Moving between geographic data structures for advanced spatial analysis

Thomas A Statham

Outline

1. Introduction: data structures
2. Showcase how fundamental data structures are in spatial analysis; yet they are often overlooked
3. How to move between geographic data structures
4. (Hopefully) make you question what spatial and non-spatial data structures you are using in your next analysis.

Introduction

In the most general sense, a data structure is:

> *any data representation and its associated operations.*

In geo, there are 3 common data structures:

1. Table
2. Surface
3. Graph

Geographic data structures should be selected/leveraged as part of spatial analysis:

> *To organise and embed spatial relationships as a first class citizen*

Get directions from Engine Shed to the Pub, the Sidings using a routing engine: OpenRouteService.

Before making the query, we first need to grab the coordinates for these two locations using a Geocoder.

In [2]:

```
engine_shed = ox.geocoder.geocode(query="Engine Shed, Bristol, BS1 6QH")  # uses the o
pub = ox.geocoder.geocode(query="The Sidings, Bristol BS1 6PL")
type(pub)
```

Out[2]:

```
tuple
```

We then use OpenRouteService (ORS) Python API, which gives access to this routing engine API.

In [3]:

```
coords = [list(engine_shed)[::-1], list(pub[::-1])]  # transform tuple to list and rev
```

In [4]:

```
client = ors.Client(key=os.environ.get("ORS_API"))  # Define the client using the ORS .
route = client.directions(coordinates=coords, profile="foot-walking")
```

Python API returns a dictionary with different information, including a summary of the distance and duration.

In [5]:

```
route["routes"][0]["summary"]  # returns a dictionary
```

Out[5]:

```
{'distance': 199.5, 'duration': 143.6}
```

Next, we have to decode Google's polyline strings to list to map our route.

In [6]:

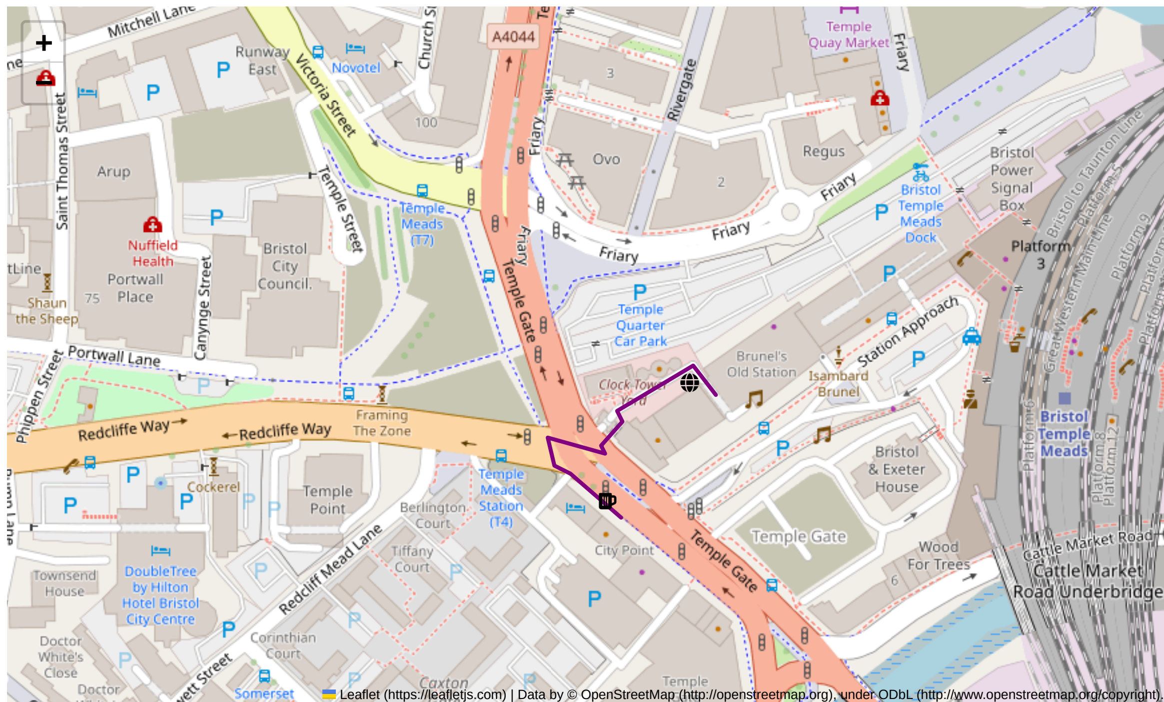
```
decoded_linstring = ors.convert.decode_polyline(route["routes"][0]["geometry"])["coord
folium_coords = [i[::-1] for i in decoded_linstring]  # reverse list again
```

In not many lines of Python, we have traversed through multiple data structures, data types and algorithms to get our answer: give me directions from the engine shed to the pub.

In [7]:

```
m = folium.Map(location=[engine_shed[0], engine_shed[1]], zoom_start=17)
folium.Marker(location=[engine_shed[0], engine_shed[1]], icon=folium.Icon(color="green
folium.Marker(location=[pub[0], pub[1]], icon=folium.Icon(color="blue",icon="beer", pr
folium.PolyLine(locations=folium_coords, color="purple").add_to(m)
m
```

Out[7]:



Moving between Tables (Polygons, Points) and Surfaces

Say we want to create a simple linear model that predicts crime aggregated at Uber H3 polygons from Jan 23 to now using some features. However, the features are not aggregated to the same level of spatial support. We can use some smart techniques to transfer the values from one level of spatial support to another.

The bike routes are currently imported as a table, which we will later convert to a surface to perform some analysis.

In [9]:

```
crime = duckdb.sql(
    "SELECT Longitude as lon, Latitude as lat "
    "FROM read_csv_auto('/home/tastatham/site/content/blog/crime/*/*.csv') "
    "WHERE lon IS NOT NULL AND lat IS NOT NULL"
).df()
crime.head()
```

Out[9]:

	lon	lat
0	-0.831066	51.825189
1	0.139935	51.563952
2	-2.516590	51.417444
3	-2.509285	51.409716
4	-2.491420	51.423811

I downloaded data for the Avon and Somerset force, but wasn't true! Here I use the indices of Multiple Deprivation 2019 to clip the spatial points, which I will use later to predict crime in Bristol.

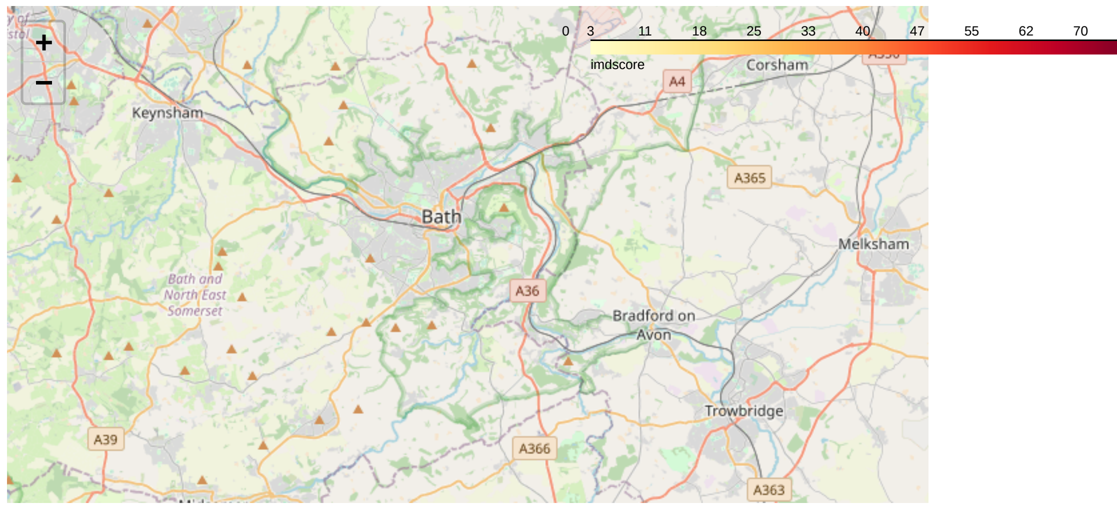
In [10]:

```
imd = gpd.read_file(
    "https://services2.arcgis.com/a4vR8lmmksFixzmB/arcgis/rest/services/Indices_Of_Dep
)
imd = imd[["LSOA11CD", "LSOA11NM", "IMDSCORE", "geometry"]]
imd.columns = map(str.lower, imd.columns)
```

In [11]:

```
m = folium.Map(location=[engine_shed[0], engine_shed[1]], zoom_start=11)
imd[["imdscore", "geometry"]].explore(m=m, column="imdscore", cmap="YlOrRd")
```

Out[11]:



Leaflet (https://leafletjs.com) | Data by © OpenStreetMap (http://openstreetmap.org), under ODbL (http://www.openstreetmap.org/copyright).

I will also convert the data to British National Grid for support later analysis.

In [12]:

```
imd_bng = imd.to_crs(27700)

crime = gpd.GeoDataFrame(
    data=crime,
    geometry=gpd.points_from_xy(crime["lon"], crime["lat"]),
    crs=4326,
).to_crs(27700).clip(imd_bng)
m = folium.Map(location=[engine_shed[0], engine_shed[1]], zoom_start=11)
crime.explore(m=m)
```

Out[12]:

Leaflet (https://leafletjs.com) | Data by © OpenStreetMap (http://openstreetmap.org), under ODbL (http://www.openstreetmap.org/copyright).

WorldPop 2020 Constrained

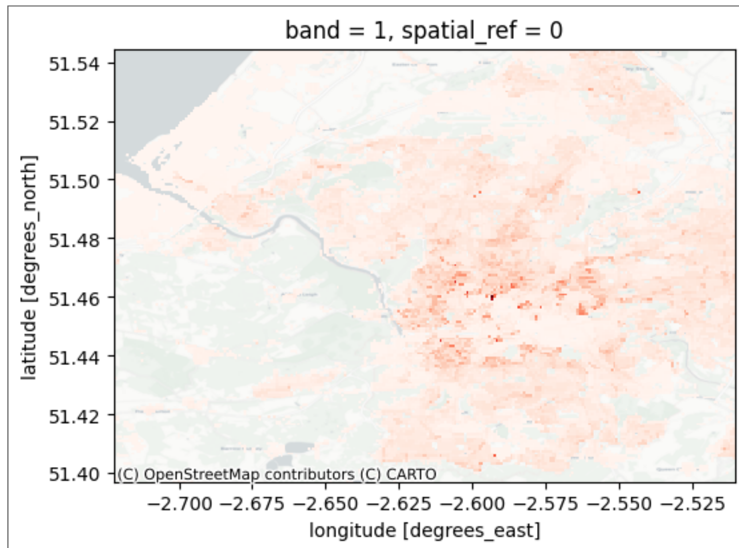
In [13]:

```
#!wget https://data.worldpop.org/GIS/Population/Global_2000_2020_Constrained/2020/BSGM
worldpop = rio.open_rasterio("gbr_ppp_2020_constrained.tif", masked=False)
xmin, ymin, xmax, ymax = imd.total_bounds
worldpop = (worldpop
    .rio.clip_box(xmin, ymin, xmax, ymax)
    .where(worldpop!= worldpop.rio.nodata)
)
```

Which looks something like...

In [14]:

```
fig, ax = plt.subplots(1,1, figsize=(6,6), facecolor="white")
worldpop.plot(ax=ax, cmap="Reds", add_colorbar=False)
ctx.add_basemap(ax=ax, crs=worldpop.rio.crs, source=ctx.providers.CartoDB.Positron)
```



Our first feature is population, which we will grab by sampling from Worldpop at the point level.

In [16]:

```
coords = crime[["lon", "lat"]].values.tolist()
pop = [x for x in worldpop.sample(coords)]
crime["pop"] = np.array(pop)
crime.head()
```

Out[16]:

	lon	lat	geometry	pop
69659	-2.593910	51.420584	POINT (358799.019 169230.950)	35.656986
45029	-2.596468	51.401671	POINT (358604.026 167129.013)	NaN
31255	-2.594939	51.404547	POINT (358712.985 167447.998)	33.756023
6007	-2.594939	51.404547	POINT (358712.985 167447.998)	33.756023
6004	-2.594939	51.404547	POINT (358712.985 167447.998)	33.756023

Now we create the Uber H3 polygons for Bristol.

