soluions_ex4

November 11, 2018

1 Weighted K-Means

In this exercise we will simulate finding good locations for production plants of a company in order to minimize its logistical costs. In particular, we would like to place production plants near customers so as to reduce shipping costs and delivery time.

We assume that the probability of someone being a customer is independent of its geographical location and that the overall cost of delivering products to customers is proportional to the squared Euclidean distance to the closest production plant. Under these assumptions, the K-Means algorithm is an appropriate method to find a good set of locations. Indeed, K-Means finds a spatial clustering of potential customers and the centroid of each cluster can be chosen to be the location of the plant.

Because there are potentially millions of customers, and that it is not scalable to model each customer as a data point in the K-Means procedure, we consider instead as many points as there are geographical locations, and assign to each geographical location a weight w_i corresponding to the number of inhabitants at that location. The resulting problem becomes a weighted version of K-Means where we seek to minimize the objective:

$$J(c_1, \ldots, c_K) = \frac{\sum_i w_i \min_k ||x_i - c_k||^2}{\sum_i w_i},$$

where c_k is the kth centroid, and w_i is the weight of each geographical coordinate x_i. In order to minimize this cost function, we iteratively perform the following EM computations:

- **Expectation step:** Compute the set of points associated to each centroid:

$$\forall\, 1 \leq k \leq K : \quad \mathcal{C}(k) \leftarrow \Big\{ i \;:\; k = \arg\min_k \|x_i - c_k\|^2 \Big\}$$

- **Minimization step:** Recompute the centroid as a the (weighted) mean of the associated data points:

$$\forall\, 1 \leq k \leq K : \quad c_k \leftarrow \frac{\sum_{i \in \mathcal{C}(k)} w_i \cdot x_i}{\sum_{i \in \mathcal{C}(k)} w_i}$$

until the objective $J(c_1, \ldots, c_K)$ has converged.

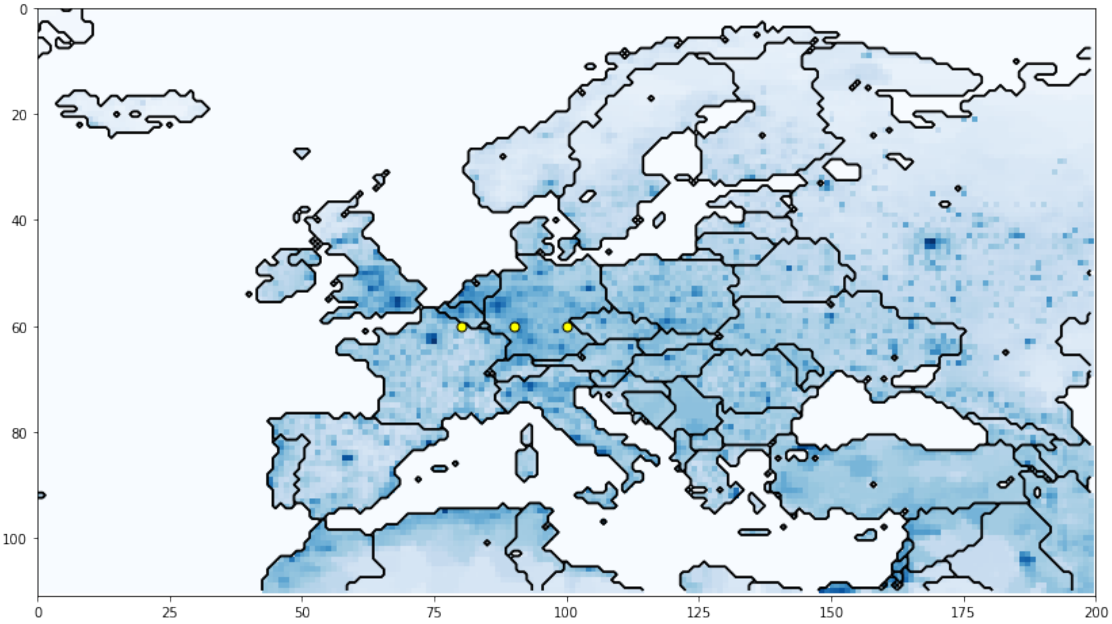
1.1 Getting started

In this exercise we will use data from http://sedac.ciesin.columbia.edu/, that we store in the files `data.mat` as part of the zip archive. The data contains for each geographical coordinates (latitude and longitude), the number of inhabitants and the corresponding country. Several variables and methods are provided in the file `utils.py`:

- `utils.population` A 2D array with the number of inhabitants at each latitude/longitude.
- `utils.countries` A 2D array with the country indicator at each latitude/longitude.
- `utils.nx` The number of latitudes considered.
- `utils.ny` The number of longitudes considered.
- `utils.plot(latitudes,longitudes)` Plot a list of centroids given as geographical coordinates in overlay to the population density map.

The code below plots three factories (white squares) with geographical coordinates (60,80), (60,90),(60,100) given as input.

```
In [1]: import utils
        %matplotlib inline
        utils.plot([60,60,60],[80,90,100])
```



1.2 Initializing Weighted K-Means (15 P)

Because K-means has a non-convex objective, choosing a good initial set of centroids is important. Centroids are drawn from from the following discrete probability distribution:

$$P(x,y) = \frac{1}{Z} \cdot \text{population}(x,y)$$

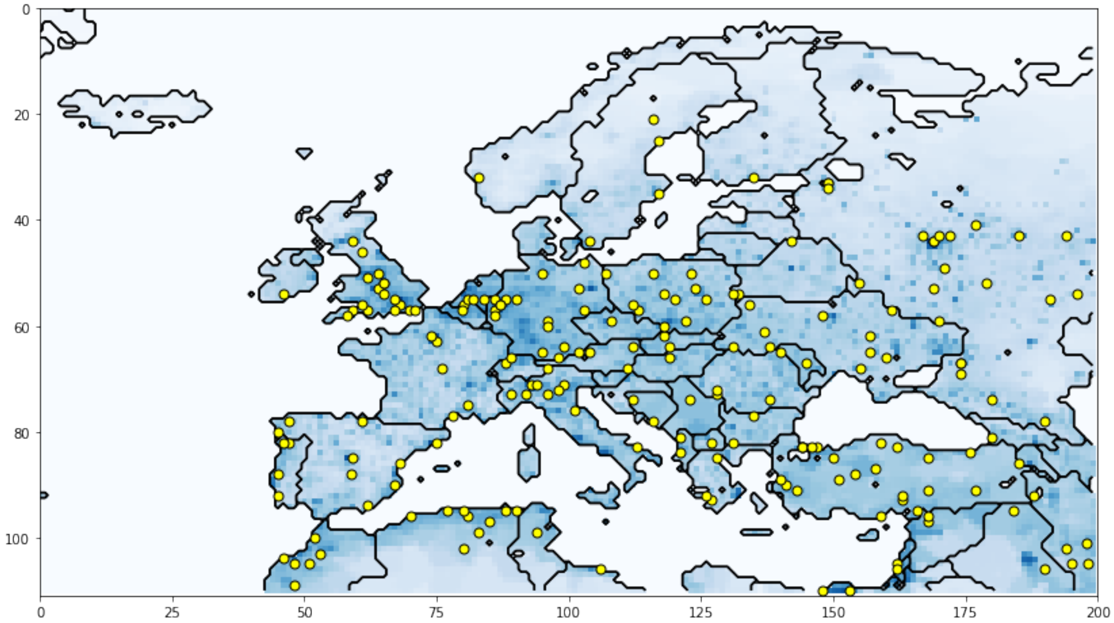
where Z is a normalization constant. Furthermore, to avoid identical centroids, we add a small Gaussian noise to the location of centroids, with standard deviation 0.01.

Tasks:

- **Implement the initialization procedure above.**
- **Run the initialization procedure for K=200 clusters.**
- **Visualize the centroids obtained with your initialization procedure using `utils.plot`.**

```
In [2]: # YOUR CODE HERE
        %matplotlib inline

        import numpy as np

        K = 200

        pop = utils.population
        print(pop.shape)
        pop_dist = pop/np.sum(pop)

        pop_1d = np.reshape(pop,pop.shape[0]*pop.shape[1])
        pop_dist_1d = np.reshape(pop_dist,pop_dist.shape[0]*pop_dist.shape[1])

        cen_ind_1d = np.random.choice(range(len(pop_dist_1d)),K,p=pop_dist_1d)

        centroids = np.empty((len(cen_ind_1d),2))

        for i,v in enumerate(cen_ind_1d):
            #get the y coordinate
            centroids[i][0] = int(v/utils.ny)
            #get the x coordinate
            centroids[i][1] = v%utils.ny

        #apply gaussian noisse to centers
        centroids = np.random.normal(centroids,0.01)

        utils.plot(np.transpose(centroids)[0],np.transpose(centroids)[1])



        # --------------
```

```
(111, 200)
```



1.3 Implementing Weighted K-Means (30 P)

Tasks:

