Faculty of Engineering and Architectural Science

DS8015 - Machine Learning non-DS Stud

W2022

Final Project Report (Due April 25th)

Team 8 - Minions

Prof. Roy Kucukates

E: roy.kucukates@ryerson.ca

Table of Contents

Problem Statement:

At a given user location (Point P), suggest the nearest cluster that a user can cycle based on user preference citing maximum safety or shortest distance (random ride using any type of bike lane) or minimum passing through cross-intersections.

Purpose: Shared bike routes along with city traffic and multiple intersections present many hazards to users. The purpose of the following analysis is to provide the safest route through street crossings and minimum shared-use paths.

Considerations

- Design paths to minimize the number of shared paths.
- Provide insights on the type of path being crossed, the path may warrant the shortest walk-through from all clusters.
- Compute community detection and clustering with minimum threshold in the strongly connected stations i.e., nodes.
- Modularity Maximization to provide strong connected communities/clusters for maximum safety
- Vegetation and other landscape features should allow adequate safety parameters e near intersections.

Case Study: YIELD TO BIKES AND CROSS IN GREEN. St. Petersburg, Florida

Bicycle Safety Guide and Countermeasure Selection System (pedbikesafe.org)

Many Case studies have been applied using machine learning practices to provide safer road pathways to cyclist. One of the case study - Green Pavement Markings at Bike Lane Weaving Area is published using the data sets in the city of ST. Petersburg, Florida.

- **Reference**: http://www.pedbikesafe.org/bikesafe/casestudies_detail.cfm?CS_NUM=719&op=L&subop=I&state_name=Florida

Dataset Description

Attributes Description:

The bike lane dataset is retrieved from the Chicago data portal (City of Chicago, 2022) consisting of 884 entries of different bike path sections around the city. The different classes that are present include:

- the_geom: A representation of the section using an array of geo-coordinates;
- street;
- pre_dir: A prefix direction of the given street;
- F_street: The intersection where the specific bike path begins;
- T_street: The intersection in which the specific bike path ends; and
- Display Row: contains the type of bike path the section falls under such as protected, shared, etc.
- MI_CTRLINE: This value did not contain any descriptions and its relationship to the dataset was not provided

Statistics:

Given the nature of the dataset, there were not many statistics that could be generated. There were a total of 884 entries. The number of unique fields in each of the attributes were retrieved using df.unique() function and are as follows:

- Street : 211 Unique names
- F_Street: 424 Unique names
- T_Street: 438 Unique names
- DisplayRou: 5 Unique names

Attributes Used:

Due to the nature of the problem and solution, the key attributes that were used were: “the_geom” and “DisplayRou” columns. “the_geom” provided the necessary spatial information that was needed for clustering and graph drawing. “DisplayRou” provided information on the lane type which was important to determine the safety of a given stretch of road. statement. “street”, “f_street”, and “t_street” attributes were also used within graph analysis as they represented the nodes and edges that were present within the drawn graph. Lastly, attributes such as the “pre_dir” and “MI_Ctrline” were not used for analysis because they did not provide useful information for solving the problem

Attribute Cleaning:

The only pre-processing that was needed on the dataset was the parsing of “the_geom” attribute prior to calculations as it is initially provided in a string format with punctuation and brackets. These strings were parsed using functions that were written by our team to replace unnecessary information with blank spaces and then geo coordinates were returned in a list format.

Work Distribution:

Team 8- (Minions) consist of 3 people. We use a two stage process for the data analysis, and perform regression/classification on the Chicago_bike_routes datasets. In the first stage, we use models and techniques used in course DS8015 specifics to Group Clustering and frequency distribution as explained. In the second stage, we use graphs for visualisation of the datasets and perform hierarchy clustering along with linear regression to determine modularity of the connected communities.

Stage 1: (Assigned to: Hai Yi Wang and Aquil Suleimanyar)

Perform K-means Clustering and Frequency Analysis

- Geo-coordinates of bike paths were grouped using K-means clustering.
- The elbow method was used to determine cluster amount.
- Frequency analysis was used to determine the different types of bike paths present in each cluster.

Stage 2: (Assigned to: Karandeep Saini)

Graph Traversal and Node analysis of Bike Paths (Karandeep)

- Bike paths and stations are plotted in an undirected graph as nodes and edges.
- Hierarchical Clustering and Decision Tree (Knn – Modularity).
- Degree centrality: longest and shortest paths within the graph are analysed.

Team Member	Project Report	Project Presentation
Karandeep Saini	- Graph Visualisation and Hierarchy clustering - Modularity and degree closeness Prediction	- Result Outputs - Shortest and longest walks - Optimum Clustering
Haiyi Wang	- Dataset Description - K-Means (Simplified)	-Dataset Description - Models and Techniques - K-Means (Single Node)
Aquil Suleimanyar	- K-Means (All Nodes) - Future Work	- Introduction - K-Means (All Nodes) - Lessons Learned

Solution and Description of Model

Stage1: K-Means and Frequency Evaluation:

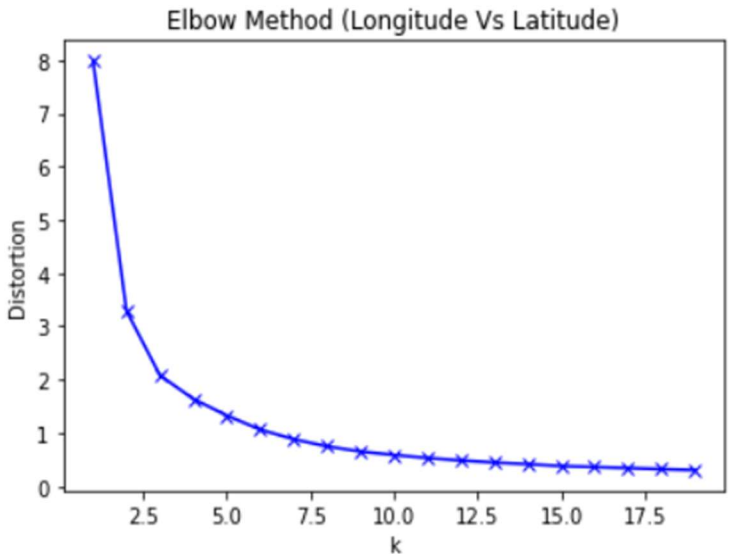
For K-means clustering, libraries from sklearn were used, specifically K-means and clone packages. To plot the locations of the cluster centers onto a map for visualisation, folium libraries were used. Lastly, frequency analysis for bike lane types were graphed using matplotlib in a stacked bar graph format.

K-means clustering is a technique that is widely used for finding groups within data that have not been previously labelled. In this specific problem, K-means is a useful method in grouping sections of bike paths into clusters through their geocoordinate representations. This technique was important into identifying the major areas cyclists can cycle around Chicago city.

Frequency Analysis was used as a generalised estimation of the level of safety within the identified clusters. As there were 5 different types of bike lanes present in the city, they each had varying levels of safety for bikers. Lanes that had more safety measures such as barriers, buffers, etc. were considered safer than others. Thus, through identifying the number of safer bike lanes in a specific cluster than another, it can be generalised that one cluster is safer than the other.

Parameter Selection:

A common method of selecting the number of clusters the algorithm should target is through the elbow method. This method was used to determine the number of clusters in the following K-mean models. An example of such technique can be seen in the above figure where the optimal number of clusters for the K-means model using simplified coordinates for bike routes resulted in 3.



K-Means (Bike Routes Simplified to One Point)

Data Frame Creation

```
def getMiddleLonLat(coordList):
  lat = coordList[len(coordList)//2][1]
  lon = coordList[len(coordList)//2][0]
  return [lat, lon]
```

The above snipping is a function that was created to retrieve the middle geocoordinate from the array of coordinate pairs found in the dataset. The return values were used to generalize each bike lane section to one point to reduce computational resources when clustering.

```
# Clustering section (Clustering using middle lon
clusterDF = my_df.loc[:,["the_geom","F_STREET","T
tempLatHolder = []
tempLongHolder = []
for row in clusterDF.itertuples(index=False):
  geomList = parseMultiString(row.the_geom)
  [latitude, longitude]= getMiddleLonLat(geomList
  tempLatHolder.append(latitude)
  tempLongHolder.append(longitude)
clusterDF['Latitude']=tempLatHolder
clusterDF['Longitude']=tempLongHolder
clusterDF.head()
```

The above code iterates through the individual entries of the data frame that was created with the geocoordinate, street names, and bike routes.

The goal is to parse the string of geocoordinates and to store them into an array so the getMiddleLonLat function can be used to simplify the coordinates to a single point for each lane. This is then stored into the data frame under respective Latitude and Longitude columns. The output of the above code can seen below(figure1):

Elbow Method

```
# import k means
from sklearn.cluster import KMeans
#Elbow method to find optimal clusters
#create an array to store distortions
distortions = []
#create a range of K to test to find optimal
K = range(1,20)
for k in K:
  #fit the model with the number of clusters
  tempKMeansModel = KMeans(n_clusters=k)
  tempKMeansModel.fit(coordClusterDF)
  # append the inertia value to distortions array for plotting to find elbow
  distortions.append(tempKMeansModel.inertia_)

#plot out the inertia values using matplotlib
plt.plot(K, distortions, 'bx-')
plt.xlabel('k')
plt.ylabel('Distortion')
plt.title('Elbow Method (Longitude Vs Latitude)')
plt.show()
```

The above code is used to implement the elbow method over the data frame created in previous sections. It will iterate through 1 to 20 clusters and determine the inertia values of the data set when fit to the respective number of clusters. Once finished, these values are then plotted using pyplot to visually determine the number of optimal clusters that are required for this specific problem.

	the_geom	F_STREET	T_STREET	DISPLAYROU	Latitude	Longitude
0	MULTILINESTRING ((-87.70296713856054 41.771674...	S CENTRAL PARK AVE	S KEDZIE AVE	BIKE LANE	41.7716132433186	-87.70684344031086
1	MULTILINESTRING ((-87.69808324194781 41.771749...	S KEDZIE AVE	S SACRAMENTO AVE	SHARED-LANE	41.77170431980887	-87.7010413713209
2	MULTILINESTRING ((-87.66131985714 41.8578756...	S DAMEN AVE	S LOOMIS ST	BIKE LANE	41.85778264716298	-87.66675025717153
3	MULTILINESTRING ((-87.54056320645704 41.748437...	E 81ST ST	E 87TH ST	BUFFERED BIKE LANE	41.742981004959795	-87.5404322755676
4	MULTILINESTRING ((-87.67673906927449 41.884982...	N KEDZIE AVE	N DAMEN AVE	PROTECTED BIKE LANE	41.8844318137297	-87.69153171479023

Figure 1

Clustering and Determining Cluster Centroids:

```
#Running k-means clustering using 3 clusters (chosen using the elbow method)
kmeans = KMeans(n_clusters=3)
#cluster the data and get the labels for all the points in the data
clusters = kmeans.fit(coordClusterDF)

#find the cluster centers
centroids = kmeans.cluster_centers_
print(centroids)
cLat = []
cLong = []
for row in centroids:
  cLat.append(row[0])
  cLong.append(row[1])
centroidDF=pd.DataFrame({"Latitude":cLat,"Longitude":cLong})
```

After determining that the optimal number of clusters is 3, the parameter is then passed into the clustering algorithm so the dataset can be fit accordingly. Once the clusters have been determined, the centroids of the clusters are outputted for reference then stored into a data frame on there own to be used for visualisation in the following section.

Visualization of Cluster Centers

```
import folium

m = folium.Map([41.8781, -87.6298], zoom_start=11.5)
centroid = centroidDF.values.tolist()

for point in range(0, len(centroid)):
    print(centroid[point])
    folium.Marker(centroid[point], popup = "Cluster"+str(point)).add_to(m)
m
```

The folium library was used to visualise the cluster centers over a map of Chicago. A map object is created and centered around the city of Chicago and is stored into the variable ‘m’. The centroid values retrieved

in the previous snippet is then converted to a list, and iteratively plotted onto the map using folium markers. The resulting map can be seen in the predictions section in figure.

Frequency Analysis of Bike Lane Types and Visualisation

The obtained cluster labels are appended to the previous data frame. Then a new data frame is created that only contains the “DISPLAY ROU” column which contains the respective bike lane types per street and their respective cluster labels. The count of occurrences of each type for the individual clusters are calculated using the groupby.size method and added into the data frame under the column “Count”. A stacked bar graph was then plotted to visually see the separation of lane types within each cluster using pyplot subplots and subplot.bar methods. The labels of each bar was added using subplot.bar_label methods. The resulting plot can be seen in the results section of this report.

```
#Add the labels back into the ClusterDF
labels = clusters.labels_
clusterDF['Cluster Labels']=labels
clusterDF.head()
laneCount = clusterDF.groupby(['DISPLAYROU','Cluster Labels']).size().reset_index(name = "Count")
print(laneCount)
print(laneCount.iloc[0]["Count"]+laneCount.iloc[3]["Count"])
print(laneCount.iloc[3:6]["Count"])

fig, ax = plt.subplots()
p1 = ax.bar(laneCount.iloc[0:3]['Cluster Labels'],laneCount.iloc[0:3]['Count'],0.35,label="Bike Lane")
p2 = ax.bar(laneCount.iloc[3:6]['Cluster Labels'],laneCount.iloc[3:6]['Count'],0.35,bottom=calculateGraphBottom(3,laneCount),label="Buffered Bike Lane")
p3 = ax.bar(laneCount.iloc[6:9]['Cluster Labels'],laneCount.iloc[6:9]['Count'],0.35,bottom=calculateGraphBottom(6,laneCount),label="Neighborhood Greenway")
p4 = ax.bar(laneCount.iloc[9:12]['Cluster Labels'],laneCount.iloc[9:12]['Count'],0.35,bottom=calculateGraphBottom(9,laneCount),label="Protected Bike Lane")
p5 = ax.bar(laneCount.iloc[12:15]['Cluster Labels'],laneCount.iloc[12:15]['Count'],0.35,bottom=calculateGraphBottom(12,laneCount),label="Shared Lane")

#set the quantities inside the bars
ax.bar_label(p1,label_type='center')
ax.bar_label(p2,label_type='center')
ax.bar_label(p3,label_type='center')
ax.bar_label(p4,label_type='center')
ax.bar_label(p5,label_type='center')
#Set Titles and Axes
ax.set_title("Frequency of Bike Lane Types Within Each Cluster")
ax.set_ylabel("Frequency")
ax.set_xlabel("Cluster")
#Set Legend
ax.legend(title='Bike Lane Type',bbox_to_anchor=(1.05, 1), loc='upper left')
```

The above function was created to help aid the calculation of the ‘bottom’ function argument of the bar graph. The purpose of the ‘bottom’ variable is to tell pyplot where the next portion of the stacked bar graph should begin to ensure that bars are properly stacked on top of each other rather than over lap. It accepts a row and data variable. Because there were 3 different clusters there are three for loops that calculate the sum for each respective cluster. Each for loop has the responsibility of summing up all count numbers that fall under the respective cluster (0, 1 ,2) up to the startRow variable which represents the beginning of the next bar that is being plotted.

K-Means (All nodes used)

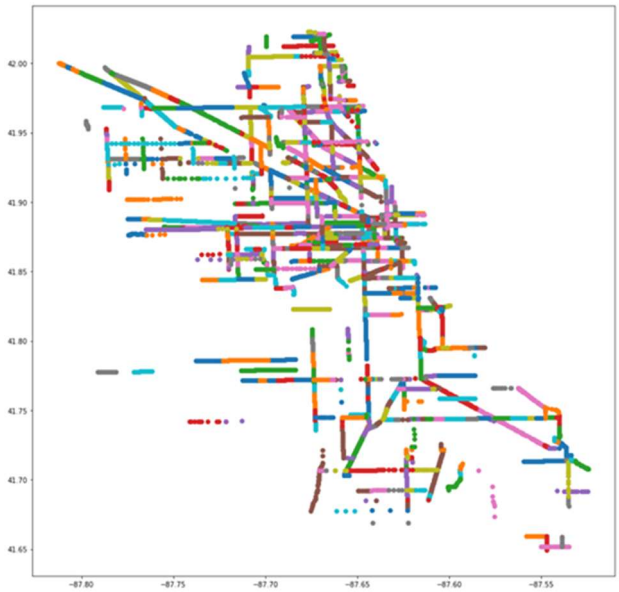
Organising the Data

The purpose of this part of the code was to isolate the columns from the dataset that we are interested in. In this case, we are interested in the OBJECTID which acts as an index, as well as the_geom, which gives us our primary data for this model. This being the geo-coordinates of the bike routes in the dataset. We print this to simplify and see the structure of that data, so that we may manipulate and organise as we see fit.

```
# read all the points of the to a new array

newMappedPoints = []
for idx, (objectid, geomPoint) in enumerate(zip(objectids,geom)):

    # remove the trailing brackets.
    geomPoint = geomPoint.replace('MULTILINESTRING ((','')
    geomPoint = geomPoint.replace('))','')
    geomPoint = geomPoint.replace(')','')
    geomPoint = geomPoint.replace('(','')


    # split the data by comma

    geomPoint = geomPoint.split(',')

    # print(geomPoint)

    for coordinate in geomPoint:
        # remove the leading space
        coordinate = coordinate.lstrip()

        # split by the white space to have x,y cordinates
        coordinate = coordinate.split(' ')
        singlePoint = [idx+1,objectid,float(coordinate[0]),float(coordinate[1])]
        newMappedPoints.append(singlePoint)
```

To organise this data, we decided to create an array, read all the points and assign them to the new array. We do this by first creating the array, then iterating through each ID while removing the unnecessary brackets and splitting the data by the commas. After the data is cleaned up, we can print the array again to see the structure of the array for plotting.

Plotting the Data



After importing matplotlib.pyplot, we can plot the scatter of every geo-coordinate in the dataset with its corresponding x and y coordinates. This plot will look like the one shown above.

Importing Necessary packages

```
# find clusters with K means
from sklearn.cluster import KMeans
from sklearn.preprocessing import MinMaxScaler

km = KMeans(n_clusters= 7)
km
```

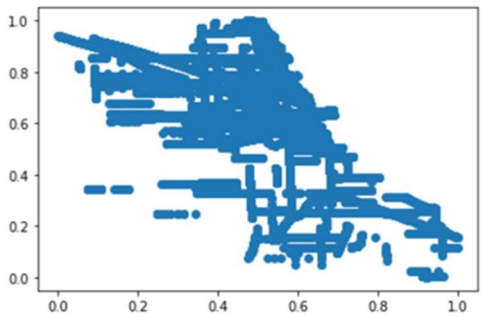
The above code is importing the KMeans and MinMaxScaler packages from the sklearn library and from the cluster and preprocessing modules, respectively. We then initialise the KMeans algorithm with the number of clusters of 7, obtained from the plot.

Scale the Data with MinMaxScaler

```
scaler = MinMaxScaler()
scaler.fit(df[['x']])
df.x = scaler.transform(df[['x']])
scaler.fit(df[['y']])
df.y = scaler.transform(df[['y']])
df
```

This dataset obtains a lot of points and those points having their own set of coordinates can lead to confusing and taxing plots. The above code is aimed to scale the data using the MinMaxScaler package for more accurate and easier plotting in general.

All of the coordinates are converted to a number between 0 and 1. Plotting this would now have the following result:

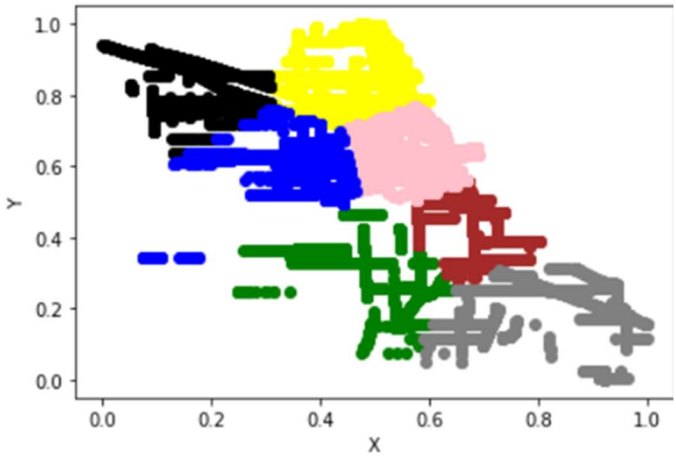


Find number of Clusters using KMeans

```
y_predicted = km.fit_predict(df[['x','y']])
```

```
df['cluster'] = y_predicted
```

This will provide a table with the OBJECTID, x, y, and cluster number the object ID falls under. When we plot this graph to visualise the clusters, we get the following result:



From this graph, we can clearly visualise the 7 different clusters with their corresponding colours.

Finding the most common bike lane type in each cluster.

To find the most common bike lane type in each cluster, we first create two empty dictionaries that will be used for associating the rows to the OBJECTID and the OBJECTID to the cluster number. We then run iterations to count the number of each bike lane type in each cluster.

After this has been completed, we run iterations again. However, this time the goal of the iteration is to find the maximum number that any bike lane type occurs in each cluster. We can then print this to obtain the cluster number, the type of bike lane that exists most often, and the number of times that type of lane occurs in that cluster of points. This results in the following output:

```
objectid_to_cluster={}
for index,foo in df.iterrows():
  objectid_to_cluster[foo['OBJECTID']] = foo['cluster']

rows_by_objectid={}
for index,foo in lanes.iterrows():
  rows_by_objectid[foo['OBJECTID']] = foo

cluster_to_bikelanetype_count={}
for objectid,cluster in objectid_to_cluster.items():
  if not (cluster in cluster_to_bikelanetype_count):
    cluster_to_bikelanetype_count[cluster] = {}
  row = rows_by_objectid[objectid]
  bikelanetype = row[6]
  if not (bikelanetype in cluster_to_bikelanetype_count[cluster]):
    cluster_to_bikelanetype_count[cluster][bikelanetype] = 0
  cluster_to_bikelanetype_count[cluster][bikelanetype] += 1

cluster_to_most_common_bikelanetype_and_count={}
for cluster, bikelanetype_count in cluster_to_bikelanetype_count.items():
  max = list(bikelanetype_count.items())[0]
  for bikelanetype, count in bikelanetype_count.items():
    if max[1] < count:
      max = bikelanetype, count
  cluster_to_most_common_bikelanetype_and_count[cluster] = max

print(cluster_to_most_common_bikelanetype_and_count)
```

```
{0.0: ('BUFFERED BIKE LANE', 25)
```

Cluster 0 has buffered bike lanes more than any other type of lane, and this type of lane occurs 25 times throughout the cluster.

Stage 2: Graph Visualisation and Hierarchical Clustering

As per given data description, the prerequisite of the datasets is to plot all the bike-routes on the graph with precise geo-location coordinates. The datasets provided are decoded to the python-igraph library and plotted as an undirected graph with stations as vertices and routes as edges. {G =(V,E)}

The tuple-list is derived from the input datasets and the_geom - multiLineString coordinates are represented by vertex nodes depending upon longitude and latitude points.

The stage 2 starts with visual representation of all stations into geometric axis as (x.y) coordinates around the Chicago city as depicted in the figure:

Figure: Represented by longitude/latitude axis in two-dimension layout

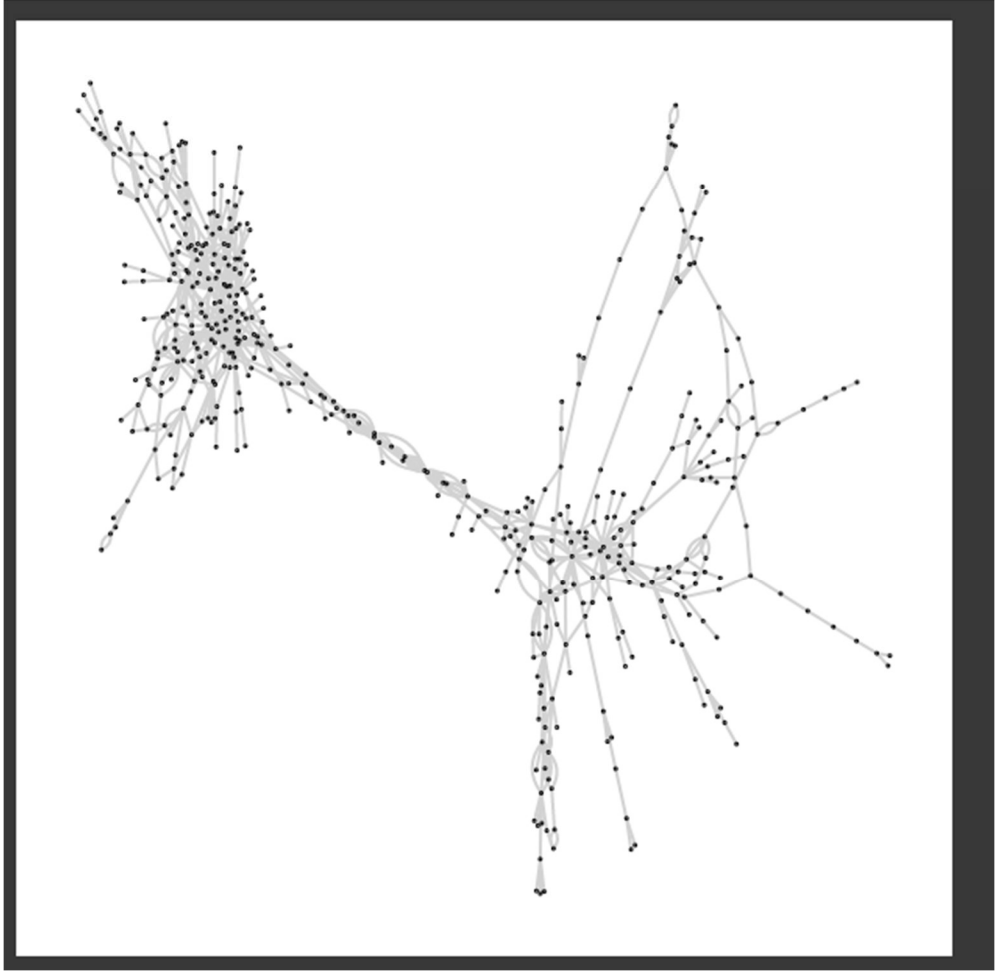
```
def plotRoutes(iterRows, g):
  #print(iterRows)
  list = parseMultiString(iterRows[0])
  g = g.simplify(multiple=False)
  g.vs()['layout'] = [(float(index[0]), float(index[1])) for index in list]
  g.vs()['size'] = sz[0]
  g.es()['arrow_size'] = 0.0
  # call color combination function
  colorCodes(iterRows[3],g)

l= my_df.loc[:,["the_geom","F_STREET","T_STREET","DISPLAYROU"]]

for row in l.itertuples(index=False):
  plotRoutes(row,g)
## plot using lat/lon as layout
ly = ig.Layout(g.vs['layout'])
ly.mirror(0)
g = g.simplify(multiple=False)
```

Having said that, let me mention that the igraph library generalises the definition of disconnected graphs. In this generalisation, one simply takes the maximum distance (u, v) over all pairs of nodes u, v that belong to the same strongly connected component. To reduce the sparse disconnected graph into strongly connected components, we need to reduce the graph into well connected edges where the vertex degree is greater than 2. A subgraph is plotted while computing the degree distribution of the graph with all nodes connected and reachable from any node.

```
## undirected degree centrality
def degree_centrality(g):
    n = g.vcount()
    if g.is_directed():
        dc = [sum(x)/(2*(n-1)) for x in zip(g.strength(mode='in'),\
              g.strength(mode='out'))]
    else:
        dc = [x/(n-1) for x in g.strength()]
    return dc

## compute several centrality measures
C = pd.DataFrame({'station':g.vs()['name'],\
                  'degree':degree_centrality(g),})

## sort w.r.t. degree centrality, look at top stations
df = C.sort_values(by='degree', ascending=False)
df.head()
```

Computing degrees, we find the min and max degree vertex.
Maximum degree: 28 Vertex ID with the maximum degree: 152
Minimum degree: 1 Vertex ID with the minimum degree: 6

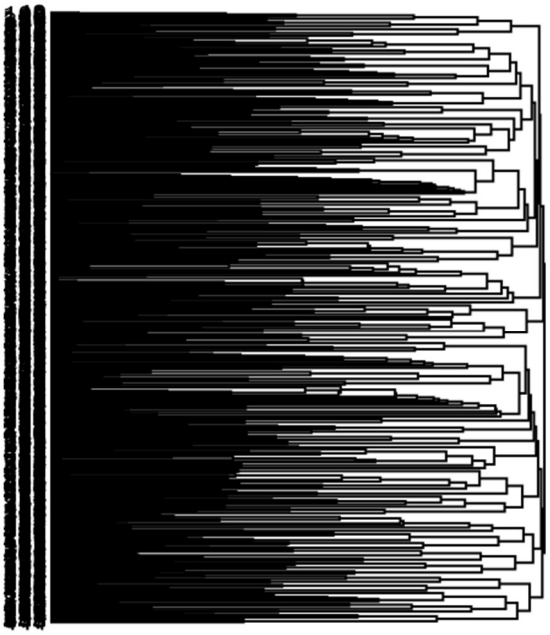
```
g.vs['color'] = [cls[i] for i in g.vs['comm']]
g.vs['size'] = sz[0]
g.es['color'] = 'lightgrey'
ig.plot(g, bbox=(0,0,300,300))
```



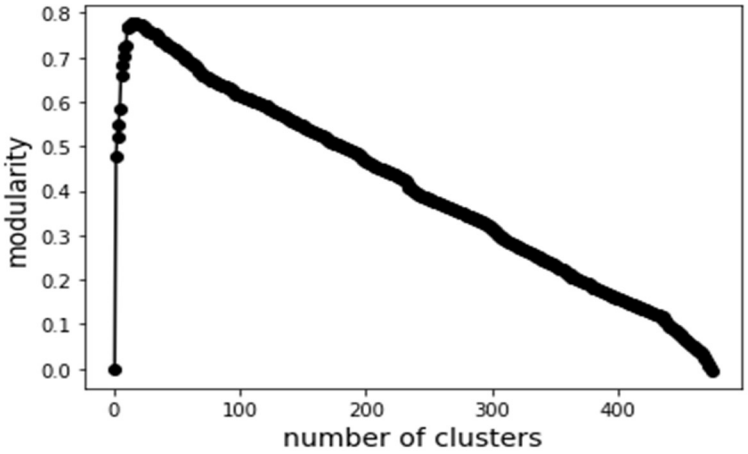
I. Model Used:

Hierarchical clustering algorithms are unsupervised machine learning tools. In particular, it is assumed that the decision about the number of the communities the set of nodes should be partitioned into should be made by the algorithm. This contrasts with, for example, k-means clustering algorithm of data points in the form of d-dimensional vectors in which the number of clusters, k, is provided as a parameter. Independently, other algorithms may be run on top of k-means to select the number of clusters. At each step of the process, an algorithm selects a pair of clusters with the largest similarity and then these two clusters are merged into one.

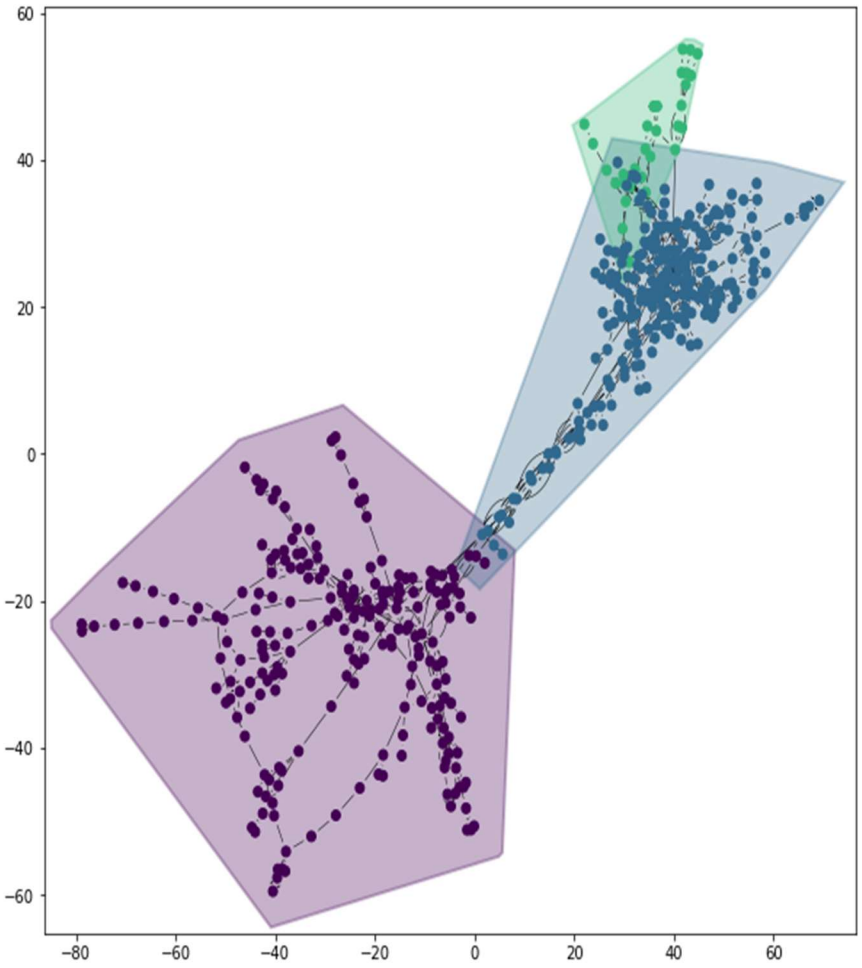
Such outcomes can be conveniently represented by a **dendrogram** known also as a **hierarchical tree. (See figure)**



and communities are detected using python-igraph betweenness. The modularity is highest at cluster size=3, and thus selected for righteous distribution. The clustering coefficient that tries to measure the tendency for nodes to cluster together. It has been experimentally verified that many real-world examples (including Uber Delivery Clusters) consist of tightly connected groups of nodes; the likelihood of seeing such behaviour in a random network, in which each pair of nodes is independently connected with a relatively small probability, is very low.



Provided with three clusters formation, the graph will be represented (See figure below). Total '3' clusters formed out of connected graph



Clustered strongly connected graph

II. How they were used:

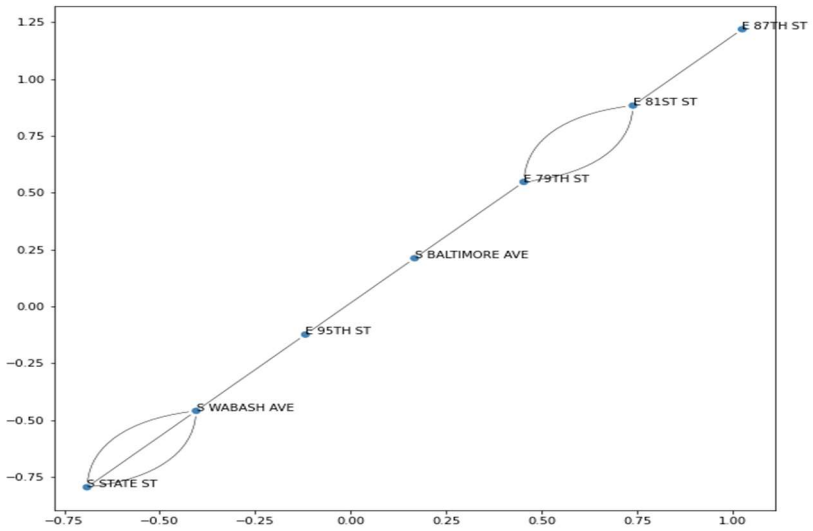
We use the python-igraph library for creating and manipulating graphs. You can look at it in two ways: first, python-igraph contains the implementation of quite a lot of graph algorithms. These include classic graph algorithms like graph isomorphism, and connectivity and also the new wave graph algorithms like transitivity, community structure detection. Second, python-igraph provides a platform for developing and/or implementing graph algorithms. It has an efficient data structure for representing graphs that are useful for implementing graph algorithms. In fact, these data structures evolved along with the implementation of the classic and non-classic graph algorithms which make up the major part of the library. This way, they were fine-tuned and checked for correctness several times.

III. Why they were chosen:

Our main goal is developing a graph model, which is efficient on large, but not extremely large graphs. More precisely, it is assumed that the graph(s) fit into the physical memory of the computer.

Conclusively, bike routes are referred to as connected stations which are represented by graph vertex/nodes with connected bike paths represented by graph edges. Thus, python-igraph fits our

requirement for quick traversal of routes and its distribution. For quick **results, we** find the busiest location, in other words, find the node with the maximum total degree and traverse to all the neighbouring communities in the shortest random walk (See figure).



Random walks on undirected graphs

V. Code Snippets

```
[vertices, layers, parents] = g.bfs(vMin, mode='OUT')

#Shortest Path from min degree node to max degree node
[vertices] = g.get_shortest_paths(vMin, vMax)
G= g.subgraph(vertices)

G.es()['arrow_size'] = 0.5
fig1, ax1 = plt.subplots()
fig1.set_size_inches(10, 10)
ig.plot(
    G,
    target=ax1,
    vertex_label = G.vs()['name'],
    vertex_size=sz[1],
    edge_width=0.5,
    margin=10
)
```

```
from sklearn.metrics import adjusted_mutual_info_score as AMI
from copy import deepcopy
visual_style = {}
gc = g.copy()
## show result with optimal modularity (3 clusters)
communities = gn.as_clustering(n=3)
membership = communities.membership
gc.vs['label'] = membership
print('AMI:',AMI(gc.vs['comm'],gc.vs['label']))
print('q:',gc.modularity(gc.vs['label']))
gc.es['color'] = 'lightgrey'
```

```
# Set community colors
num_communities = len(communities)
palette1 = ig.RainbowPalette(n=num_communities)
for i, community in enumerate(communities):
    gc.vs[community]["color"] = i
    community_edges = gc.es.select(_within=community)
    community_edges["color"] = i

# Plot the communities
fig1, ax1 = plt.subplots()
fig1.set_size_inches(10, 10)
ig.plot(
    communities,
    target=ax1,
    mark_groups=True,
    palette=palette1,
    vertex_size=sz[1],
    edge_width=0.5,
    margin=10
)
```

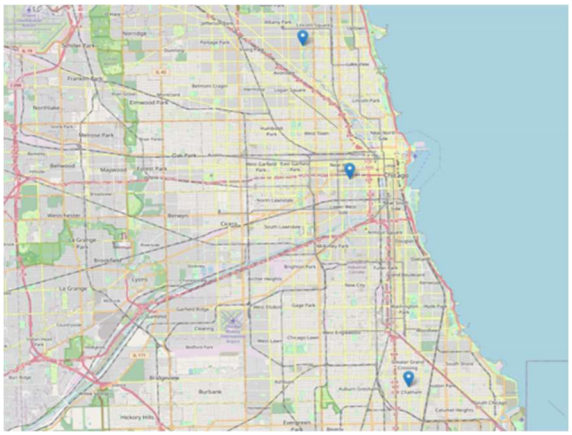
Predictions:

1. Centroid Locations and Map Plot

•0 (Downtown, City Center) : (41.87277229 , -87.66057474)

•1 (Uptown, Northern Cluster) : (41.95522707 , -87.69695233)

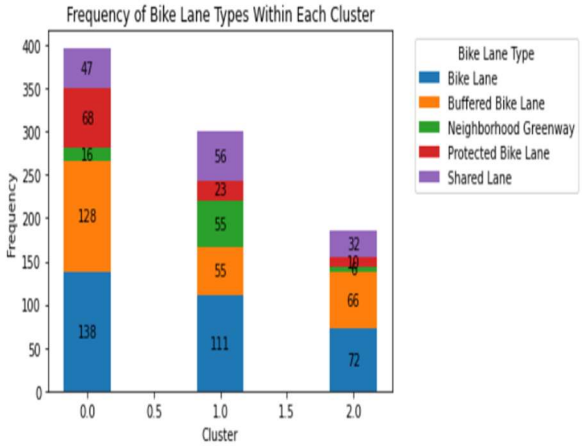
•2 (Southern Cluster): (41.74395235 , -87.61503366)



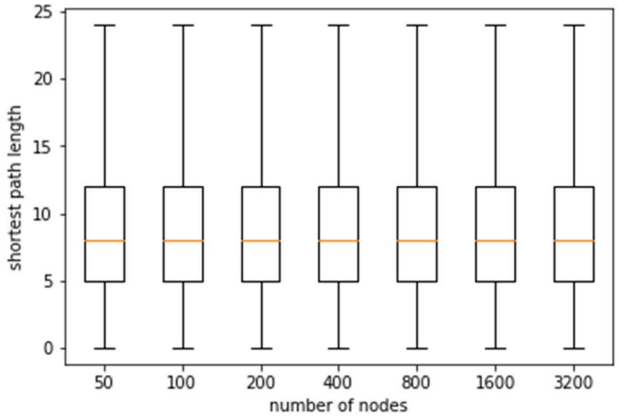
K-Means Clustering models

- Finding clusters using geo-coordinates of bike-route nodes

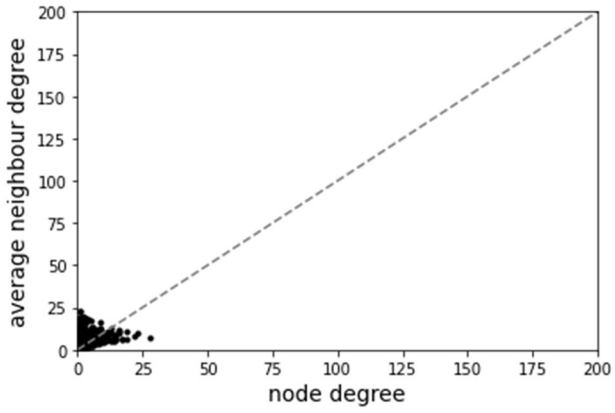
2. Determining the frequency of each bike lane type in each of the three determined clusters



3. Synthesise models with community Structure
   - Best clustering Model using hierarchy dendrogram.
   - longest & shortest walk among clusters.



4. Decision Tree(Knn) Modularity. Computing degree-Centrality on the nodes and centroid of each cluster.



Future Work:

There currently exist many limitations to the dataset that do not allow the models to provide solutions to the questions they were aimed to solve. As discussed previously, the statistical data that was provided with this dataset is limited to geo-coordinates of nodes along bike paths, as well as their respective bike lane type. In the topic of safety, especially in scenarios where unsafe could mean lethal, there are several different parameters that need to be considered. Some of these parameters can include, but are not limited to:

- Collision data in the city:
    - Roadway and intersection collision data, Collision data between vehicles, cyclists, pedestrians, and all combinations of different transportation users.
- Construction and roadway maintenance zones in the area affecting typical traffic flows or geometric design
- Intersection design in the area:
    - Signal timing standards, Designated cyclist signals, Points of conflict with other traffic.
- Alternate use roadways:
- School Zones, Designated/Undesignated trails, Etc.

Theoretically, when data for some, if not all of these different parameters are provided, the model could provide a much more accurate representation of the relative safety a cyclist can expect when deciding which route to take. Similar to the way Google offers Maps users both fastest and most fuel-efficient routes when driving, the ideal scenario for this model would be to pull the GPS location of the user, calculate using the bike route data as well as the safety parameters listed to suggest a spectrum of options for the user ranging from most practical (fastest routes) to safest routes near them.

Unsupervised Learning

The key ingredient for many clustering algorithms is modularity that we discuss in this section. As already mentioned earlier, modularity is at the same time a global criterion to define communities, a quality function of community detection algorithms, and a way to measure the presence of community structure in a network. Modularity for graphs is based on the comparison between the actual density of edges inside a community. Since the ground-truth is usually unknown, this is the best one can do in an unsupervised setting. However, we do have the ground-truth in this case and so we may compare the quality of different partitions. We illustrated several measures and distribution factors based on the unsupervised framework presented in this section. In the case of unsupervised learning, we can often do better by carefully selecting the partitioning based on the true objective function for the task at hand. This is typically done by dividing the labelled data into training, validation, and test sets. We use the training and the validation set to select the best model and parameters. We didn't use entropy and

embedding in an unsupervised framework due to limitations of the datasets. With advanced datasets, we can identify anomalous stations and run numerous embedding and entropy algorithms on the induced datasets.

References

- https://data.cityofchicago.org/Transportation/Bike-Routes/3w5d-sru8
- http://www.pedbikesafe.org/bikesafe/casestudies_detail.cfm?CS_NUM=719&op=L&subop=I&state_name=Florida
- python-igraph
- Newbies Guide to Python-igraph. A simple guide to common functions of… | by Vijini Mallawaarachchi | Towards Data Science
- Hierarchical clustering explained | by Prasad Pai | Towards Data Science