In [17]:

```
h3 = h3fy(imd_bng, 9).reset_index()
h3.head()
```

Out[17]:

	hex_id	geometry
0	89195876933ffff	POLYGON ((358991.624 167427.732, 358845.551 16...
1	89195876127ffff	POLYGON ((356867.933 172888.200, 356721.847 17...
2	891958764bbffff	POLYGON ((356265.765 178199.587, 356119.697 17...
3	8919587640bffff	POLYGON ((355286.137 178102.760, 355140.048 17...
4	89195839287ffff	POLYGON ((359385.229 174629.761, 359239.207 17...

Then aggregate and merge the crime data to h3 polygons using a spatial join.

In [18]:

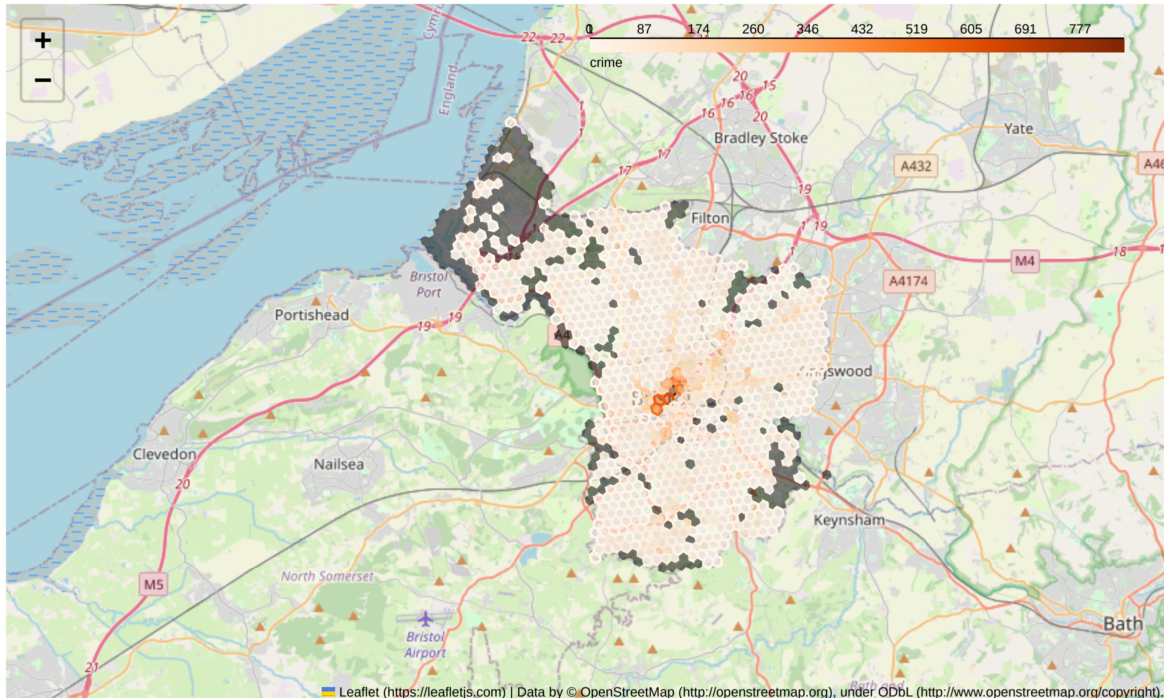
```
crime["crime"] = 1
h3_crime_sjoined = gpd.sjoin(crime, h3).groupby("hex_id")[["crime", "pop"]].sum().rese
h3_crime = pd.merge(h3, h3_crime_sjoined, on="hex_id", how="outer")
```

Now check it out..

In [19]:

```
m = folium.Map(location=[engine_shed[0], engine_shed[1]], zoom_start=11)
h3_crime.explore(m=m, column="crime", cmap="Oranges")
```

Out[19]:



There's lots of different areal interpolation methods, but the `choice of areal targets plays the biggest role in minimising bias and uncertainty.`

In [21]:

```
gdf_h3_updated = aw._areal_weighting(
    sources=imd_bng,
    targets=h3_crime,
    extensive=None,
    intensive="imdscore",
    weights="sum",
    sid="lsoa11cd",
    tid="hex_id",
    geoms=True,
    all_geoms=False,
)
```

This spatial feature engineering through map matching allows us to add new features to statistical models.

In [22]:

```
h3_all = pd.merge(h3_crime, gdf_h3_updated[["hex_id", "imdscore"]], on="hex_id", how="
model = smf.ols(formula='crime ~ pop + imdscore', data=h3_all).fit()
model.summary()
```

Out[22]:

OLS Regression Results

Dep. Variable:	crime	R-squared:	0.544
Model:	OLS	Adj. R-squared:	0.543
Method:	Least Squares	F-statistic:	528.3
Date:	Wed, 06 Sep 2023	Prob (F-statistic):	1.01e-151
Time:	22:42:59	Log-Likelihood:	-4600.6
No. Observations:	888	AIC:	9207.
Df Residuals:	885	BIC:	9222.
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	7.2519	2.983	2.431	0.015	1.398	13.106
pop	0.0148	0.000	31.452	0.000	0.014	0.016
imdscore	0.3725	0.100	3.738	0.000	0.177	0.568

Omnibus:	1285.596	Durbin-Watson:	2.029
Prob(Omnibus):	0.000	Jarque-Bera (JB):	369101.988
Skew:	8.086	Prob(JB):	0.00
Kurtosis:	101.561	Cond. No.	7.16e+03

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 7.16e+03. This might indicate that there are
strong multicollinearity or other numerical problems.

Moving between Tables and graphs

In this example a user is interested in checking out the cycle routes that start from within the Bristol boundary.
To do this I grab the Sustran cycle routes

In [23]:

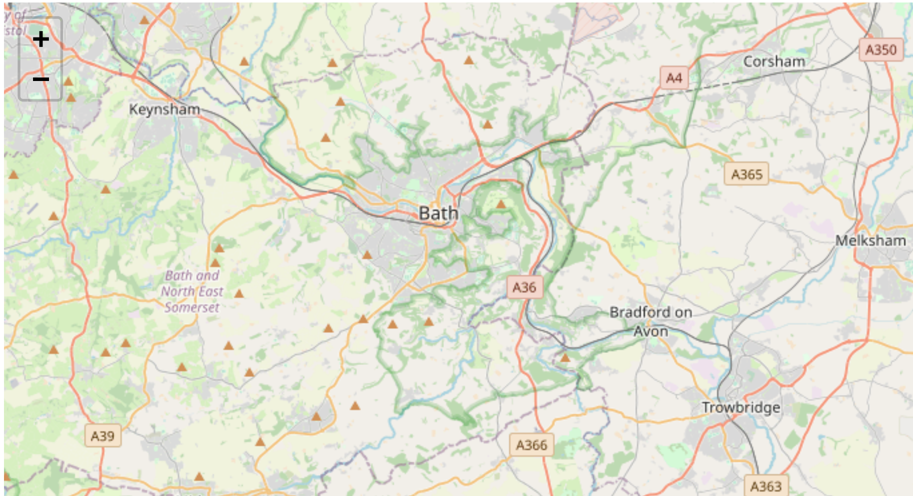
```
bike_routes = gpd.read_file("https://maps2.bristol.gov.uk/server2/rest/services/ext/ll_
bike_routes= bike_routes[["ROUTE_NAME", "DIFFICULTY", "DISTANCE", "geometry"]]
bike_routes.columns = map(str.lower, bike_routes.columns)
bike_routes["distance"] = bike_routes[["distance"]].apply(pd.to_numeric, errors="coerc
```

and clip out bike routes that completely fall outside of bris local authority

In [24]:

```
clipped_bike_routes = gpd.sjoin(bike_routes, imd).drop_duplicates(subset="route_name")
m = folium.Map(location=[engine_shed[0], engine_shed[1]], zoom_start=11)
clipped_bike_routes.explore(m=m)
```

Out[24]:



Leaflet (https://leafletjs.com) | Data by © OpenStreetMap (http://openstreetmap.org), under ODbL (http://www.openstreetmap.org/copyright).

