Machine Learning IN ACTION

Peter Harrington

SAMPLE CHAPTER

MANNING

Machine Learning in Action

by Peter Harrington

Chapter 7

brief contents

7 Improving classification with the AdaBoost meta-algorithm

This chapter covers

- Combining similar classifiers to improve performance
- Applying the AdaBoost algorithm
- Dealing with classification imbalance

If you were going to make an important decision, you'd probably get the advice of multiple experts instead of trusting one person. Why should the problems you solve with machine learning be any different? This is the idea behind a meta-algorithm. Meta-algorithms are a way of combining other algorithms. We'll focus on one of the most popular meta-algorithms called AdaBoost. This is a powerful tool to have in your toolbox because AdaBoost is considered by some to be the best-supervised learning algorithm.

In this chapter we're first going to discuss different ensemble methods of classification. We'll next focus on boosting and AdaBoost, an algorithm for boosting.

We'll then build a decision stump classifier, which is a single-node decision tree. The AdaBoost algorithm will be applied to our decision stump classifier. We'll put our classifier to work on a difficult dataset and see how it quickly outperforms other classification methods.

Finally, before we leave the subject of classification, we're going to talk about a general problem for all classifiers: classification imbalance. This occurs when we're trying to classify items but don't have an equal number of examples. Detecting fraudulent credit card use is a good example of this: we may have 1,000 negative examples for every positive example. How do classifiers work in such a situation? You'll see that you may need to use alternate metrics to evaluate a classifier's performance. This subject isn't unique to AdaBoost, but because this is the last classification chapter, it's a good time to discuss it.

7.1 Classifiers using multiple samples of the dataset

AdaBoost

Pros: Low generalization error, easy to code, works with most classifiers, no parameters to adjust

Cons: Sensitive to outliers

Works with: Numeric values, nominal values

You've seen five different algorithms for classification. These algorithms have individual strengths and weaknesses. One idea that naturally arises is combining multiple classifiers. Methods that do this are known as *ensemble methods or meta-algorithms*. Ensemble methods can take the form of using different algorithms, using the same algorithm with different settings, or assigning different parts of the dataset to different classifiers. We'll next talk about two methods that use multiple instances of the same classifier and alter the dataset applied to these classifiers. Finally, we'll discuss how to approach AdaBoost with our general framework for approaching machine-learning problems.

7.1.1 Building classifiers from randomly resampled data: bagging

Bootstrap aggregating, which is known as bagging, is a technique where the data is taken from the original dataset S times to make S new datasets. The datasets are the same size as the original. Each dataset is built by randomly selecting an example from the original with replacement. By "with replacement" I mean that you can select the same example more than once. This property allows you to have values in the new dataset that are repeated, and some values from the original won't be present in the new set.

After the S datasets are built, a learning algorithm is applied to each one individually. When you'd like to classify a new piece of data, you'd apply our S classifiers to the new piece of data and take a majority vote.

There are more advanced methods of bagging, such as random forests. A good discussion of these methods can be found at http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm. We'll now turn our attention to *boosting*: an ensemble method similar to bagging.

7.1.2 Boosting

Boosting is a technique similar to bagging. In boosting and bagging, you always use the same type of classifier. But in boosting, the different classifiers are trained sequentially. Each new classifier is trained based on the performance of those already trained. Boosting makes new classifiers focus on data that was previously misclassified by previous classifiers.

Boosting is different from bagging because the output is calculated from a weighted sum of all classifiers. The weights aren't equal as in bagging but are based on how successful the classifier was in the previous iteration.

There are many versions of boosting, but this chapter will focus on the most popular version, called AdaBoost.

General approach to AdaBoost

1. Collect: Any method.
2. Prepare: It depends on which type of weak learner you're going to use. In this chapter, we'll use decision stumps, which can take any type of data. You could use any classifier, so any of the classifiers from chapters 2–6 would work. Simple classifiers work better for a weak learner.
3. Analyze: Any method.
4. Train: The majority of the time will be spent here. The classifier will train the weak learner multiple times over the same dataset.
5. Test: Calculate the error rate.
6. Use: Like support vector machines, AdaBoost predicts one of two classes. If you want to use it for classification involving more than two classes, then you'll need to apply some of the same methods as for support vector machines.

We're now going to discuss some of the theory behind AdaBoost and why it works so well.

7.2 Train: improving the classifier by focusing on errors

An interesting theoretical question is can we take a weak classifier and use multiple instances of it to create a strong classifier? By "weak" I mean the classifier does a better job than randomly guessing but not by much. That is to say, its error rate is greater than 50% in the two-class case. The "strong" classifier will have a much lower error rate. The AdaBoost algorithm was born out of this question.

AdaBoost is short for *adaptive boosting*. AdaBoost works this way: A weight is applied to every example in the training data. We'll call the weight vector `D`. Initially, these weights are all equal. A weak classifier is first trained on the training data. The errors from the weak classifier are calculated, and the weak classifier is trained a second time with the same dataset. This second time the weak classifier is trained, the weights of the training set are adjusted so the examples properly classified the first time are weighted less and the examples incorrectly classified in the first iteration are weighted more. To get one answer from all of these weak classifiers, AdaBoost assigns α values to each of the classifiers. The α values are based on the error of each weak classifier. The error ε is given by

$$\varepsilon = \frac{\textit{number of incorrectly classified examples}}{\textit{total number of examples}}$$

and α is given by

$$\alpha = \frac{1}{2} ln\left(\frac{1-\varepsilon}{\varepsilon}\right)$$

The AdaBoost algorithm can be seen schematically in figure 7.1.

