking', 'Cycling', 'Running'],
                                    range=['orange', 'green', 'blue']),
                    legend=alt.Legend(title="Activity Type")),
    tooltip=[
        alt.Tooltip('cadence:Q', title='Cadence'),
        alt.Tooltip('count()', title='Frequency')
    ]
)

histogram = histogram.add_params(selection)
histogram = histogram.transform_filter(activity_filter)
histogram = histogram.properties(
    title="Cadence Histogram by Activity Type (with Dropdown Filter)",
    width=900,
```

```
    height=400
)
```

Step 5: Final Code Implementation

In [45]:
```
df["activity_type"] = df["cadence"].apply(lambda c: "Walking" if c < 60 else "Running" if c > 85 else "Cycling")

activity_options = ["All Activities"] + df["activity_type"].unique().tolist()
dropdown = alt.binding_select(options=activity_options, name="Select Activity: ")
selection = alt.param(name="Activity", bind=dropdown, value="All Activities")

activity_filter = alt.expr.if_(selection == "All Activities", True, alt.datum.activity_type == selection)

histogram = alt.Chart(df).mark_bar().encode(
    x=alt.X('cadence:Q', bin=alt.Bin(maxbins=120), title='Cadence (RPM)'),
    y=alt.Y('count()', title='Frequency'),
    color=alt.Color('activity_type:N',
                    scale=alt.Scale(domain=['Walking', 'Cycling', 'Running'],
                                    range=['orange', 'green', 'blue']),
                    legend=alt.Legend(title="Activity Type")),
    tooltip=[
        alt.Tooltip('cadence:Q', title='Cadence'),
        alt.Tooltip('count()', title='Frequency')
    ]
)

histogram = histogram.add_params(selection)
histogram = histogram.transform_filter(activity_filter)
histogram = histogram.properties(
    title="Cadence Histogram by Activity Type (with Dropdown Filter)",
    width=900,
    height=400
)

histogram
```

Out[45]:



Select Activity: All Activities

Chart 2: GPS Path of Activity by Date (Line Chart + Zoom + Dropdown)

Overview:

This chart plots the GPS path of recorded Strava activities using latitude and longitude coordinates. It includes:

- A dropdown filter to select a specific date
- A zoom interaction to inspect segments of the path in detail

Step 1: Create a copy and extract relevant columns

In [49]:
```
df_copy = df.copy()

df_subset = df_copy[['position_long', 'position_lat', 'timestamp']]
df_subset
```

Out[49]:

	position_long	position_lat	timestamp
0	NaN	NaN	2019-07-08 21:04:03
1	NaN	NaN	2019-07-08 21:04:04
2	NaN	NaN	2019-07-08 21:04:07
3	-999063637.0	504432050.0	2019-07-08 21:04:14
4	-999064534.0	504432492.0	2019-07-08 21:04:15
...	...	...	...
40644	-999308618.0	504554553.0	2019-10-03 23:04:54
40645	-999309466.0	504553919.0	2019-10-03 23:04:56
40646	-999309432.0	504553588.0	2019-10-03 23:04:57
40647	-999308808.0	504552459.0	2019-10-03 23:05:02
40648	-999308613.0	504552222.0	2019-10-03 23:05:05

40649 rows × 3 columns

Step 2: Drop rows with missing GPS coordinates and Grab the Date from the timestamp column

In [51]:
```
df_clean = df_subset.dropna(subset=['position_long', 'position_lat'])

df_clean['date'] = pd.to_datetime(df_clean['timestamp']).dt.date
df_clean['date'] = df_clean['date'].astype(str)

df_clean
```

Out[51]:

	position_long	position_lat	timestamp	date
3	-999063637.0	504432050.0	2019-07-08 21:04:14	2019-07-08
4	-999064534.0	504432492.0	2019-07-08 21:04:15	2019-07-08
5	-999064622.0	504432667.0	2019-07-08 21:04:16	2019-07-08
6	-999064796.0	504432736.0	2019-07-08 21:04:17	2019-07-08
7	-999064984.0	504432914.0	2019-07-08 21:04:18	2019-07-08
...	...	...	...	...
40644	-999308618.0	504554553.0	2019-10-03 23:04:54	2019-10-03
40645	-999309466.0	504553919.0	2019-10-03 23:04:56	2019-10-03
40646	-999309432.0	504553588.0	2019-10-03 23:04:57	2019-10-03
40647	-999308808.0	504552459.0	2019-10-03 23:05:02	2019-10-03
40648	-999308613.0	504552222.0	2019-10-03 23:05:05	2019-10-03

40457 rows × 4 columns

Step 3: Drop rows with missing GPS coordinates and Grab the Date from the timestamp column

In [53]:
```
min_lat = df_clean['position_lat'].min()
print(f"The minimum latitude is {min_lat}")

max_lat = df_clean['position_lat'].max()
print(f"The maximum latitude is {max_lat}\n")

min_long = df_clean['position_long'].min()
print(f"The minimum longitude is {min_lat}")

max_long = df_clean['position_long'].max()
print(f"The maximum longitude is {max_lat}")
```

```
The minimum latitude is 503986812.0
The maximum latitude is 508927195.0

The minimum longitude is 503986812.0
The maximum longitude is 508927195.0
```

Step 4: Extract unique dates and set up dropdown options

In [55]:
```
date_options = ["All Dates"] + sorted(df_clean["date"].astype(str).unique().tolist())

dropdown = alt.binding_select(options=date_options, name="Select Date: ")
date_selection = alt.param(name="DateSelector", bind=dropdown, value="All Dates")
```

Step 5: Create filter expression for selected date

In [57]:
```
date_filter = alt.expr.if_(date_selection == "All Dates", True, alt.datum.date == date_selection)
```

Step 6: Define zoom interaction

In [59]:
```
zoom_selection = alt.selection_interval(bind='scales')
```

Step 7: Build the line chart with filtering and interactive dropdown

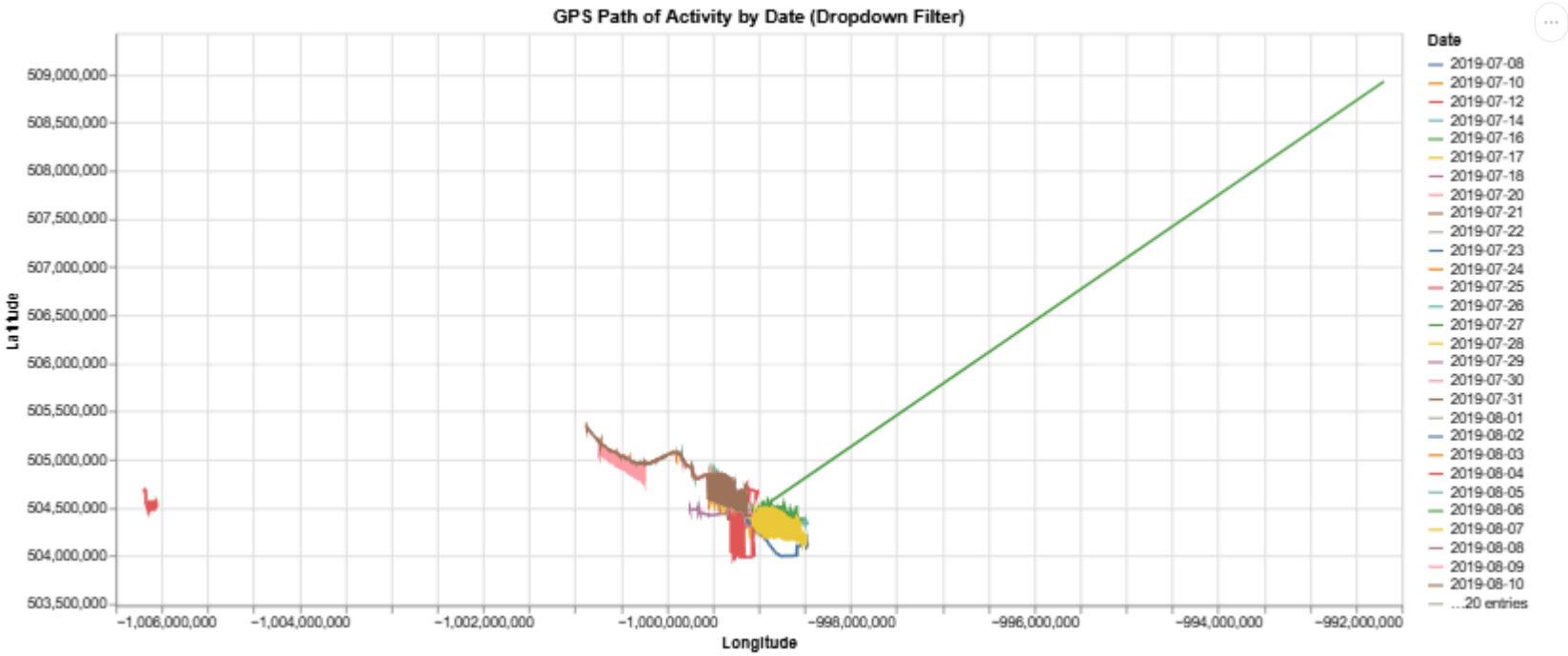
In [61]:
```
path_chart = alt.Chart(df_clean).mark_line().encode(
    x=alt.X('position_long:Q', title='Longitude'),
    y=alt.Y(
        'position_lat:Q',
        title='Latitude',
        scale=alt.Scale(domain=[min_lat - 500000, max_lat + 500000])
    ),
    color=alt.Color('date:N', title='Date'),
    tooltip=[
        alt.Tooltip('timestamp:T', title='Timestamp'),
        alt.Tooltip('position_long:Q', title='Longitude'),
        alt.Tooltip('position_lat:Q', title='Latitude'),
        alt.Tooltip('date:N', title='Date')
    ]
)

path_chart = path_chart.transform_filter(date_filter)
path_chart = path_chart.add_params(zoom_selection, date_selection)
path_chart = path_chart.properties(
    title='GPS Path of Activity by Date (Dropdown Filter)',
    width=900,
    height=400,
)
```

Step 8: Final Code Implementation

In [63]:
```
date_options = ["All Dates"] + sorted(df_clean["date"].astype(str).unique().tolist())

dropdown = alt.binding_select(options=date_options, name="Select Date: ")

date_selection = alt.param(name="DateSelector", bind=dropdown, value="All Dates")

date_filter = alt.expr.if_(date_selection == "All Dates", True, alt.datum.date == date_selection)

zoom_selection = alt.selection_interval(bind='scales')

path_chart = alt.Chart(df_clean).mark_line().encode(
    x=alt.X('position_long:Q', title='Longitude'),
    y=alt.Y(
        'position_lat:Q',
        title='Latitude',
        scale=alt.Scale(domain=[min_lat - 500000, max_lat + 500000])
    ),
    color=alt.Color('date:N', title='Date'),
    tooltip=[
```