One user may be interested in selecting the smallest route, whilst another user may wish to find the largest route.

This is a pretty simple non-spatial query.

In [25]:

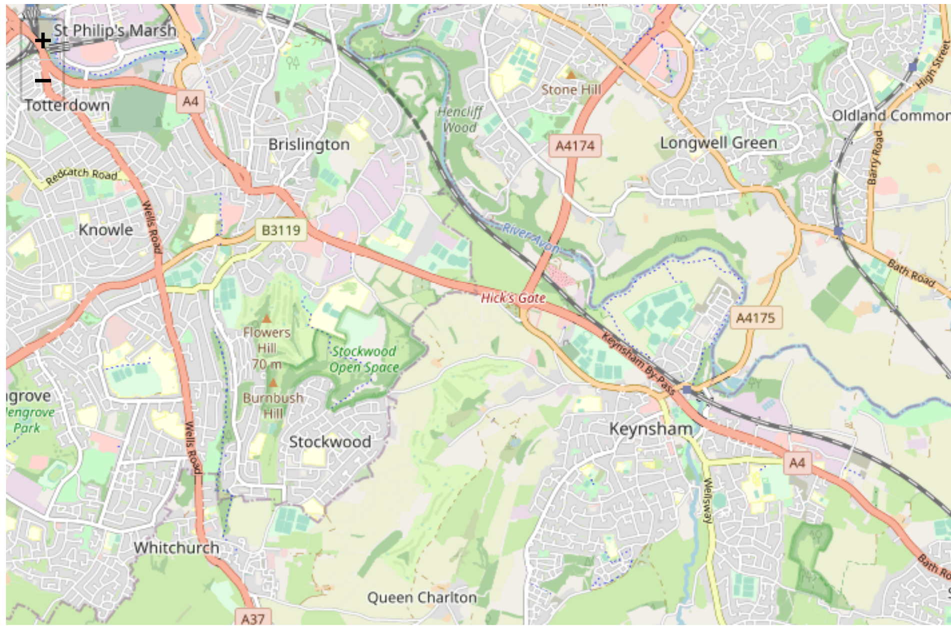
```
smallest_dis = clipped_bike_routes[clipped_bike_routes["distance"] == clipped_bike_rou
largest_dis = clipped_bike_routes[clipped_bike_routes["distance"] == clipped_bike_rout
```

which looks like...

In [26]:

```
m = folium.Map(location=[engine_shed[0], engine_shed[1]], zoom_start=13)
gpd.GeoDataFrame(smallest_dis, geometry="geometry", crs=4326).explore(m=m, color="red"
gpd.GeoDataFrame(largest_dis, geometry="geometry", crs=4326).explore(m=m, color="blue"
```

Out[26]:



Leaflet (https://leafletjs.com) | Data by © OpenStreetMap (http://openstreetmap.org), under ODbL (http://www.openstreetmap.org/copyright).

An alternative user may be interested in a more complex question. This user wants to split all of the routes over the course of a weekend: Saturday and Sunday. So we need to find out how to split the routes over Saturday and Sunday,

You could use traditional clustering techniues like k-means clustering, where k=2, but this requires defining point that represents the LineString. Instead, we can leverage the power of graph algorithms to partition our network into two pairs of nodes.

In [27]:

```
G = momepy.gdf_to_nx(clipped_bike_routes.to_crs(27700))
```

Then use the Kernighan–Lin algorithm to partition the network into two pairs of nodes...

In [28]:

```
bisect = nx.community.kernighan_lin_bisection(G, seed=0)
bisect
```

Out[28]:

```
({(347684.088364584, 170781.23213460977),
  (354211.04278352734, 184715.417715867),
  (358771.5173283669, 172558.91904138785),
  (360135.780452416, 173164.18407326954),
  (360470.74972742616, 173229.27782924206),
  (362907.737672483, 172517.46147224365)},
 {(354979.5526282616, 175881.63394465472),
  (358180.4017600689, 169196.47302706086),
  (358764.8240964034, 172484.9220697746),
  (360470.90699713555, 173228.13219716027),
  (361360.5410734009, 169837.463854736),
  (372232.5713307257, 165236.69615670637)})
```

But this returns a set of two dictionaries of the last node along each LineString.

In [29]:

```
group1 = pd.DataFrame(data=list(bisect[0]), columns=["lat", "lon"])
group2 = pd.DataFrame(data=list(bisect[1]), columns=["lat", "lon"])

group1["group"] = "Sunday"
group2["group"] = "Saturday"

bisect_df = pd.concat([group1, group2], axis=0)
```

Let's plot the routes for Saturday and Sunday

In [30]:

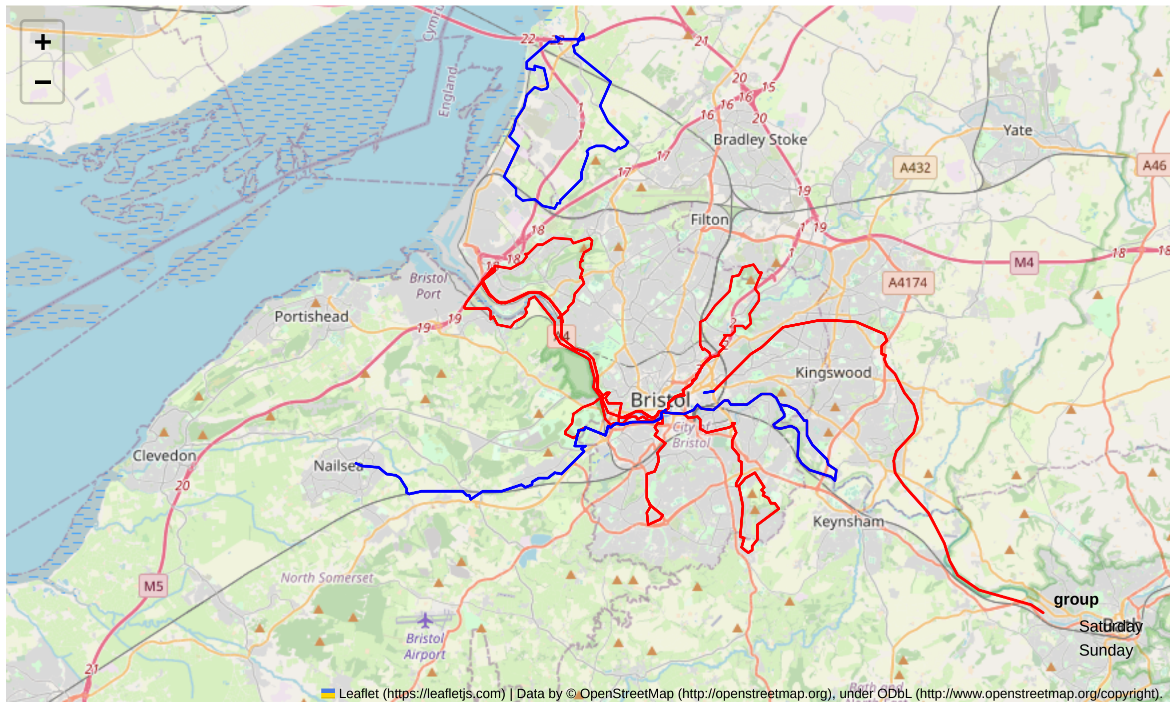
```
bisect_gdf = gpd.GeoDataFrame(
    data=bisect_df,
    geometry=gpd.points_from_xy(bisect_df["lat"], bisect_df["lon"]),
    crs=27700,
)
clipped_bike_routes_updated = gpd.sjoin_nearest(clipped_bike_routes.drop("index_right"
```

Well, I wouldn't recommend this to someone without a very high level of fitness.

In [31]:

```
m = folium.Map(location=[engine_shed[0], engine_shed[1]], zoom_start=11)
clipped_bike_routes_updated.explore(m=m, column="group", cmap=["#FF0000", "#0000ff"])
```

Out[31]:



Summary: Geographic data structures matter

There is no right or wrong answer when selecting a data structure. `Many geospatial problems can be solved using different representations:`

> *Ultimately, as a Geospatial expert, it's up to you how and when to leverage different data structures.*