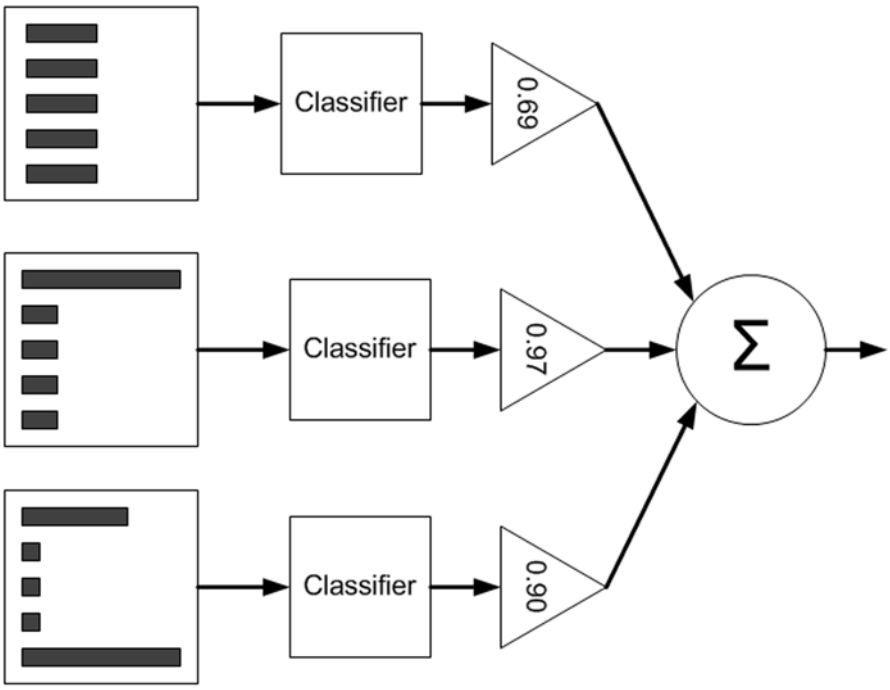

Figure 7.1 Schematic representation of AdaBoost; with the dataset on the left side, the different widths of the bars represent weights applied to each instance. The weighted predictions pass through a classifier, which is then weighted by the triangles (α values). The weighted output of each triangle is summed up in the circle, which produces the final output.

After you calculate α, you can update the weight vector D so that the examples that are correctly classified will decrease in weight and the misclassified examples will increase in weight. D is given by

$$D_i^{(t+1)} = \frac{D_i^{(t)} e^{-\alpha}}{Sum(D)}$$

if correctly predicted and

$$D_i^{(t+1)} = \frac{D_i^{(t)} e^{\alpha}}{Sum(D)}$$

if incorrectly predicted.

After D is calculated, AdaBoost starts on the next iteration. The AdaBoost algorithm repeats the training and weight-adjusting iterations until the training error is 0 or until the number of weak classifiers reaches a user-defined value.

We're going to build up to the full AdaBoost algorithm. But, before we can do that, we need to first write some code to create a weak classifier and to accept weights for the dataset.

7.3 Creating a weak learner with a decision stump

A *decision stump* is a simple decision tree. You saw how decision trees work earlier. Now, we're going to make a decision stump that makes a decision on one feature only. It's a tree with only one split, so it's a stump.

While we're building the AdaBoost code, we're going to first work with a simple dataset to make sure we have everything straight. You can create a new file called adaboost.py and add the following code:

```
def loadSimpData():
    datMat = matrix([[ 1. ,  2.1],
        [ 2. ,  1.1],
        [ 1.3,  1. ],
        [ 1. ,  1. ],
        [ 2. ,  1. ]])
    classLabels = [1.0, 1.0, -1.0, -1.0, 1.0]
  return datMat,classLabels
```

You can see this data in figure 7.2. Try choosing one value on one axis that totally separates the circles from the squares. It's not possible. This is the famous 45 problem that decision trees are notorious for having difficulty with. AdaBoost will need to use multiple decision stumps to properly classify this dataset. By using multiple decision stumps, we'll be able to build a classifier to completely classify the data.

You can load the dataset and class labels by typing in

```
>>> import adaboost
>>> datMat,classLabels=adaboost.loadSimpData()
```

Now that you have the dataset loaded, we can create a few functions to build our decision stump.

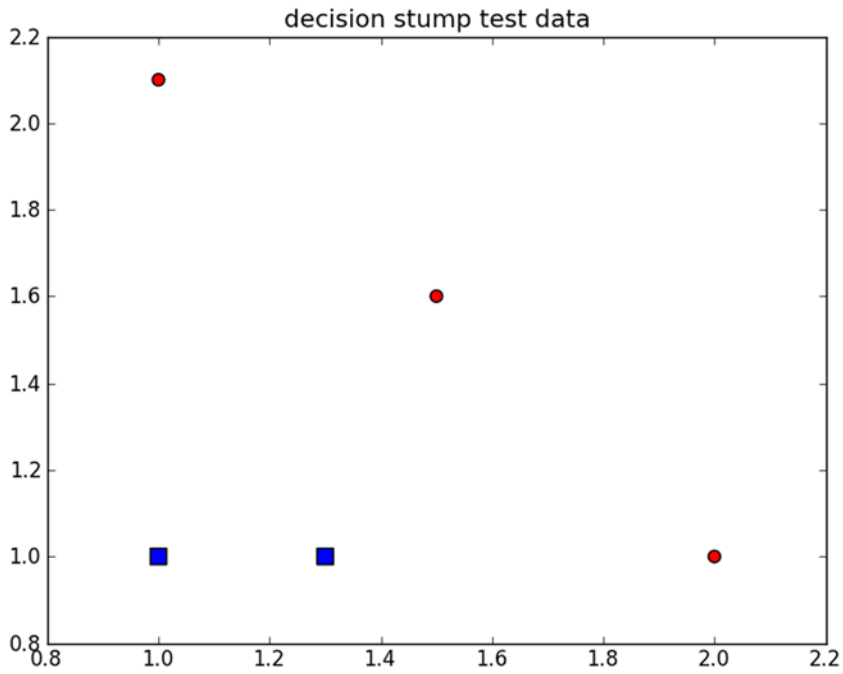

Figure 7.2 Simple data used to check the AdaBoost building functions. It's not possible to choose one threshold on one axis that separates the squares from the circles. AdaBoost will need to combine multiple decision stumps to classify this set without error.

