# CS 178: Machine Learning & Data Mining: Fall 2020

# Homework 4

Due Date: **Tuesday, November 24, 2020**

---

The submission for this homework should be ***a single PDF file*** containing all of the relevant code, figures, and any text explaining your results. Be sure to include copies of any code you write as answers to the appropriate question, so that it may be graded.

## Problem 1: Decision Trees for Spam Classification (25 points)

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 where both classes have equal probability, we will prefer to predict class +1.

1. Calculate the entropy $H(y)$, in bits, of the binary class variable $y$. *Hint:* Your answer should be a number between 0 and 1. *(5 points)*

2. Calculate the information gain for each feature $x_i$. Which feature should I split on for the root node of the decision tree? *(10 points)*

3. Determine the complete decision tree that will be learned from these data. (The tree should perfectly classify all training data.) Specify the tree by drawing it, or with a set of nested if-then-else statements. *(10 points)*

## Problem 2: Decision Trees in Python (50 points)

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 gain some experience with more realistic machine learning problems, and have an opportunity to compare methods and ideas amongst ourselves. Our in-class Kaggle page is `https://www.kaggle.com/c/uci-cs178-f20`; you can join using the participation URL: `https://www.kaggle.com/t/bd21e4e5d61540888ed61f438bac55b8` Follow the instructions on the CS178 Canvas page to create an appropriate Kaggle account (if necessary), join our in-class competition, and download the competition data. **Note:** although you will eventually form teams for the project, please **do not** merge yet, as you will want to be able to submit individually for this homework. For convenience, the data are also included in the HW4 code zip, in the `data` subdirectory.

1. The following code may be used to load the training features $X$ and class labels $Y$:

```
X = np.genfromtxt('data/X_train.txt', delimiter=',')
Y = np.genfromtxt('data/Y_train.txt', delimiter=',')
X,Y = ml.shuffleData(X,Y)
# and simlarly for test data features. Test target values are withheld for the
    ↪ competition.
```

The first 41 features are numeric (real-valued features); we will restrict our attention to these:

```
X = X[:,:41]   # keep only the numeric features for now
```

Print the minimum, maximum, mean, and variance of each of the first 5 features. *(5 points)*

2. To enable us to do model selection, partition your training data $X$ into training data `Xtr,Ytr` and validation sets `Xva,Yva` of approximately equal size. Learn a decision tree classifier from the training data using the method implemented in the `mltools` package (this may take a minute):

```
learner = ml.dtree.treeClassify(Xtr, Ytr, maxDepth=50)
```

Here, we set the maximum tree depth to 50 to avoid potential recursion limits or memory issues. Compute and report your decision tree's training and validation error rates. *(5 points)*

3. Now try varying the `maxDepth` parameter, which forces the tree learning algorithm to stop after at most that many levels. Test `maxDepth` values in the range `0, 1, 2, ..., 15`, and plot the training and validation error rates versus `maxDepth`. Do models with higher `maxDepth` have higher or lower complexity? What choice of `maxDepth` provides the best decision tree model? *(10 points)*

4. The `minParent` parameter controls the complexity of decision trees by lower bounding the amount of data required to split nodes when learning. Fixing `maxDepth=50`, compute and plot the training and validation error rates for `minParent` values in the range `2.^[0:13]=[1,2,4,8,...,8192]`. Do models with higher `minParent` have higher or lower complexity? What choice of `minParent` provides the best decision tree model? *(10 points)*

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

6. We discussed in class that we could understand our model's performance as we vary our preference for false positives compared to false negatives using the ROC curve, or summarize this curve using a scalar *area under curve* (AUC) score. For the best decision tree model trained in the previous parts, use the `roc` function to plot an ROC curve summarizing your classifier performance on the training points, and another ROC curve summarizing your performance on the validation points. Then using the `auc` function, compute and report the AUC scores for the training and validation data. *(10 points)*

7. Based on your results in the previous parts, pick `maxDepth` and `minParent` values that you think will perform well. Retrain your decision tree model using all the data in `X_train.txt`. Score your performance on the same data (accuracy rate and AUC).

   Then, using code like the following, make predictions on the test points (feature vectors found in `X_test.txt`), and export your predictions in the format required by Kaggle:

```
learner = ... # train a model using training data X,Y
Xte = np.genfromtxt('data/X_test.txt', delimiter=',')
Yte = np.vstack((np.arange(Xte.shape[0]), learner.predictSoft(Xte)[:,1])).T
# Output a file with two columns, a row ID and a confidence in class 1:
np.savetxt('Y_submit.txt',Yte,'%d,%.2f',header='Id,Predicted',comment='',delimiter=','
    ↪ )
```

   Submit your predictions on all of the test data to Kaggle, and include your Kaggle username and leaderboard AUC in your homework solutions. *(10 points)*

Note that we use `predictSoft` to output probabilistic predictions (that test examples are members of class 1) for upload to Kaggle. While you may also use "hard" predictions (class values), accounting for the learned model's confidence in each prediction will produce a smoother ROC curve and (usually) a better AUC score.

## Problem 3: Ensemble Methods (20 points)

Choose **either part** of this question to answer (your choice): a random forest classifier, which is a bagged ensemble of decision trees; **or** an boosted ensemble of regression trees learned with gradient boosting.

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"
```

**Option 1: 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.

1. 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.)

2. Now choose an ensemble size and build an ensemble using the full training set, make predictions on the test data, and evaluate (via Kaggle's leaderboard) and report your performance.

**Option 2: Gradient boosting:**

Gradient boosted trees are boosted collections of decision trees, which are build sequentially to predict the residual error in the current ensemble. You'll write a for-loop to build the ensemble members, and another to compute the prediction of the ensemble.

Since this is a classification problem, we will do gradient boosting on a logistic negative log likelihood loss, i.e., we will regress the log-odds ratio in a manner similar to logistic regression (except that, instead of a linear regression on the log-odds, we will use a collection of decision trees). Use `treeRegress` to fit each learner to the (real-valued) log-odds update; `treeRegress` works analagously to `treeClassify`.

In practice, start out with a baseline predictor $f(x) = 0$, and compute probabilies $p(y = 1) = \sigma(f(x))$ where $\sigma$ is the usual logistic function. Then, update $f(x)$ by regressing the derivative of $J$:

$$\frac{dJ^{(i)}}{df^{(i)}} = \begin{cases} 1 - \sigma(f(x^{(i)})) & y^{(i)} = 1 \\ -\sigma(f(x^{(i)})) & y^{(i)} = 0 \end{cases}$$

Fit $dJ$ using a regression tree $t(x)$, and update $f(x) = f(x) + \alpha t(x)$ for some step size $\alpha$.

1. Using your validation split, learn a gradient boosted ensemble of decision trees on the training data and evaluate validation performance. (See the pseudocode from lecture slides.) For your individual learners, use very strong complexity control (depth cutoff 2–3, or large minParent, etc.), since the boosting process will be adding complexity to the overall learner. 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, and then evaluate the results using fewer of them.)

2. Now choose an ensemble size and repeat on the full training data, make predictions on the test data, and evaluate your accuracy and AUC scores. Report your performance.

## Problem 4: Statement of Collaboration (5 points)

It is **mandatory** to include a *Statement of Collaboration* in each submission, that follows the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed.

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments in particular, I encourage students to organize (perhaps using Piazza) to discuss the task descriptions, requirements, possible bugs in the support code, and the relevant technical content *before* they start working on it. However, you should not discuss the specific solutions, and as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (no photographs of the blackboard, written notes, referring to Piazza, etc.). Especially *after* you have started working on the assignment, try to restrict the discussion to Piazza as much as possible, so that there is no doubt as to the extent of your collaboration.