HW6 Solution

Original problem set description: http://www.cs.tufts.edu/comp/135/2019s/hw6.html (http://www.cs.tufts.edu/comp/135/2019s/hw6.html)

Problem 1: Principal Components Analysis (PCA)

You can assume you have a training set X with N examples, each one a feature vector with F features.

1a: True/False

1a(i) We compute principal components by finding eigenvectors of the dataset's feature matrix X.

Answer: FALSE.

We compute eigenvectors/eigenvalues of the **covariance** matrix for a dataset X.

1a(ii): To select the number of components K to use, we can find the value of K that minimizes reconstruction error on the training set. This choice will help us manage accuracy and model complexity.

Answer: FALSE

If we chose K to minimizing reconstruction error on training set, we'll always choose K = F (perfect reconstruction using all available features).

1a(iii): If we already have fit a PCA model with $K = 10$ components, fitting a PCA model with $K = 9$ components for the same dataset is easy.

Answer: TRUE

The PCA model with K=9 just requires keeping the 9 eigenvectors with largest eigenvalues, out of the 10 we already have.

Problem 1b

Suppose we had a dataset for a medical application, with many measurements describing the patient's height, weight, income, daily food consumption, and many other attributes.

1b(i): Would the PCA components learned be the same if you used feet or centimeters to measure height? Why or why not?

```
**Answer**: No, the components learned would NOT be the same.

The choice of units can impact the scalar values appearing in the height
feature column. Thus, the choice can potentially make this the column with
the *largest* variance, or *smallest* variance, or somewhere in between.

Because the components are chosen to maximize variance, the direction of
these vectors is dependent on the choice of units.
```

1b(ii): Before applying PCA to this dataset, what (if any) preprocessing would you recommend and why?

Probably the ideal thing to do is make sure each numerical feature column is "normalized" to have mean of zero and variance of one.

This is the strategy recommended in the ISL textbook Chapter 10 (you can read more there about why, see Fig 10.3 and related discussion).

Problem 1c

Stella has a training dataset $X = \{x_n\}_{n=1}^N$, which has N example feature vectors x_n each with F features. She applies PCA to her training set to learn a matrix W of shape (K, F), where each row represents the basis vector for component k.

She would like to now project her test dataset $X' = \{x'_t\}_{t=1}^T$ using the same components W. She remembers that she needs to **center** her data, so she applies the following 3 steps to project each F-dimensional test vector x'_t to a K-dimensional vector z'_t:

$$
\begin{aligned}
m &= \frac{1}{T}\sum_{t=1}^{T} x'_t \qquad \text{(compute the mean)} \\
\tilde{x}'_t &= x'_t - m \qquad \text{(center the data by subtracting mean)} \\
z_t &= W\tilde{x}'_t \qquad \text{(project down to K-dimensions)}
\end{aligned}
$$

Is this correct? If so, explain why. If not, explain what Stella should do differently.

Answer: No, this is not quite correct. Stella should compute the mean of the TRAINING set, not the test set, to do the centering. Otherwise she can't use the component vectors W learned on the training set.

One way to think about this is that the test-time reconstruction should NOT depend on what data you happen to be testing with. If you only have one vector in the test set, you shouldn't just set it to all zeros and then project. That wouldn't make any sense, because any single vector you test with

would be transformed to all zeros and then reconstructed to the **same** vector.

In [ ]:

Problem 2: K-Means Clustering

Problem 2a: True/False

2a(i): We always get the same clustering of dataset X when applying K-means with $K = 1$, no matter how we initialize the cluster centroids μ

Answer: TRUE.

When $K = 1$, we get a special case where $z_{n1} = 1$ for all examples n, and the minimum distance objective is a simple convex quadratic:

$$\min_{\mu} \sum_{n=1}^{N} (x_n - \mu)^2$$

Thus, we get a unique answer every time: $\mu = \text{mean}(x_1, \ldots x_N)$

2a(ii): We always get the same clustering of dataset X when applying K-means with $K = 2$, no matter how we initialize the cluster centroids μ

Answer: FALSE

Consider the 4 example, 2 cluster case on the slides discussed in class here:
http://www.cs.tufts.edu/comp/135/2019s/slides/23_clustering.pdf#page=30
(http://www.cs.tufts.edu/comp/135/2019s/slides/23_clustering.pdf#page=30)

The cluster centroid initialization matters!

2a(iii): The only way to find the cluster centroids μ that minimize the K-means cost function (minimize sum of distances to nearest centroid) is to apply the K-means algorithm, alternatively updating assignments and cluster centroid locations.

Answer: FALSE

Many other algorithms are possible. Remember that an optimization problem can be solved in many ways. For example, we could enumerate all possible clusterings of the data given its fixed size N and a number of clusters K, compute the cost function for each one, and pick the best. This would be terribly inefficient for large datasets, but still a way to solve the problem.

