UDA_FinalProject_Stapleton

January 2, 2025

1 UDA Final Project

1.1 Joshua Stapleton

1.2 CID: 02301776

1.3 03/01/2025

1.4 0. Introduction

Over the last year, I've done a decent amount of running. As expected, there are a few routes I frequently follow, and a long tail of once-off anomalous routes (eg: for races / random runs with friends). While the common routes followed are relatively consistent, every route is fully geospatialtemporally unique! I use an Apple Watch to record my runs, and Apple Health offers the ablity to export raw workout data. As such, I have a very detailed dataset of my runs, and this assignment is an in-depth analysis of it, with the goals of:

1) Answering various questions detailed in Section 2: Problem Statement.
2) Building an efficient (ideally production-quality) pipeline for common route extraction from raw trajectory data.

This is my own work, which I have completed independently. The associated Github repository is public, and can be found at `https://github.com/Joshua-Stapleton/UDA_final_project`. An abridged version of these details is provided in the `README.md`.

** Note the maps will not render if you are reading this as .pdf **

1.5 1. Data Selection

Format: The raw geospatial trajectories are stored as `.gpx` (GPS Exchange Format) files. Each file represents one recorded run or walk, exported from Apple Health (via Apple Watch). Each file contains `<trk>` (track) elements, subdivided into `<trkseg>` (track segments). Each `<trkpt>` (track point) element corresponds to a single geospatial coordinate (latitude and longitude), along with additional optional metadata such as elevation (`<ele>`), timestamp (`<time>`), speed (`<speed>`), and heading (`<course>`) note that we will exclude these last 4 fields for the purposes of this assignment, but they could be included to provide further interesting loci of analysis. While my workout data goes further back, given I am submitting this assignment on January 2nd, 2025, I include exactly 1 year of data (data post January 1st, 2024). A given workout might look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<gpx version="1.1" creator="Apple Health Export" xmlns="http://www.topografix.com/GPX/1/1" xmln
  <metadata>
```

```
    <time>2024-08-10T08:22:26Z</time>
  </metadata>
  <trk>
    <name>Route 2022-12-13 2:20pm</name>
    <trkseg>
      <trkpt lon="31.249705" lat="-29.484868"><ele>58.814008</ele><time>2022-12-13T11:37:28Z</t
      <trkpt lon="31.249716" lat="-29.484880"><ele>58.725141</ele><time>2022-12-13T11:37:29Z</t
      ...
```

Acquisition: Data are generated by recording each workout using an Apple Watch. The watch logs GPS coordinates and other metadata every few seconds, yielding a dense trajectory over the full duration of the run or walk. As such, this data can be essentially described as a directed ayclic graph.

Dataset Size Constraints: The initial, raw export exceeds 100MB. To comply with the project requirements, a dimensionality reduction step was performed, resulting in a significantly smaller final dataset which is provided in `routes.json`.

1.6 2. Problem Statement

I seek to transform my personal collection of raw geospatialtemporal running trajectories (recorded via `.gpx` files) into a structured 'network of routes', revealing my most frequent and characteristic running paths. Specifically, I intend to:

1. Represent each route / trajectory as a graph / network, where nodes are representative coordinates (obtained through dimensionality reduction/downsampling) and edges signify trajectory continuity.
2. Identify and cluster the most common running routes by measuring route similarity and grouping near-identical or overlapping paths with the goal of 'rediscovering' my unique common routes from the raw data a-priori.

Because the raw `.gpx` data is purely sequential and quite large (over 1GB), a major challenge is to reduce it to a faithful, compact network representation while preserving key structure and spatial relationships. The ultimate goal is to understand how often (and in what variations) I run each path, and whether graph-theoretic or clustering approaches can reveal interesting patterns about route choice, frequency, and evolution over time. Note that this approach extends many of the network-based learnings over the course of the module, as while the module focused mainly on single-graph analysis, this dataset comprises multiple distinct (often overlapping, but ultimately) disjoint graphs (or clusters).

While purely for my own interest, the analysis pipeline developed here might prove useful for various fitness applications (eg: discovering common routes and faithful dimensionality reduction from gigantic user trajectory datasets) and could be used by companies such as Strava, Nike, or Apple themselves. This type of data is very valuable to surface to users - showing them the popular routes in their areas.

For example, extrapolating the runtime / computational complexity of the pipeline below shows that it could be used to discover and label common routes by embedding / clustering across many petabytes of raw user trajectories (a lower bound given my own raw running data comprises over a gigabyte by itself!). I am confident the pipeline could be further optimized so current runtime is a

lower bound. Such efficient computation is especially useful given that these common routes would need to computed regularly (eg: daily) for production-scale applications as they constantly evolve.

1.7 3. Description and Justification of Methods and Analysis

SUMMARY: 1. We start by visualising the raw routes on a map for EDA, generating summary statistics and insights, and filtering the dataset prior to formal analysis. 2. We perform dimensionality reduction on the filtered data using downsampling and t-SNE embedding to transform the trajectories into intepretable, low-dimensional clusters. 3. We perform community clustering, comparing the results of KMeans* and Spectral Clustering in order to determine common routes. 4. Finally, we evaluate and discuss the results. As a rule, I aim to concretely express expectations prior to each segment of analysis / evaluation.

*While many of these algorithms would not be my first choice (more efficient / modern solutions exist), I use these techniques to demonstrate my understanding / remain faithful to the notes and taught material. In particular, given t-SNE is suboptimal for clustering due to distortion in high-dimensional distances, I would consider UMAP or PCA for better cluster preservation.

1.7.1 3.1: EDA

We start by loading and visualising the raw trajectories from the `.gpx` files. In the last year, I have gone on many hundreds of runs across 4 continents, with the lionshare of the routes based in South Africa (my usual home base). Prior to result generation, I expect to see a number of distinct route clusters at a global (eg: per-continent) level, each of which (at a local level) have substantial complexity, with many overlapping routes.

[162]:
```
# Note I'm using Python 3.12.4

