

Project 3: Applied Learning Theory

Introduction

In project 2, we are looking at the applied side of learning theory. There are two components of this project: sides of learning theory we've considered:

1. Boosting (AdaBoost)
2. Reductions from multiclass to binary classification

There are two data sets: `5ng-test` for real testing (this is a five-class subset of the 20 newsgroups data) and `simplifiedata`, a synthetic and small data set.

Task 1: Boosting

(40%)

In this task, we will implement AdaBoost. A shell for AdaBoost is in `Boost.m`. You will need to fill in the `BoostTrain` and `BoostPredict` functions. See the comments in `Boost.m` for a description of what each function should do and how it should store its outputs. Note that in order to do boosting properly, you need to have binary classification algorithms that can accept weighted data. I have provided an implementation of decision trees that does so (in `DT.m`). The DT implementation has a maximum-depth parameter (as opposed to threshold) for pruning. This is to make it easier to train decision stumps.

Once you've implemented `Boost.m`, I suggest testing it on the `simplifiedata` data set using decision stumps as the base learner. To do this, first load the data:

```
>> trX = load('simplifiedataX');  
>> trY = load('simplifiedataY');
```

Next, let's just train a simple decision stump and evaluate its performance on the training data:

```
>> model = DT('train', trX, trY, ones(size(trY)), 1);  
>> mean(trY == DT('predict', model, trX))  
ans = 0.82000
```

In the first line, we're building the decision tree. `trX` and `trY` are the training data points and labels. The `ones(...)` command is for the *importance weights* of the training points—they are all equally weighted. The final argument is the maximum depth. A stump has depth 1.

In the second line, we are computing the accuracy of the decision tree on the training data, by computing the mean of whether the predictions match the truth. In this case, we get 82% accuracy.

(If you retrain with maximum depth 2, you should get 86% accuracy.)

Now, we boost the model. In order to call `Boost`, we need to supply it with the base learner we want to use (in this case, DT), the training data, the number of rounds of boosting to perform, and any extra arguments to the base learner. In this case, to build stumps, we want to pass 1 as an additional argument. We then evaluate. Let's try boosting a few different number of iterations (one through 4). Note the alphas and errors that are created as part of the boosted model:

```

>> model = Boost('train', @DT, trX, trY, 1, 1)
model =
  models: {[1x1 struct]}
  alphas: {[0.7582]}
  errors: {[0.1800]}

>> mean(trY == Boost('predict', @DT, model, trX))
ans =
  0.8200

>> model = Boost('train', @DT, trX, trY, 2, 1)
model =
  models: {[1x1 struct] [1x1 struct]}
  alphas: {[0.7582] [0.5367]}
  errors: {[0.1800] [0.2547]}

>> mean(trY == Boost('predict', @DT, model, trX))
ans =
  0.8200

>> model = Boost('train', @DT, trX, trY, 3, 1)
model =
  models: {[1x1 struct] [1x1 struct] [1x1 struct]}
  alphas: {[0.7582] [0.5367] [0.3186]}
  errors: {[0.1800] [0.2547] [0.3459]}

>> mean(trY == Boost('predict', @DT, model, trX))
ans =
  0.8600

>> model = Boost('train', @DT, trX, trY, 4, 1)
model =
  models: {[1x1 struct] [1x1 struct] [1x1 struct] [1x1 struct]}
  alphas: {[0.7582] [0.5367] [0.3186] [0.0625]}
  errors: {[0.1800] [0.2547] [0.3459] [0.4688]}

>> mean(trY == Boost('predict', @DT, model, trX))
ans =
  0.8600

```

What we see is that after three iterations of boosting, performance has gone from 82% to 86%. If we boost more, it doesn't really help. We can try, for instance, ten iterations:

```

>> model = Boost('train', @DT, trX, trY, 10, 1)
model =
  models: {1x10 cell}
  alphas: {[0.7582] [0.5367] [0.3186] [0.0625] [0.0570]
           [0.0301] [0.0060] [0.0022] [0.0018] [5.3642e-04]}
  errors: {[0.1800] [0.2547] [0.3459] [0.4688] [0.4715]
           [0.4850] [0.4970] [0.4989] [0.4991] [0.4997]}

```

Task 2: All-versus-all Classification

(30%)

The second thing we'll implement is all-versus-all classification. As a reference, and so we can compare results later, I've implemented *one-versus-all* classification in `OVA.m`. Your job is to implement `AVA.m`, which, as usual, is partially written for you. Since `simplifiedata` is just binary, we can't test OVA and AVA on it, so we'll have to use the `5ng` data set. Let's load the data and inspect it:

```
>> load 5ng
>> size(trX)
ans =
    2795    3530

>> size(teX)
ans =
    1873    3530

>> myunique(trY)
ans =
     1
     2
     3
     4
     5
```

(Note, `myunique` returns the unique values in a vector: here, we see that there are five unique classes.)