A better way that's not brute force (also discussed in class) is a stochastic gradient descent algorithm. This approach is outlined here:

http://papers.nips.cc/paper/989-convergence-properties-of-the-k-means-algorithms.pdf (http://papers.nips.cc/paper/989-convergence-properties-of-the-k-means-algorithms.pdf)

2a(iv): The K-means cost function requires computing the Euclidean distance from each example x_n to its nearest cluster centroid μ_k. Because the Euclidean distance requires a square root operation (e.g. `np.sqrt` or `np.pow(___, 0.5)`), no implementation of the K-means coordinate descent algorithm can be correct unless a "sqrt" operation is performed when computing distances between examples and clusters.

Answer: FALSE

All we care about when performing the assign-to-nearest-cluster step (the "E" step) is which cluster is **closest** to the current example. So as long as we consistently compute any monotonic transform of the distance (for example, the log of the distance, the square of the distance, etc.), and compute the minimum among those transformed distance values, the algorithm can be correct.

In practice, this often means the "squared" Euclidean distance is used, because it's cheaper than computing the square root.

$$sqEuclideanDist(n,k) = \sum_f (x_{nf} - \mu_{kf})^2, \quad \text{for } k \in 1, 2, \dots K$$

$$k^* = \text{argmin}_k \; sqEuclideanDist(n,k)$$

$$z_{nk^*} = 1$$

Problem 2b

2b: Suppose you are given a dataset X with N examples, as well as a group of $K = 5$ cluster locations $\{\mu_k\}_{k=1}^K$ fit by applying the K-means coordinate descent algorithm to this dataset. You know $N > 5$. Describe how you could initialize K-means using $K = 6$ clusters to obtain a better cost than the $K = 5$ solution.

Answer:

Initialize the first five cluster centers to the locations found via k-means with K=5: $\mu_1, \mu_2, \dots \mu_5$.

Then, for the remaining centroid location, select a current example n whose location is not already exactly the same as one of the current centroids. That is, the selected example's distance to its nearest center must be **strictly greater** than zero. Then, we set $\mu_6 = x_n$. There must be such an

example, because there are more examples than the 5 clusters (We'll assume all example vectors are distinct).

This K=6 solution must have strictly lower cost than the K=5 solution, because the distance from each example to its closest center can only decrease. Either each example will have its nearest cluster in the first 5 (in which case that example's cost is the same) or the new centroid (in which case that example's cost goes down). There's at least one example guaranteed to decrease its cost.

Problem 2c

2c: Suppose you are using sklearn's implementation of K-means (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html) to fit 10 clusters to a dataset.

You start with code like this:

```
    kmeans = sklearn.cluster.KMeans(
        n_clusters=10, random_state=42, init='random', n_init=1, al
gorithm='full')
    kmeans.fit(X)
```

List at least two changes you might make to these keyword arguments to improve the quality of your 10 learned cluster location vectors (as measured by their K-means cost function applied to the **training data**).

Answer:

- Set the number of separate initializations `n_init > 1` , to as large a value as we can reasonably afford (in runtime). Taking the best of multiple initializations is **always recommended** to achieve the best cost function on the training set.
- Set the initialization procedure to 'kmeans++'. This has guarantees that the random selection procedure does not, and should in general yield much better cost function values.

Problem 2d

2d: Consider the two steps of the k-means coordinate descent algorithm, `assign_examples_to_clusters` and `update_cluster_locations` .

2d(i): What is the big-O runtime of `assign_examples_to_clusters` ? Justify your answer. Express in terms of $N, K, and F$.

Answer: $O(NKF)$.

At each example, we need to compute its distance from each current cluster center.

Each euclidean distance calculation between two vectors of size F has cost $O(F)$.

There are $N * K$ such calculations.

Thus, the total cost is $O(N * K * F)$

2d(ii): What is the big-O runtime of `update_cluster_locations`? Justify your answer. Express in terms of N, K, and F.

Answer: $O(NF + KF)$

Justification: we have to touch each example exactly once, and add all its F features to one of the cluster centroids. So this part is O(NF). There's also some steps that are

A good implementation would look like:

```
mu_KF = np.zeros((K, F)) # O(K * F) storage allocation
n_K = np.zeros((K))      # O(K), counts # examples assigned to each
k)

for n in range(N):       # Entire loop is O(N*F)
    k = z_N[n]           # -- one access, O(1)
    mu_KF[k] += x_NF[n]  # -- one addition op for each feature, O
(F)
    n_K[k] += 1          # -- one addition, O(1)

for k in range(K):       # Entire loop is O(K*F)
    mu_KF[k] /= n_K[k]   # -- one division for each feature, O(F)

return mu_KF
```

In [ ]: