EECS 440: Machine Learning Fall 2018 Programming Problem Set

General Instructions: There are 4 problems. Each problem is worth 5 points (graded on 60 points in Canvas). From the website/canvas, you can download (i) the data files for the problems referred to in the questions below and (ii) utility code we have made available to parse these files. Each problem will have the following files:

- A "problem.names" file, which will list the attributes, their types and values. One attribute will be an "example-id" attribute which will be useful for identifying examples, but which you will not use when learning.
- A "problem.data" file, which will list all the examples and their class labels.
- A "problem.info" file, which gives additional information about the problem, such as how the data was generated. This is for your information only and does not affect the implementation in any way.

Please read all the questions before you start implementation, because some of the later questions build on the previous ones. Try to build modular and flexible frameworks to make implementing the later questions easier.

Your programs must be written in MATLAB or Python. You may *not* use external libraries that implement any significant component of the problems. You may *not* reference open source code, or copy snippets from other code not written by you. Besides standard system libraries, you *can* use libraries that give you access to data structures or libraries for math operations (like numpy or scipy) or libraries for optimization (like cvxopt). **If in doubt, you must ask first before using a library.**

Your code must be efficient, cleanly written and easy to understand. It should handle errors and corner cases properly. You will lose points if the TAs cannot follow the logic easily or if your code breaks during testing. Note that we will test your code on data other than what is provided. Your code should work on standard installations of (one of) python 2.7, python 3 and MATLAB 8 or above.

Your commits are due on csevcs.case.edu by 11:59pm on the due dates specified after the question. You will receive a 10% bonus for any solution turned in a week or more in advance of the due date. You must notify us of such a commit by email. You can use one late day each week (up to Saturday 11:59pm) with a penalty of 20%. Submissions after Saturday 11:59pm for any week will not be graded. Other bonus points may be awarded for well-written and commented, easy to read, clean and efficient code.

Note that we will use the git commit logs to track commits to and ensure that each person contributes equally to all assignments. **Therefore, please do not ask anyone else to commit code you have written. You will not receive credit for code committed by anyone other than yourself. There will be NO exceptions to this rule.**

1. In this problem, you will implement and evaluate a decision tree learning algorithm. Due dates: First commit 9/14: 10 points, Second commit 9/21: 20 points, Final commit 9/28: 30 points.

a. Decision Tree Learner (45 points)

Implement the ID3 decision tree algorithm discussed in class. Your implementation should be able to handle nominal and continuous attributes, and use both information gain and gain ratio as the split criterion (depending on a provided option below). You will also need to implement stratified cross validation as described below. The main file should be called **dtree**.

Your code should take four options.

1. Option 1 is the path to the data. If this is "/a/b/someproblem" then you will load "/a/b/someproblem.names" and "/a/b/someproblem.data".
2. Option 2 is a 0/1 option. If 0, use cross validation. If 1, run the algorithm on the full sample.
3. Option 3 is a nonnegative integer that sets the maximum depth of the tree (the number of tests on any path from root to leaf). If this value is zero, you should grow the full tree. If this value is positive, grow the tree to the given value. Note that if this value is too large, it will have the same effect as when the option is zero.
4. Option 4 is a 0/1 option. If 0, use information gain as the split criterion. If 1, use gain ratio.

When your code is run, it should first construct 5 folds using stratified cross validation if this option is provided. To ensure repeatability, set the random seed for the PRNG to 12345. Then it should produce decision trees on each fold (or the sample according to the option) and report the following.

b. Output format

When your code is run on any problem, it must produce output in the following format:

Accuracy: 0.xyz

Size: xyz

Maximum Depth: xyz

First Feature: <name>

"Accuracy" is the (average) fraction of examples in the test sets (if cross validating) or training set (if not) that were correctly classified by the learned decision tree. "Size" is the size of the tree in number of nodes, and maximum depth is the length of the longest sequence of tests from root to leaf. The "First Feature" is the name of the first feature that was used to partition the data, or "None" if the tree was empty. (In MATLAB, you can output the index of the feature instead of the name.)

c. Writeup (15 points)

Prepare a writeup on your experiments. In your writeup, answer the following questions. "CV" means "cross validated":

(a) What is the CV accuracy of the classifier on each dataset when the depth is set to 1?

(b) For *spam* and *voting*, look at first test picked by your tree. Do you think this looks like a sensible test to perform for these problems? Explain.

(c) For *volcanoes* and *spam*, plot the CV accuracy as the depth of the tree is increased. On the x-axis, choose depth values to test so there are at least five evenly spaced points. Does the accuracy improve smoothly as the depth of the tree increases? Can you explain the pattern of the graph?

(d) Pick 3 different depth values. How do the CV accuracies change for gain and gain ratio for the different problems for these values?

(e) Compare the CV accuracies and the accuracy on the full sample for depths 1 and 2. Are they comparable?

Write down any further insights or observations you made while implementing and running the algorithms, such as time and memory requirements, the complexity of the code, etc. Especially interesting insights may be awarded extra points.

Because this is a long assignment, you are expected to make intermediate weekly commits to the repository. Each week, we expect to see substantial progress made in the implementation. Create a separate subdirectory "P1" in your git repository, place your code in it and push to csevcs. Include a short README file containing (i) a summary of the work done for the week and (ii) your experience with the API and documentation, and anything you found confusing. Do NOT commit any other code.

2. In this problem, you will implement and evaluate (i) naïve Bayes and (ii) logistic regression. Due dates: First commit 10/5: 10 points, Second commit 10/12: 20 points, Final commit 10/19: 30 points.

a. Naïve Bayes (15 points)

Implement the naïve Bayes algorithm discussed in class. Discretize continuous values. To do this, partition the range of the feature into *k* bins (value set through an option). Then replace the feature with a discrete feature that takes value *x* if the original feature's value was in bin *x*. Use *m*-estimates to smooth your probability estimates. Use logs whenever possible to avoid multiplying too many probabilities together. Your main file should be called **nbayes**. It should take four options:

1. Option 1 will be the path to the data (see problem 1).
2. Option 2 is a 0/1 option. If 0, use cross validation. If 1, run the algorithm on the full sample.
3. Option 3 (at least 2) is the number of bins for any continuous feature.
4. Option 3 is the value of *m* for the *m*-estimate. If this value is negative, use Laplace smoothing. Note that *m*=0 is maximum likelihood estimation. The value of *p* in the *m*-estimate should be fixed to 1/*v* for a variable with *v* values.

When your code is run, it should first construct 5 folds using stratified cross validation if this option is provided. To ensure repeatability, set the random seed for the PRNG to 12345. Then it should produce naïve Bayes models on each fold (or the sample according to the option) and report as in part (c).

b. Logistic Regression (20 points)

Implement the logistic regression algorithm described in class. During learning, minimize the negative conditional log likelihood plus a constant (λ) times a penalty term, half of the 2-norm of the weights squared. You can use standard gradient descent for the minimization. Nominal attributes should be encoded as follows: map each value to a number 1…*k* if there are *k* values. The main file should be called **logreg**. It should take three options: the first two options above and a third which is a nonnegative real number that sets the value of the constant λ. The same notes about 5 fold stratified CV from above, etc. apply in this case.

c. Output format

When either algorithm is run on any problem, it must produce output in exactly the following format:

Accuracy: 0.xyz 0.abc

Precision: 0.xyz 0.abc

Recall: 0.xyz 0.abc

Area under ROC: 0.xyz

For all metrics expect Area under ROC, "0.xyz" is the average value of each quantity over five folds. "0.abc" is the standard deviation. For Area under ROC, use the "pooling" method described in problem 2.

d. Writeup (15 points)

Prepare a writeup on your experiments. In your writeup, answer the following questions:

(a) What is the accuracy of naïve Bayes with Laplace smoothing and logistic regression (λ =1) on the different learning problems? For each problem, perform a *t*-test to determine if either is superior with 95% confidence.