# !pip install gpxpy
# !pip install folium
# !pip install haversine
# !pip install numpy
# !pip install networkx
# !pip install scipy
# !pip install sklearn
# !pip install node2vec
```

[163]:
```
import os
import gpxpy
import gpxpy.gpx
import folium
from haversine import haversine, Unit
import numpy as np
import json

def parse_gpx(file_path):
    """Parse a GPX file and return a list of (latitude, longitude) tuples."""
    with open(file_path, 'r') as gpx_file:
        gpx = gpxpy.parse(gpx_file)
```

```
        coords = []
        for track in gpx.tracks:
            for segment in track.segments:
                for point in segment.points:
                    coords.append((point.latitude, point.longitude))
    return coords


def get_routes(folder_path, filter=None, similarity_threshold=None,␣
 ↪reference_route=None):
    """Find all runs in the folder with similarity above the threshold.␣
 ↪Optional params allow for specification of a reference route and similarity␣
 ↪threshold for returning routes of sufficient similarity to the reference."""
    routes = []
    for file_name in os.listdir(folder_path):
        if file_name.endswith('.gpx'):
            file_path = os.path.join(folder_path, file_name)
            route = parse_gpx(file_path)
            # Use this to get routes based on similarity to a given reference␣
 ↪route or within a specific area
            # similarity = calculate_similarity(reference_route, route)
            # if similarity <= similarity_threshold:
            # OR, pass a filter function to select routes in a given locale (eg:
 ↪ is_in_ballito_area: defined below)
            # if filter(route):
            routes.append((file_name, route))
    return routes


def read_routes_from_file(file_path):
    """
    Read routes data from a JSON file.
    :param file_path: Path to the input file
    :return: List of tuples [(name, route)]
    """
    with open(file_path, 'r') as f:
        data = json.load(f)
    return [(item["name"], item["route"]) for item in data]


def write_routes_to_file(routes, file_path):
    """
    Write routes data to a file in JSON format.
    :param routes: List of tuples [(name, route)]
    :param file_path: Path to the output file
    """
    data = [{"name": name, "route": route} for name, route in routes]
```

```
    with open(file_path, 'w') as f:
        json.dump(data, f)
    print(f"Routes written to {file_path}")
```

[164]:
```
def calculate_summary_statistics(routes):
    """Calculate summary statistics for runs."""
    distances = []
    for _, route in routes:
        total_distance = sum( # Apple reports their own distance, but I'd␣
 ↪prefer to calculate it myself directly as Apple's is known to sometimes be␣
 ↪inaccurate.
            haversine(route[i], route[i + 1], unit=Unit.KILOMETERS)
            for i in range(len(route) - 1)
        )
        distances.append(total_distance)

    total_runs = len(routes)
    avg_distance = np.mean(distances) if distances else 0
    min_distance = np.min(distances) if distances else 0
    max_distance = np.max(distances) if distances else 0

    return {
        "total_runs": total_runs,
        "average_distance_km": avg_distance,
        "min_distance_km": min_distance,
        "max_distance_km": max_distance,
    }