Warning: if you're on Octave you'll have to load `5ng-octave` instead. Moreover, the lack of support for sparse matrices will make this part slow.

For initial code-testing purposes, let's use a small subset of the data for training and testing. We can create these as follows:

```
>> trX1 = trX(1:20:end,1:10:end);
>> trY1 = trY(1:20:end);
>> trX2 = trX(2:20:end,1:10:end);
>> trY2 = trY(2:20:end);
```

We'll use the "1" data for training and the "2" data for testing. These should contain 140 examples each and 353 features.

We'll start by running `OVA`, which will work without any effort on your part. We'll use a decision tree as the base learner with a maximum depth of 3. As `OVA` runs, it will tell you which class it's building a classifier for. We'll then check training error and test error:

```
>> ova = OVA('train', @DT, trX1, trY1, 3);
1...2...3...4...5...
>> mean(trY1 == OVA('predict', @DT, ova, trX1))
ans = 0.50714
>> mean(trY2 == OVA('predict', @DT, ova, trX2))
ans = 0.27857
```

These accuracies are pretty crummy, but that's largely because we built very shallow DTs on a very small subset of the data.

Now, that's all well and good, so let's try your AVA code. We'll do basically the exact same thing:

```
>> ava = AVA('train', @DT, trX1, trY1, 3);
1...2...3...4...5...
>> mean(trY1 == AVA('predict', @DT, ava, trX1))
ans = 0.50714
>> mean(trY2 == AVA('predict', @DT, ava, trX2))
ans = 0.29286
```

So we see that on the test data, AVA seems to be doing slightly better. Note that since DT supports multiclass “naturally,” we can also just run the DT:

```
>> model = DT('train', trX1, trY1, ones(size(trY1)), 3);
>> mean(trY1 == DT('predict', model, trX1))
ans = 0.40714
>> mean(trY2 == DT('predict', model, trX2))
ans = 0.27857
```

As we can see, the AVA actually *outperforms* the internal multiclass handling of DTs (though I wouldn't trust this too much—this is a really tiny, unrealistic dataset).

Putting it all together...

(30%)

The `run.m` script is just like all other projects. When you execute `run.m`, it will attempt to load data from `results.mat` and will also attempt to detect if you're running Matlab or Octave (plotting commands vary slightly between the two, much to my chagrin). It will tell you what it's doing and if it's wrong, you might want to go in and just hard-code the correct responses. Next it will load data from `5ng` (or `5ng-octave`, if it thinks you're running Octave). It then runs the following set of experiments, each of which leads to a single plot. (If you rerun `run.m` later, with `result.mat` saved, it will still regenerate the plots, but will not run all the classifiers.) I've also listed the amount of time that it takes my solution to run for each part, so you have a sense. The total time was 32 minutes.

1. Boosted decision stumps on binary data, with variable number of boosting iterations. (2 minutes)
2. Boosting decision *trees* on binary data, with varying tree depth. (5 minutes)
3. Multiclass-to-binary reduction with DT, with learning curve. (13 minutes)
4. Multiclass-to-binary reduction with DT, with varying number of classes. (14 minutes)

Each figure is now labeled better and the questions below refer to each of the previous steps (each of which produces it's own figure).

- F1:** (a) In this figure, we see training (blue) and test (red) accuracy curves for different amounts of boosting. Do you observe any overfitting or underfitting?
 (b) How would you decide when to stop boosting?
- F2:** (a) Here we see the effect of boosting not just stumps (decision trees with depth 1), but also fuller decision trees. The curves are both for test accuracy. Where does boosting help the most and the least?
 (b) What do you think would happen if we let the the maximum depth get larger?

- F3:** (a) Here, we see learning curves for multiclass reductions for perceptron, using both AVA (green, purple) and OVA (blue, red). The two top curves (blue, green) are training data; the two bottom (red, purple) are test data. Between OVA and AVA, which seems to work better. Why do you think this is?
- (b) Discuss the effect that you see on the training curves, based on what we know about these two algorithms.
- F4:** (a) In this plot, we see OVA and AVA test accuracies on a varying number of classes (from 2 to 5). Note that we expect the accuracies to go down as the number of classes increases (since the problem is getting harder). However, does one reductions seem to fare better (relatively) as the number of classes increases? How does this relate to the theoretical analysis we had of these algorithms?

What to hand in

Please hand in (a) your `results.mat` file and (b) a `writeup.pdf` containing the answers to all the questions in the previous section. Note that you do *not* have to discuss anything related to the “toy” data set.