CS178 Homework #4 Solutions

March 10, 2017

```
In [1]: import numpy as np
        np.random.seed(0)
        import mltools as ml
        import matplotlib.pyplot as plt   # use matplotlib for plotting with inline plots
        %matplotlib inline
```

Problem 1: Decision Trees

You can most easily do this problem by hand, but since I have to type a solution I will put it into the Python notebook.

```
In [2]: Xy = np.array(
             [[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] ]);
        ep = 1e-12;  # to move values away from log(0)=-infty
        X = Xy[:,0:-1];
        Y = Xy[:,-1];
```

(a) Calculate the entropy of the class variable y

```
In [3]: p = np.mean(Y>0)
        Hy = -(p*np.log2(p) + (1-p)*np.log2(1-p))
        print "H(y) = {} bits".format(Hy)
```

```
H(y) = 0.970950594455 bits
```

(b) Calculate the information gain for each feature xi. Which feature should be split first?

```
In [4]: IG = [0.0]*5
        for i in range(5):
            idx = X[:,i]>0;
            if (idx.sum()==0 or (1-idx).sum()==0):
                IG[i]=0
                continue
            p1 = np.mean( Y[idx]>0 )+ep;
            p0 = np.mean(Y[idx==False]>0)+ep;
            a = np.mean(idx)+ep;
            IG[i] = Hy + a*(p1*np.log2(p1)+(1-p1)*np.log2(1-p1))+(1-a)*(p0*np.log2(p0)+(1-p0)*np

        print IG
```

```
[0.046439344670192784, 0.60998654699205646, 0.0058021490136882514, 0.091277446241109894, 0.00580
```

So, we should select the 2nd feature.

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

```
In [5]: # Split on feature 2:
        print "Splitting on feature 2:"
        print "Left data: \n", Xy[X[:,1]==0,:]
        print "Right data: \n", Xy[X[:,1]==1,:]
```

```
Splitting on feature 2:
Left data:
[[ 0  0  1  1  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]]
Right data:
[[ 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  1  1  1  1 -1]]
```

```
In [6]: # On the right data, we will always predict "-1".
        # On the left data, we'll need to split again.
        # You can see by inspection that the next best feature is the first
        #   or you can recompute it as above
        print "Splitting left data on feature 1 next:"
        XLeft = Xy[X[:,1]==0,:]
        print "Left data: \n", XLeft[ XLeft[:,0]==0, :]
        print "Right data: \n", XLeft[ XLeft[:,0]==1, :]
```

```
Splitting left data on feature 1 next:
Left data:
[[ 0  0  1  1  0 -1]
 [ 0  0  1  0  0  1]]
Right data:
[[1 0 1 1 1 1]
 [1 0 0 0 0 1]
 [1 0 1 1 0 1]]
```

```
In [7]: # On the right data, we always predict "+1"
        # On the left data, we can see that only the fourth feature is informative

        # So the final rule is:
        # if (long):
        #     discard
        # else:
        #     if (known):
        #         read
        #     else:
        #         if (has "grade"): discard
        #         else: read
```

Problem 2: Decision trees on Kaggle

```
In [8]: import mltools.dtree as dt;
        reload(dt);
```

(Part A) Load the data

```
In [13]: X = np.genfromtxt("project/X_train.txt",delimiter=' ')
         Y = np.genfromtxt("project/Y_train.txt",delimiter=' ')
         Xt,Xv,Yt,Yv = ml.splitData(X,Y,0.80)

         Xe = np.genfromtxt('project/X_test.txt',delimiter=' ')
```

(Part B) Train a basic decision tree:

```
In [16]: lr = dt.treeClassify(Xt,Yt, maxDepth=50)
```

```
In [17]: print "Train ERR: ",lr.err(Xt,Yt)
         print "Valid ERR: ",lr.err(Xv,Yv)
```

```
Train ERR:  0.0341125
Valid ERR:  0.33205
```