(b) Examine the effect of the number of bins when discretizing continuous features in naïve Bayes. Do this by comparing accuracy across several different values of this parameter using *volcanoes*.

(c) Examine the effect of *m* in the naïve Bayes *m*-estimates. Do this by comparing accuracy across *m*=0, Laplace smoothing, *m*=10 and 100 on the given problems.

(d) Examine the effect of λ on logistic regression. Do this by comparing accuracy across λ=0, 1, and 10 for the given problems.

Write down any further insights or observations you made while implementing and running the algorithms, such as time and memory requirements, the complexity of the code, etc. Especially interesting insights may be awarded extra points.

Because this is a long assignment, you are expected to make intermediate weekly commits to the repository. Each week, we expect to see substantial progress made in the implementation. Create a separate subdirectory “P2” in your git repository, place your code in it and push to csevcs. Include a short README file containing (i) a summary of the work done for the week and (ii) your experience with the API and documentation, and anything you found confusing. Do NOT commit any other code.

3. In this problem, you will implement and evaluate (i) bagging and (ii) boosting. Due dates: First commit 10/26: 20 points, Final commit 11/2: 40 points.

a. Bagging (15 points) and Boosting (30 points)

Implement the bagging and boosting algorithms discussed in class. Your main files should be called **bag** and **boost**. They should take four options:

1. Option 1 is the path to the data (see problem 1).
2. Option 2 is a 0/1 option. If 0, use cross validation. If 1, run the algorithm on the full sample.
3. Option 3 is a learning algorithm. This can be **dtree**, **nbayes** or **logreg**. For each value, the base classifier will be the corresponding algorithm you have implemented in your previous assignments. You should use the decision stumps (trees of depth 1, containing just the root), for the **dtree** option. All parameters for these base classifiers should be "reasonable" values, i.e. use values that you have previously found to work well with these algorithms.
4. Option 4 is the number of iterations to perform bagging or boosting.

Remember that for boosting, you will need to modify your previous code so it works with weighted examples. Some of these modifications were described in class.

When your code is run, it should first construct 5 folds using stratified cross validation if this option is provided. To ensure repeatability, set the random seed for the PRNG to 12345. Then it should produce bagged/boosted models on each fold (or the sample according to the option) and report the following.

b. Output format

When your code is run on any problem, it must produce output in exactly the following format:

Accuracy: 0.xyz 0.abc

Precision: 0.xyz 0.abc

Recall: 0.xyz 0.abc

Area under ROC: 0.xyz

For all metrics expect Area under ROC, "0.xyz" is the average value of each quantity over five folds. "0.abc" is the standard deviation. For Area under ROC, use the "pooling" method described in problem 2.

c. Writeup (15 points)

Prepare a writeup on your experiments. In your writeup, answer the following questions:

(a) For all problems and any learning algorithm, compare the accuracy of the ensemble versions (10 iterations) to the base learners. Produce a table with the accuracies of the base learner, the bagged learner, and the boosted learner. Perform paired *t*-tests to determine if any of the ensemble methods are significantly better than the base learner with 95% confidence.

(b) For any two problems and any learning algorithm, evaluate how the accuracy of bagging changes with the number of iterations. Pick at least three iteration values between 2 and 50, and plot the accuracy on a graph. Do you see any difference by problem?

(c) Repeat (b) for boosting.

(d) Evaluate the sensitivity of bagging to noise as follows. When training, after constructing the training sample, flip an example's label with probability *p*. Then use this noisy sample in your bagging algorithm and evaluate the resulting classifier on the usual (noise free) test set. For any two problems and any learning algorithm, plot a graph with *p* on the x-axis and the test-set accuracy of bagging (30 iterations) on the y-axis. You can use results from the previous questions for a $p=0$ point. Discuss how resilient bagging is to noise based on your observations.

(e) Repeat (d) for boosting.

Write down any further insights or observations you made while implementing and running the algorithms, such as time and memory requirements, the complexity of the code, etc. Especially interesting insights may be awarded extra points.

Because this is a long assignment, you are expected to make intermediate weekly commits to the repository. Each week, we expect to see substantial progress made in the implementation. Create a separate subdirectory "P4" in your git repository, place your code in it and push to csevcs. Include a short README file containing (i) a summary of the work done for the week and (ii) your experience with the API and documentation, and anything you found confusing. Do NOT commit any other code.

4. In this problem, you will implement and evaluate feature selection strategies. Due dates: First commit 11/9: 20 points, Final commit 11/16: 40 points.

a. Forward and Backward Feature Selection (35 points)

Implement forward feature selection and backward feature selection as discussed in class. Your main files should be called **ffeatsel** and **bfeatsel**. They should take three options:

1. Option 1 is the path to the data (see problem 1).
2. Option 2 is the learning algorithm. This can be **dtree**, **nbayes** or **logreg**. For each value, the underlying classifier will be the corresponding algorithm you have implemented in your previous assignments. You should use full decision trees for the **dtree** option. All parameters for these base classifiers should be "reasonable" values, i.e. use values that you have previously found to work well with these algorithms.
3. Option 3 is the number of features to return (1 or more).

(Note that the 0/1 option for folds is not needed here.) When your code is run, it should first construct 5 folds using stratified cross validation. To ensure repeatability, set the random seed for the PRNG to 12345. Then it should perform forward or backward feature selection, using an *internal* 3-fold cross validation loop to construct validation sets. The feature selection procedure should terminate once it has selected at most <Option 3> features. In each outer fold, the methods should return the best set of features they found, so that the size of the set is less than or equal to Option 3. To compute this set of features, first compute the number of folds (out of 3) where each feature was chosen. Sort features by this number, and pick the best ones, breaking ties randomly. Finally, the learning algorithm should retrain a classifier on each training fold using just the returned set of features, evaluate it on the (outer) test set and report the following.

b. Output format

When your code is run on any problem, it must produce output in exactly the following format:

```
Accuracy: 0.xyz 0.abc

Precision: 0.xyz 0.abc

Recall: 0.xyz 0.abc

Area under ROC: 0.xyz
```

For all metrics expect Area under ROC, "0.xyz" is the average value of each quantity over five folds. "0.abc" is the standard deviation. For Area under ROC, use the "pooling" method described in problem 2.

c. Writeup (25 points)

Prepare a writeup on your experiments. In your writeup, answer the following questions:

(a) Examine the effect of the number of features selected on the accuracy of the resulting classifier. Do this by plotting the accuracy of an algorithm using at least four different values of features retained on the x axis and the accuracy on the y axis. As the fifth point on the x axis, use all the features. Discuss the resulting graphs.

(b) Compare the accuracy of the classifiers obtained with forward and backward feature selection using the graphs from (a). Do you notice a difference between the two?

(c) Compare the accuracy of forward and backward feature selection to boosted feature selection, which involves running boosting with decision stumps for *k* iterations, where *k* is the desired number of features.

(d) Discuss the differences in average runtime between the three feature selection approaches for a given number of features to be selected. In this runtime, do *not* include the time to train the final classifier. Which approach has the best combination of speed and accuracy in your opinion?

Write down any further insights or observations you made while implementing and running the algorithms, such as time and memory requirements, the complexity of the code, etc. Especially interesting insights may be awarded extra points.

Because this is a long assignment, you are expected to make intermediate weekly commits to the repository. Each week, we expect to see substantial progress made in the implementation. Create a separate subdirectory “P5” in your git repository, place your code in it and push to csevcs. Include a short README file containing (i) a summary of the work done for the week and (ii) your experience with the API and documentation, and anything you found confusing. Do NOT commit any other code.