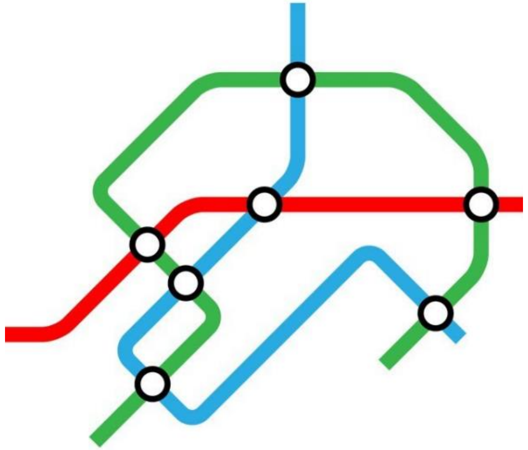
Working with Spatial Data. Network Analysis

Reading, exploring and analyzing, feature extraction



Yordan Darakchiev
Technical Trainer
iordan93@gmail.com

sli.do

#DataScience

Table of Contents

Geospatial Data

Exploring, analyzing and visualizing

Geospatial Data

- Data that has a geographic component to it
  - Most commonly: coordinates (latitude, longitude)
  - Sometimes: country, city, ZIP code, address
  - Not necessarily on Earth ([Google Mars](#))
- Sources
  - Satellite images
  - GPS data
  - Geotagging (e.g., photos at Facebook)
  - Manual entry, etc.
- Working with spatial data isn't trivial...
  - E.g., we need geometry on a sphere to calculate distances
  - ... but we have libraries that make our lives easier

Reading and Exploring Geospatial Data

- In some cases, we have convenient datasets
- In other cases, it's in specific formats
  - GeoJSON, Shapefile, KML, etc.
  - Some libraries (like geopandas) can read these automatically
- Data cleaning
  - Non-spatial columns: proceed as usual
    - Tidy up the data, impute or remove missing values, explore outliers, normalize columns, etc.
  - Spatial columns: fixing or changing coordinates is easier when you visualize them
- Exploratory data analysis
  - Most commonly: look for clusters and other patterns
  - Also: compare attributes across different regions
    - E.g., income by country

Example: Earthquake Data

- Dataset: `earthquakes.csv`, info
  - Read the dataset, look at missing values
  - Leave only columns you're interested in

```
["Date", "Time", "Latitude", "Longitude", "Magnitude", "Depth"]
```

  - Explore the dataset
    - Examples: how is the magnitude distributed? When and where did the most powerful earthquakes happen? What are the recent ones?
  - Perform additional data cleaning, exploration and visualization of the non-spatial columns
  - Fix dates (remove invalid date format, convert to `datetime`)

```
dt_info = earthquake_data.Date + " " + earthquake_data.Time
earthquake_data = earthquake_data.drop(
  index = dt_info[dt_info.str.len() > 20].index)
earthquake_data["DateTime"] = pd.to_datetime(
  earthquake_data.Date + " " + earthquake_data.Time)
```

Plotting Data on a Map

- To plot data, we'll use the geopandas package

```
conda install -c conda-forge geopandas
conda install -c conda-forge geodatasets
```

```
import geopandas as gpd
```

- Setting up and displaying a world map

```
land = gpd.read_file(geodatasets.data.naturalearth.land.url)

land.plot(figsize = (20, 10), color = "coral", facecolor = "aqua")
plt.gca().set_facecolor("aqua")
plt.show()
```

- Projections (docs, EPSG)
  - Different ways to show a sphere in a 2D plane
  - **Every projection has distortions**

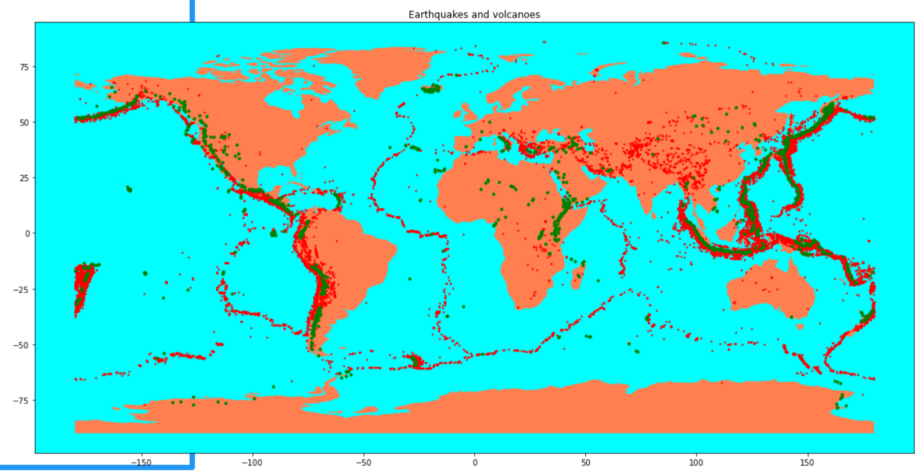
Plotting Data on a Map (2)

- Data (features)
  - Use as common pandas columns (`Series`)
- Selection, projection, grouping, etc.
  - Work as expected
- Geometry
  - Points, lines, polygons
  - [Quick guide]
  - Contains useful info and methods, such as area, bounds, centroids and distances
  - Allows for **very** easy plotting
- Using / changing projections
  - `dataframe.to_crs(name)`
  - Commonly used with EPSG (4326 by default)

Adding Data on Volcanoes

- Dataset: `volcanoes.csv`, [info]
- Read the data and convert to x, y coordinates
- Plot just after the earthquakes
  - And before the "map decorations"

```
volcano_data = pd.read_csv(...)
geometry = [Point(xy) for xy in zip(
    volcano_data.Longitude, volcano_data.Latitude)
]
volcano_data = gpd.GeoDataFrame(volcano_data,
    geometry = geometry, crs = "EPSG:4326")
land.plot(figsize = (20, 10), color = "coral")
plt.gca().set_facecolor("aqua")
earthquake_data.plot(ax = plt.gca(),
    c = "r", markersize = 2)
volcano_data.plot(...)
plt.show()
```



Drawing a Choropleth Map

- Like a heatmap
  - Shows different countries (or US states) in different colors according to a scale
- Dataset: `ufo_sightings_scrubbed.csv`, info
  - Clean the data (careful with "longitude")
  - Narrow down the data to US

```
ufos = pd.read_csv("ufo_sightings_scrubbed.csv", low_memory = False)
ufos = ufos[["datetime", "country", "state", "latitude", "longitude "]]
ufos.columns = ["datetime", "country", "state", "latitude", "longitude"]

ufos = ufos[ufos.country == "us"]
```

- Use the shape files from States_shapefile

Drawing a Choropleth Map (2)

- Read the shape file
- Read the state names from `state_names.csv`
  - Use them to add the full names to the UFOs dataset

```
states = gpd.read_file("States_shapefile.shp")
state_names = pd.read_csv("states.csv")
state_names.abbreviation = state_names.abbreviation.str.lower()
state_names_dict = {state.abbreviation: state["name"]
  for index, state in state_names.iterrows()}

ufos.state.replace(state_names_dict, inplace = True)
```

- Get the number of sightings per state

```
num_sightings_by_state = ufos.groupby("state").size()
num_sightings_by_state.state = num_sightings_by_state.state.str.upper()
```

Drawing a Choropleth Map (3)

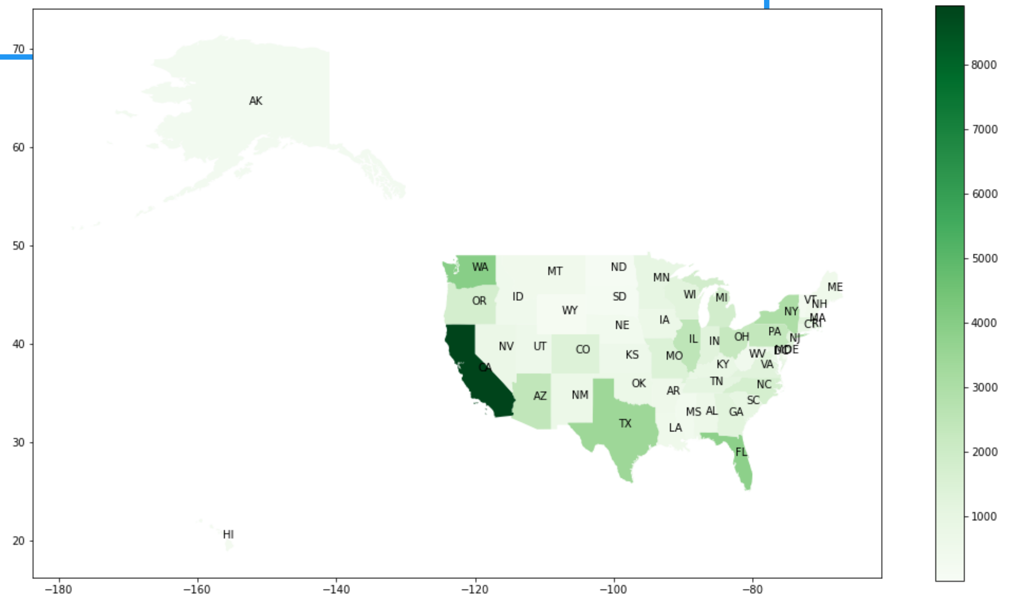
- Combine the two datasets

```
states = states.merge(num_sightings_by_state,
    left_on = "State_Name", right_on = "state")
states.plot(column = 0, legend = True, cmap = "Greens", figsize = (8, 5))
```

- Add text annotations

```
states["centers"] = states.geometry.apply(lambda x: x.centroid.coords[0])
for idx, row in states.iterrows():
    plt.annotate(text = row["State_Code"], xy = row["centers"])
plt.show()
```

- * Other ideas
  - Remove Alaska / show it separately
  - Use a transformation (e.g. sqrt)



Analyzing Maps

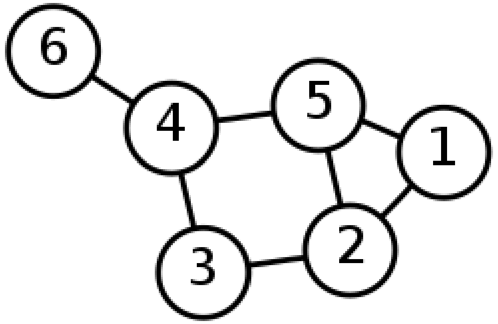
- There are many algorithms used to model spatial data
  - Most commonly, we look for density patterns and clusters of points
  - Common algorithms are
    - KDE – Kernel Density Estimation
    - kMeans Clustering
    - Hierarchical Clustering
    - kNN – k Nearest Neighbors
  - This course doesn't deal with modelling, so we won't get into more detail
    - But feel free to explore the algorithms as you wish
    - You can see details on these on machine learning-related articles
- We can also represent the map as a network
  - E.g., road maps, railway maps, or other "sets of connected dots"

Network Analysis

Working with graphs

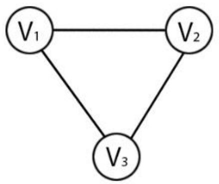
Networks = Graphs

- A graph is a geometrical object consisting of objects which are related by some attribute
  - Nodes (vertices, points) – describe objects
  - Edges (arcs, lines) – connect nodes

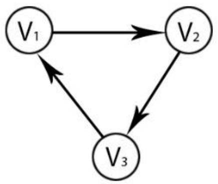


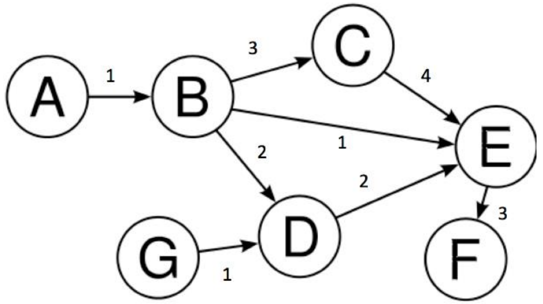
- Types of graphs
  - Directed / undirected
    - In a directed graph, there is only one way to travel between the nodes
  - Weighted / unweighted
    - A weighted graph contains some quantity ("weight", usually ≥ 0) over each of its edges

Undirected Graph



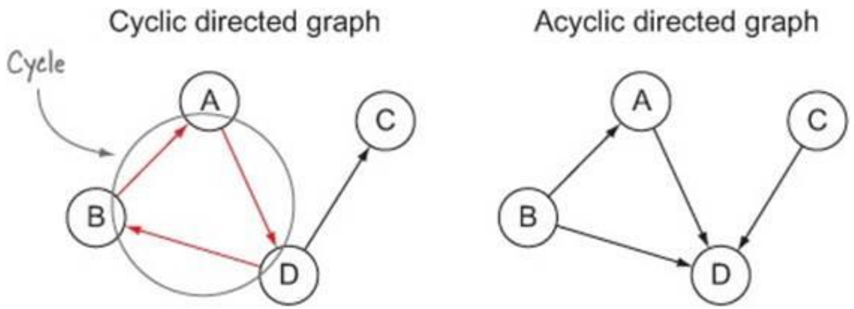
Directed Graph



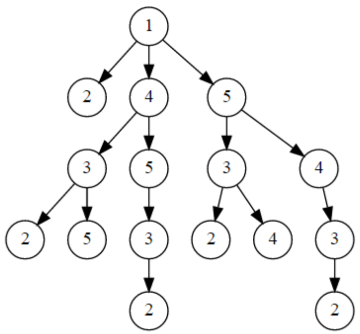


Graphs

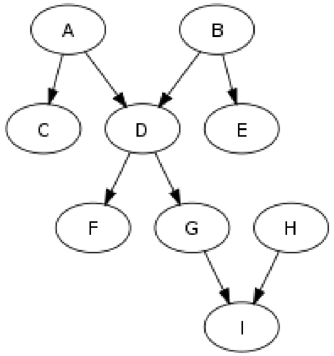
- Types of graphs (cont'd)
  - Cyclic / acyclic
    - When you travel along a cyclic graph, you will visit one node more than once
  - These types are independent
    - i.e. a graph can be "acyclic directed unweighted graph"



- Special cases
  - **Tree** – each node has at most one "parent"



  - **DAG** – directed acyclic graph



Representing Graphs

- We can use the library `networkx`
  - Installed by default with Anaconda
- Create a simple weighted undirected graph

```
import networkx as nx
g = nx.Graph()
g.add_edge("a", "b", weight = 0.1)
g.add_edge("b", "c", weight = 1.5)
g.add_edge("a", "c", weight = 1.0)
g.add_edge("c", "d", weight = 2.2)
```

- Display the graph

```
nx.draw(g, with_labels = True)
plt.show()
```

Finding a Shortest Path

- Advanced graph display
  - Show the weights at each edge
  - Make the edge width proportional to its weight

```
pos = nx.spring_layout(g)
weights = nx.get_edge_attributes(g, "weight")
nx.draw(g, pos, with_labels = True)

nx.draw_networkx_edge_labels(g, pos,
  edge_labels = weights)
nx.draw_networkx_edges(g, pos,
  width = [v * 2 for v in weights.values()])
plt.show()
```

- Shortest paths

```
print(nx.shortest_path(g, "b", "d"),
  nx.shortest_path_length(g, "b", "d"))
print(nx.shortest_path(g, "b", "d", weight = "weight"),
  nx.shortest_path_length(g, "b", "d", weight = "weight"))
```

Creating Directed Graphs

- Directed graph (digraph)
  - Simply change the definition of g
  - Now each edge is directed
  - The visualization will include arrows
    - They point at the direction of each connection

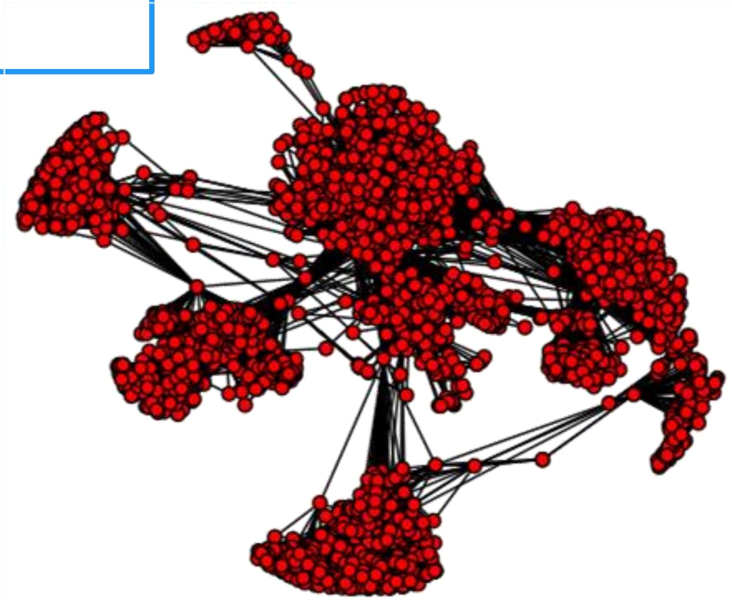
```
g = nx.DiGraph()
g.add_edge("a", "b", weight = 0.1)
g.add_edge("b", "c", weight = 1.5)
g.add_edge("a", "c", weight = 1.0)
g.add_edge("c", "d", weight = 2.2)
```

```
print(nx.shortest_path(g, "b", "d")) # ['b', 'c', 'd']
print(nx.shortest_path(g, "d", "b")) # Error: No path between d and b.
```

Example: Social Circles

- Dataset: `facebook.zip`, info
  - Format: `first_user_id second_user_id`
    - I.e. edge list
- Read the graph
  - Extremely simple

```
facebook_graph = nx.read_edgelist("facebook_combined.txt")
print(len(facebook_graph.nodes)) # 4039
print(len(facebook_graph.edges)) # 88234
```

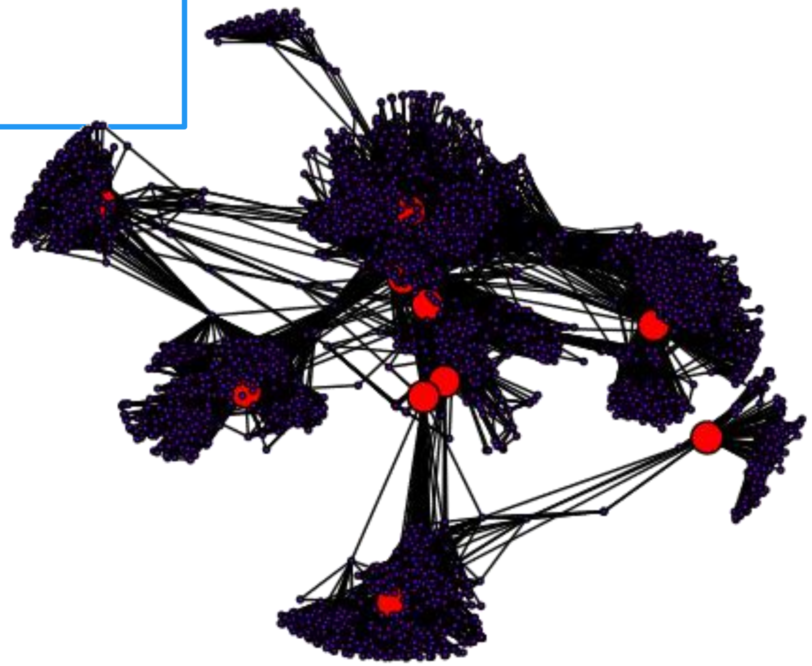


Calculating Important Nodes

- Measure: centrality
  - Different types of centrality, according to different formulas
    - E.g. "betweenness centrality"
  - Measures how important a node is
- To exemplify, let's use a smaller graph

```
karate_graph = nx.karate_club_graph()
centrality = nx.betweenness_centrality(karate_graph)
# Returns a dictionary
```

- Ten most important nodes in the Facebook graph
  - Look similar to cluster centroids

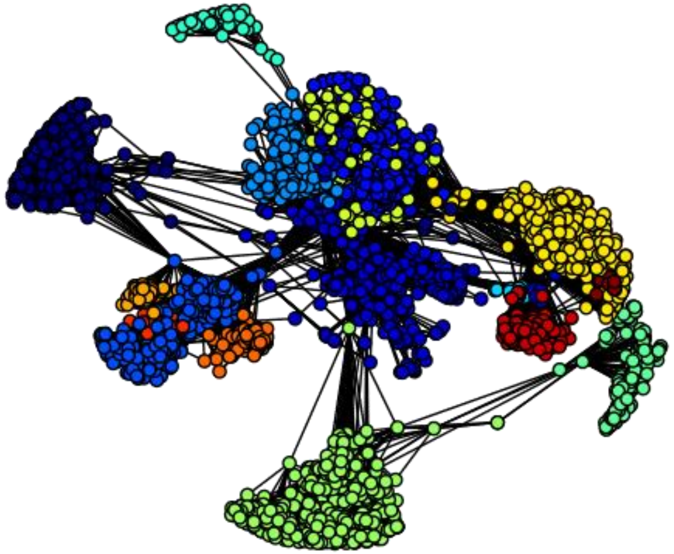


Finding Communities

- Measure: cliques
  - Most commonly used algorithm: [Girvan – Newman]
    - Uses edge betweenness as the measure

```
from networkx.algorithms import community
nx.draw(karate_graph, with_labels = True)
communities_generator = community.girvan_newman(karate_graph)
for i in range(1, 4):
  communities = next(communities_generator)
  print("level " + str(i), communities)
```

- We can find communities in the Facebook graph
  - Look similar to different clusters



Summary

- Geospatial data
  - Reading and exploring
  - Projections
  - Visualization
    - Scatter plots
    - Choropleth maps
- Network analysis
  - Graphs, types of graphs
  - Shortest path between nodes
  - Centrality
  - Communities

Questions?