```
        alt.Tooltip('timestamp:T', title='Timestamp'),
        alt.Tooltip('position_long:Q', title='Longitude'),
        alt.Tooltip('position_lat:Q', title='Latitude'),
        alt.Tooltip('date:N', title='Date')
    ]
)

path_chart = path_chart.transform_filter(date_filter)
path_chart = path_chart.add_params(zoom_selection, date_selection)
path_chart = path_chart.properties(
    title='GPS Path of Activity by Date (Dropdown Filter)',
    width=900,
    height=400,
)

path_chart
```

Out[63]:



Select Date: All Dates

Chart 3: Activity Segmentation

Overview:

This dashboard breaks down exercise sessions and allows users to explore metrics like distance, cadence, power, and heart rate.

- It uses click-based interactivity to link a summary chart with detailed metrics like Distance, Avg. Heart Rate, Avg. Cadence, and Avg. Power for the selected activity.

Step 1: Aggregate data for each activity session

In [67]:
```
activity_summary = df.groupby('datafile').agg({
    'distance': 'sum',
    'heart_rate': 'mean',
    'Leg Spring Stiffness': 'mean',
    'Ground Time': 'mean',
    'cadence': 'mean',
    'Power': 'mean'
})

activity_summary = activity_summary.reset_index()

activity_summary.head()
```

Out[67]:

	datafile	distance	heart_rate	Leg Spring Stiffness	Ground Time	cadence	Power
0	activities/2675855419.fit.gz	302980.56	118.337398	NaN	NaN	74.406504	NaN
1	activities/2677658978.fit.gz	27331.10	122.117647	NaN	NaN	77.191176	NaN
2	activities/2677658993.fit.gz	21677.25	135.592593	NaN	NaN	81.777778	NaN
3	activities/2677659014.fit.gz	22203.87	134.571429	NaN	NaN	79.517857	NaN
4	activities/2682705331.fit.gz	3266077.74	117.687157	NaN	NaN	74.135016	NaN

Step 2: Convert distance from meters to kilometers

In [69]:
```
activity_summary['distance_km'] = activity_summary['distance'] / 1000

print(activity_summary['distance'].head())
print('-')
print(activity_summary['distance_km'].head())
```

```
0     302980.56
1      27331.10
2      21677.25
3      22203.87
4    3266077.74
Name: distance, dtype: float64
-
0     302.98056
1      27.33110
2      21.67725
3      22.20387
4    3266.07774
Name: distance_km, dtype: float64
```

Step 3: Define a single-click selection on 'datafile'

In [71]:
```
click = alt.selection_point(fields=['datafile'])
```

Step 4: Create a main bar chart with single-click selection

We build the main chart using:

- X-axis: each activity session (datafile)
- Y-axis: total distance in km
- Color: average heart rate (using a red-yellow-green scale)

In [73]:
```
activity_chart = alt.Chart(activity_summary).mark_bar().encode(
        x=alt.X('datafile:N', sort='-x', title='Activity (datafile)'),
        y=alt.Y('distance_km:Q', title='Total Distance (km)'),
        color=alt.Color(
            'heart_rate:Q',
            title='Avg. Heart Rate',
            scale=alt.Scale(scheme='redyellowgreen')
        ),
        tooltip=[
            alt.Tooltip('datafile:N', title='Activity'),
            alt.Tooltip('distance_km:Q', format='.2f', title='Total Distance (km)'),
            alt.Tooltip('heart_rate:Q', format='.1f', title=' Avg. Heart Rate'),
            alt.Tooltip('cadence:Q', format='.1f', title='Avg. Cadence'),
            alt.Tooltip('Power:Q', format='.1f', title='Avg. Power')
        ]
    )

activity_chart = activity_chart.add_params(click)

activity_chart = activity_chart.properties(
    title='Activity Segmentation: Total Distance by Activity',
    width=850,
    height=300
)
```

Step 5: Create a linked bar chart below showing heart rate and cadence for the selected activity session

In [75]:
```
metrics_chart = alt.Chart(activity_summary).transform_filter(click)

metrics_chart = metrics_chart.transform_fold(['heart_rate', 'cadence'], as_=['Metric', 'Value'])

metrics_chart = metrics_chart.transform_calculate(PrettyMetric="{'heart_rate': 'Avg. Heart Rate', 'cadence': 'Avg. Cadence'}[datum.Metric]")

metrics_chart = metrics_chart.mark_bar(size=40).encode(
    y=alt.Y('PrettyMetric:N', title='Activity Metrics'),
    x=alt.X('Value:Q', title='Value'),
    color=alt.Color('PrettyMetric:N', title='Activity Metrics', scale=alt.Scale(scheme='category10')),
    tooltip=[alt.Tooltip('Value:Q', format='.1f')]
)

metrics_chart = metrics_chart.properties(
    title='Selected Activity Metrics (Avg. Heart Rate and Avg. Cadence)',
    width=850,
    height=200
)

advanced_chart = activity_chart & metrics_chart
```

Step 6: Final Code Implementation

In [77]:
```
click = alt.selection_point(fields=['datafile'])

activity_chart = alt.Chart(activity_summary).mark_bar().encode(
        x=alt.X('datafile:N', sort='-x', title='Activity (datafile)'),
        y=alt.Y('distance_km:Q', title='Total Distance (km)'),
        color=alt.Color(
            'heart_rate:Q',
            title='Avg. Heart Rate',
            scale=alt.Scale(scheme='redyellowgreen')
        ),
        tooltip=[
            alt.Tooltip('datafile:N', title='Activity'),
            alt.Tooltip('distance_km:Q', format='.2f', title='Total Distance (km)'),
            alt.Tooltip('heart_rate:Q', format='.1f', title=' Avg. Heart Rate'),
            alt.Tooltip('cadence:Q', format='.1f', title='Avg. Cadence'),
            alt.Tooltip('Power:Q', format='.1f', title='Avg. Power')
        ]
    )

activity_chart = activity_chart.add_params(click)

activity_chart = activity_chart.properties(
    title='Activity Segmentation: Total Distance by Activity',
    width=850,
    height=300
)

metrics_chart = alt.Chart(activity_summary).transform_filter(click)

metrics_chart = metrics_chart.transform_fold(['heart_rate', 'cadence'], as_=['Metric', 'Value'])

metrics_chart = metrics_chart.transform_calculate(PrettyMetric="{'heart_rate': 'Avg. Heart Rate', 'cadence': 'Avg. Cadence'}[datum.Metric]")

metrics_chart = metrics_chart.mark_bar(size=40).encode(
    y=alt.Y('PrettyMetric:N', title='Activity Metrics'),
    x=alt.X('Value:Q', title='Value'),
    color=alt.Color('PrettyMetric:N', title='Activity Metrics', scale=alt.Scale(scheme='category10')),
    tooltip=[alt.Tooltip('Value:Q', format='.1f')]
)

metrics_chart = metrics_chart.properties(
    title='Selected Activity Metrics (Avg. Heart Rate and Avg. Cadence)',
    width=850,
    height=200
)

advanced_chart = activity_chart & metrics_chart

advanced_chart
```

Out[77]:

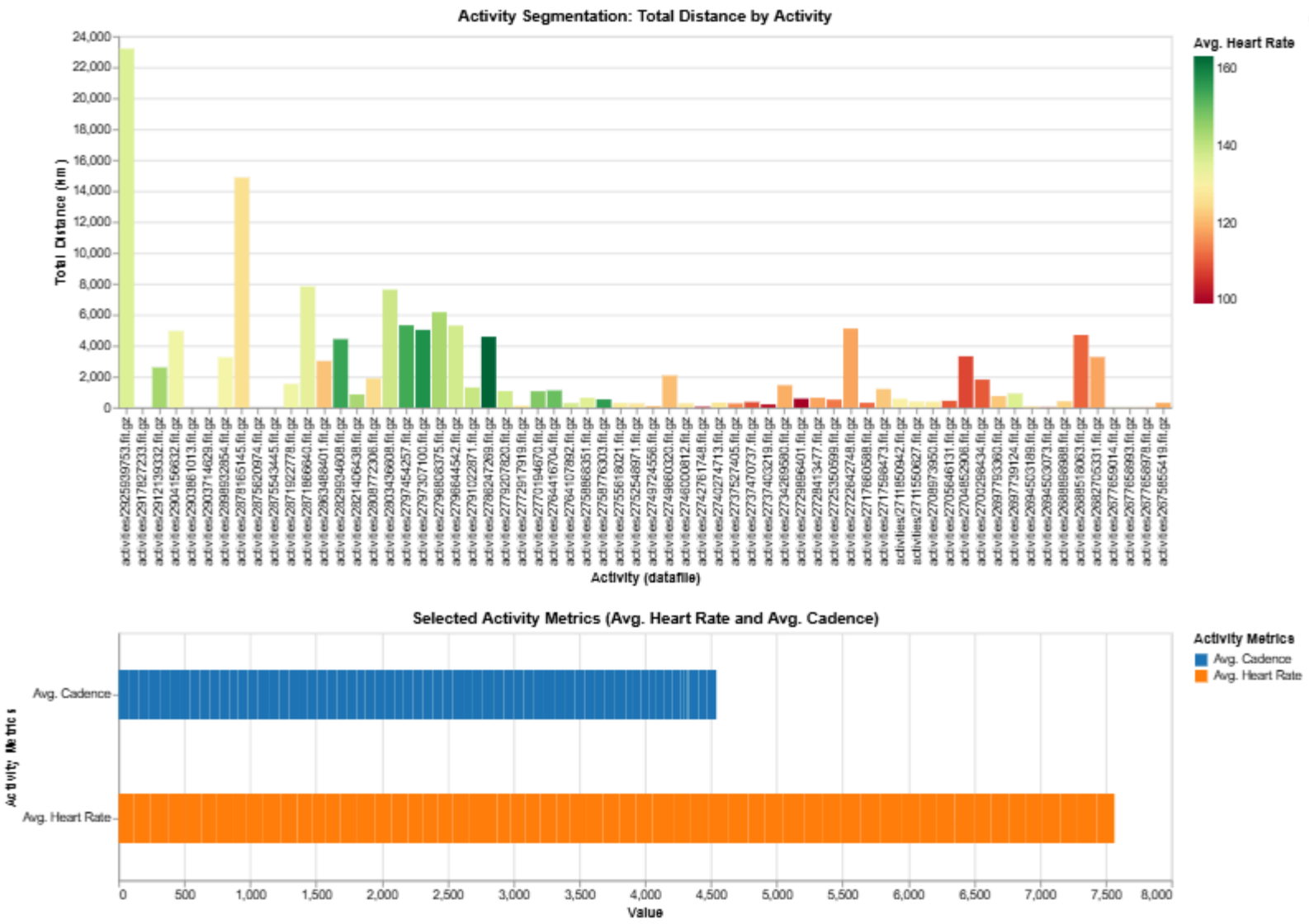


Chart 4: Leg Spring Stiffness & Ground Time Analysis

Overview:

This scatterplot explores the relationship between Leg Spring Stiffness and Ground Time. The chart includes:

- A color-coded efficiency metric
- A regression trendline
- An interactive slider to filter data by efficiency

Step 1: Compute efficiency and explore its distribution

In [81]:
```
df['Ground Time'].replace(0, pd.NA, inplace=True)
df['efficiency'] = df['Leg Spring Stiffness'] / df['Ground Time']

