CS434 Machine Learning and Data Mining – Homework 4

Decision Trees and k-Means Clustering

Overview and Objectives. In this homework, we are going to do some exercises to understand Decision Trees a bit better and then get some hand-on experience with k-means clustering.

How to Do This Assignment.

- Each question that you need to respond to is in a blue "Task Box" with its corresponding point-value listed.
- We prefer typeset solutions (LaTeX / Word) but will accept scanned written work if it is legible. If a TA can't read your work, they can't give you credit.
- Programming should be done in Python and numpy. If you don't have Python installed, you can install it from here. This is also the link showing how to install numpy. You can also search through the internet for numpy tutorials if you haven't used it before. Google and APIs are your friends!

You are NOT allowed to...

- Use machine learning package such as `sklearn`.
- Use data analysis package such as `panda` or `seaborn`.
- Discuss low-level details or share code / solutions with other students.

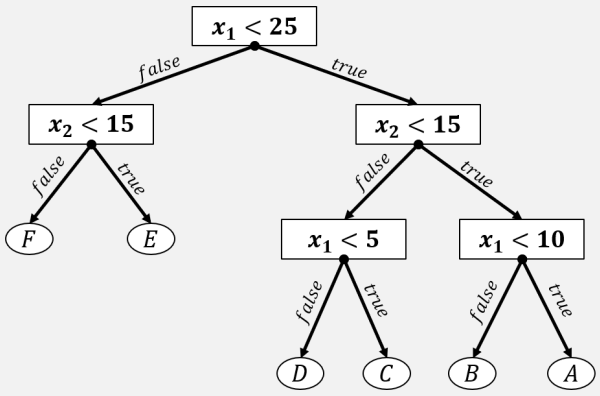
Advice. Start early. There are two sections to this assignment – one involving small exercises (20% of grade) and another focused more on programming (80% of the grade). Read the whole document before deciding where to start.

How to submit. Submit a zip file to Canvas. Inside, you will need to have all your working code and `hw4-report.pdf`.

1 Exercises: Decision Trees and Ensembles [5pts]

To get warmed up and reinforce what we've learned, we'll do some light exercises with decision trees – how to interpret a decision tree and how to learn one from data.

▶ `Q1 Drawing Decision Tree Predictions [2pts].` Consider the following decision tree:



a) Draw the decision boundaries defined by this tree over the interval $x_1 \in [0, 30]$, $x_2 \in [0, 30]$. Each leaf of the tree is labeled with a letter. Write this letter in the corresponding region of input space.

b) Give another decision tree that is syntactically different (i.e., has a different structure) but defines the same decision boundaries.

c) This demonstrates that the space of decision trees is syntactically redundant. How does this redundancy influence learning – i.e., does it make it easier or harder to find an accurate tree?

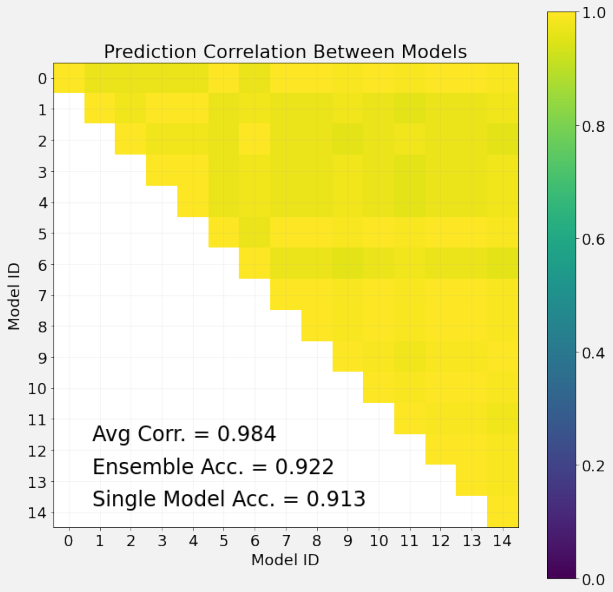
▶ Q2 Manually Learning A Decision Tree [2pts]. Consider the following training set and learn a decision tree to predict Y. Use information gain to select attributes for splits.

A	B	C	Y
0	1	1	0
1	1	1	0
0	0	0	0
1	1	0	1
0	1	0	1
1	0	1	1

For each candidate split include the information gain in your report. Also include the final tree and your training accuracy.

Now let's consider building an ensemble of decision trees (also known as a random forest). We'll specifically look at how decreasing correlation can lead to further improvements in ensembling.

▶ Q3 Measuring Correlation in Random Forests [1pts]. We've provided a Python script `decision.py` that trains an ensemble of 15 decision trees on the Breast Cancer classification dataset we used in HW1. We are using the `sklearn` package for the decision tree implementation as the point of this exercise is to consider ensembling, not to implement decision trees. When run, the file displays the plot:



The non-empty cells in the upper-triangle of the figure show the correlation between predictions on the test set for each of 15 decision tree models trained on the same training set. Variations in the correlation are due to randomly breaking ties when selecting split attributes. The plot also reports the average correlation (a very high 0.984 for this ensemble) and accuracy for the ensemble (majority vote) and a separately-trained single model. Even with the high correlation, the ensemble managed to improve performance marginally.

As discussed in class, uncorrelated errors result in better ensembles. Modify the code to train the following ensembles (each separately). Provide the resulting plots for each and describe what you observe.

a) Apply bagging by uniformly sampling train datapoints with replacement to train each ensemble member.

b) The `sklearn` API for the DecisionTreeClassifier provides many options to modify how decision trees are learned, including some of the techniques we discussed to increase randomness. When set less than the number of features in the dataset, the `max_features` argument will cause each split to only consider a random subset of the features. Modify line 43 to include this option at a value you decide.

2 Implementation: k-Means Clustering [20pts]

2.1 Implementing k-Means Clustering

As discussed in lecture, k-Means is a clustering algorithm that divides an input dataset into k separate groups. In this section, we'll get an implementation running for a simple toy problem and then apply it to a unsorted image collection to discover underlying structure.

k-Means Algorithm. More formally, given an input dataset $X = \{\mathbf{x}_i\}_{i=1}^n$[1] and the number of clusters k, k-Means produces a set of assignments $Z = \{z_i\}_{i=1}^n$ where $z_i \in \{1, 2, ..., k\}$ mapping each point to a one of the k clusters and a set of k cluster centers $C = \{c_j\}_{j=1}^k$ (also referred to as centroids). The k-Means algorithm attempts to minimize the sum-of-squared-error (SSE) between each datapoint and it's assigned centroid – we can write this objective as

$$SSE(X, Z, C) = \sum_{i=1}^{n} ||x_i - c_{z_i}||_2^2 \tag{1}$$

where c_{z_i} is the centroid vector that point i is assigned to. k-Means tries to minimize this objective by alternating between two stages updating either the assignments or the centroids:

1. **Update Assignments.** For each point x_i, compute which centroid is closest (i.e., has the minimum distance) and set z_i to the id of this nearest centroid.

$$z_i = \underset{j=1,2,...,k}{\arg\min} ||\mathbf{x}_i - \mathbf{c}_j||_2^2 \tag{2}$$

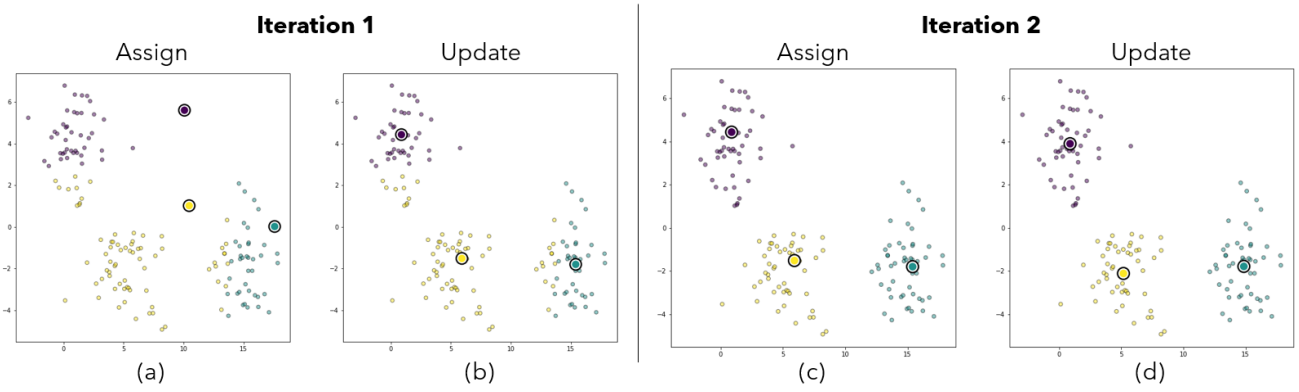
   This operation is similar to finding the nearest neighbor of x_i within the set of centroids.

2. **Update Centroids.** For each centroid c_j, update its value as the mean of all assigned points:

$$\mathbf{c}_j = \frac{1}{\sum_{i=1}^n \mathbb{I}[z_i == j]} \sum_{i=1}^{n} \mathbb{I}[z_i == j]\mathbf{x}_i \tag{3}$$

   where $\mathbb{I}[z_i == j]$ is the indicator function that is 1 if $z_i == j$ and 0 otherwise. For instance, the denominator of the leading term effectively just counts the number of points assigned to cluster j.