Looks overfit!
(Part C) Let's look at controlling the depth:

```
In [18]: for depth in range(16):
             lr.train(Xt,Yt, maxDepth=depth)
             print "Depth {:02d}: {} train, {} validation".format(depth, lr.err(Xt,Yt), lr.err(X
```

```
Depth 00: 0.3411125 train, 0.34165 validation
Depth 01: 0.3331625 train, 0.33065 validation
Depth 02: 0.32165 train, 0.32045 validation
Depth 03: 0.3172875 train, 0.3177 validation
Depth 04: 0.315225 train, 0.31675 validation
Depth 05: 0.3109375 train, 0.31475 validation
Depth 06: 0.3077 train, 0.31035 validation
Depth 07: 0.3020125 train, 0.30795 validation
Depth 08: 0.2972375 train, 0.30765 validation
Depth 09: 0.290225 train, 0.30375 validation
Depth 10: 0.2812625 train, 0.30265 validation
Depth 11: 0.272975 train, 0.3038 validation
Depth 12: 0.263225 train, 0.30285 validation
Depth 13: 0.2514875 train, 0.3042 validation
Depth 14: 0.2382875 train, 0.30635 validation
Depth 15: 0.22445 train, 0.3073 validation
```

(Part D, E) and now at the number of data required to split, either in the resulting child (leaf), or in the parent:

```
In [19]: for i in range(2,13):
             lr.train(Xt,Yt, maxDepth=20, minLeaf=2**i)
             print "> 2^{:d} data per leaf: {} train, {} validation".format(i, lr.err(Xt,Yt), lr
```

```
> 2^2 data per leaf: 0.1820625 train, 0.3167 validation
> 2^3 data per leaf: 0.201175 train, 0.3172 validation
> 2^4 data per leaf: 0.225275 train, 0.3149 validation
> 2^5 data per leaf: 0.2542375 train, 0.31015 validation
> 2^6 data per leaf: 0.2766125 train, 0.3086 validation
> 2^7 data per leaf: 0.28855 train, 0.3077 validation
> 2^8 data per leaf: 0.2980125 train, 0.3064 validation
> 2^9 data per leaf: 0.3036125 train, 0.3079 validation
> 2^10 data per leaf: 0.3104125 train, 0.3103 validation
> 2^11 data per leaf: 0.313875 train, 0.3134 validation
> 2^12 data per leaf: 0.32165 train, 0.32045 validation
```

```
In [20]: for i in range(2,13):
             lr.train(Xt,Yt, maxDepth=20, minParent=2**i)
             print "> 2^{:d} data per parent: {} train, {} validation".format(i, lr.err(Xt,Yt),
```

```
> 2^2 data per parent: 0.158525 train, 0.3167 validation
> 2^3 data per parent: 0.1708625 train, 0.3152 validation
> 2^4 data per parent: 0.190225 train, 0.31695 validation
```

```
> 2^5 data per parent: 0.213675 train, 0.3161 validation
> 2^6 data per parent: 0.2358125 train, 0.3128 validation
> 2^7 data per parent: 0.2566 train, 0.3103 validation
> 2^8 data per parent: 0.2746125 train, 0.3099 validation
> 2^9 data per parent: 0.2885 train, 0.3076 validation
> 2^10 data per parent: 0.2996375 train, 0.30785 validation
> 2^11 data per parent: 0.3081 train, 0.30985 validation
> 2^12 data per parent: 0.311025 train, 0.31225 validation
```

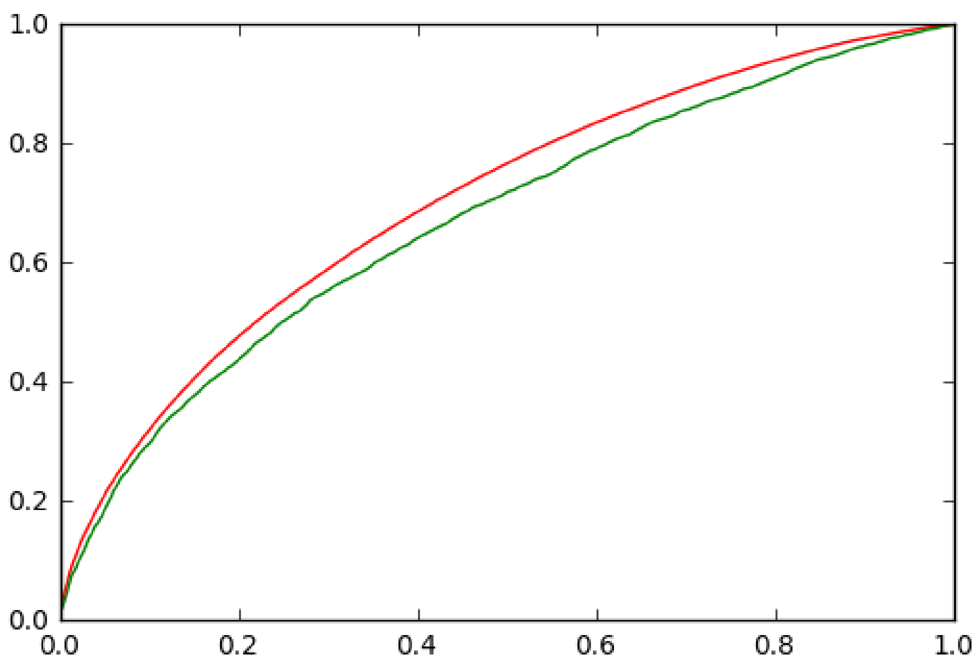
I'll pick minLeaf = 2^8:

```
In [21]: lr.train(Xt,Yt,maxDepth=20,minLeaf=2**8)
```

