Homework 9
APPM 5650 Fall 2021
Randomized Algorithms

Due date: Monday, Oct 25 2021 **Instructor**: Prof. Becker
Theme: Randomization for K-means clustering **Revision date**: 10/22/2021

Instructions Collaboration with your fellow students is allowed and in fact recommended, although direct copying is not allowed. Please write down the names of the students that you worked with. The internet is allowed for basic tasks only, not for directly looking for solutions.

An arbitrary subset of these questions will be graded.

In this homework, we look at the K-means clustering problem and the standard algorithm to solve it (called Lloyd's algorithm, or usually just called the K-means algorithm). Let $X \in \mathbb{R}^{n\times p}$ be a dataset where each row is an observation, so we have observations $\{x^{(i)}\}_{i=1}^n \subset \mathbb{R}^p$ (so technically each $x^{(i)}$ is a row vector, but think of it as just a vector). I use this row-wise convention since it matches Matlab's default clustering convention. We let K denote the total number of clusters we'd like to find. In K-means algorithms, a cluster is defined by a cluster center $\mu^{(j)}$, and let $\mathcal{M} = \{\mu^{(j)}\}_{j=1}^K$ be the set of all cluster centers. The clustering/partition of a data point x is then determined by computing

$$d_x \stackrel{\text{def}}{=} \min_{\mu\in\mathcal{M}} \|x-\mu\|_2, \quad j_x \stackrel{\text{def}}{=} \operatorname*{argmin}_{\mu\in\mathcal{M}} \|x-\mu\|_2. \tag{1}$$

We will look at a randomized variant known as "K-means++"[1] which has had a big impact on machine learning. This method takes only K steps, and produces a set $\mathcal{M}$ which is guaranteed, in expectation, to be not so far from the optimal set of cluster centers; specifically, the expected value of the clustering objective value is within $8(\ln(K)+2)$ times the minimum objective value. This is a big deal because finding the true optimal set of cluster centers is a NP-hard problem, even if $K = 2$ (see [2]) or even if $p = 2$ (see [3]).

procedure K-MEANS++(X, K) ▷ X is $n \times p$, K is number of clusters
  $\mathcal{M} \leftarrow \emptyset$ ▷ The set of cluster centers
  Choose $j \sim$ Uniform$[1,\ldots,n]$
  $\mathcal{M} \leftarrow \mathcal{M} \cup \{x^{(j)}\}$
  for $k = 2,\ldots,K$ **do**
    Compute $d_i \equiv d_{x^{(i)}}$ via (1) for $x^{(i)}$, $i = 1,\ldots,n$ ▷ This depends on $\mathcal{M}$ of course
    Choose j at random from a weighted distribution with probabilities $\{d_i^2\}_{i=1}^n$
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{x^{(j)}\}$
  end for
end procedure

The very basic idea is that selecting points that are far away from the existing cluster centers is a good way to find new clusters. If this is such a good idea, why not just deterministically pick the one point that is farthest away? Such a short-sighted choice is a "greedy choice". We'll formalize that method in the procedure below:

[1] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding", SODA 2007
[2] D. Aloise, A. Deshpande, P. Hansen, P. Popat, "NP-hardness of Euclidean sum-of-squares clustering". Machine Learning. 75: 245–249, 2009
[3] M. Mahajan, P. Nimbhorkar, K. Varadarajan, The Planar k-means Problem is NP-hard, Lecture Notes in Computer Science. 5431: 274–285, 2009

1: **procedure** GREEDY K-MEANS(X, K)
Steps 2–6 and 8–10 same as K-means++
7: Choose j deterministically as the index of the maximal value of $\{d_i^2\}_{i=1}^n$
10: **end procedure**

Note that in practice, K-means++ is not a full algorithm itself but used to initialize a K-means algorithm (since Lloyd's algorithm can only improve the K-means objective every iteration). For our homework, we'll just stick with the initial K-means++ and not do anything afterwards.

Problem 1: [CODING] Compare the performance of K-means++ and "Greedy K-means" (do this over many runs; each run, K-means++ makes new random choices, and Greedy K-means makes a new initial selection for step 3).

Do this on the full MNIST test data, in `mnist_data_all.mat` which is on Canvas. This has 4 variables, `Train` which is 60000×784 (e.g., $n = 60000$ and $p = 784 = 28^2$); `Test` which is 10000×784; `Train_labels` which is 60000×1 and contains the true labels of the training data points, so an integer 0–9; and `Test_labels` which has the true labels of the testing data points.

Run the algorithms on the training data, for $K = 10$, then evaluate error on both testing and training data. Note: evaluate the classification error (the number of correctly classified points), *not* the K-means objective error (though you can do that too if you like).

Deliverable: A plot (e.g., a box-plot aka box-and-whiskers plot) showing the performance of the two methods for both testing and training error on the MNIST data; and, a short discussion of the results.

Hints: the amount of coding can be reduced if you use Matlab's `pdist2` function (requires Stats toolbox), or my variant `pdist2_faster` available on github [which as faster than `pdist2` a few years ago, but may not be anymore]; other routines like `dsearchn` could probably be used instead as well. Read the documentation carefully; you may want to get the *squared* Euclidean norm; you can also request the code to output the quantity j_x from (1) which is useful in evaluating the accuracy of your method on the training and testing data. In python, with SciPy, see scipy.spatial.distance.cdist.html; or with Scikit-learn, sklearn.metrics.pairwise_distances.html.

Perhaps the biggest amount of coding will be to actually evaluate the accuracy of your code. There's one major problem: the true data has labels $(0, 1, \ldots, 9)$, and your algorithm will give $K = 10$ clusters (say, $(a, b, c, d, e, f, g, h, i, j)$), but you don't know what the correspondence between these clusters is. One method to do this, which is optimal under a certain criteria, is the "Hungarian algorithm" or "Kuhn-Munkres" algorithm. For code to do this in Matlab, see github.com/ZJULearning/MatlabFunc/tree/master/Tools and the `bestMap.m` file (which relies on `hungarian.m`). For python, see scipy.optimize.linear_sum_assignment.html (in old versions of Scikit-learn, you can use linear_assignment_.py but this is now deprecated in favor of the scipy function). In Python, this `linear_sum_assignment` does the Hungarian algorithm, but you still need to build up a cost matrix / contingency table to give to the algorithm. See the `bestMap.m` file for how to do that, or try following `contigency_matrix` from github.com/mwburke/cluster-stability/blob/master/cluster_stability.py (or maybe you can get `scipy.stats.contingency.crosstab` to do this? I haven't tested any of these solutions in Python myself).

You may want to run plain K-means on the MNIST data, to get a feel for what to expect. Using Matlab's `kmeans`, I get about $51\% \pm 2\%$ accuracy on the training and testing data (compared to a benchmark of 10% accuracy for random guessing, since we have 10 classes). You may also want to randomly subsample the training data to speed up the algorithm; for example, I found that reducing the training data from 10000 to 2000 had very little effect on overall accuracy.

Bonus (not required, not for credit): how does performance change if you select weights based on $\{d_i\}$ instead of $\{d_i^2\}$?