efficienty_metrics = df['efficiency'].describe()
print(efficienty_metrics)
```

```
count     17625.000000
unique     1464.000000
top           0.041667
freq        212.000000
Name: efficiency, dtype: float64
```

Step 2: Create a Scatter Plot

In [83]:
```
scatter = alt.Chart(df).mark_circle(size=60).encode(
    x=alt.X('Leg Spring Stiffness:Q', title='Leg Spring Stiffness'),
    y=alt.Y('Ground Time:Q', title='Ground Time'),
    color=alt.Color('efficiency:Q', scale=alt.Scale(scheme='viridis'), title='Efficiency'),
    tooltip=[
        alt.Tooltip('Leg Spring Stiffness:Q', title='Leg Spring Stiffness'),
        alt.Tooltip('Ground Time:Q', title='Ground Time'),
        alt.Tooltip('efficiency', format='.3f', title='Efficiency')
    ]
)
```

Step 3: Add a Regression Trendline

In [85]:
```
trendline = alt.Chart(df).transform_regression('Leg Spring Stiffness', 'Ground Time')
trendline = trendline.mark_line(color='black')
trendline = trendline.encode(x='Leg Spring Stiffness:Q', y='Ground Time:Q')
```

Step 4: Define a slider for the Efficiency value

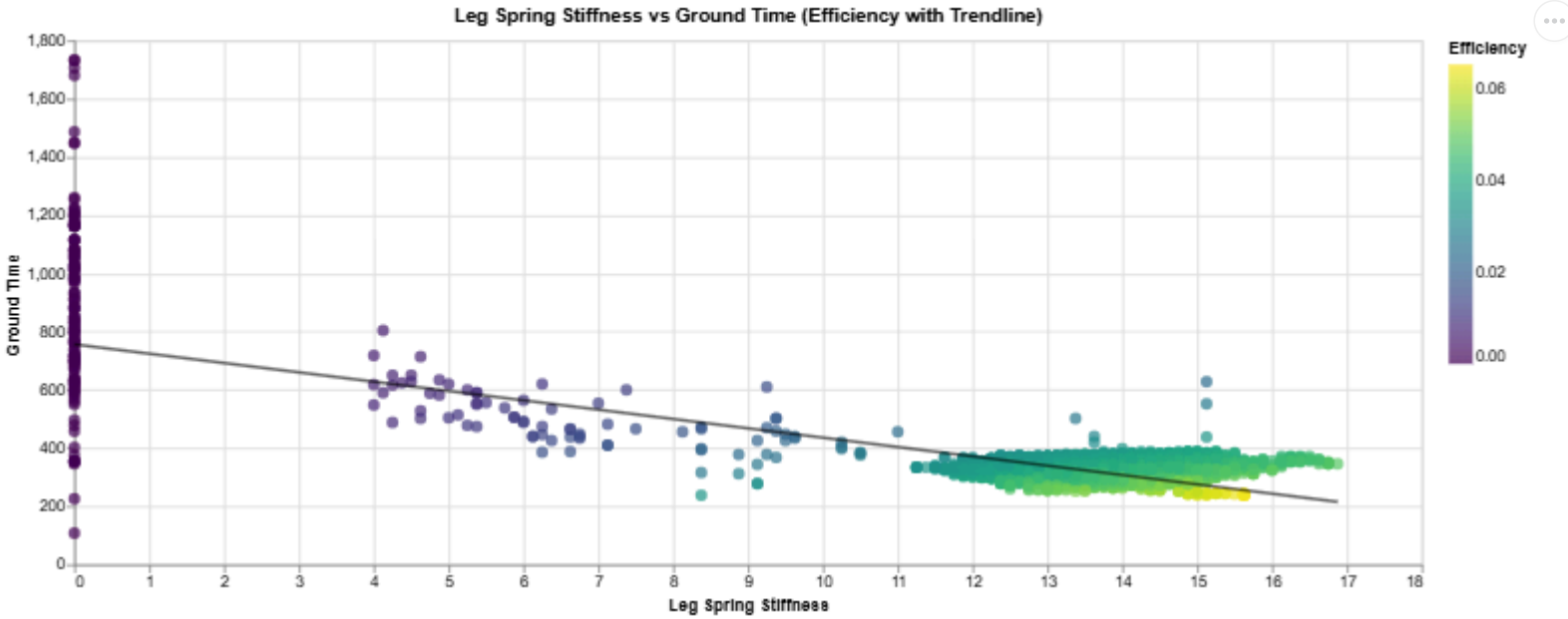
In [87]:
```
efficiency_slider = alt.binding_range(
    min=df['efficiency'].min(),
    max=df['efficiency'].max(),
    step=0.001,
    name='Efficiency Slider Filter:'
)

