ECOLE POLYTECHNIQUE

INFO 442: DATA ANALYSIS IN C++

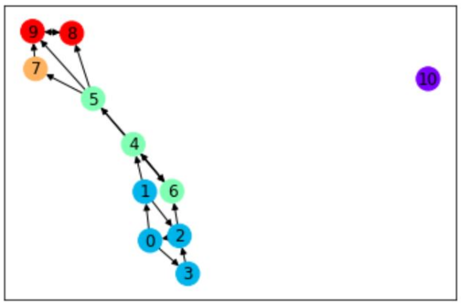
CONNECTED COMPONENTS AND DBSCAN

JOSHUA BENABOU

We have chosen to complete this project in Jupyter Notebook using Python, for ease of display. Graph visualization is done using the Networkx module.

1 Algorithms to find strongly connected components

A digraph $G = (V, E)$ is strongly connected (SC) if it contains a path from u to v for all nodes $u, v \in V$. A sub-digraph of G is a strongly connected component (SCC) of G if it is strongly connected and maximal with this property. Any digraph has a unique SCC-decomposition. Below, SCC's color-coded on graph with $|V| = 11$, $|E| = 17$.



We have implemented two algorithms for finding SCC's in digraphs in time $O(|V| + |E|)$.

1.1 Kosaraju's algorithm

Kosaraju's algorithm (1981) identifies the SCC of a node u as the set of nodes reachable from u by both forward and backward traversal. Given a digraph $G = (V, E)$ let G^t denote its transpose (i.e all edge directions are reversed). Kosaraju works as follows:

1. Initialize an empty stack st. Perform a depth-first search (DFS) on G, pushing each node u to st once it is is processed (i.e once the DFS-subtree of u has been explored).
2. Do DFS on G^t as follows: while st is not empty, pop a node u from st.
    - If u has already been assigned a cluster label, move on.
    - Else, assign u a new label l, and do a DFS to assign l to all members of u's connected component in G^t.

Kosaraju works because, after the forward-traversal DFS (step 1), st is such that: if G contains an edge $u \to v$, then u appears before v in st, unless u and v are in the same SCC in which case order does not matter.

1.2 Tarjan's algorithm

By contrast, Tarjan's algorithm (1972) requires only one round of DFS. Performing a DFS with a counter (starting from an arbitrary node), we define for each node u: $dt(u)$ as the "time of discovery" of u, i.e the counter value when we encounter u during the DFS; at any moment we also define $low(u)$ as the smallest value of $dt(v)$ over all nodes already-visited nodes v in the DFS-subtree of u. Note that $low(u) \leq dt(u)$. Keeping track of these two quantities in two arrays, Tarjan is as follows:

Initialize an empty stack st. Do a DFS on G, pushing each node u to st as it is explored. Then:

- Define $dt(u)$, and explore children of u to update $low(u)$.

- If a node u is such that $dt(u) = low(u)$, then assign a new cluster label l to u and all the nodes above u in the stack. Pop all these nodes (including u) from st.

Tarjan works because of the remarkable property that the nodes of an SCC of G form a subtree in the DFS spanning tree of G. As a consequence, if during the algorithm a node u is encountered such that $dt(u) = low(u)$, then u is node of its SCC that was explored first.

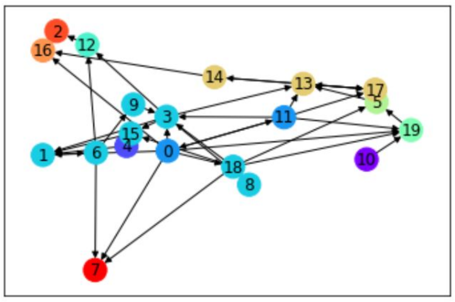
1.3 Performance on real datasets

Note that the time complexity $O(|V|+|E|)$ for the above SCC-finders is simply that of DFS. Below, performance on SNAP datasets. DK="Dead Kernel" (data too large to be loaded in Jupyter).

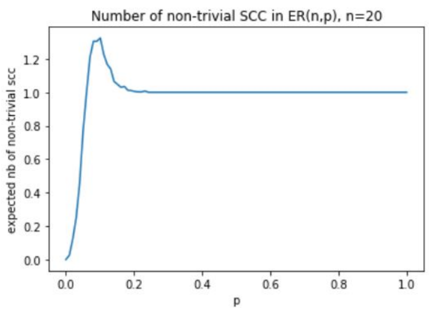
Dataset					Runtimes (ms)	
Name	$\|V\|$	$\|E\|$	#SCC	largest SCC	Kosaraju	Tarjan
p2p-Gnutella08	6301	20777	4234	2068	18	12
p2p-Gnutella24	26518	65369	20167	6352	88	62
Wiki-Vote	7115	103689	5816	1300	69	22
p2p-Gnutella31	62586	147892	?	14149	DK	DK

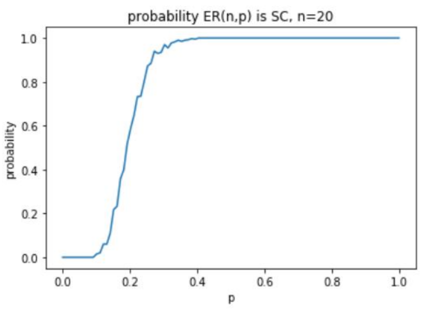
2 Erdos-Renyi Graphs

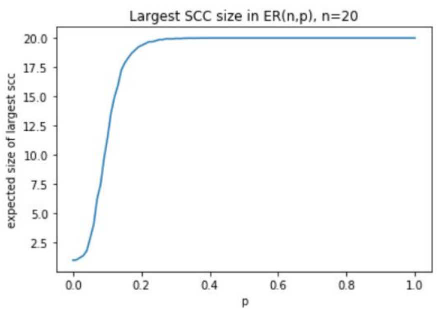
Given $p \in [0,1]$, an Erdos-Renyi digraph $ER(n,p)$ on n vertices is constructed by defining, for each pair of distinct vertices (i,j) an edge $i \to j$ with probability p. How do SCC's behave in these graphs?



Above, a generation of $ER(20, 0.1)$. For small p, $ER(n,p)$ has fewer edges and is less likely to be strongly connected than for large p. Generating $ER(20,p)$ for several different p, we remarked that the graph almost always has at most one non-trivial SCC. We thus collected statistics on the expected number and expected maximal size of SCC's in $ER(20,p)$ for 50 equally-spaced p in $[0,1]$ (100 generations for each p):

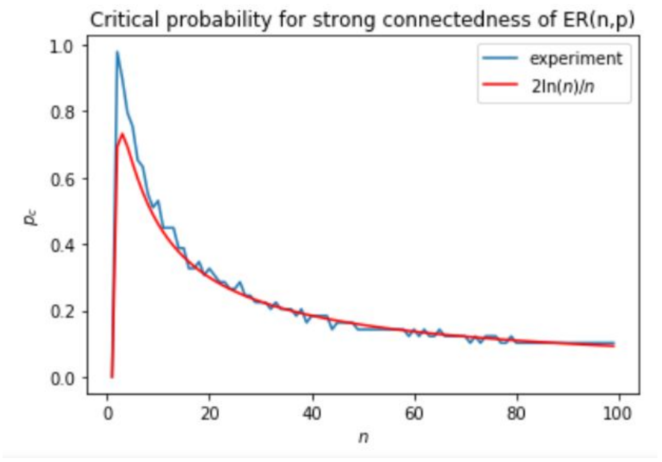






As expected, the expected number of non-trivial SCC's is 0 for $p = 0$ and 1 for $p = 1$. The maximal expected value attained is only slightly greater than 1, confirming our early observation. Furthermore, the interval of p for which the graph is likely neither strongly connected nor has 0 non-trivial SCC's is small.

In 1960, Erdos and Renyi showed $p = \ln(n)/n$ is a sharp bound for the *connectedness* of $ER(n,p)$. We define the critical probability $p_c(n)$ for *strong connectedness* of $ER(n,p)$ as the smallest p such that the $\Pr[ER(n,p)$ is SC$] > 0.95$. Plotting $p_c(n)$ for $n \leq 100$, we conjecture that $p_c = 2\ln(n)/n$ is a sharp asymptotic bound for strong-connectedness:



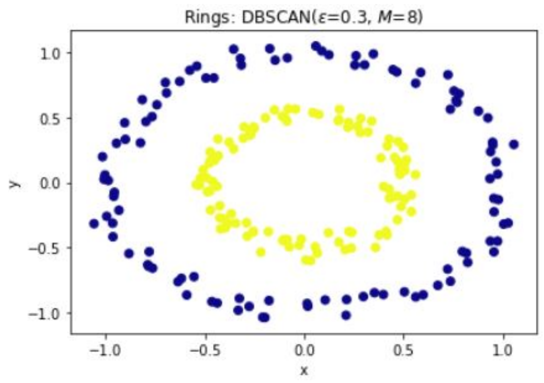
3 DBSCAN

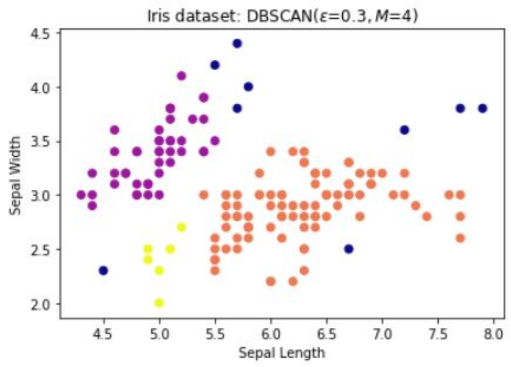
3.1 Description of Algorithm

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) clusters a *geometric* dataset by grouping together points with many nearby neighbors, and marking as outliers points whose nearest neighbors are too far away. The input is a set P of points and a distance function $d : P \times P \to \mathbb{R}$. DBSCAN requires two hyperparameters: a neighborhood radius ϵ, and the minimum number M the of points required to form a dense region.

Informally, DBSCAN partitions the data into clusters and outliers as follows: Given some unvisited point p, if $|P| \cap B(p,\epsilon) \geq M$, p will be part of a cluster, which will also contain all its ϵ-neighbors $|P| \cap B(p,\epsilon)$. Otherwise, p is tentatively labeled as an outlier (it could be determined later to be part of a cluster). Exploring the ϵ-neighbors of p, their ϵ-neighbors and so on via DFS, the cluster of p is determined. To find remaining clusters, the procedure is repeated on any unvisited points.

3.2 Sample results





We give here just two DBSCAN results (many more can be in the Jupyter Notebook). Above, left, DBSCAN succesfully finds two clusters and 0 outliers on a synthetic dataset with $N = 200$ (silhouette=0.13). Above right, we find 3 clusters and 9 outliers on the Iris dataset for $N = 150$ flowers belonging to two species (silhouette=0.45).

3.3 Heuristics for selecting M and ϵ

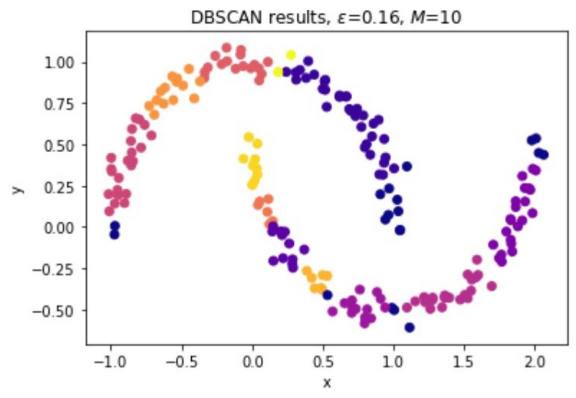
We have generally selected M using prior knowledge about the dataset. For data in $\mathbb{R}^d$ with low noise, M on the order d gave moderate results. For fixed M, we tried two heuristics for choosing ϵ:

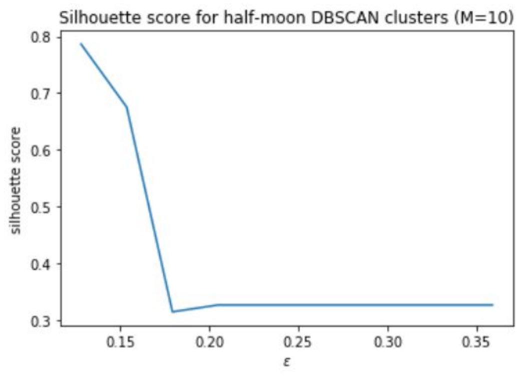
I. KNN plot: For each point, we plot the distance to its M-nearest neighbor, and order these distances from smallest to largest. Optimal ϵ is chosen as the distance coinciding with the ”elbow” of this plot, where the distance increases sharply (see bottom-right plot).

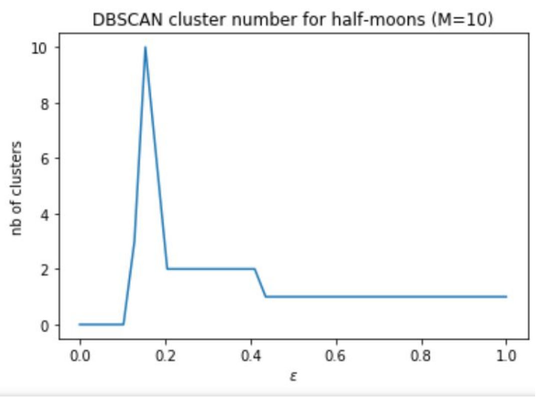
II. Silhouette score: Given a DBSCAN output consisting of some outliers and a set of clusters, we define for each non-outlier point p: $a(p)$ as the average distance of p to points in its cluster C; $b(p)$ as the minimum over $C' \neq C$ of the average distance to points in cluster C'; the silhouette coefficient

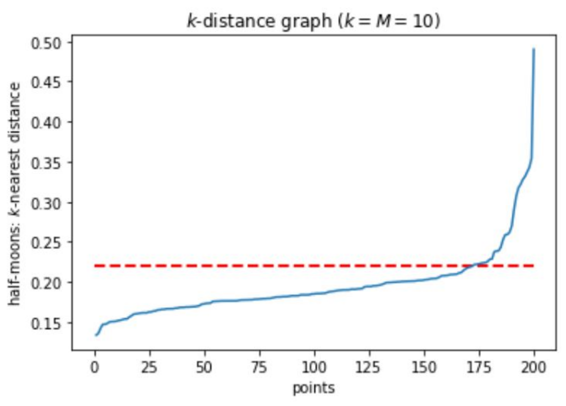
$$s(p) = \frac{a(p) - b(p)}{\max(a(p), (b(p))}$$

and the silhouette score s as the average of $s(p)$ over all non-outliers. We have $s \in [-1, 1]$. We choose the ϵ which maximises the silhouette score. In practice, computing the silhouette is not an efficient heuristic for choosing ϵ *before* running DBSCAN; it is more useful a metric for solution quality. We note however that the highest silhouette coefficient does not always correspond to optimal clustering, as shown below for the ”half-moons” dataset, with 8 detected clusters (better results listed in table **3.4**). Silhouette is plotted where defined. Indeed, if DBSCAN has chosen too many points as outliers, giving more than the optimal number of clusters, the silhouette coefficient can be higher than for the optimal solution if these clusters have sufficient separation. For ”half-moons”,the knn-plot gives an optimal $\epsilon = 0.22$, which gives the correct result (2 clusters).









3.4 Runtime performance

Our neighbor query to find $B(p, \epsilon)$ is implemented using a linear scan, however using a kd-tree this could be done in $O(\log n)$ on average if ϵ is not too large. DBSCAN does one neighbor query for each point, thus average runtime $O(\log n)$ in that case, and in any implementation, the worst case is $O(n^2)$. Below, runtime performance for DBSCAN on real and synthetic datasets.

Dataset	#points	M	ϵ	#clusters	#outliers	Runtime (s)
Iris	150	4	0.3	3	9	0.027
Airports (US)	1298	50	300	4	238	4.3
Airports (World)	7698	10	50	21	7318	215
half-moons (synthetic)	1000	10	0.22	2	2	2
half-moons (synthetic)	10000	10	0.22	2	1	227
half-moons (synthetic)	30000	10	0.22	2	1	2006

4 SCC-finders vs DBSCAN: US flight network

We now seek to compare clustering results from SCC-finders and DBSCAN on the same dataset. We thus need a geometrical dataset which also takes the form of a graph. For this have chosen to use aviation data.

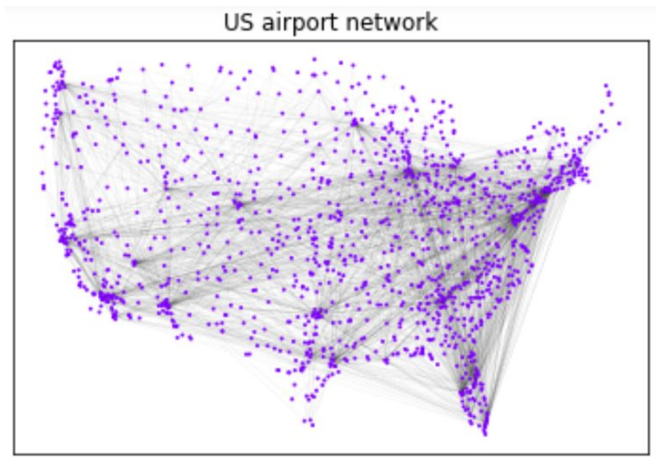
The airports.dat and routes.dat collections from https://openflights.org/data.html#airport contain:

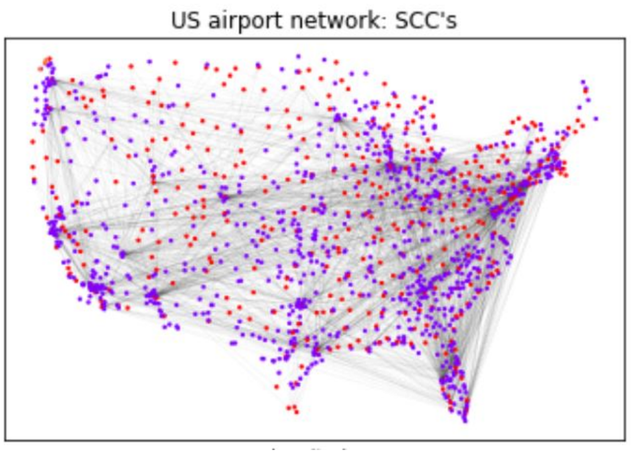
- Information about airports across the globe (airport name, city, country, and latitude/longitude, etc)
- A list of commercial routes between airports (airline, source airport, destination airport, etc)

W focus on the flight network of mainland USA for easy visualization (although we have also tested the global network, see **3.4**), and apply both forms of clustering:

- DBSCAN, using great-circle distance between airports, will tell us where the airports are clustered geographically. We expect the US aiports to be clustered on the coasts, where there are many large cities, and that there be many outliers corresponding to small airports spread out across the country.
- Constructing a graph with airports as nodes and flight routes as directed edges, finding the SCC's will tell us about the connectivity of the US airport system. Ideally,the US network should form a single SCC: every airport should be reachable from any other.

Below,left, are the location of the 1298 mainland US airports along with 4948 logged commercial routes. We can already begin to see which areas are more busy and where the clusters might be located.



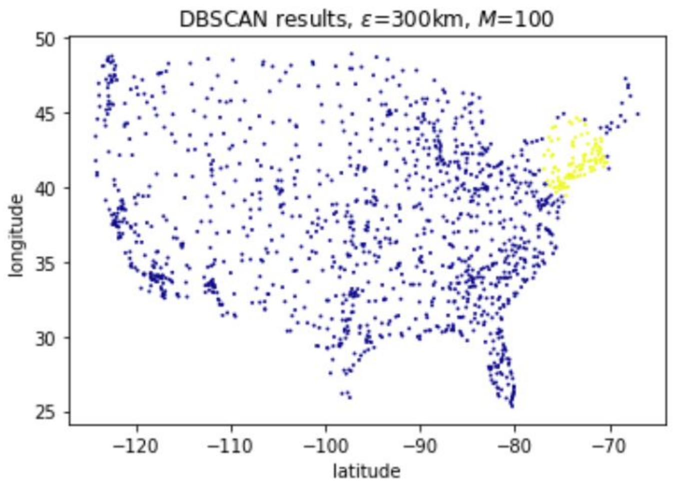


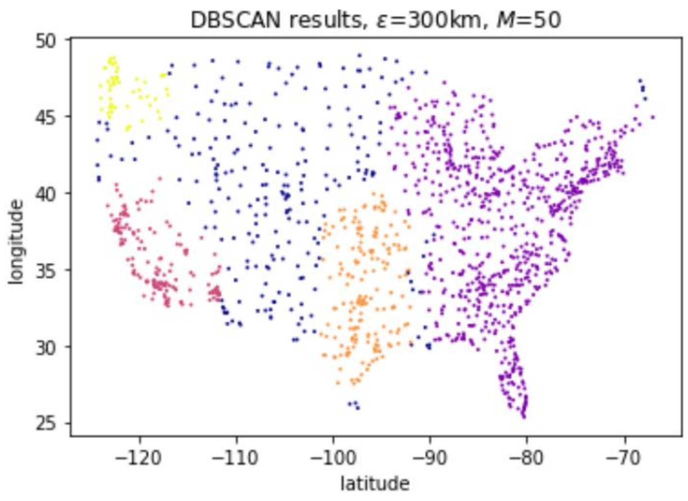
4.1 SCC results

Above, right, We identify: a cluster of size 398 (purple), one of size 4, and curiously, 896 singletons (red). This would suggest the majority of US airports only offer one-way routes: one can leave, but not return (or vice versa). Absurd! In fact, all the airports in red have 0 logged in or out flights - the route data is very incomplete for these small airports. As for the 4-cluster, it consists of 4 nearby small airports in Washington state. It would likely merge into the main cluster with updated data. The US flight network is thus strongly connected! DBSCAN is therefore a more meaningful method of clustering here.

4.2 DBSCAN results

We will fix search radius ϵ to 300 km and vary M. For large M, DBSCAN detects only sufficiently dense clusters. For $M = 100$, only the Northeast cluster (JFK, Boston, etc) survives, the remaining points (dark blue) being outliers. For $M = 50$ we obtain 4 large clusters (silhouette score: 0.46), which correspond to the principal hubs of US flight activity: Southern California (e.g LAX) in maroon, Northern California (e.g SFO) in yellow, Eastern seaboard (JFK, Washington, Atlanta, etc) in purple, and center-south (Dallas, Denver, etc) in orange.


DBSCAN results, ε=300km, M=100
longitude
latitude


DBSCAN results, ε=300km, M=50
longitude
latitude