CS5785 Homework 4

The homework is generally split into programming exercises and written exercises.
This homework is due on **Novmeber 26, 2019 at 11:59 PM EST**. Upload your homework to CMS. Please upload the submission as a single `.zip` file. A complete submission should include:

1. A write-up as a single `.pdf` file, or as a single `.ipynb` file.
2. Source code for all of your experiments (AND figures) in `.py` files if you use Python or `.ipynb` files if you use the IPython Notebook. If you use some other language, include all build scripts necessary to build and run your project along with instructions on how to compile and run your code.

The write-up should contain a general summary of what you did, how well your solution works, any insights you found, etc. On the cover page, include the class name, homework number, and team member names. You are responsible for submitting clear, organized answers to the questions. If you submit the writeup portion as a `.ipynb` file it should still be a clean and organized report, as with a `.pdf` submission, with only the minimal amount of code needed to generate figures etc. (note that you can import functions defined in a separate `.py` file into a IPython notebook). You could use online LaTeX templates from Overleaf, under "Homework Assignment" and and "Project / Lab Report".
Please include all relevant information for a question, including text response, equations, figures, graphs, output, etc. If you include graphs, be sure to include the source code that generated them. Please pay attention to Slack for relevant information regarding updates, tips, and policy changes. You are encouraged (but not required) to work in groups of 2.

Programming Exercises

1. **Multidimensional scaling for genetic population differences.** In this exercise, we will look at a dataset of 42 human geographic populations collected by Cavalli-Sforza *et al.* in *The History and Geography of Human Genes,* 1994. There are many ways to measure genetic similarity between populations. This work uses the *modified Nei's distance,* which compares allele frequencies at specific locations within the human genome. Other ways to measure genetic similarity include the edit distance between DNA sequences or Euclidean distance of gene expressions.

   This dataset contains comparisons between every pair of populations in the study as a distance matrix. You can use the following code to interact with it:

   ```
   import numpy as np
   data = np.load("mds-population.npz")
   print data['D'] # Distance matrix
   print data['population_list'] # List of populations
   ```

Here, $D_{i,j}$ is the dissimilarity between population i and j. Higher scores indicate more dissimilarity.

Since this distance is not based on an underlying vector representation, we cannot directly use traditional techniques like classification or clustering to analyze it. Our first step will be to *embed* this distance matrix into an m-dimensional vector space.

(a) First, use multidimensional scaling (MDS) to coerce D into a 2-dimensional vector representation. You can use the functions in `sklearn.manifold` for this.

i. MDS will attempt to output a set of points $\mathbf{x} \in \mathscr{R}^{42,m}$ such that the Euclidean distance between every pair of points approximates the Nei's distance between these populations, or $\|x_i - x_i\|_2 \approx D_{i,j}$. What assumptions are being made? Under what circumstances could this fail? How could we measure how much information is being lost? Please explain.

ii. One way of increasing the quality of the output is by increasing the dimensionality of the MDS result. How many dimensions are necessary to capture most of the variation in the data? There are several ways of making this judgment: for instance, you might sample some quality measure between x and D while varying m, or you might count the nonzero singular values of D, or inspect the singular values of x at some high dimension like $m = 20$. Briefly explain your method and justify why it makes sense.

iii. Use MDS to embed the distance matrix into only two dimensions and show the resulting scatterplot. Label each point with the name of its population.

(b) *k-means on 2D embedding.* Select an appropriate k and run k-means on the scatterplot. Show the resulting clusters.

Since we are working with only two dimensions, this clustering is likely to lose a lot of high-dimensional structure. Do you agree with the resulting clustering? What information seems to be lost?

(c) *Comparing hierarchical clustering with K-Means.* Use hierarchical clustering to cluster the original distance matrix. We suggest using the functions in `scipy.cluster.hierarchy` for this, as this library can plot the resulting graph structure. Show the resulting tree as a *dendrogram*, labeling the x axis with the categorical population names and the y axis with the Nei's distance between clusters. (It's also possible to do this with `sklearn`, but traversing the resulting structure is much harder).

To turn the resulting tree into a flat clustering of points, cut off the dendrogram at a certain distance by merging all subclusters within this distance together. This can be done with the `scipy.cluster.hierarchy.fcluster` function. Select a distance cutoff that roughly corresponds with your chosen k earlier, balancing the number of points in each cluster with the number of resulting clusters. Visualize the resulting clustering by coloring the corresponding points on your 2D MDS embedding. How does this clustering compare with the k-means clustering you computed earlier?

(d) *Compare k-medoids with k-means.* Repeat the above experiment, but applying k-medoid clustering on the original distance matrix. Show the resulting clusters on a 2D scatterplot. Are there any significant differences between the clustering chosen by k-medoids compared to k-means?

2. **Random forests for image approximation** In this question, you will use random forest regression to approximate an image by learning a function, $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, that takes *image* (x, y) *coordinates*

as input and outputs *pixel brightness*. This way, the function learns to approximate areas of the image that it has not seen before.

a. Start with an image of the Mona Lisa. If you don't like the Mona Lisa, pick another interesting image of your choice.

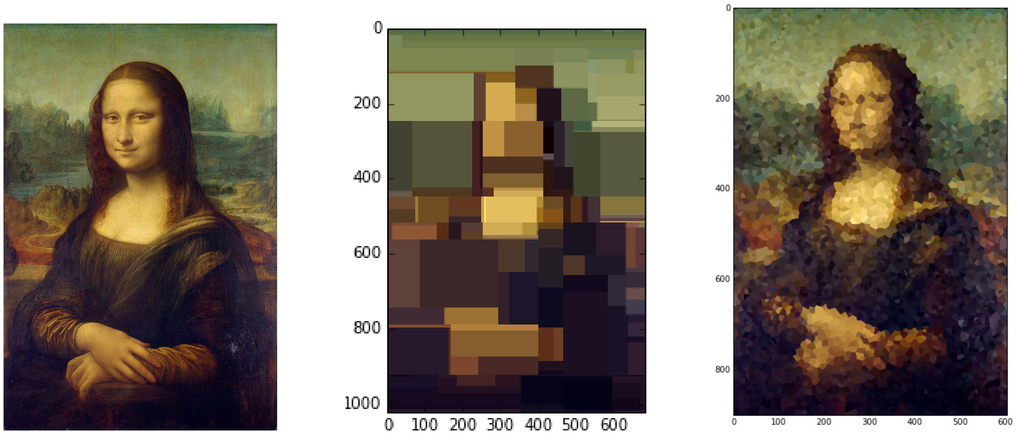

Figure 1: **Left**: `http://tinyurl.com/mona-lisa-small` *Mona Lisa*, Leonardo da Vinci, via Wikipedia. Licensed under Public Domain. **Middle**: Example output of a decision tree regressor. The input is a "feature vector" containing the (x, y) coordinates of the pixel. The output at each point is an (r, g, b) tuple. This tree has a depth of 7. **Right**: Example output of a k-NN regressor, where $k = 1$. The output at each pixel is equal to its closest sample from the training set.

b. **Preprocessing the input.** To build your "training set," uniformly sample 5,000 random (x, y) coordinate locations. Note if you use an image other than the Mona Lisa image linked above you may need to sample a different number of coordinates, so that you sample approximately the same percentage of the total number of pixels in the image (around 1%).

- What other preprocessing steps are necessary for random forests inputs? Describe them, implement them, and justify your decisions. In particular, do you need to perform mean subtraction, standardization, or unit-normalization?

c. **Preprocessing the output.** Sample pixel values at each of the given coordinate locations. Each pixel contains red, green, and blue intensity values, so decide how you want to handle this. There are several options available to you:

- Convert the image to grayscale
- Regress all three values at once, so your function maps (x, y) coordinates to (r, g, b) values: $f : \mathbb{R}^2 \to \mathbb{R}^3$
- Learn a different function for each channel, $f_{Red} : \mathbb{R}^2 \to \mathbb{R}$, and likewise for f_{Green}, f_{Blue}.

Note that you may need to rescale the pixel intensities to lie between 0.0 and 1.0. (The default for pixel values may be between 0 and 255, but your image library may have different defaults.)