(Part F) Check the AUC scores and the ROC curve:

```
In [22]: fprT,tprT,tnrT = lr.roc(Xt,Yt)
         fprV,tprV,tnrV = lr.roc(Xv,Yv)
         plt.plot(fprT,tprT,'r-',fprV,tprV,'g-')
         print "AUC: Train ",lr.auc(Xt,Yt),"   Valid ",lr.auc(Xv,Yv)
```

```
AUC: Train  0.703419880746    Valid  0.671362503932
```



(Part G) Make predictions & upload them to Kaggle

```
In [23]: lr.train(X,Y, minLeaf=2**8, maxDepth=20)
```

```
In [25]: # Output our predictions to a file:
         YeHat = lr.predictSoft(Xe)[:,1]
         np.savetxt('project/predict_dtree.csv', np.vstack((np.arange(len(YeHat)),YeHat)).T,'%d,

         # and then upload them!
```

Problem 3: Random Forests

```
In [41]: M = Xt.shape[0]
         Mv= Xv.shape[0]
         rforest = [None]*25
         YtHat = np.zeros((M,25))
         YvHat = np.zeros((Mv,25))
         for l in range(25):
             #print "Training {}".format(l)                # uncomment if you want to monitor pr
             Xi,Yi = ml.bootstrapData(Xt,Yt, M)            # draw this member's random sample of
             rforest[l] = dt.treeClassify()                #   and train the model on that draw
             rforest[l].train(Xi,Yi,maxDepth=20,nFeatures=10)
             YtHat[:,l] = rforest[l].predict(Xt)           # predict on training data
             YvHat[:,l] = rforest[l].predict(Xv)           #   and validation data & save result

             if l+1 in [1,5,10,15,25]:
                 # Make the prediction (mean of columns 0...l-1) and score the error rate:
                 errT = ((Yt - YtHat[:,0:l+1].mean(axis=1)>.5)).mean()
                 errV = ((Yv - YvHat[:,0:l+1].mean(axis=1)>.5)).mean()
                 print "{:02d} members: {} train, {} valid".format(l+1,errT,errV)
```

```
01 members: 0.1380125 train, 0.2017 valid
05 members: 0.1261375 train, 0.2076 valid
10 members: 0.106425 train, 0.1953 valid
15 members: 0.1262375 train, 0.21415 valid
25 members: 0.1238625 train, 0.2151 valid
```

To compute AUC, it is easiest to wrap our ensemble in a classifier object:

```
In [46]: class randomForest(ml.base.classifier):
             def __init__(self, learners):
                 self.learners=learners;
                 self.classes=learners[0].classes;
             def predictSoft(self,X):
                 ysoft = np.zeros((X.shape[0],len(self.classes)));
                 for i in range(len(self.learners)): ysoft[:,1]+=self.learners[i].predict(X);
                 return ysoft/len(self.learners);

         rf = randomForest(rforest);
         print "AUC Train: ",rf.auc(Xt,Yt)," Valid: ",rf.auc(Xv,Yv)
```

```
AUC Train:  0.954917550798  Valid:  0.72930380735
```

In a slightly different variant, we can try averaging the trees' confidence (instead of their class prediction):

```
In [48]: class randomForest2(ml.base.classifier):
             def __init__(self, learners):
                 self.learners=learners
                 self.classes=learners[0].classes
             def predictSoft(self,X):
                 ysoft = np.zeros((X.shape[0],len(self.classes)));
                 for i in range(len(self.learners)): ysoft+=self.learners[i].predictSoft(X);
                 return ysoft/len(self.learners);

         rf2 = randomForest2(rforest);
         print "AUC Train: ",rf2.auc(Xt,Yt)," Valid: ",rf2.auc(Xv,Yv)
```

```
AUC Train:  0.964688724452  Valid:  0.735652765839
```

```
In [ ]:
```