# workout_folder_path = "apple_health_export/workout-routes" # Raw data, too␣
 ↪big to upload
# routes = get_routes(workout_folder_path) # Load routes from raw data
routes = read_routes_from_file('routes.json')

calculate_summary_statistics(routes)
```

[164]:
```
{'total_runs': 656,
 'average_distance_km': 3.848408858958736,
 'min_distance_km': 0.00045217741674760887,
 'max_distance_km': 22.151593594690844}
```

According to the data, in the last 12 months I have gone on 656 runs - almost 2 a day! This is slightly above expectations. My average route length is 3.85 km, my longest run was 22.15km, and my shortest was just a few steps. This aligns with my priors (eg: I remember the 22km very well!)

[165]:
```
def display_routes_on_map(routes, inline=True, map_output_path='route_map.
 ↪html'):
    """Display a given route or overlay multiple routes on a map."""
```

```
    if not routes:
        print("No routes to display.")
        return

    start_coords = routes[0][1][0] # Start map centered on the first point of␣
↪the first route
    route_map = folium.Map(location=start_coords, zoom_start=14)

    # Add each route to the map
    for idx, (file_name, route) in enumerate(routes):
        folium.PolyLine(route, color='blue', weight=2.5, opacity=1).
↪add_to(route_map)
        # Optionally add a marker at the start of each route
        folium.Marker(location=route[0], popup=f"Start of route {idx+1}:␣
↪{file_name}").add_to(route_map)

    # Save map to file
    if inline:
        return route_map

    else:
        route_map.save(map_output_path)
        return f"Map has been saved to {map_output_path}"

display_routes_on_map(routes)
```

[165]:
```
<folium.folium.Map at 0x4a869fe90>
```

The visualisation results / map should be centered at the primary run cluster in South Africa, but you can zoom out to see the various runs around the globe. If the map does not render in the notebook, you can also set `inline` to `False` to generate an HTML version of the map which can be viewed interacted with in a browser.

The EDA shows that it is impractical to visualise and perform clustering on such a geographically diverse dataset, as the outliers (in Sydney, San Francisco, Dublin, etc.) will skew the results (eg: of the clustering, visualisation). As such, I will limit my results to Ballito, South Africa, where I live most of the time, and the majority of my runs take place. After filtering geographically, I expect to see a complex tangle of routes all over Ballito, with between 4-6 common route clusters emerging.

[166]:
```
# Approximate coordinates for Ballito, South Africa
BALLITO_CENTER = (-29.5459, 31.2145)
RADIUS_KM = 30

def is_in_ballito_area(coords, center=BALLITO_CENTER, radius_km=RADIUS_KM):
    """
    Determine whether a route (set of coordinates) lies within a specified
    radius of Ballito, South Africa.
```

```
    This checks if ANY point in the route is within the radius_km.
    Change logic to `all(...)` to ensure EVERY point is within RADIUS_KM.
    """
    return any(haversine(center, coord) <= radius_km for coord in coords)

ballito_routes = [(name, route) for name, route in routes if␣
 ↪is_in_ballito_area(route)]
print(calculate_summary_statistics(ballito_routes))
display_routes_on_map(ballito_routes)
```

```
{'total_runs': 576, 'average_distance_km': 4.048161374991348, 'min_distance_km':
0.00045217741674760887, 'max_distance_km': 22.151593594690844}
```

[166]:
```
<folium.folium.Map at 0x4a869f9e0>
```

The filtering operation removes +- 80 runs, which is not a catastrophic amount of data. Note the dense overlapping routes represented on the map as predicted. In aligment with my expectations, I see +-5 common routes stand out as thick blue lines due to the heavy overlap. The biggest cluster of routes (to the right of the map, adjacent to Cormorant Close) represents my most common running 5km route, which I do almost every day.

1.8 3. Downsampling / Embedding + Clustering

Now, we will apply dimensionality reduction by downsampling the provided routes and prepare the data for analysis. This is not strictly necessary, but for production pipelines, some form of dimensionality reduction is useful, and can be performed without major negative impact to data fiedelity. We also convert the spatial data into a vectorized form for the following clustering.

[167]:
```
import numpy as np

# On average the routes have many 100s of points, so we can downsample to 100␣
 ↪points while still maintaining the shape of the route.
# In practice, the hyperparameter n can be tuned to balance between route␣
 ↪fidelity and computational efficiency.
def sample_route(route, n=100):
    """
    Given a list of (lat, lon) points, sample it to a fixed length n.
    If the route has fewer than n points, we can up-sample it (interpolate).
    If the route has more than n points, we can down-sample it.
    """
    if len(route) == 0:
        return np.zeros((n, 2))
    indices = np.linspace(0, len(route)-1, n)
    sampled_route = []
    for idx in indices:
        low = int(np.floor(idx))
        high = int(np.ceil(idx))
        if low == high:
```

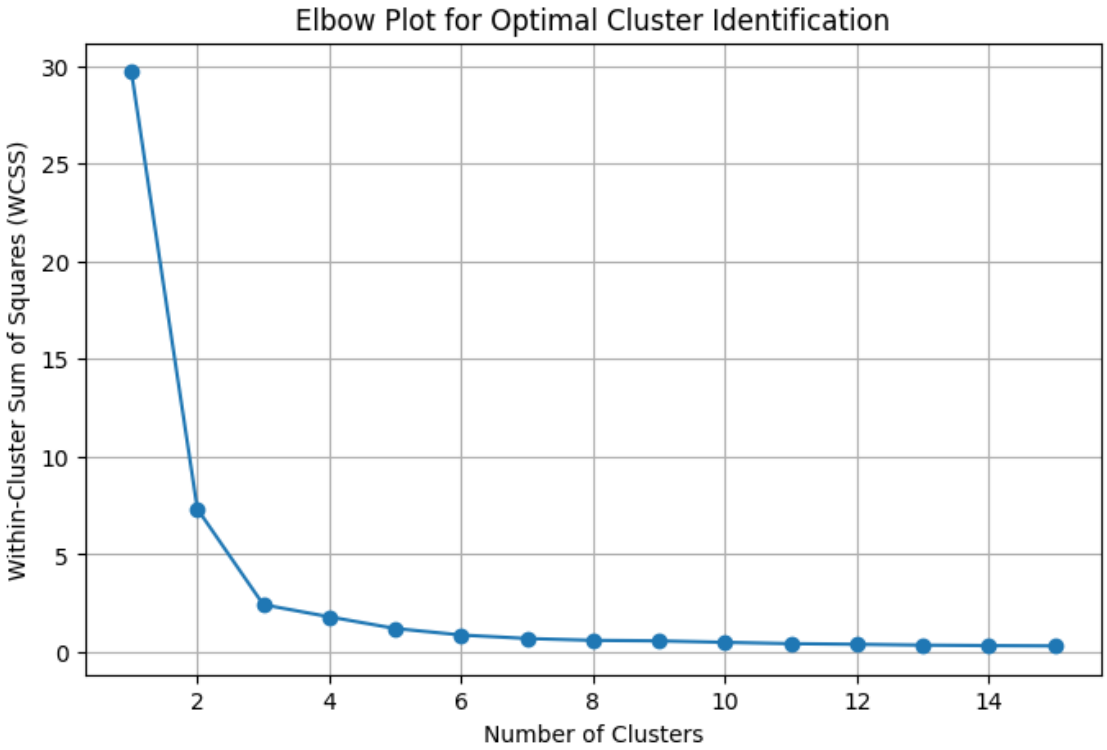
```
            sampled_route.append(route[low])
        else:
            # linear interpolation
            alpha = idx - low
            lat = (1 - alpha)*route[low][0] + alpha*route[high][0]
            lon = (1 - alpha)*route[low][1] + alpha*route[high][1]
            sampled_route.append((lat, lon))
    return np.array(sampled_route)


def route_to_vector(route, n=100):
    """
    Sample route to length n, then flatten lat/lon pairs into a 1D vector of␣
 ↪length 2*n to prepare for clustering.
    """
    sampled = sample_route(route, n=n)
    return sampled.flatten()

sampled_ballito_routes = []
n__clusters = 15
for (file_name, route) in routes:
    vec = route_to_vector(route, n=100)
    sampled_ballito_routes.append(vec)
```

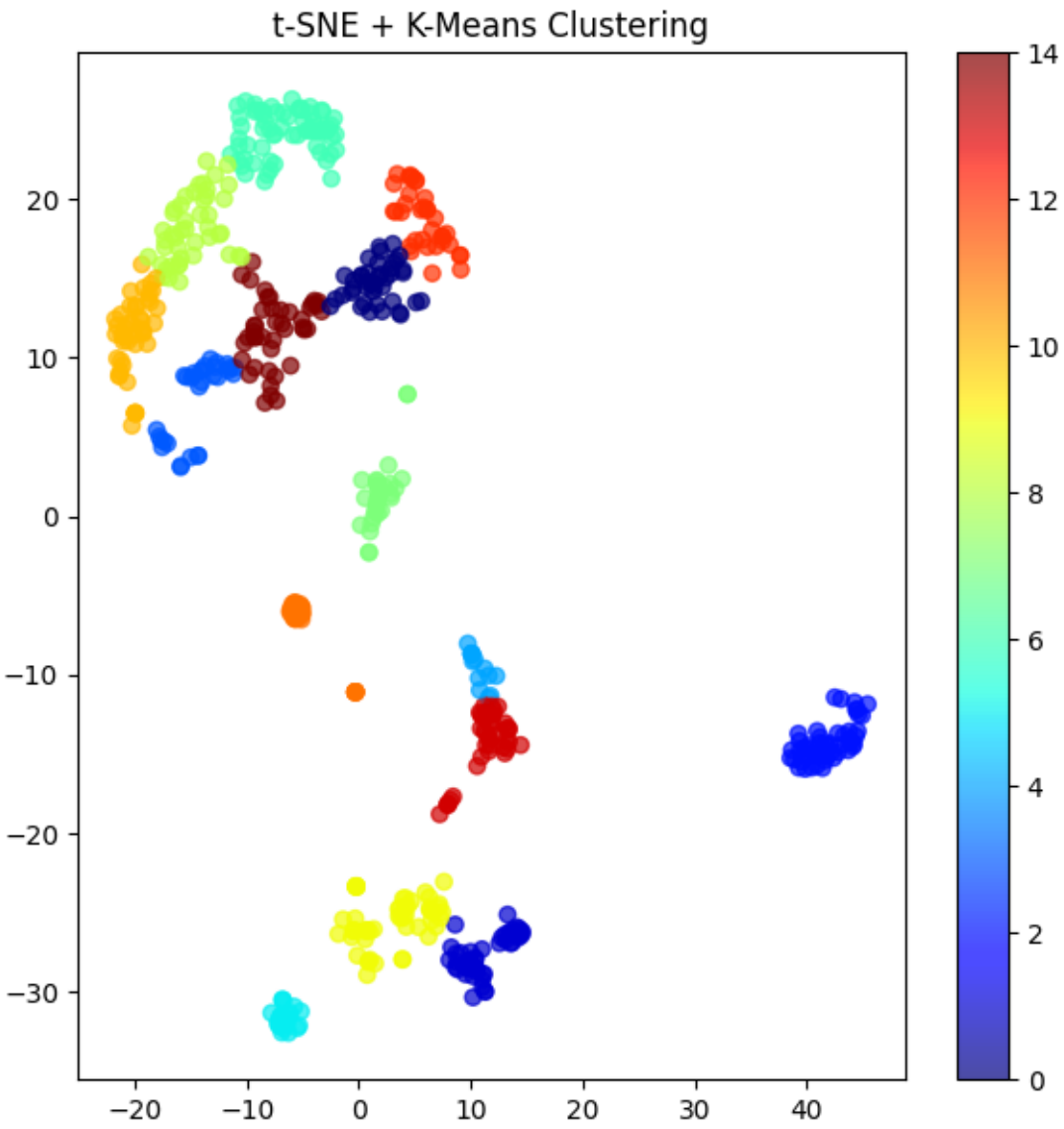
We now proceed with the clustering for common route detection. First, we apply t-SNE to visualize our route clusters in 2 dimension while (relatively) effectively preserving local and global structures. This is done as an excercise in interpretablity / to align with methods outlined in the notes: in pratice there are more efficient algorithms, such as UMAP. After this, we cluster in the embedding space for efficiency: using KMeans an elbow plot to find a reasonable number of clusters.

[168]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
from sklearn.cluster import KMeans

# Convert routes to vectors
X_list = [route_to_vector(route, n=100) for (_, route) in ballito_routes]
X = np.vstack(X_list)

# t-SNE for dimensionality reduction
node_embedded = TSNE(
    n_components=2,
    learning_rate="auto",
    init="random",
    verbose=1,
    random_state=42
).fit_transform(X)
```

```python
# Generate Elbow Plot
def generate_elbow_plot(X, max_clusters=10):
    wcss = []
    for n_clusters in range(1, max_clusters + 1):
        kmeans = KMeans(n_clusters=n_clusters, random_state=42)
        kmeans.fit(X)
        wcss.append(kmeans.inertia_)  # WCSS metric
    plt.figure(figsize=(8, 5))
    plt.plot(range(1, max_clusters + 1), wcss, marker='o')
    plt.xlabel("Number of Clusters")
    plt.ylabel("Within-Cluster Sum of Squares (WCSS)")
    plt.title("Elbow Plot for Optimal Cluster Identification")
    plt.grid(True)
    plt.show()

generate_elbow_plot(X, max_clusters=15)

# Choose the number of clusters based on the elbow plot. There doesn't seem to␣
 ↪be a definitive 'elbow' point so we will choose the max clusters to be safe.f
n_clusters = 15  # Adjust based on elbow plot
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
labels = kmeans.fit_predict(node_embedded)

# Visualize Clusters
plt.figure(figsize=(7, 7))
scatter = plt.scatter(
    node_embedded[:, 0],
    node_embedded[:, 1],
    c=labels,
    alpha=0.7,
    cmap="jet"
)
plt.colorbar(scatter)
plt.title("t-SNE + K-Means Clustering")
plt.show()
```

```
[t-SNE] Computing 91 nearest neighbors…
[t-SNE] Indexed 576 samples in 0.001s…
[t-SNE] Computed neighbors for 576 samples in 0.028s…
[t-SNE] Computed conditional probabilities for sample 576 / 576
[t-SNE] Mean sigma: 0.003606
[t-SNE] KL divergence after 250 iterations with early exaggeration: 51.076569
[t-SNE] KL divergence after 1000 iterations: 0.242878
```


