CPSC 340 Assignment 2 (due October 7)

K-Nearest Neighbours, Random Forests, K-Means, Density-Based Clustering

1 K-Nearest Neighbours

In this question we revisit the *citiesSmall* dataset from the previous assignment. In this dataset, nearby points tend to receive the same class label because they are part of the same state. This indicates that a $k-$nearest neighbours classifier might be a better choice than a decision tree. The file *knn.m* has implemented the training function for a $k-$nearest neighbour classifier (which is to just memorize the data).

1.1 KNN Prediction

Fill in the *predict* function in *knn.m* so that the model file implements the k-nearest neighbour prediction rule (based on Euclidean distance).

1. Hand in your predict function.
2. Report the training and test error obtained on the *citiesSmall.mat* dataset for $k = 1$, $k = 3$, and $k = 10$.
3. Hand in the plot generatied by classifier2Dplot on the *citiesSmall.mat* dataset for $k = 1$. (Note that this version of the function also plots the test data.)
4. Why is the training error 0 for $k = 1$?
5. If you didn't have an explicit test set, how would you choose k?

Note: Matlab can be slow at executing operations in 'for' loops, but allows extremely-fast hardware-dependent vector and matrix operations. By taking advantage of SIMD registers and multiple cores (and faster matrix-multiplication algorithms), vector and matrix operations in Matlab will often be several times faster than if you implemented them yourself in a fast language like C. If you find that calculating the Euclidean distances between all pairs of points takes too long, the following code will form a matrix containing the squared Euclidean distances between all training and test points:

```
[n,d] = size(X);
[t,d] = size(Xtest);
D = X.^2*ones(d,t) + ones(n,d)*(Xtest').^2 - 2*X*Xtest';
```

Element $D(i, j)$ gives the squared Euclidean distance between training point i and testing point j.

1.2 Condensed Nearest Neighbours

The file *citiesBig1.mat* contains a version of this dataset with more than 30 times as many cities. KNN can obtain a lower test error if it's trained on this dataset, but the prediction time will be very slow. A common strategy for applying KNN to huge datasets is called *condensed nearest neighbours*, and the main

idea is to only store a *subset* of the training examples (and to only compare to these examples when making predictions). The most common variation of this algorithm works as follows:

- Our initial subset just contains the first training example.
- At training time, we go through the examples in order.
- If the example is incorrectly classified by the KNN classifier using the current subset, add it to the subset.
- If the example is correctly classified by the KNN classifer, do *not* add it to the subset.

Implement the *condensed nearest neighbours* algorithm as described above in a function called *cnn*.

1. Hand in your *cnn.m* code.
2. Report the training and testing errors, as well as the number of variables in the subset, on the *citiesBig1.mat* dataset with $k = 1$.
3. Hand in the plot generated by *plotClassifier* on the *citiesBig1.mat* dataset for $k = 1$
4. Why is the training error with $k = 1$ now greater than 0?
5. If you have s examples in the subset, what is the cost of running the predict function on t test examples in terms of n, d, t, and s?
6. Try out your function on the dataset in *citiesBig2.mat*. Why are the test error *and* training error so high (even for $k = 1$) for this method on this dataset?

2 Random Forests

The file *vowels.mat* containing a supervised learning dataset where we are trying to predict which of the 11 "steady-state" English vowels that a speaker is trying to pronounce.

2.1 Random Trees

The functions *decisionTree* and *decisionStump* implement a (non-random) decision tree that is built using greedy recursive splitting and information gain. The function *randomTree* calls a function *randomStump* (which is not included) to fit a random decision tree.

1. Make a plot of the training error and the test error for the *decisionTree* model, as the depth is varied from 1 to 15.
2. Why does the *decisionTree* function terminate if you set the *depth* parameter to ∞?
3. Copy the *decisionStump* function to a new function called *randomStump*. Modify the *randomStump* so that it only considers $\lfloor \sqrt{d} \rfloor$ randomly-chosen features. Hand in the new training function and make a plot of the training and test error of the now-working *randomTree* model as the depth is varied from 1 to 15 with this method (it should not be the same every time, so just include one run of the method).
4. Make a third training/test plot where you use the *decisionTree* model but for each depth you train on a different boostrap sample of the training data (but evaluate the training error on the original training data).

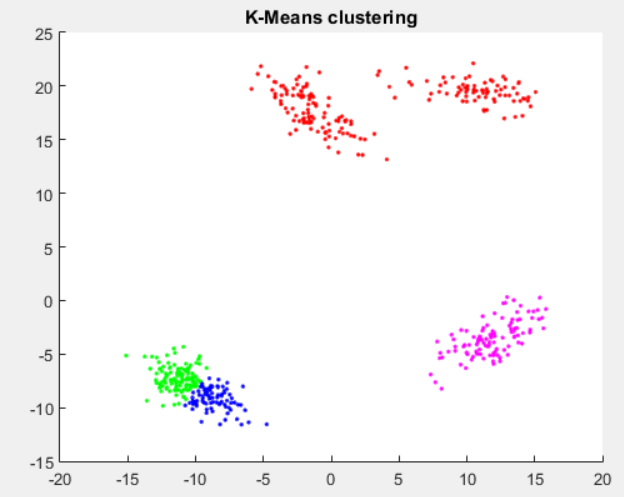
2.2 Random Decision Forests

The function *decisionforest* function repeatedly calls *decisionTree* to fit a set of models, and for prediction averages their results.

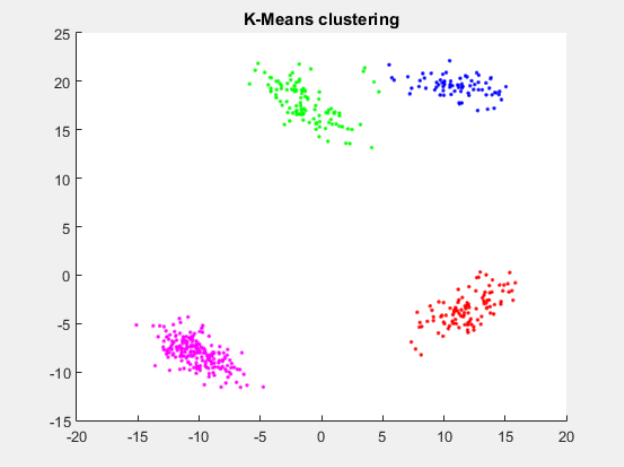
1. Report the test error of the *decisionForest* classifier with a depth of ∞ and 50 trees.
2. Report the test error of the *decisionForest* classifier with a depth of ∞ and 50 trees, if each tree is trained on a different boostrap sample.
3. Report the test error of the *decisionForest* classifier with a depth of ∞ and 50 trees, if you train on the original dataset but use *randomTree* instead *decisionTree* to fit the models.
4. Report the test error of the *decisionForest* classifier with a depth of ∞ and 50 trees, if you use *randomTree* and each tree is trained on a different boostrap sample. Hand in your modified *decisionForest.m* function (which is now a random forest model).
5. What is the effect of the two ingredients of random forests, bootstrapping and random splits, on the performance on this dataset?

3 K-Means Clustering

If you run the function *example_Kmeans*, it will load a dataset with two features and a very obvious clustering structure. It will then apply the k-means algorithm with a random initialization. The result of applying the algorithm will thus depend on the randomization, but a typical run might look like this:



But the 'correct' clustering (that was used to make the data) is this:



(Note that the colours are arbitrary due to label switching.)

3.1 Selecting among Initializations

If you run the demo several times, it will find different clusterings. To select among clusterings for a *fixed* value of k, one strategy is to minimize the sum of squared distances between examples x_i and their means w_{c_i},

$$f(w_1, w_2, \ldots, w_k, c_1, c_2, \ldots, c_n) = \sum_{i=1}^{n} \|x_i - w_{c_i}\|_2^2 = \sum_{i=1}^{n} \sum_{j=1}^{d} (x_{ij} - w_{c_i j})^2.$$

where c_i is the closest mean to x_i. This is a natural criterion because the steps of k-means alternately optimize this objective function in terms of the w_c and the c_i values.

1. In the *clusterKmeans.m* file, add a new function called *error* that takes the same input as the *predict* function but that returns the value of this above objective function. Hand in your code.
2. Using the *clustering2Dplot* file, output the clustering obtained by running k-means 50 times (with $k = 4$) and taking the one with the lowest error.

3.2 Selecting k

We now turn to the much-more-difficult task of choosing the number of clusters k.

1. Explain why the above objective function cannot be used to choose k.
2. Explain why even evaluating this objective function on test data still wouldn't be a suitable approach to choosing k.
3. Hand in a plot of the minimum error found across 50 random initializations, as you vary k from 1 to 10.
4. The *elbow method* for choosing k consists of looking at the above plot and visually trying to choose the k that makes the sharpest "elbow" (the biggest change in slope). What values of k might be reasonable according to this method?

3.3 k-Medians

The data in *clusterData2.mat* is the exact same as the above data, except it has 4 outliers that are very far away from the data.

1. Using the *clustering2Dplot* file, output the clustering obtained by running k-means 50 times (with $k = 4$) on *clusterData2.mat* and taking the one with the lowest error.
2. What values of k might be chosen by the elbow method for this dataset?
3. Implement the *k-medians* algorithm, which assigns examples to the nearest w_c in the L1-norm and to updates the w_c by setting them to the "median" of the points assigned to the cluster (we define the d-dimensional median as the concatenation of the median of the points along each dimension). Hand in your code.
4. Using the L1-norm version of the error (where c_i now represents the closest median in the L1-norm),

$$f(w_1, w_2, \ldots, w_k, c_1, c_2, \ldots, c_n) = \sum_{i=1}^{n} \|x_i - w_{c_i}\|_1 = \sum_{i=1}^{n} \sum_{j=1}^{d} |x_{ij} - w_{c_i j}|,$$

what value of k would be chosen by the elbow method under this strategy?

4 Vector Quantization and Density-Based Clustering

In this question we'll look at vector quantization, an alternative way that k-means is commonly used. Then we'll start to explore alternative clustering methods like density-based clustering.

4.1 Image Colour-Space Compression

Discovering object groups is one motivation for clustering. Another motivation is vector quantization, where we find a prototype point for each cluster and replace points in the cluster by their prototype. The file *dog.mat* contains a 3D-array I representing the RGB values of a picture of a dog. You can view the picture by typing *image(I/255)*. Write a funtion called *quantizeImage* with the header

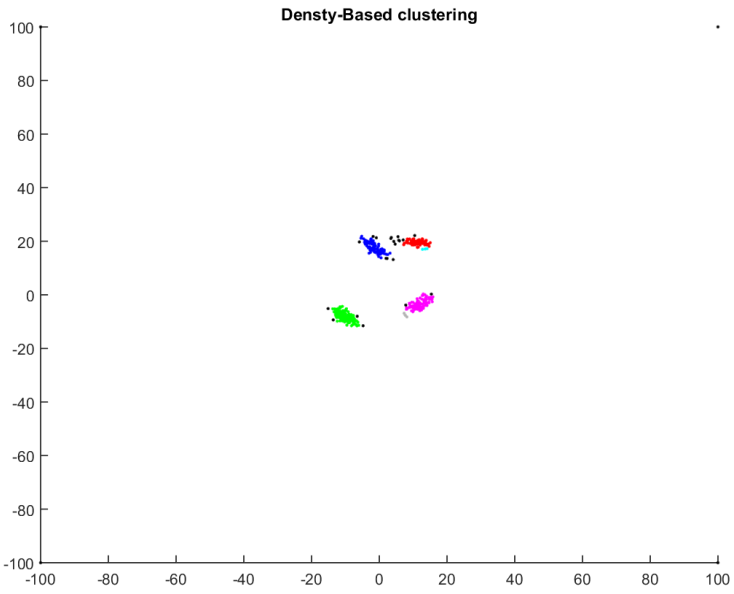
- function [Iquant] = quantizeImage(I,b)

where I is an image, b is the number of bits used to represent the colour-space, and $Iquant$ is a version of the image where each pixel's colour can be encoded using only b bits by replacing the original pixel's colour with the nearest prototype. You should find the prototypes using the *clusterKmeans* function, and recall that with b bits you can represent 2^b numbers, so the number of clusters should be $k = 2^b$.

1. Hand in your *quantizeImage* function.
2. Show the image obtained if you encode the colours using 1, 2, 4, and 6 bits.

4.2 Effect of Parameters on DBSCAN

If you run the function *example_DBscan*, it will apply the basic density-based clustering algorithm to the dataset from the previous assignment. The final output should look like this:



Even though we know that each object was generated from one of four clusters (and we have 4 outliers), the algorithm finds 6 clusters and does not assign some objects to any cluster. However, the assignments will change if we change the parameters of the algorithm. Find and report values for the two parameters (*radius* and *minPts*) such that the density-based clustering method finds:

1. The 4 "true" clusters.
2. 3 clusters (merging the top two, which also seems like a reasonable interpretaition).
3. 2 clusters.
4. 1 cluster.

4.3 K-Means vs. DBSCAN Clustering

If you run the function *example_animals*, it will load a dataset containing 85 attribute values for 50 animals. It will then apply a k-means clustering to the animals, and report the resulting clusters. The exact clustering will depend on the initialization of k-means and the value of k, but below is the result of one of the runs:

- Cluster 1: killer+whale blue+whale hippopotamus humpback+whale seal walrus dolphin
- Cluster 2: elephant ox sheep rhinoceros buffalo giant+panda pig cow
- Cluster 3: skunk mole hamster squirrel rabbit mouse raccoon
- Cluster 4: antelope horse moose spider+monkey gorilla chimpanzee giraffe zebra deer
- Cluster 5: grizzly+bear beaver dalmatian persian+cat german+shepherd siamese+cat tiger leopard fox bat wolf chihuahua rat weasel otter bobcat lion polar+bear collie

Some of these groupings make sense (cluster 1 contains fairly-large animals that live in or near the water) while others do not (grizzly bears and bats are both in cluster 5).

1. Sometimes when you run this demo, an entire row of $model.W$ has NaN values. Why does this happen?

2. Modify this demo to use the density-based clustering method, and report the clusters obtained if you set *minPoints* to 3 and the radius such that it finds 5 clusters.