The first one will be used to test if any of values are less than or greater than the threshold value we're testing. The second, more involved function will loop over a weighted version of the dataset and find the stump that yields the lowest error.
The pseudo-code will look like this:

Set the minError to +∞
For every feature in the dataset:
    For every step:
        For each inequality:
            Build a decision stump and test it with the weighted dataset
            If the error is less than minError: set this stump as the best stump
Return the best stump

Let's now build this function. Enter the code from the following listing into adaboost.py and save the file.

Listing 7.1 Decision stump–generating functions

```
def stumpClassify(dataMatrix,dimen,threshVal,threshIneq):
    retArray = ones((shape(dataMatrix)[0],1))
    if threshIneq == 'lt':
```

```
        retArray[dataMatrix[:,dimen] <= threshVal] = -1.0
    else:
        retArray[dataMatrix[:,dimen] > threshVal] = -1.0
    return retArray

def buildStump(dataArr,classLabels,D):
    dataMatrix = mat(dataArr); labelMat = mat(classLabels).T
    m,n = shape(dataMatrix)
    numSteps = 10.0; bestStump = {}; bestClasEst = mat(zeros((m,1)))
    minError = inf
    for i in range(n):
        rangeMin = dataMatrix[:,i].min(); rangeMax = dataMatrix[:,i].max();
        stepSize = (rangeMax-rangeMin)/numSteps
        for j in range(-1,int(numSteps)+1):
            for inequal in ['lt', 'gt']:
                threshVal = (rangeMin + float(j) * stepSize)
                predictedVals = \
                        stumpClassify(dataMatrix,i,threshVal,inequal)
                errArr = mat(ones((m,1)))
                errArr[predictedVals == labelMat] = 0
                weightedError = D.T*errArr
                #print "split: dim %d, thresh %.2f, thresh ineqal: \
                       %s, the weighted error is %.3f" %\
                       (i, threshVal, inequal, weightedError)
                if weightedError < minError:
                    minError = weightedError
                    bestClasEst = predictedVals.copy()
                    bestStump['dim'] = i
                    bestStump['thresh'] = threshVal
                    bestStump['ineq'] = inequal
    return bestStump,minError,bestClasEst
```

Calculate weighted error (1)

The code in listing 7.1 contains two functions. The first function, `stumpClassify()`, performs a threshold comparison to classify data. Everything on one side of the threshold is thrown into class -1, and everything on the other side is thrown into class +1. This is done using array filtering, by first setting the return array to all 1s and then setting values that don't meet the inequality to -1. You can make this comparison on any feature in the dataset, and you can also switch the inequality from greater than to less than.

The next function, `buildStump()`, will iterate over all of the possible inputs to `stumpClassify()` and find the best decision stump for our dataset. Best here will be with respect to the data weight vector `D`. You'll see how this is done in a bit. The function starts out by making sure the input data is in the proper format for matrix math. Then, it creates an empty dictionary called `bestStump`, which you'll use to store the classifier information corresponding to the best choice of a decision stump given this weight vector D. The variable `numSteps` will be used to iterate over the possible values of the features. You also initialize the variable `minError` to positive infinity; this variable is used in finding the minimum possible error later.

The main portion of the code is three nested `for` loops. The first one goes over all the features in our dataset. You're considering numeric values, and you calculate the minimum and maximum to see how large your step size should be. Then, the next

`for` loop loops over these values. It might make sense to set the threshold outside the extremes of your range, so there are two extra steps outside the range. The last `for` loop toggles your inequality between greater than and less than.

Inside the nested three `for` loops, you call `stumpClassify()` with the dataset and your three loop variables. `stumpClassify()` returns its class prediction based on these loop variables. You next create the column vector `errArr`, which contains a 1 for any value in `predictedVals` that isn't equal to the actual class in `labelMat`. You multiply these errors by the weights in `D` and sum the results to give you a single number: `weightedError`. (1) This is the line where AdaBoost interacts with the classifier. You're evaluating your classifier based on the weights `D`, not on another error measure. If you want to use another classifier, you'd need to include this calculation to define the best classifier for `D`.

You next print out all the values. This line can be commented out later, but it's helpful in understanding how this function works. Last, you compare the error to your known minimum error, and if it's below it, you save this decision stump in your dictionary `bestStump`. The dictionary, the error, and the class estimates are all returned to the AdaBoost algorithm.

To see this in action, enter the following in the Python shell:

```
>>> D = mat(ones((5,1))/5)
>>> adaboost.buildStump(datMat,classLabels,D)
split: dim 0, thresh 0.90, thresh ineqal: lt, the weighted error is 0.400
split: dim 0, thresh 0.90, thresh ineqal: gt, the weighted error is 0.600
split: dim 0, thresh 1.00, thresh ineqal: lt, the weighted error is 0.400
split: dim 0, thresh 1.00, thresh ineqal: gt, the weighted error is 0.600
                                    .
                                    .
split: dim 1, thresh 2.10, thresh ineqal: lt, the weighted error is 0.600
split: dim 1, thresh 2.10, thresh ineqal: gt, the weighted error is 0.400
({'dim': 0, 'ineq': 'lt', 'thresh': 1.3}, matrix([[ 0.2]]), array([[-1.],
       [ 1.],
       [-1.],
       [-1.],
       [ 1.]]))
```

As `buildStump` iterates over all of the possible values, you can see the output, and finally you can see the dictionary returned. Does this dictionary correspond to the lowest possible weighted error? Are there other settings that have this same error?

The decision stump generator that you made is a simplified version of a decision tree. It's what you'd call the weak learner, which means a weak classification algorithm. Now that you've built the decision stump–generating code, we're ready to move on to the full AdaBoost algorithm. In the next section, we'll create the AdaBoost code to use multiple weak learners.

7.4 Implementing the full AdaBoost algorithm

In the last section, we built a classifier that could make decisions based on weighted input values. We now have all we need to implement the full AdaBoost algorithm.

We'll implement the algorithm outlined in section 7.2 with the decision stump built in section 7.3.

Pseudo-code for this will look like this:

For each iteration:
Find the best stump using buildStump()
Add the best stump to the stump array
Calculate alpha
Calculate the new weight vector – D
Update the aggregate class estimate
If the error rate ==0.0 : break out of the for loop

To put this function into Python, open adaboost.py and add the code from the following listing.