The figure below demonstrates this process for two iterations on a simple problem. The centroids shown as large circles with black borders are initialized randomly. In the first assignment step (a), points are colored according to the nearest centroid. Then in the update step (b), the centroids are moved to the mean of their assigned points. The process repeats in iteration 2, assignments are updated (c) and then the centroids are updated (d).



Additional Implementation Details. The algorithm above does not specify how the to initialize the centroids. Popular options include uniformly random samples between the min/max of the dataset, starting the centroids at random data points, and iterative furthest point sampling (bit tricky to implement).

[1] Notice that we do not have any associated y_i, as we are in an unsupervised setting.

Another issue that can arise during runtime is having "dead" clusters – or clusters that no or very few points assigned. Commonly, clusters with fewer members than some threshold are re-initialized.

The skeleton code provided in `kmeans.py` implements this algorithm in the `kMeansClustering` function shown below. You will implement the helper functions to perform each step.

```
def kMeansClustering(dataset, k, max_iters=10, min_size=0, visualize=False):

  # Initialize centroids
  centroids = initalizeCentroids(dataset, k)

  # Keep track of sum of squared error for plotting later
  SSE = []

  # Main loop for clustering
  for i in range(max_iters):

    # 1. Update Assignments Step
    assignments = computeAssignments(dataset, centroids)

    # 2. Update Centroids Step
    centroids, counts = updateCentroids(dataset, centroids, assignments)

    # Re-initalize any cluster with fewer then min_size points
    for c in range(k):
      if counts[c] <= min_size:
        centroids[c] = initalizeCentroids(dataset, 1)

    if visualize:
      plotClustering(centroids, assignments, dataset, "Iteration "+str(i))
    SSE.append(calculateSSE(dataset,centroids,assignments))

    # Get final assignments
    assignments = computeAssignments(dataset, centroids)

  # Return our clustering and the error over training
  return centroids, assignments, SSE
```

▶ Q4 Implement k-Means [10pts]. The provided skeleton code file `kmeans.py` implements the main k-means function but includes stubs for the following functions: `initializeCentroids`, `computeAssignments`, `updateCentroids`, and `calculateSSE`. Implement these functions to finish the implementation. The code provides input/output specifications.

a) `initializeCentroids` – Given the dataset and a integer k produce k centroid vectors. We recommend doing this by selecting k random points from the dataset, but you are welcome to try alternatives.

b) `computeAssignments` – Given the dataset and a set of centroids, implement Eq.2 to produce a vector of assignments where the i'th entry is the id of the centroid nearest to point x_i. This will involve computing distances between centroids and the dataset and will take up most of the computation in k-means. To keep things running fast, we strongly recommend using similar efficient matrix calculations as in HW1's implementation of kNN. See the HW1 solutions if you did not use these fast operations.

c) `updateCentroids` – Given the dataset, the centroids, and the current assignments, implement Eq.3 to produce a new matrix of centroids where the j'th row stores the j'th centroid. Also return the count of datapoints assigned to each centroid as a vector.

d) `calculateSSE` – Given the dataset, the centroids, and the current assignments, implement Eq.1 to calculate the sum-of-squared-errors for a clustering. This will be a scalar value. We will use this to track progress in the clustering and compare across clusterings.

When executing `kmeans.py`, k-means with $k = 3$ will be applied to the toy problem shown above with three Gaussian blobs and the result will be plotted. Use this to verify your implementation is accurate – note that depending on your choice of initialization you may get different colors for clusters.

Hint: If you want help debugging your algorithm, you can set `visualize=True` in the k-means function call to have each iteration plotted to look at intermediate steps.

▶ Q5 Randomness in Clustering [2pt]. k-Means is fairly sensitive to how the centroids are initialized. Finish the following code in the `toyProblem` function to run k-means with $k = 5$ fifty times on the toy dataset and plot the final SSE achieved for each clustering.

```
##############################
# Q5 Randomness in Clustering
##############################
k = 5
max_iters = 20

SSE_rand = []
# Run the clustering with k=5 and max_iters=20 fifty times and
# store the final sum-of-squared-error for each run in the list SSE_rand.
raise Exception('Student error: You haven\'t implemented the randomness
  experiment for Q5.')

# Plot error distribution
plt.figure(figsize=(8,8))
plt.hist(SSE_rand, bins=20)
plt.xlabel("SSE")
plt.ylabel("# Runs")
plt.show()
```

Provide the resulting plot and discuss how this might affect how you apply k-means to a real dataset.

