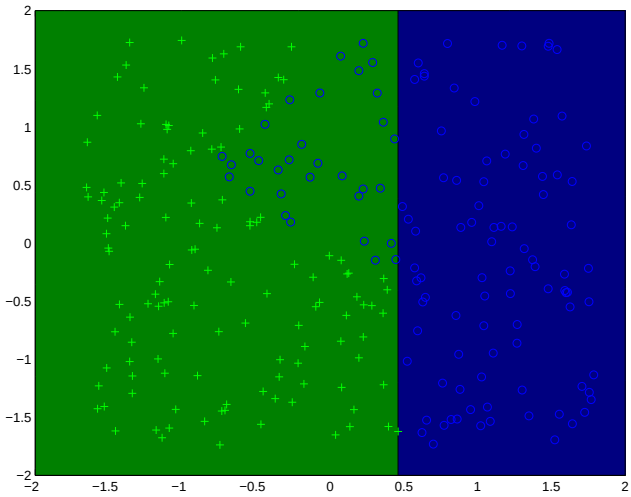
CPSC 340 Assignment 2 (due October 2nd)

Frequency-Based Supervised Learning, K-Means Clustering

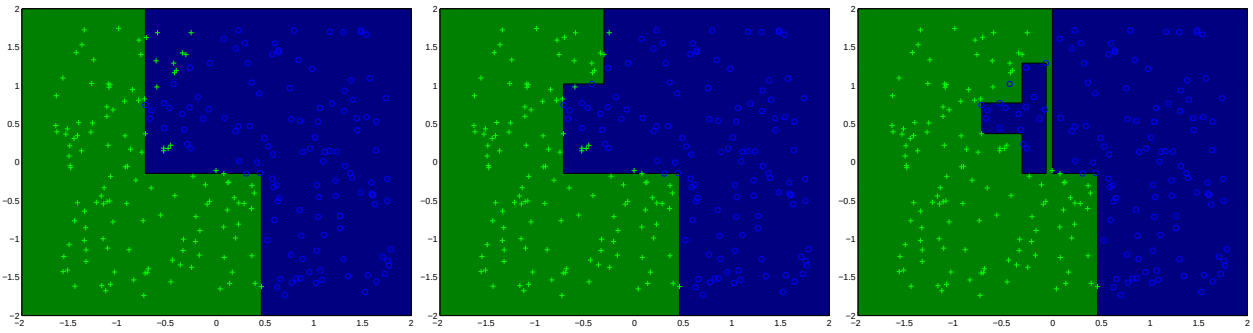
- You can work in groups on the assignments. However, please hand in your own assignments and state the group members that you worked (as well as other sources of help like online material).
- Place your name and student number on the first page, and submit all answers as a single PDF file to handin.
- Please organize your submission sequentially according to the sections used in this document.
- All Sections (1-4) are equally weighted.
- There may be updates/clarifications to the assignment after the first version is put online. Any modifications will be marked in red.

1 K-Nearest Neighours

Download and expand *a2.zip*, then run *classifier2Ddemo* in Matlab. This loads a dataset with 2 features, fits a decision stump, and then produces a plot similar to the one below:



This is a scatterplot of the feature values across all the training examples. The x-axis is given by the first feature, the y-axis is given by the second feature, and the symbol ('+' or 'o') gives the class label. The green region shows the area of the space that the decision stump labels '+', and the blue region shows the area of the space that the decision stump labels 'o'. In this case, you can see the decision stump is using a rule that tests whether the first feature is larger than a number that is close to 0.5. You can also see that it incorrectly classifies some of the 'o' points where the first feature is less than 0.5. If you change the depth of the decision tree in the demo and re-run it, you can see how the depth of the decision tree affects the predictions that the model will make. For example, below are the plots with depths of 3, 5, and 7.



1.1 Prediction Function

In this dataset, nearby points seem to receive the same class label. 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). Fill in the *predict* function in *knn.m* so that the model file implements the k-nearest neighbour prediction rule (based on Euclidean distance). Hand in your predict function as well plots similar to the above for $k = 1$, 5, and 10.

Hint: 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, 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:

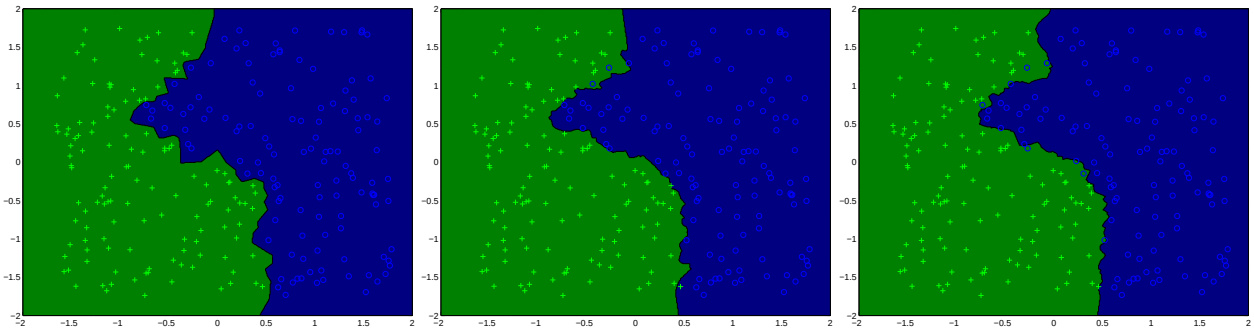
```
[N,D] = size(X);
[T,D] = size(Xtest);
D = X.^2*ones(D,T) + ones(N,D)*(Xtest').^2 - 2*X*Xtest';
```

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

Answer:
The code should roughly have the form:

```
function [yhat] = predict(model,Xtest)
X = model.X;
y = model.y;
C = model.C;
K = model.K;
[N,D] = size(X);
[T,D] = size(Xtest);

% Let's pre-compute all the distances to save time
D = X.^2*ones(D,T) + ones(N,D)*(Xtest').^2 - 2*X*Xtest';

yhat = zeros(T,1);
for i = 1:T
    % Sort the distances to the other points
   [minDist,sorted] = sort(D(:,i));

   % Count the number of labels of the K closest points
   counts = zeros(C,1);
   for j = 1:K
       trainLabel = y(sorted(j));
       counts(trainLabel) = counts(trainLabel)+1;
   end

   % Pick the label that occured most often among the neighbours
   [numberVals,yhat(i)] = max(counts);
end
end
```

The figures should roughly look like this:



1.2 Choosing the Number of Neighbours

Recall the two parts of the fundamental trade-off in machine learning:

1. How small you can make the training error?
2. How well does the training error approximates the test error?

In a $k-$nearest neighbour classifier, how does the parameter k affect each of the two parts of the trade-off? What is one way that we could choose k in practice?

Answer:

1. As k increases, the training error gets larger.
2. As k increases, the training error better approximates the test error.

To choose k, you could use a validation set or cross-validation.

1.3 Runtime

If you have pre-computed the distances to all N training examples, the cost of classifying a new example by sorting these distances is bounded by $O(N \log N)$. Show how we can classify a new example more quickly, even for large values of k, if we use an $O(N)$-time *selection* algorithm (a *selection* algorithm is an algorithm that returns the kth largest element in an unsorted list).

Answer:
The naive answer is to use the $O(n)$ selection algorithm k times (to compute the smallest distance, then the second smallest distance, and so on). This gives us a cost of $O(nk)$, which is faster more small k but is slower once $k > \log N$. A better strategy would be:

1. Use the selection algorithm to get the kth smallest distance.
2. Go through all the distances to find the points that are smaller than the k smallest.

The first step involves calling the selection algorithm once for a cost of $O(n)$, and the second step costs $O(n)$ to look at each distance once. Once you have the k smallest values, you can compute the prediction in $O(k)$, so since $k \leq n$ the total cost is $O(n)$.

2 Naive Bayes

If you run the function *example_trainValidate* it will perform the following:

1. Load the *newsgroups* data from Assignment 1, except we have now renamed *Xtest* to *Xvalidate* and *ytest* to *yvalidate* since we have already looked at them in the previous assignment.
2. Train a decision tree of depth 20, and report its validation error.
3. Call the *knn* function and report its validation error. Once you have finshined Question 1.1, it should report a lower validation error than the decision tree.
4. Call the *naiveBayes* function and report its validation error. Note that the current *naiveBayes* function is extra-naive, because it assumes that $p(x_{ij}|y_i) = 1/2$ for all features j.

2.1 Estimating the Conditional Probabilities

The *predict* function of the naive Bayes classifier is already implemented, but the calculation of the variable *p_xy* is incorrect (right now, it just sets all values to 1/2). Modify this function so that *p_xy* computes the conditional probability of these values based on the frequencies in the data set. Hand in your code and report the validation error that you obtain.

Answer:
The code should look roughly like this:

```
% We will store:
%   p(x(i,j) = 1 | y(i) = c) as p_xy(j,1,c)
%   p(x(i,j) = 0 | y(i) = c) as p_xy(j,2,c)
p_xy = zeros(D,2,C);
for c = 1:C
    for j = 1:D
        p_xy(j,1,c) = sum(X(y==c,j)==1)/counts(c);
        p_xy(j,2,c) = sum(X(y==c,j)==0)/counts(c);
    end
end
```

The test error is approximately 0.19, lower than the other two methods.

2.2 Interpreting the Naive Bayes Model

The predictions made by naive Bayes are bit harder to explain than the explicit rules used by decision trees, since they simultaneously combine information from the all the features. However, we can get an idea of which features matter the most by looking at the conditional probabilities. For each of the four classes 'c', report the three words that have the highest value of $p(x_{ij} = 1|y_i = c)$.

Answer:
For *comp.**, the top 3 words are *windows*, *help*, and *email*.
For *rec.**, the top 3 words are *team*, *car*, and *games*.
For *sci.**, the top 3 words are *space*, *help*, and *nasa*.
For *talk.**, the top 3 words are *god*, *fact*, and *question*.

2.3 Runtime of Naive Bayes for Binary Data

If you have D features for N objects, you can implement the training phase of a naive Bayes classifier in $O(ND)$ since the dominant cost is looking at each element $X(i, j)$ one time. (You do not have to actually implement it in this way for Question 3.1, but you should think about how this could be done). What is the cost of classifying T test examples with the model?

Answer:
For each of the T examples, the dominant cost is computing $p(x_{ij}|y_i)$ for all D values of j and y_i. You can do this with three ''for' loops (as in the naive Bayes *predict* function in the given code), giving a total time of $O(TDC)$ if you have C classes (we'll accept answers that ignore the dependence on C, although in practice this is important if C is large). Note that this is higher than using a depth M decision tree if $M < DC$. (However, the training and testing phases can be much faster if the examples are *sparse*, meaning that most values of x_{ij} are zero.)

2.4 Explaining the Performance of Different Methods

The naive Bayes model is much less powerful than a k-nearest neighbour classifier (or even a decision tree) because it can only model very simple distributions, yet on this dataset it actually gives lower error. What theorem from class could explain why a less-flexible model worked better on a particular dataset?

Answer:
No free lunch theorem! (Naive Bayes model makes particular assumptions, and these seem to be better-suited to the newsgroups data than the k-nearest neighbours assumptions.)

3 Random Forests

If you run the function *example_randomForest* it will perform the followinng:

1. Load Fisher's famous iris flower dataset, split into a training and validation set.
2. Fit a depth-4 decision tree, and report the validation error.
3. Fit a 'random forest' classifier using depth-4 decision trees, and report the validation error.

You will notice that the validation errors of the two models are the same, because the 'random' forest classifier doesn't yet incorporate any randomization.

3.1 Boostrap Aggregation

Currently, the *randomForest.m* functions trains each tree on the entire N training examples. Modify this function so that each tree is trained on a bootstrap sample of the training data (N samples chosen with replacement). Hand in your code, and report the validation error with this modification.

Answer:
A simple way to implement bootstrap sampling is like this:

```
for k = 1:nBootstraps
    sample = ceil(N*rand(N,1));
    model.subModel{k} = randomTree(X(sample,:),y(sample),depth);
end
```

The test error should be approximately 0.04, which is slightly lower than the single decision tree which achieves 0.05.

3.2 Random Feature Selection

Using bootstrap samples introduces a small amount of variation among the trees, but they are still very similar to each other. In order to introduce a larger variation among the trees, modify the *randomStump.m*

function so that it splits using *a random feature* instead of searching over all features. Hand in your code, and report the validation error with this modification.

Answer:
A simple way to implement the random stump is by modifying this part:

```
% Loop over features looking for the best split
if any(y ~= y(1))
    for d = ceil(rand*D)
        thresholds = sort(unique(X(:,d)));

        for t = thresholds'
```

The test error should now be down to something like 0.01 (or 0.00, depending on the random seed).