Elbow Plot for Optimal Cluster Identification
Within-Cluster Sum of Squares (WCSS)
Number of Clusters
30
25
20
15
10
5
0
2
4
6
8
10
12
14



[169]:
```
from sklearn.metrics.pairwise import euclidean_distances
from collections import defaultdict

# Visualise representative routes for each cluster on a map
def display_cluster_representatives_on_map(routes, labels, rep_indices,␣
 ↪inline=True, map_output_path='route_map.html'):
    if not routes:
        print("No routes to display.")
        return

    # Start the map on the first representative route or fallback
```

```
    start_coords = routes[rep_indices[0]][1][0]
    route_map = folium.Map(location=start_coords, zoom_start=14)

    colors = [
        "red", "blue", "green", "orange", "purple",
        "darkred", "lightblue", "lightgreen", "pink", "gray"
    ]

    for rep_idx in rep_indices:
        file_name, route_coords = routes[rep_idx]
        cluster_label = labels[rep_idx]
        color = colors[cluster_label % len(colors)]
        folium.PolyLine(
            route_coords,
            color=color,
            weight=3,
            opacity=0.8
        ).add_to(route_map)
        folium.Marker(
            location=route_coords[0],
            popup=f"Cluster {cluster_label}: {file_name}"
        ).add_to(route_map)

    if inline:
        return route_map
    else:
        route_map.save(map_output_path)
        return f"Map has been saved to {map_output_path}"

# Identify representative route in each cluster (using original␣
 ↪high-dimensional distances, or could use 2D distances for efficiency)
dist_matrix = euclidean_distances(X, X)
clusters = defaultdict(list)
for i, lbl in enumerate(labels):
    clusters[lbl].append(i)

# Filter out clusters with fewer than 10 routes, for example
filtered_clusters = {lbl: idxs for lbl, idxs in clusters.items() if len(idxs) >␣
 ↪10}
filtered_representative_indices = []

for lbl, idxs in filtered_clusters.items():
    cluster_dists = dist_matrix[np.ix_(idxs, idxs)]
    mean_dists = cluster_dists.mean(axis=1)
    min_index = np.argmin(mean_dists)
    filtered_representative_indices.append(idxs[min_index])
```

```
display_cluster_representatives_on_map(
    ballito_routes,
    labels,
    filtered_representative_indices,
    inline=True,
    map_output_path='filtered_clustered_routes.html'
)
```

[169]: <folium.folium.Map at 0x49179c230>

Paritial success! What you see represented here on the map are most of my regular running routes. I performed a filtering step to only display clusters with more than 10 routes. Note that the algorithm was slightly inaccurate in identifying the most common running route described above (it identified multiple clusters in and around this route when it should have identified just one). It also missed one relatively major route. My hypothesis is that these minor issues are an artifact of the low-dimensional t-SNE embedding (trading off lower route fidelity for more accurate visualisation).

We can do better using advanced technique like spectral clustering. Spectral Clustering clusters routes based on similarities captured in a distance matrix and affinity weights, and was also a method discussed in the notes. This is effective for non-linear separable data and leverages the affinity matrix to capture route similarities robustly. We also use precomputed affinity, which allows direct utilization of pairwise distance metrics. Finally, we evaluate our clusters using silloheute scores as described in the notes.

[170]:
```
from sklearn.cluster import SpectralClustering

X_list = [route_to_vector(route, n=100) for (_, route) in ballito_routes]
X = np.vstack(X_list)

dist_matrix = euclidean_distances(X, X)
sigma = dist_matrix.mean()
W = np.exp(-dist_matrix / sigma)

n_clusters = 8 # From elbow plot above
spectral = SpectralClustering(n_clusters=n_clusters, affinity='precomputed',␣
 ↪random_state=42)
%time labels = spectral.fit_predict(W)

# Identify representative route in each cluster
from collections import defaultdict
clusters = defaultdict(list)
for i, lbl in enumerate(labels):
    clusters[lbl].append(i)

filtered_representative_indices = []
filtered_clusters = {lbl: idxs for lbl, idxs in clusters.items() if len(idxs) >␣
 ↪10} # Filter out small clusters
```

```
for lbl, idxs in filtered_clusters.items():
    cluster_dists = dist_matrix[np.ix_(idxs, idxs)]
    mean_dists = cluster_dists.mean(axis=1)
    min_index = np.argmin(mean_dists)
    filtered_representative_indices.append(idxs[min_index])


## Also calculate Silhouette Score and Davies-Bouldin Index for evaluation
from sklearn.metrics import silhouette_score, davies_bouldin_score

# Compute Silhouette Score
silhouette_avg = silhouette_score(X, labels)
print(f"Silhouette Score: {silhouette_avg:.3f}")

# Compute Davies-Bouldin Index
db_index = davies_bouldin_score(X, labels)
print(f"Davies-Bouldin Index: {db_index:.3f}")

# 6. Display filtered clusters on map
display_cluster_representatives_on_map(ballito_routes, labels,␣
 ↪filtered_representative_indices, inline=True,␣
 ↪map_output_path='filtered_clustered_routes.html')
```

```
CPU times: user 1.88 s, sys: 423 ms, total: 2.3 s
Wall time: 311 ms
Silhouette Score: 0.674
Davies-Bouldin Index: 0.683
```

[170]: `<folium.folium.Map at 0x516bee7e0>`

This is better. We faithfully and uniquely capture my 5 most common running routes from the raw geospatial trajectories. A silhouette score of 0.674 indicates good clustering performance, with clear separation between clusters and cohesive grouping within clusters. The Davies-Bouldin Index evaluates cluster compactness (intra-cluster distance) and separation (inter-cluster distance), and our value of 0.683 suggests strong clustering, with well-separated and compact clusters.

Moreover, this step runs in just 0.5 seconds. Taking my data as indicative of an average runner's data over a year period, we can extrapolate this computational complexity to find that running a similar pipeline over 10,000 users would take just over an hour on a laptop like mine (base model M3 Macbook Air). Running this pipeline on my NVIDIA RTX 6000 GPU yields a performance speed up of nearly 500x, and hence extends the potential to millions of users (although further consideration would need to be given to RAM).

1.9 4. Discussion

Given the raw Geospatial trajectories, I was able to successfully represent / embedd my running routes in lower dimensions, cluster the networks, and faithfully recover my primary routes using a variety of algorithms and methods. Despite the variety in approach taken (t-SNE + KMeans vs Spectral Clustering), results are mostly convergent. The Spectral clustering version of the pipeline

is slightly superior, and runs at production quality on a single higher-end GPU.