▶ Q6 Error Vs. K [2pt]. One important question in k-means is how to choose k. In prior exercises, we've chosen hyperparmeters like this based on performance on some valdiation set; however, k-means is an unsupervised method so we don't have any labels to compute error. One idea would be to pick k such that the sum-of-squared-errors is minimized; however, this ends up being a bad idea – let's see why.

Finish the following code tn the `toyProblem` function to run k-means with $k = 1, 2, ..., 150$ on the toy dataset and plot the final SSE achieved for each clustering.

```
########################
# Q6 Error vs. K
########################

SSE_vs_k = []
# Run the clustering max_iters=20 for k in the range 1 to 150 and
# store the final sum-of-squared-error for each run in the list SSE_vs_k.
raise Exception('Student error: You haven\'t implemented the k experiment for
  Q6.')

# Plot how SSE changes as k increases
plt.figure(figsize=(16,8))
plt.plot(SSE_vs_k, marker="o")
plt.xlabel("k")
plt.ylabel("SSE")
plt.show()
```

Provide the resulting plot and discuss why choosing k based on the sum-of-squared error doesn't make sense.

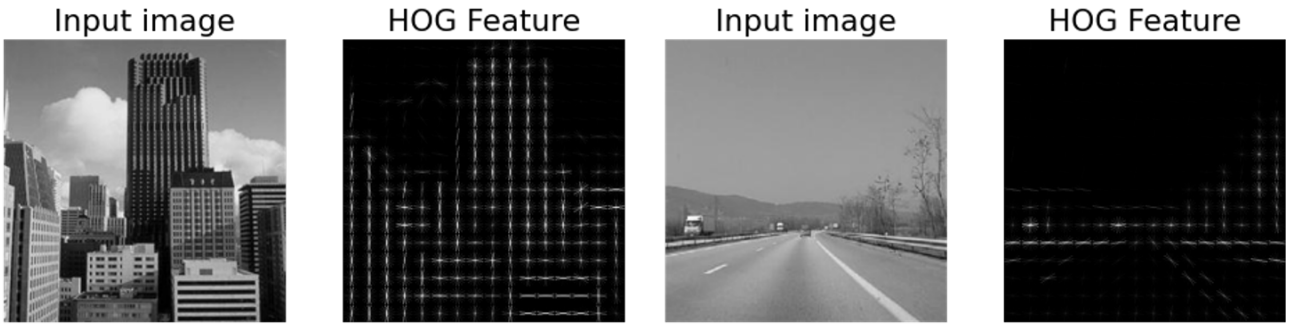
One popular heuristic for choosing k is to make a plot of SSE vs k like you just did and then selecting k to be at the "elbow" of the chart – the point where the rapid decrease of SSE changes to a more asymptotic shape. This would be roughly around 3-6 in our chart.

2.2 k-Means on an Unorganized Image Collection.

Unsupervised learning methods are often used to find unknown structure in data. In this section, we'll try to figure out clusters in a collection of images.

Representing Images with HOG. Directly clustering images may not be very successful due to their high dimensionality – even a small 256x256 greyscale image has over 65,000 values. Instead, old-fashion computer vision used to rely on extracting hand-designed features of the image. One such feature is the Histogram of Oriented Gradients (HOG)

which captures the direction of different edges in local areas of an image. A couple of examples are shown below:



where each cell of the HOG feature contains a histogram over 8 orientations, visualized as the star pattern per block. These features are useful for identifying similar structures in images but have since been largely replaced with features from Convolutional Neural Networks in practice. We'll use them here because they are easy to work with.

Clustering Images. We've prepared a collection of images and their extracted HOG features as saved `numpy` arrays `img.npy` and `hog.npy`. You'll need to have these in the same directory as your `kmeans.py` code for this next part.

▶ `Q7 Clustering Images. [4pt]` Inside `kmeans.py`, the `imageProblem` function runs your k-means algorithm on this set of images and features and then visualizes the resulting clusters by showing up to 50 samples. By default, the code runs with $k = 10$ for this dataset. Answer:

a) From the images displayed, describe what types of images are in the dataset. Does the default parameter of $k = 10$ seem to be too high, too low, or fine as it is?

b) Adjust k until you are happy with the clusters you observe. Include the samples from your clusterings.

c) Compare the SSE for your chosen k against the SSE from $k = 10$. Is SSE a good indicator of clustering quality?

▶ `Q8 Evaluating Clustering as Classification [2pt]` Suppose after you finish clustering, you want to assign a name to each cluster corresponding to what is in them.

- Provide a "label" for each of your clusters.
- Estimate the purity of your clusters as the fraction of examples in the plots that seem to match your labelling.

3 Debriefing (required in your report)

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone or did you discuss the problems with others?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?