3.3 Choosing the Number of Random Features/Trees

You can do either problem 1 or problem 2 below:

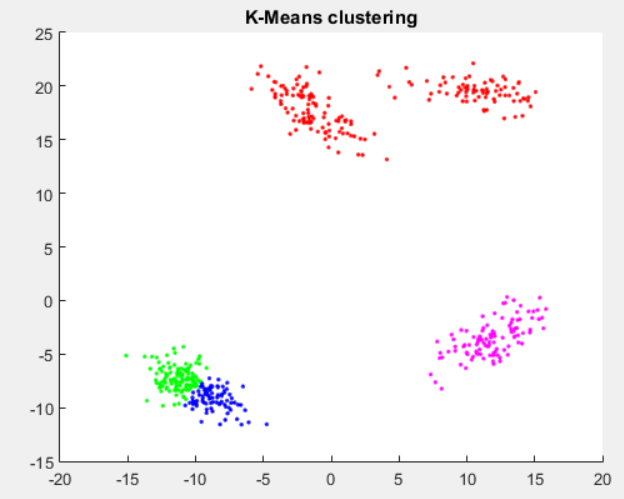
1. Consider the problem of learning a decision tree of fixed depth, where at each depth you consider a random subset of the features as possible candidates for the split variable. Let's use k to denote the number of random features that each decision stump considers (so in the previous question, we had $k = 1$). How does the value k affect the two parts of the fundamental trade-off in machine learning?
2. Consider the problem of learning a random forest, and let's use k to denote the number of random trees that we learn. How does the value k affect the two parts of the fundamental trade-off in machine learning?

Answer:
Only one of these variations on the question needs to be answered:

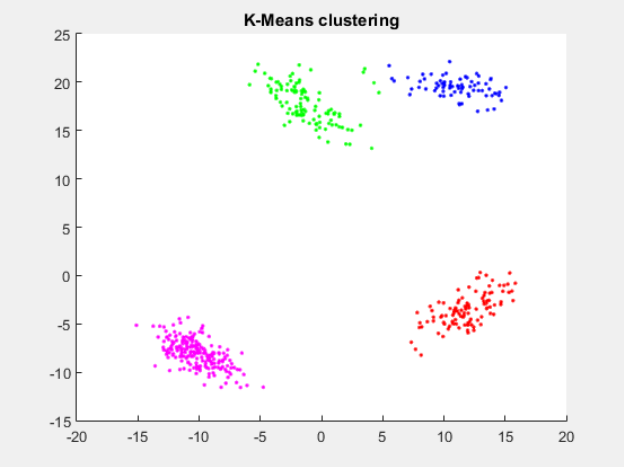
1. If the number of features k is large, we are searching over more rules so have the ability to make the training error smaller. However, since we are searching over more models the training error is likely to be a worse approximation of the test error (higher optimization bias).
2. If the number of trees k is large, we are averaging out the effect of fitting trees very precisely (the average has a smaller optimization bias than the individual trees). This will give us a higher training error, but should make the training error a better approximation of the testing error.

4 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.)

4.1 K-Means++

If you run the demo several times, you will see that k-means occasionally discovers the four clusters in the data. However, on many runs it does not. To try to improve its performance on a typical run, modify the *clusterKmeans.m* file so that it uses the k-means++ initialization. Hand in your modified code. Report whether k-means++ increase the chances of finding the four true clusters, and comment on whether k-means runs faster with the k-means++ initialization.

Hint: you can use the function *sampleDiscrete* to sample from a discrete probability distribution. For example, if you call *sampleDiscrete([0.2 0.3 0.5])*, then 50% of the time it will return 3, 30% of the time it will return 2, and 20% of the time it will return 1.

Answer:
One way to implement the initialization is like this:

```
means = X(ceil(rand*N),:); % Initial cluster is random
for k = 2:K
    % Compute squared Euclidean distance between each data point
    % and existing clusters
    distances = X.^2*ones(D,k-1) + ones(N,D)*(means').^2 - 2*X*means';

    % Compute minimum distances
    minDistances = min(distances,[],2);

    % Sample proportional to minimum distances
    i = sampleDiscrete(minDistances/sum(minDistances));
    means(k,:) = X(i,:);
end
```

The k-means++ seems to increase the chance of finding the right clusters. For me, it increased the number of times it found the right cluster from 5 to 7 when I tried out 10 initializations. It also tended to make the algorithm converge a bit faster. But the effect of the solution and the speed of convergence will vary quite a bit based on the individual runs.