Listing 7.2 AdaBoost training with decision stumps

```
def adaBoostTrainDS(dataArr,classLabels,numIt=40):
    weakClassArr = []
    m = shape(dataArr)[0]
    D = mat(ones((m,1))/m)
    aggClassEst = mat(zeros((m,1)))
    for i in range(numIt):
        bestStump,error,classEst = buildStump(dataArr,classLabels,D)
        print "D:",D.T
        alpha = float(0.5*log((1.0-error)/max(error,1e-16)))
        bestStump['alpha'] = alpha
        weakClassArr.append(bestStump)
        print "classEst: ",classEst.T
        expon = multiply(-1*alpha*mat(classLabels).T,classEst)
        D = multiply(D,exp(expon))
        D = D/D.sum()
        aggClassEst += alpha*classEst
        print "aggClassEst: ",aggClassEst.T
        aggErrors = multiply(sign(aggClassEst) !=
                    mat(classLabels).T,ones((m,1)))
        errorRate = aggErrors.sum()/m
        print "total error: ",errorRate,"\n"
        if errorRate == 0.0: break
    return weakClassArr
```

1 Calculate D for next iteration

2 Aggregate error calculation

```
>>> classifierArray = adaboost.adaBoostTrainDS(datMat,classLabels,9)
D: [[ 0.2  0.2  0.2  0.2  0.2]]
classEst:  [[-1.  1. -1. -1.  1.]]
aggClassEst:  [[-0.69314718  0.69314718 -0.69314718 -0.69314718
               0.69314718]]
total error:  0.2

D: [[ 0.5    0.125  0.125  0.125  0.125]]
classEst:  [[ 1.  1. -1. -1. -1.]]
aggClassEst:  [[ 0.27980789  1.66610226 -1.66610226 -1.66610226
              -0.27980789]]
total error:  0.2
```

```
D: [[ 0.28571429  0.07142857  0.07142857  0.07142857  0.5       ]]
classEst:  [[ 1.  1.  1.  1.  1.]]
aggClassEst:  [[ 1.17568763  2.56198199 -0.77022252 -0.77022252
              0.61607184]]
total error:  0.0
```

The AdaBoost algorithm takes the input dataset, the class labels, and one parameter, `numIt`, which is the number of iterations. This is the only parameter you specify for the whole AdaBoost algorithm.

You set the number of iterations to 9. But the algorithm reached a total error of 0 after the third iteration and quit, so you didn't get to see all nine iterations. Intermediate output from each of the iterations comes from the `print` statements. You'll comment these out later, but for now let's look at the output to see what's going on under the hood of the AdaBoost algorithm.

The `DS` at the end of the function names stands for decision stump. Decision stumps are the most popular weak learner in AdaBoost. They aren't the only one you can use. This function is built for decision stumps, but you could easily modify it for other base classifiers. Any classifier will work. You could use any of the algorithms we explored in the first part of this book. The algorithm will output an array of decision stumps, so you first create a new Python list to store these. You next get `m`, the number of data points in your dataset, and create a column vector, `D`.

The vector `D` is important. It holds the weight of each piece of data. Initially, you'll set all of these values equal. On subsequent iterations, the AdaBoost algorithm will increase the weight of the misclassified pieces of data and decrease the weight of the properly classified data. `D` is a probability distribution, so the sum of all the elements in `D` must be 1.0. To meet this requirement, you initialize every element to 1/m. You also create another column vector, `aggClassEst`, which gives you the aggregate estimate of the class for every data point.

The heart of the AdaBoost algorithm takes place in the `for` loop, which is executed `numIt` times or until the training error becomes 0. The first thing that is done in this loop is to build a decision stump with the `buildStump()` function described earlier. This function takes `D`, the weights vector, and returns the stump with the lowest error using `D`. The lowest error value is also returned as well as a vector with the estimated classes for this iteration `D`.

Next, alpha is calculated. This will tell the total classifier how much to weight the output from this stump. The statement `max(error,1e-16)` is there to make sure you don't have a divide-by-zero error in the case where there's no error. The alpha value is added to the `bestStump` dictionary, and the dictionary is appended to the list. This dictionary will contain all you need for classification.

The next three lines ❶ are used to calculate new weights `D` for the next iteration. In the case that you have 0 training error, you want to exit the `for` loop early. This is calculated ❷ by keeping a running sum of the estimated class in `aggClassEst`. This value is a floating point number, and to get the binary class you use the `sign()` function. If the total error is 0, you quit the `for` loop with the `break` statement.

Let's look at the intermediate output. Remember, our class labels were [1.0, 1.0, -1.0, -1.0, 1.0]. In the first iteration, all the D values were equal; then only one value, the first data point, was misclassified. So, in the next iteration, the D vector puts 0.5 weight on the first data point because it was misclassified previously. You can see the total class by looking at the sign of aggClassEst. After the second iteration, you can see that the first data point is correctly classified, but the last data point is now wrong. The D value now becomes 0.5 for the last element, and the other values in the D vector are much smaller. Finally, in the third iteration the sign of all the values in aggClassEst matches your class labels and the training error becomes 0, so you can quit.

To see classifierArray type in

```
>>> classifierArray
[{'dim': 0, 'ineq': 'lt', 'thresh': 1.3, 'alpha': 0.6931471805599453},
    {'dim': 1, 'ineq': 'lt', 'thresh': 1.0, 'alpha': 0.9729550745276565},
    {'dim': 0,'ineq': 'lt', 'thresh': 0.90000000000000002, 'alpha':
      0.8958797346140273}]
```

This array contains three dictionaries, which contain all of the information you'll need for classification. You've now built a classifier, and the classifier will reduce the training error to 0 if you wish. How does the test error look? In order to see the test error, you need to write some code for classification. The next section will discuss classification.

7.5 Test: classifying with AdaBoost

Once you have your array of weak classifiers and alphas for each classifier, testing is easy. You've already written most of the code in adaBoostTrainDS() in listing 7.2. All you need to do is take the train of weak classifiers from your training function and apply these to an instance. The result of each weak classifier is weighted by its alpha. The weighted results from all of these weak classifiers are added together, and you take the sign of the final weighted sum to get your final answer. The code to do this is given in the next listing. Add the following code to adaboost.py, and then you can use it to classify data with the classifier array from adaboostTrainDS().

Listing 7.3 AdaBoost classification function

```
def adaClassify(datToClass,classifierArr):
    dataMatrix = mat(datToClass)
    m = shape(dataMatrix)[0]
    aggClassEst = mat(zeros((m,1)))
    for i in range(len(classifierArr)):
        classEst = stumpClassify(dataMatrix,classifierArr[i]['dim'],\
                                 classifierArr[i]['thresh'],\
                                 classifierArr[i]['ineq'])
        aggClassEst += classifierArr[i]['alpha']*classEst
        print aggClassEst
    return sign(aggClassEst)
```

The function in listing 7.3 is adaClassify(), which, as you may have guessed, classifies with a train of weak classifiers. The inputs are datToClass, which can be multiple data

instances or just one to be classified, and `classifierArr`, which is an array of weak classifiers. The function `adaClassify()` first converts `datToClass` to a NumPy matrix and gets `m`, the number of instances in `datToClass`. Then it creates `aggClassEst`, which is a column vector of all 0s. This is the same as `adaBoostTrainDS()`.

Next, you look over all of the weak classifiers in `classifierArr`, and for each of them you get a class estimate from `stumpClassify()`. You saw `stumpClassify()` earlier when you were building stumps. At that time you iterated over all of the possible stump values and chose the stump with the lowest weighted error. Here you're simply applying the stump. This class estimate is multiplied by the alpha value for each stump and added to the total: `aggClassEst`. I've added a `print` statement so you can see how `aggClassEst` evolves with each iteration. Finally, you return the sign of `aggClassEst`, which gives you a +1 if its argument is greater than 0 and a -1 if the argument is less than 0.

Let's see this in action. After you've added the code from listing 7.3, type the following at the Python shell:

```
>>> reload(adaboost)
<module 'adaboost' from 'adaboost.py'>
```

If you don't have the classifier array, you can enter the following:

```
>>> datArr,labelArr=adaboost.loadSimpData()
>>> classifierArr = adaboost.adaBoostTrainDS(datArr,labelArr,30)
```

Now you can classify by typing this:

```
>>> adaboost.adaClassify([0, 0],classifierArr)
[[-0.69314718]]
[[-1.66610226]]
[[-2.56198199]]
matrix([[-1.]])
```

You can see that the answer for point [0,0] gets stronger with each iteration. You can also do this with multiple points:

```
>>> adaboost.adaClassify([[5, 5],[0,0]],classifierArr)
[[ 0.69314718]
            .
            .
[-2.56198199]]
matrix([[ 1.],
        [-1.]])
```

The answer for both points gets stronger with each iteration. In the next section we're going to apply this to a much bigger and harder dataset from the real world.

7.6 Example: AdaBoost on a difficult dataset

In this section we're going to try AdaBoost on the dataset from chapter 4. It's the horse colic dataset. In chapter 4 we tried to predict whether a horse with colic would live or die by using logistic regression. Let's see if we can do better with AdaBoost and the decision stumps.

Example: using AdaBoost on a difficult dataset

1. Collect: Text file provided.
2. Prepare: We need to make sure the class labels are +1 and -1, not 1 and 0.
3. Analyze: Manually inspect the data.
4. Train: We'll train a series of classifiers on the data using the `adaBoostTrainDS()` function.
5. Test: We have two datasets. With no randomization, we can have an apples-to-apples comparison of the AdaBoost results versus the logistic regression results.
6. Use: We'll look at the error rates in this example. But you could create a website that asks a trainer for the horse's symptoms and then predicts whether the horse will live or die.

Before you use the functions from the previous code listings in this chapter, you need to have a way to load data from a file. The familiar `loadDataSet()` is given in the following listing.

Listing 7.4 Adaptive load data function

```
def loadDataSet(fileName):
    numFeat = len(open(fileName).readline().split('\t'))
    dataMat = []; labelMat = []
    fr = open(fileName)
    for line in fr.readlines():
        lineArr =[]
        curLine = line.strip().split('\t')
        for i in range(numFeat-1):
            lineArr.append(float(curLine[i]))
        dataMat.append(lineArr)
        labelMat.append(float(curLine[-1]))
    return dataMat,labelMat
```

The function in listing 7.4 is `loadDataSet()`, which you've seen many times before. It's slightly improved this time because you don't have to specify the number of features in each file. It automatically detects this. The function also assumes that the last feature is the class label.

To use it, enter the following in your Python shell after you've saved adaboost.py:

```
>>> datArr,labelArr = adaboost.loadDataSet('horseColicTraining2.txt')
>>> classifierArray = adaboost.adaBoostTrainDS(datArr,labelArr,10)
total error: 0.284280936455
total error: 0.284280936455
                    .
                    .
total error: 0.230769230769
>>> testArr,testLabelArr = adaboost.loadDataSet('horseColicTest2.txt')
>>> prediction10 = adaboost.adaClassify(testArr,classifierArray)
To get the number of misclassified examples type in:
```

```
>>> errArr=mat(ones((67,1)))
>>> errArr[prediction10!=mat(testLabelArr).T].sum()
16.0
```

To get the error rate, divide this number by 67.

I've repeated the process for a number of weak classifiers between 1 and 10,000. The results are listed in table 7.1. The test error is excellent for this dataset. If you remember, in chapter 5 we looked at this dataset with logistic regression. At that time, the average error rate was 0.35. With AdaBoost we never have an error rate that high, and with only 50 weak learners we achieved high performance.

If you look at the Test Error column in table 7.1, you'll see that the test error reaches a minimum and then starts to increase. This sort of behavior is known as *overfitting*. It has been claimed in literature that for well-behaved datasets the test error for AdaBoost reaches a plateau and won't increase with more classifiers. Perhaps this dataset isn't "well behaved." It did start off with 30% missing values, and the assumptions made for the missing values were valid for logistic regression but they may not work for a decision tree. If you went back to our dataset and replaced all the 0s with other values—perhaps averages for a given class—would you have better performance?

Number of Classifiers	Training Error	Test Error
1	0.28	0.27
10	0.23	0.24
50	0.19	0.21
100	0.19	0.22
500	0.16	0.25
1000	0.14	0.31
10000	0.11	0.33

Table 7.1 AdaBoost test and training errors for a range of weak classifiers. This dataset is particularly difficult. Usually AdaBoost reaches a test error plateau, and the error doesn't increase with more classifiers.

AdaBoost and support vector machines are considered by many to be the most powerful algorithms in supervised learning. You can draw a number of similarities between the two. You can think of the weak learner in AdaBoost as a kernel in support vector machines. You can also write the AdaBoost algorithm in terms of maximizing a minimum margin. The way these margins are calculated is different and can lead to different results, especially with higher dimensions.

In the next section we're going to leave AdaBoost and talk about a problem common to all classifiers.

7.7 Classification imbalance

Before we leave the subject of classification, there's a topic that needs to be addressed. In all six chapters on classification, we assumed that the cost of classifying things is equal. In chapter 5, for example, we built a system to detect whether a horse with

stomach pain would end up living or dying. We built the classifier but didn't talk about what happens after classification. Let's say someone brings a horse to us and asks us to predict whether the horse will live or die. We say die, and rather than delay the inevitable, making the animal suffer and incurring veterinary bills, they have it euthanized. Perhaps our prediction was wrong, and the horse would have lived. Our classifier is only 80% accurate, after all. If we predicted this incorrectly, then an expensive animal would have been destroyed, not to mention that a human was emotionally attached to the animal.

How about spam detection? Is it OK to let a few spam messages arrive in your inbox as long as real email never gets put into the spam folder? What about cancer detection? Is it better to tell someone to go for a second opinion as long as you never let someone with a disease go untreated?

The examples for this abound, and it's safe to say that in most cases the costs aren't equal. In this section, we'll examine a different method for measuring performance of our classifiers and some graphical techniques for visualizing the performance of different classifiers with respect to this problem. Then we'll look at two methods of altering our classification algorithms to take into account the costs of making different decisions.

7.7.1 Alternative performance metrics: precision, recall, and ROC

So far in this book we've measured the success of the classification tasks by the error rate. The error rate was the number of misclassified instances divided by the total number of instances tested. Measuring errors this way hides how instances were misclassified. There's a tool commonly used in machine learning that gives you a better view of classification errors called a *confusion matrix*. A confusion matrix for a three-class problem involving predicting animals found around the house is shown in table 7.2.

		Dog	Cat	Rat
	Dog	24	2	5
	Cat	2	27	0
	Rat	4	2	30

Table 7.2 Confusion matrix for a three-class problem

With a confusion matrix you get a better understanding of the classification errors. If the off-diagonal elements are all zero, then you have a perfect classifier.

Let's consider another confusion matrix, this time for the simple two-class problem. The confusion matrix is given in table 7.3. In the two-class problem, if you correctly classify something as positive, it's called a True Positive, and it's called a True Negative when you properly classify the negative class. The other two possible cases (False Negative and False Positive) are labeled in table 7.3.

		+1	-1
	+1	True Positive (TP)	False Negative (FN)
	-1	False Positive (FP)	True Negative (TN)

Table 7.3 Confusion matrix for a two-class problem, with different outcomes labeled

With these definitions we can define some new metrics that are more useful than error rate when detection of one class is more important than another class. The first term is *Precision = TP/(TP+FP)*. Precision tells us the fraction of records that were positive from the group that the classifier predicted to be positive. The second term we care about is *Recall = TP/(TP+FN)*. Recall measures the fraction of positive examples the classifier got right. Classifiers with a large recall don't have many positive examples classified incorrectly.

You can easily construct a classifier that achieves a high measure of recall or precision but not both. If you predicted everything to be in the positive class, you'd have perfect recall but poor precision. Creating a classifier that maximizes both precision and recall is a challenge.

Another tool used for measuring classification imbalance is the *ROC curve*. ROC stands for receiver operating characteristic, and it was first used by electrical engineers building radar systems during World War II. An example ROC curve is shown in figure 7.3.

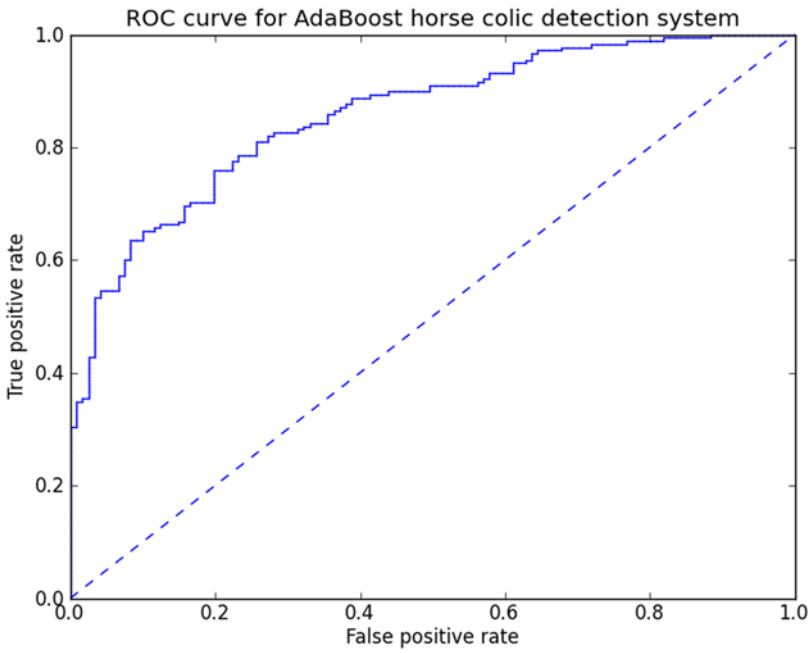

Figure 7.3 ROC for AdaBoost horse colic detection system using 10 decision stumps

The ROC curve in figure 7.3 has two lines, a solid one and a dashed one. The x-axis in figure 7.3 is the number of false positives, and the y-axis is the number of true positives. The ROC curve shows how the two rates change as the threshold changes. The leftmost point corresponds to classifying everything as the negative class, and the rightmost point corresponds to classifying everything in the positive class. The dashed line is the curve you'd get by randomly guessing.

ROC curves can be used to compare classifiers and make cost-versus-benefit decisions. Different classifiers may perform better for different threshold values, and it may make sense to combine them in some way. You wouldn't get this type of insight from simply looking at the error rate of a classifier.

Ideally, the best classifier would be in upper left as much as possible. This would mean that you had a high true positive rate for a low false positive rate. For example, in spam classification this would mean you catch all the spam and don't allow any legitimate emails to get put in the spam folder.

One metric to compare different ROC curves is the area under the curve (AUC). The AUC gives an average value of the classifier's performance and doesn't substitute for looking at the curve. A perfect classifier would have an AUC of 1.0, and random guessing will give you a 0.5.

In order to plot the ROC you need the classifier to give you a numeric score of how positive or negative each instance is. Most classifiers give this to you, but it's usually cleaned up before the final discrete class is delivered. Naïve Bayes gives you a probability. The input to the sigmoid in logistic regression is a numeric value. AdaBoost and SVMs both compute a numeric value that's input to the `sign()` function. All of these values can be used to rank how strong the prediction of a given classifier is. To build the ROC curve, you first sort the instances by their prediction strength. You start with the lowest ranked instance and predict everything below this to be in the negative class and everything above this to be the positive class. This corresponds to the point 1.0,1.0. You move to the next item in the list, and if that is the positive class, you move the true positive rate, but if that instance is in the negative class, you change the true negative rate.

This procedure probably sounds confusing but it will become clear when you look at the code in the following listing. Open adaboost.py and add the following code.

Listing 7.5 ROC plotting and AUC calculating function

```
def plotROC(predStrengths, classLabels):
    import matplotlib.pyplot as plt
    cur = (1.0,1.0)
    ySum = 0.0
    numPosClas = sum(array(classLabels)==1.0)
    yStep = 1/float(numPosClas)
    xStep = 1/float(len(classLabels)-numPosClas)
    sortedIndicies = predStrengths.argsort()
    fig = plt.figure()
    fig.clf()
    ax = plt.subplot(111)
    for index in sortedIndicies.tolist()[0]:
```

1 Get sorted index

```
        if classLabels[index] == 1.0:
            delX = 0; delY = yStep;
        else:
            delX = xStep; delY = 0;
            ySum += cur[1]
        ax.plot([cur[0],cur[0]-delX],[cur[1],cur[1]-delY], c='b')
        cur = (cur[0]-delX,cur[1]-delY)
    ax.plot([0,1],[0,1],'b--')
    plt.xlabel('False Positive Rate'); plt.ylabel('True Positive Rate')
    plt.title('ROC curve for AdaBoost Horse Colic Detection System')
    ax.axis([0,1,0,1])
    plt.show()
    print "the Area Under the Curve is: ",ySum*xStep
```

The code in listing 7.5 takes two inputs; the first is a NumPy array or matrix in a row vector form. This is the strength of the classifier's predictions. Our classifier and our training functions generate this before they apply it to the `sign()` function. You'll see this function in action in a bit, but let's discuss the code first. The second input is the `classLabels` you used earlier. You first input `pyplot` and then create a tuple of floats and initialize it to 1.0,1.0. This holds your cursor for plotting. The variable `ySum` is used for calculating the AUC. You next calculate the number of positive instances you have, by using array filtering, and set this value to `numPosClas`. This will give you the number of steps you're going to take in the y direction. You're going to plot in the range of 0.0 to 1.0 on both the x- and y-axes, so to get the y step size you take 1.0/numPosClas. You can similarly get the x step size.

You next get the sorted index (1), but it's from smallest to largest, so you start at the point 1.0,1.0 and draw to 0,0. The next three lines set up the plot, and then you loop over all the sorted values. The values were sorted in a NumPy array or matrix, but Python needs a list to iterate over, so you call the `tolist()` method. As you're going through the list, you take a step down in the y direction every time you get a class of 1.0, which decreases the true positive rate. Similarly, you take a step backward in the x direction (false positive rate) for every other class. This code is set up to focus only on the 1s so you can use either the 1,0 or +1,-1 class labels.

To compute the AUC, you need to add up a bunch of small rectangles. The width of each of these rectangles will be `xStep`, so you can add the heights of all the rectangles and multiply the sum of the heights by `xStep` once to get the total area. The height sum (`ySum`) increases every time you move in the x direction. Once you've decided whether you're going to move in the x or y direction, you draw a small, straight-line segment from the current point to the new point. The current point, `cur`, is then updated. Finally, you make the plot look nice and display it by printing the AUC to the terminal.

To see this in action, you'll need to alter the last line of `adaboostTrainDS()` to

```
return weakClassArr,aggClassEst
```

in order to get the `aggClassEst` out. Next, type in the following at your Python shell:

```
>>> reload(adaboost)
<module 'adaboost' from 'adaboost.pyc'>
>>> datArr,labelArr = adaboost.loadDataSet('horseColicTraining2.txt')
>>> classifierArray,aggClassEst =
     adaboost.adaBoostTrainDS(datArr,labelArr,10)
>>> adaboost.plotROC(aggClassEst.T,labelArr)
the Area Under the Curve is: 0.858296963506
```

You should also see an ROC plot identical to figure 7.3. This is the performance of our AdaBoost classifier with 10 weak learners. Remember, we had the best performance with 40 weak learners? How does the ROC curve compare? Is the AUC better?

7.7.2 Manipulating the classifier's decision with a cost function

Besides tuning the thresholds of our classifier, there are other approaches you can take to aid with uneven classification costs. One such method is known as *cost-sensitive learning*. Consider the cost matrix in table 7.4. The top table encodes the costs of classification as we've been using it up to this point. You calculate the total cost with this cost matrix by `TP*0+FN*1+FP*1+TN*0`. Now consider the cost matrix in the bottom frame of table 7.4. The total cost using this cost matrix will be `TP*-5+FN*1+FP*50+TN*0`. Using the second cost matrix, the two types of incorrect classification will have different costs. Similarly, the two types of correct classification will have different benefits. If you know these costs when you're building the classifier, you can select a classifier with the minimum cost.

		+1	-1
	+1	0	1
	-1	1	0

		+1	-1
	+1	-5	1
	-1	50	0

Table 7.4 Cost matrix for a two-class problem

There are many ways to include the cost information in classification algorithms. In AdaBoost, you can adjust the error weight vector `D` based on the cost function. In naïve Bayes, you could predict the class with the lowest expected cost instead of the class with the highest probability. In SVMs, you can use different `C` parameters in the cost function for the different classes. This gives more weight to the smaller class, which when training the classifier will allow fewer errors in the smaller class.

7.7.3 Data sampling for dealing with classification imbalance

Another way to tune classifiers is to alter the data used to train the classifier to deal with imbalanced classification tasks. This is done by either undersampling or oversampling the data. *Oversample* means to duplicate examples, whereas *undersample* means to delete examples. Either way, you're altering the data from its original form. The sampling can be done either randomly or in a predetermined fashion.

Usually there's a rare case that you're trying to identify, such as credit card fraud. As mentioned previously, the rare case is the positive class. You want to preserve as much information as possible about the rare case, so you should keep all of the examples from the positive class and undersample or discard examples from the negative class. One drawback of this approach is deciding which negative examples to toss out. The examples you choose to toss out could carry valuable information that isn't contained in the remaining examples.

One solution for this is to pick samples to discard that aren't near the decision boundary. For example, say you had a dataset with 50 fraudulent credit card transactions and 5,000 legitimate transactions. If you wanted to undersample the legitimate transactions to make the dataset equally balanced, you'd need to throw out 4,950 examples, which may also contain valuable information. This may seem extreme, so an alternative is to use a hybrid approach of undersampling the negative class and oversampling the positive class.

To oversample the positive class, you could replicate the existing examples or add new points similar to the existing points. One approach is to add a data point interpolated between existing data points. This process can lead to overfitting.

7.8 Summary

Ensemble methods are a way of combining the predictions of multiple classifiers to get a better answer than simply using one classifier. There are ensemble methods that use different types of classifiers, but we chose to look at methods using only one type of classifier.

Combining multiple classifiers exploits the shortcomings of single classifiers, such as overfitting. Combining multiple classifiers can help, as long as the classifiers are significantly different from each other. This difference can be in the algorithm or in the data applied to that algorithm.

The two types of ensemble methods we discussed are bagging and boosting. In bagging, datasets the same size as the original dataset are built by randomly sampling examples for the dataset with replacement. Boosting takes the idea of bagging a step further by applying a different classifier sequentially to a dataset. An additional ensemble method that has shown to be successful is random forests. Random forests aren't as popular as AdaBoost, so they aren't discussed in this book.

We discussed the most popular variant of boosting, called AdaBoost. AdaBoost uses a weak learner as the base classifier with the input data weighted by a weight vector. In the first iteration the data is equally weighted. But in subsequent iterations the

data is weighted more strongly if it was incorrectly classified previously. This adapting to the errors is the strength of AdaBoost.

We built functions to create a classifier using AdaBoost and the weak learner, decision stumps. The AdaBoost functions can be applied to any classifier, as long as the classifier can deal with weighted data. The AdaBoost algorithm is powerful, and it quickly handled datasets that were difficult using other classifiers.

The classification imbalance problem is training a classifier with data that doesn't have an equal number of positive and negative examples. The problem also exists when the costs for misclassification are different from positive and negative examples. We looked at ROC curves as a way to evaluate different classifiers. We introduced precision and recall as metrics to measure the performance classifiers when classification of one class is more important than classification of the other class.

We introduced oversampling and undersampling as ways to adjust the positive and negative examples in a dataset. Another, perhaps better, technique was introduced for dealing with classifiers with unbalanced objectives. This method takes the costs of misclassification into account when training a classifier.

We've introduced a number of powerful classification techniques so far in this book. This is the last chapter on classification, and we'll move on to regression next to complete our study of supervised learning algorithms. Regression is much like classification, but instead of predicting a nominal class, we'll be predicting a continuous value.

PYTHON/MACHINE LEARNING

Machine Learning IN ACTION

Peter Harrington

A machine is said to learn when its performance improves with experience. Learning requires algorithms and programs that capture data and ferret out the interesting or useful patterns. Once the specialized domain of analysts and mathematicians, machine learning is becoming a skill needed by many.

Machine Learning in Action is a clearly written tutorial for developers. It avoids academic language and takes you straight to the techniques you'll use in your day-to-day work. Many (Python) examples present the core algorithms of statistical data processing, data analysis, and data visualization in code you can reuse. You'll understand the concepts and how they fit in with tactical tasks like classification, forecasting, recommendations, and higher-level features like summarization and simplification.

What's Inside

- A no-nonsense introduction
- Examples showing common ML tasks
- Everyday data analysis
- Implementing classic algorithms like Apriori and Adaboost

Readers need no prior experience with machine learning or statistical processing. Familiarity with Python is helpful.

Peter Harrington is a professional developer and data scientist. He holds five US patents and his work has been published in numerous academic journals.

For access to the book's forum and a free eBook for owners of this book, go to manning.com/MachineLearninginAction

"An approachable and useful book."
—Alexandre Alves
Oracle Corporation

"Smart, engaging applications of core concepts."
—Patrick Toohey
Mettler-Toledo Hi-Speed

"Great examples! Teach a computer to learn anything!"
—John Griffin, Coauthor of *Hibernate Search in Action*

"An approachable taxonomy, skillfully created from the diversity of ML algorithms."
—Stephen McKamey
Isomer Innovations

ISBN 13: 978-1-61729-018-3
ISBN 10: 1-61729-018-1
54499
9 781617 290183

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]