efficiency_select = alt.param(
    name='EffSelector',
```

```
    bind=efficiency_slider,
    value=df['efficiency'].min()
)
```

Step 5: Final Code Implementation

In [89]:
```
efficiency_slider = alt.binding_range(
    min=df['efficiency'].min(),
    max=df['efficiency'].max(),
    step=0.001,
    name='Efficiency Slider Filter:'
)

efficiency_select = alt.param(
    name='EffSelector',
    bind=efficiency_slider,
    value=df['efficiency'].min()
)

scatter = alt.Chart(df).mark_circle(size=60).encode(
    x=alt.X('Leg Spring Stiffness:Q', title='Leg Spring Stiffness'),
    y=alt.Y('Ground Time:Q', title='Ground Time'),
    color=alt.Color('efficiency:Q', scale=alt.Scale(scheme='viridis'), title='Efficiency'),
    tooltip=[
        alt.Tooltip('Leg Spring Stiffness:Q', title='Leg Spring Stiffness'),
        alt.Tooltip('Ground Time:Q', title='Ground Time'),
        alt.Tooltip('efficiency', format='.3f', title='Efficiency')
    ]
)

scatter = scatter.add_params(efficiency_select)
scatter = scatter.transform_filter(alt.datum.efficiency >= efficiency_select)

trendline = alt.Chart(df).transform_regression('Leg Spring Stiffness', 'Ground Time')
trendline = trendline.mark_line(color='black', opacity=0.6)
trendline = trendline.encode(x='Leg Spring Stiffness:Q', y='Ground Time:Q')

scatter_with_trend = scatter + trendline

scatter_with_trend = (scatter + trendline).properties(
    title='Leg Spring Stiffness vs Ground Time (Efficiency with Trendline)',
    width=900,
    height=350
)

scatter_with_trend
```

Out[89]:



Efficiency Slider Filter: 0

To further validate the Trendline, we can calculate these Additional Statistical Measures:

In [91]:

```
df['Leg Spring Stiffness'] = pd.to_numeric(df['Leg Spring Stiffness'], errors='coerce')
df['Ground Time'] = pd.to_numeric(df['Ground Time'], errors='coerce')

df_clean = df.dropna(subset=['Leg Spring Stiffness', 'Ground Time'])

slope, intercept, r_value, p_value, std_err = linregress(df['Leg Spring Stiffness'], df['Ground Time'])

regression_results = pd.DataFrame({
    "Metric": ["Slope", "Intercept", "R-value", "R-squared", "P-value", "Std Error"],
    "Value": [slope, intercept, r_value, r_value**2, p_value, std_err]
})

regression_results
```

Out[91]:

	Metric	Value
0	Slope	NaN
1	Intercept	NaN
2	R-value	NaN
3	R-squared	NaN
4	P-value	NaN
5	Std Error	NaN

Advanced Visualization: Animated GPS Path

Overview:

- This interactive animation shows the GPS route of a selected Strava activity, step-by-step, using a time slider.
- The animation allows users to visually follow the route as it was recorded in real time.

Why Plotly instead of Altair?

This notebook uses both Altair and Plotly. An issue arose when creating the final visualization because Altair does not support frame-by-frame animation and interactive playback, which Plotly does. To overcome this, Plotly was used. Using Plotly allows for dynamic visualization of GPS path progression over time, making it possible to observe pacing, pauses, and route behavior in a way that static charts cannot. This enhances the dashboard's storytelling and interactivity.

Step 1: Import the necessary libraries

In [95]:
```
import pandas as pd
import plotly.express as px
import ipywidgets as widgets
from IPython.display import display
```

Step 2: Get all unique activity files

In [97]:
```
activity_files = df['datafile'].dropna().unique()
```

Step 3: Create an Activity Dropdown Widget

In [99]:
```
activity_dropdown = widgets.Dropdown(
    options=activity_files,
    description='Select Activity File:',
    style={'description_width': 'initial'},
    layout=widgets.Layout(width='325px')
)
```

Step 4: Define the Plotting Function

In [101…
```
def plot_gps_path(activity_file):
    df_copy = df[df['datafile'] == activity_file].copy()
    df_copy = df_copy.dropna(subset=['position_long', 'position_lat', 'timestamp'])

    df_copy['timestamp'] = pd.to_datetime(df_copy['timestamp'])
    df_copy = df_copy.sort_values('timestamp')
    df_copy['frame'] = df_copy['timestamp'].dt.strftime('%I:%M:%S %p')
```

Step 5: Build the Animated Scatter Plot with Styling and Interactivity

In [103…
```
'''
    # Create an animated scatter plot using Plotly Express
    scatter = px.scatter(
        df_copy,
        x='position_long',
        y='position_lat',
        animation_frame='frame',
        range_x=[df_copy['position_long'].min(), df_copy['position_long'].max()],
        range_y=[df_copy['position_lat'].min(), df_copy['position_lat'].max()],
        labels={
            'position_long': 'Longitude',
            'position_lat': 'Latitude',
            'cadence': 'Cadence',
            'heart_rate': 'Heart Rate',
            'frame': 'Time'
        },
        hover_data=['frame', 'cadence', 'heart_rate'],
        height=500
    )

    # Add the full GPS route as a red line behind the animated point
    scatter.add_scatter(
        x=df_copy['position_long'],
        y=df_copy['position_lat'],
        mode='lines',
        line=dict(color='red', width=0.4),
        name='Activity Path'
    )


    scatter.update_traces(
        marker=dict(size=8, color='blue'),
        selector=dict(mode='markers')
    )

    # Update chart layout
    scatter.update_layout(
        title={
            'text': f'Animated GPS Path for {activity_file}',
            'x': 0.5,
            'xanchor': 'center'
        },
        showlegend=False
    )
```

```
    scatter.show()
'''
```

Out[103…

```
"\n    # Create an animated scatter plot using Plotly Express\n    scatter = px.scatter(\n        df_copy,\n        x='position_long',\n        y='position_la
t',\n        animation_frame='frame',\n        range_x=[df_copy['position_long'].min(), df_copy['position_long'].max()],\n        range_y=[df_copy['position_la
t'].min(), df_copy['position_lat'].max()],\n        labels={\n            'position_long': 'Longitude',\n            'position_lat': 'Latitude',\n            'ca
dence': 'Cadence',\n            'heart_rate': 'Heart Rate',\n            'frame': 'Time'\n        },\n        hover_data=['frame', 'cadence', 'heart_rate'],\n
height=500\n    )\n    \n    # Add the full GPS route as a red line behind the animated point\n    scatter.add_scatter(\n        x=df_copy['position_long'],\n
y=df_copy['position_lat'],\n        mode='lines',\n        line=dict(color='red', width=0.4),\n        name='Activity Path'\n    )\n    \n    \n    scatter.updat
e_traces(\n        marker=dict(size=8, color='blue'),\n        selector=dict(mode='markers')\n    )\n    \n    # Update chart layout\n    scatter.update_layout
(\n        title={\n            'text': f'Animated GPS Path for {activity_file}',\n            'x': 0.5,\n            'xanchor': 'center'\n        },\n        sh
owlegend=False\n    )\n    \n    \n    scatter.show()\n"
```

Step 6: Final Code Implementation

In [105…

```
activity_dropdown = widgets.Dropdown(
    options=activity_files,
    description='Select Activity File:',
    style={'description_width': 'initial'},
    layout=widgets.Layout(width='325px')
)

def plot_gps_path(activity_file):
    df_copy = df[df['datafile'] == activity_file].copy()
    df_copy = df_copy.dropna(subset=['position_long', 'position_lat', 'timestamp'])

    df_copy['timestamp'] = pd.to_datetime(df_copy['timestamp'])
    df_copy = df_copy.sort_values('timestamp')
    df_copy['frame'] = df_copy['timestamp'].dt.strftime('%I:%M:%S %p')

    # Create an animated scatter plot using Plotly Express
    scatter = px.scatter(
        df_copy,
        x='position_long',
        y='position_lat',
        animation_frame='frame',
        range_x=[df_copy['position_long'].min(), df_copy['position_long'].max()],
        range_y=[df_copy['position_lat'].min(), df_copy['position_lat'].max()],
        labels={
            'position_long': 'Longitude',
            'position_lat': 'Latitude',
            'cadence': 'Cadence',
            'heart_rate': 'Heart Rate',
            'frame': 'Time'
        },
        hover_data=['frame', 'cadence', 'heart_rate'],
        height=500
    )

    # Add the full GPS route as a red line behind the animated point
    scatter.add_scatter(
        x=df_copy['position_long'],
        y=df_copy['position_lat'],
```

```
        mode='lines',
        line=dict(color='red', width=0.4),
        name='Activity Path'
    )


    scatter.update_traces(
        marker=dict(size=8, color='blue'),
        selector=dict(mode='markers')
    )

    # Update chart layout
    scatter.update_layout(
        title={
            'text': f'Animated GPS Path for {activity_file}',
            'x': 0.5,
            'xanchor': 'center'
        },
        showlegend=False
    )


    scatter.show()

display(widgets.interact(plot_gps_path, activity_file=activity_dropdown))
```

```
interactive(children=(Dropdown(description='Select Activity File:', layout=Layout(width='325px'), options=('ac…
<function __main__.plot_gps_path(activity_file)>
```

Example HTML GIF Rendering: Animated GPS Route for Activity: 2688518603.fit.gz

Overview:

- This example displays the animated GPS path of a **single selected activity**. It avoids the use of widgets so the animation can be rendered in exported `.html` and `.pdf` formats.

In [108…

```
activity_file = "activities/2688518063.fit.gz"
```

In [109…

```
def plot_gps_path(activity_file):
    df_copy = df[df['datafile'] == activity_file].copy()
    df_copy = df_copy.dropna(subset=['position_long', 'position_lat', 'timestamp'])

    df_copy['timestamp'] = pd.to_datetime(df_copy['timestamp'])
    df_copy = df_copy.sort_values('timestamp')
    df_copy['frame'] = df_copy['timestamp'].dt.strftime('%I:%M:%S %p')

    scatter = px.scatter(
        df_copy,
        x='position_long',
        y='position_lat',
        animation_frame='frame',
        range_x=[df_copy['position_long'].min(), df_copy['position_long'].max()],
        range_y=[df_copy['position_lat'].min(), df_copy['position_lat'].max()],
        labels={
            'position_long': 'Longitude',
            'position_lat': 'Latitude',
```

```
            'cadence': 'Cadence',
            'heart_rate': 'Heart Rate',
            'frame': 'Time'
        },
        hover_data=['frame', 'cadence', 'heart_rate'],
        height=500
    )

    scatter.add_scatter(
        x=df_copy['position_long'],
        y=df_copy['position_lat'],
        mode='lines',
        line=dict(color='red', width=0.4),
        name='Activity Path'
    )

    scatter.update_traces(marker=dict(size=8, color='blue'), selector=dict(mode='markers'))

    scatter.update_layout(
        title={
            'text': f'Animated GPS Path for {activity_file}',
            'x': 0.5,
            'xanchor': 'center'
        },
        showlegend=False
    )

    return scatter
```

Step 1: Import the necessary library

In [111…

```
import plotly.io as pio
```

Step 2: Export the Animated GPS Path as a Standalone HTML File

In [113…

```
fig = plot_gps_path(activity_file)

pio.write_html(fig, file="activities (2688518063).html", auto_open=False)
```

Interpreting the Visualizations (Insights and Analysis):

The visualizations here give a layered perspective on Professor Brooks' exercise data, connecting biomechanical measurements to the spatial-temporal structure of each session. Cadence is a key predictor of movement style, and the histogram is a good place to begin exploring these patterns. Sorting cadence distributions by activity type, walking, cycling, and running, highlights differences in stride dynamics. Broken down by activity type, it is evident that cadence corresponds to intuitive expectations: running workouts are bunched up at the high end, walking is skewed toward the low end, and cycling is in between. In addition, the histogram reveals variability within activity types, like fast walking or low-cadence runs. These discrepancies may reflect the impact of terrain, intentional changes in pace, or the impact of cumulative factors of fatigue that never appear in bare numbers.

Moving beyond stride measures, the GPS path chart adds geography to the mix. This date-based zoom-enabled dropdown-enabled line chart introduces the ability to inspect workout routes over time closely. Spatial plotting facilitates the identification of circular training loops, workout location changes, and route fluctuations that map onto different performance outcomes. Even though the graph is not quantitatively technical, it adds narrative depth by linking movement patterns to real geography. It supplements cadence and heart rate readings by placing them in physical space to interpret training behavior and impact within a spatial context.

The activity segmentation dashboard focuses on workout-level performance to further illuminate the movement and space analyses. The visualization combines total distance, average heart rate, and average cadence, enabling an overall understanding of how effort varies between workouts. Users can zoom into specific sessions and identify which workouts are long-duration endurance efforts or short, high-intensity intervals. For instance, reduced-distance runs coupled with high heart rate values and cadence could be marked for interval training days. At the same time, extended sessions at moderate readings align with aerobic base-building exercises. The interactivity of the chart makes it more useful as it provides an easy-to-use tool for evaluating load management and any imbalances between intensity and volume throughout the training block.

From biomechanics to trends in performance, the plot of leg spring stiffness vs. ground contact time provides a more technical layer of understanding. The chart's negative slope, indicated by a regression line, verifies the biomechanical hypothesis that stiffer will equal shorter ground contact time, which is a marker of good running mechanics. Gradients of color also contribute to interpretability, indicating which sessions fall into areas of efficiency desired. The slider allows simple, flexible filtering to extract peak-efficiency sessions or flag potential outliers. Along with cadence and distance, this plot provides valuable insight into how form quality varies between sessions and how it might be linked to fatigue, effort level, or even ambient factors like terrain.

Finally, the animated GPS track brings static route data and a dynamic life history. Unlike the line chart — GPS Path of Activity by Date, this animation illustrates how each session unfolds moment by moment. Temporal motion, with hover-over tooltips showing cadence and heart rate, allows users to identify pacing decisions, recovery periods, or random stops. This level of detail reveals nuances in training that are often lost when viewing session averages alone.

These visualizations form an integrated dashboard combining biomechanical insight with contextual data. The cadence histogram and efficiency scatterplot contextualize stride quality and physical performance, the GPS charts place those metrics in space and time, and the activity segmentation dashboard combines individual workouts with overall training trends. The result is a system enabling Professor Brooks to examine his training data in multiple dimensions, leading to more informed decisions and strategic planning for future sessions.

In [ ]: