Lab 6: Clustering and Community Detection

April 16th, 2014

Objectives

In this lab, you will get familiar with two very important concepts: clustering and community detection. In particular, you will:

- get a deeper understanding of expectation-maximization by implementing two clustering algorithms, k-means and Gaussian mixture models,
- apply them to identify clusters of geo-tagged tweets,
- implement the Louvain algorithm for community detection and apply it to a dataset of Wikipedia pages.

Deliverables

- Your code for the clustering part.
- Your answers to the questions of the clustering part.
- Your code for the `communityDetection()` method of the `ix.lab06.community.Louvain`.
- Plot of the `simple` graph and the communities found at the first level of the Louvain method.
- Plot of the `karate` graph and the best communities found by the Louvain method.
- Highest modularity of the `wikipedia` graph found by the Louvain method.
- Plot of communities of the node `Google` and its neighbors from the `wikipedia` graph.

1 Clustering

Clustering is the process of grouping observations such that elements of the same group are more similar to each other than to those of other groups. It is therefore an important tool when it comes to explore and mine data. In this lab, we will focus on two very important clustering methods that allow for finding the clusters and the assignment of the observations, by minimizing an objective function:

- The first method is k-means; it separates a given set of observations into k clusters.
- The second method, Gaussian mixture model (GMM), is a more general method that can be seen as a probabilistic version of k-means. It assigns an observation, not to one cluster only, but probabilistically to many clusters.

Both methods use expectation-maximization (EM) to find clusters and assignments which are locally optimal.

1.1 Datasets

The first dataset we will use contains the locations of geo-tagged tweets. It is located in the file `data/tweets.txt`. Each line of the file is a pair (x, y) that represents the Cartesian coordinates of geo-tagged tweets. We obtained these coordinates by projecting the longitudes and latitudes corresponding to tweets. Figure 1 shows a representation of these tweets.

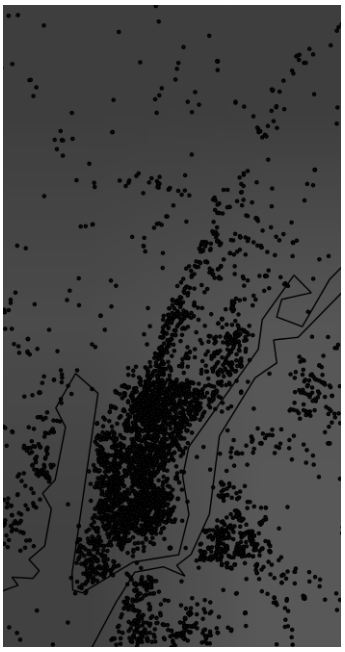


Figure 1: Snapshot showing the locations of geo-tagged tweets emitted from Manhattan.

We also provide you with synthetic data-points (x, y) generated by a mixture of 2-dimensional multivariate Gaussian distribution, located in the file `data/synthetic.txt`. The mixing coefficients π, mean and variance of the two Gaussian components (respectively μ_1 and $\mathbf{\Sigma}_1$, μ_2 and $\mathbf{\Sigma}_2$) are:

$$\pi = \begin{pmatrix} 0.256 \\ 0.744 \end{pmatrix}, \mu_1 = \begin{pmatrix} 4.490 \\ 5.201 \end{pmatrix}, \mathbf{\Sigma}_1 = \begin{pmatrix} 0.552 & 0.110 \\ 0.110 & 0.826 \end{pmatrix}, \mu_2 = \begin{pmatrix} 3.998 \\ 9.330 \end{pmatrix}, \mathbf{\Sigma}_2 = \begin{pmatrix} 0.393 & 0.544 \\ 0.544 & 0.580 \end{pmatrix}.$$

This synthetic dataset allows you to test the correctness of the results you obtain.

You can plot the data points using the `scatterPlot()` function in `DataUtils`. We encourage you to modify this function in order to plot specific clusters and their centers, as well as the points assignments. For example, in k-means, the color of each point could depend on the cluster to which it belongs.

1.2 k-MEANS

k-means uses an expectation-maximization approach in order to find locally-optimal solutions. We provide you with a partial implementation of k-means in the class `ix.clustering.Kmeans`.

- We already implemented the expectation step of k-means. Implemented the updates of the maximization step `mStep()`, and test it using the provided unit test.
- Run k-means on the synthetic dataset with $k = 2$ clusters. How does the distortion (fit of data to the clusters) measure evolve? Analyze the clusters and assignments found.
- In the `run()` method, we fix the number of iterations of k-means. Modify this method such that it stops as soon as it has converged. You can still keep the number of iterations as a maximum.
- Keeping the same number of clusters k, run your code on the dataset of tweets location. Analyze the results you obtain by plotting the evolution of different clusters and their centers. After convergence of k-means, how is the geographical distribution of the clusters? What does the clustering obtained capture?
  Hint: One might wonder if the borough of Manhattan is well captured.

1.3 MIXTURE OF GAUSSIANS

Now that you have implemented k-means, we will move to Gaussian mixture models (GMM), which can been seen as a probabilistic extension of k-means. We have provided you with a skeleton for the code in `ix.lab06.clustering.GaussianMixtureModel`.

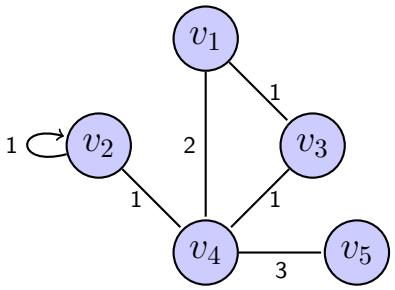
- Implement the updates of the expectation step of GMM in the `eStep()` function.
- Implement the updates of the maximization step of GMM in the `mStep()` function.
- What is the measure used in GMM that is the equivalent of the distortion measure in k-means?
- Run GMM with $k = 2$ components on the synthetic dataset. Interpret your results by observing the means, covariance matrices, responsibilities and mixture coefficients. Compare them to the parameters of the generating distribution.
- Run GMM on the tweets dataset and vary the number of components $k = 2, 3, 4$. Interpret the results you obtain by plotting the data points and the responsibilities of each cluster in generating them.
- How sensitive is GMM to initial conditions? Explain your answer by pointing out potential sources of instability.
- Give an example of a dataset where GMM might diverge.
  Hint: Imagine a very popular tweeting-place in Manhattan.

2 Community Detection

The second part of this lab is about community detection on graphs. In this part, we first explain the dataset representation, and then explore and implement the Louvain method for community detection. In this document, we use the terms vertex and node (edge and link) interchangeably.

2.1 Dataset

We study community detection on weighted and undirected graphs with self-loops. Here, each graph is represented by its edge list. In the graph's text file, each line is a TAB separated string which contains three tokens: edge-source, edge-destination and edge-weight. Figure 2a shows an example of a graph and Figure 2b the corresponding input file.



(a) Graph

```
v1      v3      1
v1      v4      2
v2      v2      1
v2      v4      1
v3      v4      1
v4      v5      3
```

(b) Input text file

Figure 2: Graph example and its corresponding input file for the community detection.

2.2 Modularity

The modularity of a weighted and undirected graph G with adjacency matrix A for a partitioning C is defined as

$$Q = \frac{1}{2m} \sum_{u,v} \left[A_{ij} - \frac{d_u d_v}{2m} \right] \delta(c_u, c_v), \tag{1}$$

where A_{uv} represents the weight of the edge between vertices u and v, $d_u = \sum_v A_{uv}$ is the sum of the weights of the edges attached to the vertex u, c_u is the community to which vertex u is assigned in the partitioning C, the function $\delta(i, j)$ is 1 if $i = j$ and 0 otherwise, and $m = \frac{1}{2} \sum_{u,v} A_{uv}$ is the total weight of the graph.

It is easy to show that modularity (1) is equivalent to

$$Q = \sum_{c=1}^{|C|} \left[\frac{\sigma_{\text{in},c}}{m} - \left(\frac{\Sigma_{\text{tot},c}}{2m} \right)^2 \right]. \tag{2}$$

Here, $|C|$ is the number of communities, $\sigma_{\text{in},c}$ is the sum of the weights of the edges which are inside the community c and $\Sigma_{\text{tot},c}$ is the sum of the degrees of the vertices in c.

2.3 Louvain Method for Community Detection

Most well-known community detection algorithms try to maximize modularity; however, maximizing modularity is a NP-hard problem. The Louvain method is a community detection algorithm that tries to maximize modularity in a greedy way. This algorithm runs iteratively in two steps:

1. First, we assign each node of the graph to its own community. Then, for each node u, we compute the gain in modularity by removing node u from its community and adding it to

each of the communities of its neighbors. The gain in modularity ΔQ obtained by moving an isolated node i into a community c can easily be computed from (2) as

$$\Delta Q = \left[\frac{\sigma_{\text{in},c} + d_{u,\text{in}}}{m} - \left(\frac{\Sigma_{\text{tot},c} + d_u}{2m}\right)^2\right] - \left[\frac{\sigma_{\text{in},c}}{m} - \left(\frac{\Sigma_{\text{tot},c}}{2m}\right)^2 - \left(\frac{d_u}{2m}\right)^2\right], \quad (3)$$

where $d_{u,\text{in}}$ is the sum of the weights of the edges from node u to nodes in the module c. It is easy to show that (3) simplifies to

$$\Delta Q = \frac{1}{m}\left(d_{u,\text{in}} - \frac{\Sigma_{\text{tot},c} d_u}{2m}\right). \quad (4)$$

The node u is added to the community which results in the highest gain in modularity, but only if this gain is positive. If no positive gain is possible, node u stays in its original community. This process is applied repeatedly and sequentially to all nodes until no further improvement can be achieved. Note that often a node is considered several times in this process.

2. We build a new network whose nodes are now the communities found during the first step. The weight of the links between new nodes is defined as the sum of the weight of the links between nodes in the two corresponding communities. Links between nodes of the same community are summed up and represented as a self-loop for this community in the new network.

These two steps are repeated iteratively until there are no more changes and a local maximum of modularity is achieved.

2.4 Implementation

The goal of this section is to implement the Louvain community detection algorithm.

- Based on (2), complete the `modularity()` function of `ix.lab06.community.Status`. Test your implementation using `ix.lab06.community.ModularityTest`.

During each iteration of the Louvain method, we need to try and assign each node to one of its neighboring communities, and check if the change increases the modularity. We thus need a method to get all neighboring communities of a node, as well as the sum of the weights of all edges going from this node to each community.

- Complete the `weightToNeighboringCommunities()` method of the `Status` class. Use `WeightToNeighboringCommunitiesTest` to test this method.

You are now ready to implement the first step of the algorithm. As a reminder, you need to iterate over each node, and remove it from its community. Then, you should try to add it to each of its neighboring communities, and compute the change in modularity. If there is at least one community such that the modularity increases, you should add the node to the one that increases the modularity the most, otherwise it should be put back in its original community.

- Complete the `assignCommunities()` method of the class `Status`. The goal of this method is to implement the first step of the Louvain algorithm. Use Equation (4) to compute the gain in modularity. You can use `weightToNeighboringCommunities()` to find all neighboring communities of a node, `removeNodeFromCommunity()` to remove a node from its community, and `insertNodeIntoCommunity()` to insert a node into a new community. Test your implementation with `AssignCommunitiesTest`.
  Hint: Do not iterate over the nodes more than `PASS_MAX` times.

We implemented the second step of Louvain in the `getNextLevel()` method of `Status`. This method computes the graph induced by the current level of Louvain, where each node represents a community. Now, we can finally combine these two steps to run the community detection algorithm.

- Implement the Louvain algorithm by completing the `communityDetection()` method of the class `Louvain`. Test the algorithm with `LouvainTest`.
- Run the algorithm for the graphs `karate.txt` and `simple.txt` found in the `data` folder:

```
java ix.lab06.community.Louvain data/karate.txt output/karate-level0.txt output/
    karate-best.txt
```

- You can use the python script `draw.py` to draw a graph and its communities:

```
python scripts/draw.py data/karate.txt output/karate-best.txt
```

  Use the above script to plot the communities at different levels, for both graphs. You can run these on your own machine, or on `ix-student{1,2}.epfl.ch`.

- Run the community detection algorithm on `data/wikipedia.txt`. Since this graph is large and it is not informative to draw all of its nodes and edges, we use `draw_wiki.py` to plot a node and its neighbors, and color them by community (note that this script does not show the edges, to keep the plot readable):

```
python scripts/draw_wiki.py data/wikipedia.txt output/wiki-best.txt data/wiki_nodes
    .txt ArticleName
```

  where `ArticleName` is the name of an article of interest, for example `Tehran` (you can find all article names in `data/wiki_nodes.txt`).

- Use script `wiki_comm.py` to print `nb` articles which are in the same community as the article `ArticleName`:

```
python scripts/wiki_comm.py output/wiki-level0.txt data/wiki_nodes.txt ArticleName
    nb
```