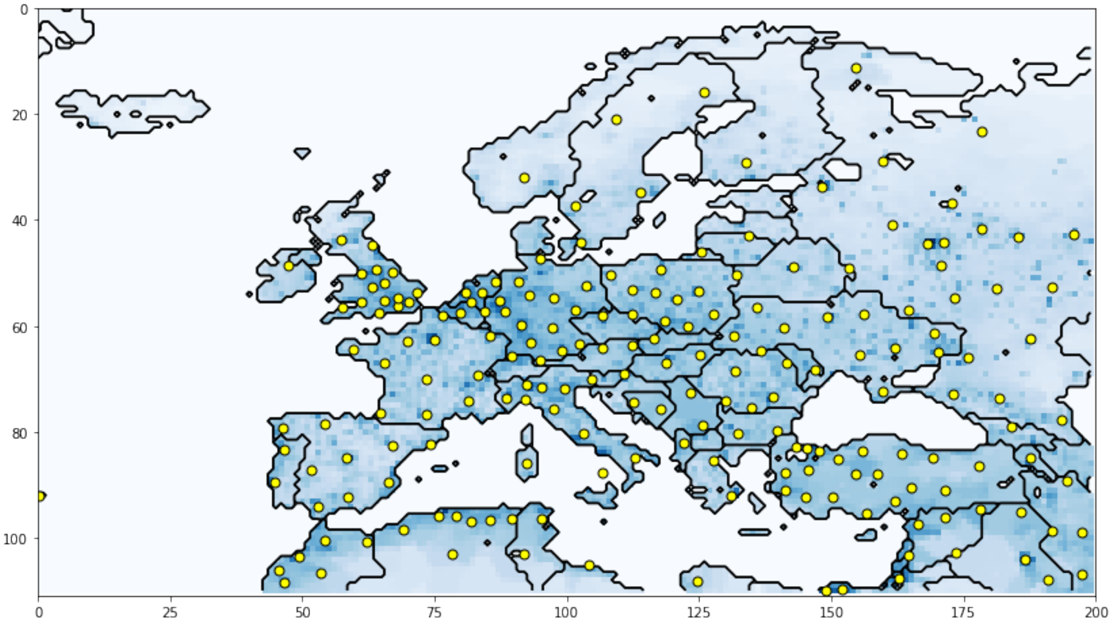
- **Implement the weighted K-Means algorithm as described in the introduction.**
- **Run the algorithm with K=200 centroids until convergence (stop if the objective does not improve by more than 0.01). Convergence should occur after less than 50 iterations. If it takes longer, something must be wrong.**
- **Print the value of the objective function at each iteration.**
- **Visualize the centroids at the end of the training procedure using the methods** `utils.plot`**.**

```
In [5]: from scipy.spatial.distance import cdist

        #create array of all possible points
        points = np.array(np.meshgrid(range(utils.nx),range(utils.ny))).T.reshape(-1,2)

        #create an matrix with column for points(X,Y) and weight(W) for each coordinate
        XYW = np.column_stack((points,pop_1d))

        #get first value for J
        JD = cdist(centroids,points) # get all distances from each centroid to each point
        JDmin = np.amin(JD,axis = 0) #get the min distances for each point, which is to the clos
        J = np.dot(XYW[:,2],JDmin)/np.sum(XYW[:,2]) # compute J
```

```
print("J before EM is ",J)

#add delta 1 so loop starts for first iteration
Jold = J+1

#cunter variable
counter = 0

#keep looping until convergence criteria reached
while(Jold-J>0.01):

    #set J to old J
    counter = counter+1
    Jold = J

    print("Starting iteration ",counter)

    #EXPECTATION STEP
    #get the indeces of closest cluster for each point
    D = cdist(centroids,points)
    DminI = D.argmin(axis = 0)

    #add array of index for closest cluster to matrix
    CiXYW = np.column_stack((DminI,XYW))

    #MINIMIZATION
    #iterate through every centroid
    for k in range(K):

        #get all points which are closest to cluster k
        ck = CiXYW[np.where(CiXYW[:,0] == k)]

        #compute weight of all points in cluster
        sumW = np.sum(ck[:,3])

        #only recompute centroid if it has any weights, since otherwise divide by 0
        if(sumW != 0):

            #comupute X*W and Y*W
            XW = ck[:,1]*ck[:,3];
            YW = ck[:,2]*ck[:,3];

            #get new centroid
            centroids[k] = [np.sum(XW)/sumW, np.sum(YW)/sumW]

        #if Wsum is 0, notify
        else:
            print("weight in c",k," is 0, moving on")
    #get new value for J
    JD = cdist(centroids,points) # get all distances from each centroid to each point
    JDmin = np.amin(JD,axis = 0) #get the min distances for each point, which is to the
    J = np.dot(XYW[:,2],JDmin)/np.sum(XYW[:,2]) # compute J

    print("new J = ",J)

#print result when converged
print("converged after",counter, "iterations")
print("final J is",J)
```

```
J before EM is  2.953941108400821
Starting iteration  1
weight in c 122  is 0, moving on
new J =  2.476300859124441
Starting iteration  2
new J =  2.35122658258938
Starting iteration  3
new J =  2.2827192068156625
Starting iteration  4
new J =  2.2489970961399335
Starting iteration  5
new J =  2.229752306510785
Starting iteration  6
new J =  2.2064036217681333
Starting iteration  7
new J =  2.1798057372640116
Starting iteration  8
new J =  2.166944815985929
Starting iteration  9
new J =  2.157783133132925
converged after 9 iterations
final J is 2.157783133132925
```

```
In [6]: utils.plot(np.transpose(centroids)[0],np.transpose(centroids)[1])
```



```
In [ ]:
```