What other preprocessing steps are necessary for random regression forest outputs? Describe them, implement them, and justify your decisions.

d. To build the final image, for each pixel of the output, feed the pixel coordinate through the random forest and color the resulting pixel with the output prediction. You can then use `imshow` to view the result. (If you are using grayscale, try `imshow(Y, cmap='gray')` to avoid fake-coloring). You may use any implementation of random forests, but you should understand the implementation and you must cite your sources.

e. Experimentation.

i. Repeat the experiment for a random forest containing a single decision tree, but with depths 1, 2, 3, 5, 10, and 15. How does depth impact the result? Describe in detail why.

ii. Repeat the experiment for a random forest of depth 7, but with number of trees equal to 1, 3, 5, 10, and 100. How does the number of trees impact the result? Describe in detail why.

iii. As a simple baseline, repeat the experiment using a k-NN regressor, for $k = 1$. This means that every pixel in the output will equal the nearest pixel from the "training set." Compare and contrast the outlook: why does this look the way it does?

iv. Experiment with different pruning strategies of your choice.

f. Analysis.

i. What is the decision rule at each split point? Write down the 1-line formula for the split point at the root node for one the trained decision trees inside the forest. Feel free to define any variables you need.

ii. Why does the resulting image look like the way it does? What shape are the patches of color, and how are they arranged?

iii. *Straightforward:* How many patches of color may be in the resulting image if the forest contains a single decision tree? Define any variables you need.

iv. *Tricky:* How many patches of color might be in the resulting image if the forest contains n decision trees? Define any variables you need.

3. **SVM Classification** This problem involves the OJ data set which is available at `https://gist.github.com/gcr/e86ca41c43accbaed32226cc63af14e7` (click the "Raw" button to download).

a. Create a training set containing a random sample of 800 observations, and a test set containing the remaining observations. Report the class fractions in your training and test sets.

b. Fit a support vector classifier to the training data using `C = 0.01`, with `Purchase` as the response and the other variables as predictors. Describe the classifier. What are the learned coefficients? (You can use `sklearn` to help with this.)

c. What are the training and test error rates?

d. Use cross validation to select an optimal cost (`C`) and report a CV error plot. Consider values in the range 0.01 to 10. What `C` is the best using the one-stderr rule?

e. Compute the training and test error rates using this new value for cost.

f. For a support vector classifier with a radial bassis function (RBF) kernel $k(x, x') = e^{-\gamma\|x-x'\|^2}$, why should we consider a *smaller* γ value as corresponding to a simpler model? (Note that sometimes we write the RBF kernel as $k(x, x') = e^{-\|x-x'\|^2/\sigma^2}$ using $\sigma = 1/\sqrt{\gamma}$.)

g. Repeat parts (b) through (e) using a support vector machine with a radial kernel. Use cross-validation (with the one-standard-error rule) to select an appropriate `gamma` .

h. Repeat parts (b) through (e) using a support vector machine with a polynomial kernel. Set `degree=2` and use cross validation again to choose an appropriate `gamma`.

i. Overall, which approach seems to give the best results on this data?

4. **Approximating images with neural networks.** In this question, you will implement your own neural network toolkit. You will be writing your own implementation from scratch, using *C*++ and CUDA. You should calculate the derivatives of each layer by hand using pencil and paper. Please attach a scan of your paper notes to the homework.

Just kidding. We're not that mean. There are several good convolutional neural network packages that have done the heavy lifting for us. One of the more interesting (and well-written!) demos is called CONVNETJS. It is implemented in Javascript and runs in a modern web browser without any dependencies.

Take a look at convnet.js's "Image Painting" demo at: `http://cs.stanford.edu/people/karpathy/convnetjs/demo/image_regression.html`

a. Describe the structure of the network. How many layers does this network have? What are the sizes of the layers? What activation function(s) are being used?

b. What does "Loss" mean here? What is the actual loss function? You may need to consult the source code, which is available on Github.

c. Plot the loss over time, after letting it run for 5,000 iterations. You can do this by watching the training and manually writing down the loss every 500 or so iterations. How good does the network eventually get?

d. Can you make the network converge to a lower loss function by lowering the learning rate every 1,000 iterations? Some *learning rate schedules*, for example, halve the learning rate every *n* iterations: does this technique let the network converge to a lower training loss?

e. Lesion study. The text box contains a small snippet of Javascript code that initializes the network. You can change the network structure by clicking the "Reload network" button, which simply evaluates the code. Let's perform some brain surgery: Try commenting out each layer, one by one. Report some results: How many layers can you drop before the accuracy drops below a useful value? How few hidden units can you get away with before quality drops noticeably?

f. Try adding a few layers by copy+pasting lines in the network definition. Can you noticeably increase the accuracy of the network?

Written Exercises

1. **Decision trees.** Suppose we modify the tree-growing algorithm presented in class to use a new impurity function. Define $f(r) = \min\{r, 1-r\}$. Then we will define the impurity of a set of examples as:

$$I(\{y_1, \ldots, y_n\}) = f(p) \tag{1}$$

where p is the fraction of positive examples in $\{y_1, \ldots, y_n\}$. Let us call this the *min-error* impurity function.

As usual, for a split where p_1 positive and n_1 negative examples reach the left branch and p_2 positive and n_1 negative examples reach the right branch, the weighted impurity of the split will be

$$(p_1 + n_1) \cdot f\left(\frac{p_1}{p_1 + n_1}\right) \quad + \quad (p_2 + n_2) \cdot f\left(\frac{p_2}{p_2 + n_2}\right). \tag{2}$$

(a) Suppose that each branch of this split is replaced by a leaf labeled with the more frequent class among the examples that reach that branch. Show that the number of training mistakes made by this truncated tree is exactly equal to the weighted impurity given above. Thus, using the min-error impurity is equivalent to growing the tree greedily to minimize training error.

(b) Suppose the dataset looks like the following. There are three $\{0,1\}$-valued attributes, and one $\{-,+\}$-valued class label y.

a_1	a_2	a_3	y
0	0	0	+
1	1	0	+
0	1	0	+
1	0	1	-
0	0	1	-
0	1	0	-
1	1	0	-
1	1	1	-
1	0	0	-
1	1	0	-

Which split will be chosen at the root when the Gini index impurity function is used? Which split will be chosen at the root when min-error impurity is used? Explain your answers.

(c) Under what general conditions on p_1, n_1, p_2, and n_2 will the weighted min-error impurity of the split be strictly smaller than the min-error impurity before making the split (i.e., of all the examples taken together)?

(d) What do your answers to the last two parts suggest about the suitability of min-error impurity for growing decision trees?

2. **Bootstrap aggregation ("bagging")** Suppose we have a training set of N examples, and we use bagging to create a bootstrap replicate by drawing N samples **with replacement** to form a new training set. Because each sample is drawn with replacement, some examples may be included in this bootstrap replicate multiple times, and some examples will be omitted from it entirely.

As a function of N, compute the expected fraction of the training set that does not appear at all in the bootstrap replicate. What is the limit of this expectation as $N \to \infty$?

Hint 1: You can express the number of examples not chosen by bootstrapping as the sum of n random variables X_i, where each of these variables equals 1 if the i'th sample was never chosen, or 0 otherwise.

Hint 2: Recall that expectation is linear.

Hint 3: You may want to use the fact that $\lim_{n\to\infty}(1+x/n)^n = e^x$.

3. **Maximum-margin classifiers** Suppose we are given $n = 7$ observations in $p = 2$ dimensions. For each observation, there is an associated class label.

X_1	X_2	Y
3	4	Red
2	2	Red
4	4	Red
1	4	Red
2	1	Blue
4	3	Blue
4	1	Blue

   a. Sketch the observations and the maximum-margin separating hyperplane.

   b. Describe the classification rule for the maximal margin classifier. It should be something along the lines of "Classify as Red if $\beta_0 + \beta_1 X_1 + \beta_2 X_2 > 0$, or classify as Blue otherwise." Provide the values for β_0, β_1, and β_2.

   c. On your sketch, indicate the margin for the maximal margin hyperplane.

   d. Indicate the support vectors for the maximal margin classifier.

   e. Argue that a slight movement of the seventh observation would not affect the maximal margin hyperplane.

   f. Sketch a hyperplane that separates the data, but is not the maximum-margin separating hyperplane. Provide the equation for this hyperplane.

   g. Draw an additional observation on the plot so that the two classes are no longer separable by a hyperplane.