CS273a Homework #4

Introduction to Machine Learning: Fall 2016

Due: Friday November 18th, 2016

Write neatly (or type) and show all your work!

Please remember to turn in at most two documents, one with any handwritten solutions, and one PDF file with any electronic solutions.

Download the provided Homework 4 code, to replace / add to last week's code (several new functions have been added).

Problem 1: Decision Trees

We'll use the same data as in our earlier homework: In order to reduce my email load, I decide to implement a machine learning algorithm to decide whether or not I should read an email, or simply file it away instead. To train my model, I obtain the following data set of binary-valued features about each email, including whether I know the author or not, whether the email is long or short, and whether it has any of several key words, along with my final decision about whether to read it ($y = +1$ for "read", $y = -1$ for "discard").

x_1 know author?	x_2 is long?	x_3 has 'research'	x_4 has 'grade'	x_5 has 'lottery'	y $\Rightarrow$ read?
0	0	1	1	0	-1
1	1	0	1	0	-1
0	1	1	1	1	-1
1	1	1	1	0	-1
0	1	0	0	0	-1
1	0	1	1	1	1
0	0	1	0	0	1
1	0	0	0	0	1
1	0	1	1	0	1
1	1	1	1	1	-1

In the case of any ties, we will prefer to predict class +1.

(a) Calculate the entropy of the class variable y

(b) Calculate the information gain for each feature x_i. Which feature should I split on first?

(c) Draw the complete decision tree that will be learned from these data.

Problem 2: Decision Trees on Kaggle

In this problem, we will use our Kaggle in-class competition data to test decision trees on real data.

Kaggle is a website designed to host data prediction competitions; we will use it to give some experience with more realistic machine learning problems, and some opportunity to compare methods and ideas amongst ourselves. Create an account **linked to / using your UCI-affiliated email** and log into `inclass.kaggle.com`, and search for the `UC Irvine CS273a 2016` competition,

https://inclass.kaggle.com/c/uc-irvine-cs273a-2016
You can download the data from Kaggle, or it is in your `data` subdirectory from the HW4 zip file. (Note: this is a private class competition; if you do not use a UCI email, you will not be able to access and submit.)

In this problem, we will build a simple decision tree model to make predictions on the data. You can use the `treeClassify` class provided to build your decision trees.

Note: Kaggle competitions only let you submit a fixed number of predictions per day, ≈ 2 in our case, so be careful. We'll use a validation split to decide what hyperparameter choices we think are most promising, and upload only one model.

(a) Load the training data, `X_train.txt` and `Y_train.txt`. There are quite a lot of data available; for the homework you can just use the first 10,000 samples. Split out a validation set as well, say samples 10001:20000.

(b) Learn a decision tree classifier on the data. To avoid any potential recursion limits, specify a max depth of 50, e.g.,

```
dt = treeClassify(Xt,Yt, maxDepth=50)
```

(This may take a minute or two.) Compute your model's training and validation error rates.

(c) Now, try varying the maximum depth parameter (`maxDepth`), which forces the tree to stop after at most that many levels. Test values `0, 1, ..., 15` and compare their performance (both training and test) against the full depth. Is complexity increasing or decreasing with the depth cutoff? Identify whether you think the model begins overfitting, and if so, when. If you use this parameter for complexity control, what depth would you select as best?

(d) Now, using high maximum depth ($d = 50$), use `minLeaf` to control complexity. Try values `2.^[2:12]=[4,8,16,...,4096]`. Is complexity increasing or decreasing as `minLeaf` grows? Identify when (if) the model is starting to overfit, and what value you would use for this type of complexity control.

(e) **(Not graded)** A related control is `minParent`; how does complexity control with `minParent` compare to `minLeaf`?

(f) Our Kaggle competition measures performance using the ROC curve, specifically the AUC (area under the curve) score. Compute and plot the ROC curve for your trained model (using e.g. the `roc` member function), and the area under the curve (`auc`).

(g) Using your best complexity control value (either depth or number of leaf data), re-train a model (you may use the first 10k data, or more, as you prefer). Load the test features `X_test.txt`, and make predictions on all 200k test points. Output your predictions in the format expected by Kaggle,

```
Ypred = learner.predictSoft( Xte )
# Now output a file with two columns, a row ID and a confidence in class 1:
np.savetxt('Yhat_knn200.txt',
   np.vstack( (np.arange(len(Ypred)) , Ypred[:,1]) ).T,
   '%d, %.2f',header='ID,Prob1',comments='',delimiter=',');
```

Upload them and report your model's performance. Compare its performance to the AUC score you estimated using the validation data.

Problem 3: Random Forests

Random Forests are bagged collections of decision trees, which select their decision nodes from randomly chosen subsets of the possible features (rather than all features). You can implement this easily in `treeClassify` using option `'nFeatures'`=n, where n is the number of features to select from (e.g., $n = 50$ or $n = 60$ if there are 90-some features); you'll write a for-loop to build the ensemble members, and another to compute the prediction of the ensemble.

In Python, it is easy to keep a list of different learners, even of different types, for use in an ensemble predictor:

```
ensemble[i] = ml.treeClassify(Xb,Yb,...)  # save ensemble member "i" in a cell array
# ...
ensemble[i].predict(Xv,Yv);              # find the predictions for ensemble member "i"
```

(a) Using your validation split, learn a bagged ensemble of decision trees on the training data and evaluate validation performance. (See the pseudocode from lecture slides.) For your individual learners, use little complexity control (depth cutoff 15+, minLeaf 4, etc.), since the bagging will be used to control overfitting instead. For the bootstrap process, draw the same number of data as in your training set after the validation split ($M' = M$ in the pseudocode). You may find `ml.bootstrapData()` helpful, although it is very easy to do yourself. Plot the training and validation error as a function of the number of learners you include in the ensemble, for (at least) $1, 5, 10, 25$ learners. (You may find it more computationally efficient to simply learn 25 ensemble members first, and then evaluate the results using only a few of them; this will give the same results as only learning the few that you need.)

(b) Now choose an ensemble size and build an ensemble using at least 10k training data points, make predictions on the test data, and upload to Kaggle. Report your preformance.