Homework Assignment 6

36-350, Data Mining

SOLUTIONS

1. *Minimum-Error Classification*

   (a) *For each fixed x, show that the probability of mis-classification, R, is $q + p - 2pq$.*

   ANSWER: Mis-classifying means either $Y = 1$ but we predict 0, or $Y = 0$ but we predict 1. Since Y and our predictions are independent (given X), the first error has probability $p(1 - q)$, and the second $(1 - p)q$. So $R = p(1 - q) + (1 - p)q = p + q - 2pq$.

   (b) *Plot this error rate as a function of q, in the interval $[0, 1]$ for $p = 0.1$, $p = 0.3$, $p = 0.5$, $p = 0.6$ and $p = 0.9$. Where are the minima?*

   ANSWER: I re-wrote $R = p + (1 - 2p)q$, and used `abline` (see next page):

```
plot(c(0,1),c(0,1),xlab="q",ylab="R",type="n")
abline(0.1,1-2*0.1); abline(0.3,1-2*0.3,lty=2)
abline(0.5,1-2*0.5,lty=3); abline(0.6,1-2*0.6,lty=4)
abline(0.9,1-2*0.9,lty=5)
```

   The minima are at $q = 0$ for $p = 0.1$ and $p = 0.3$, at $q = 1$ for $p = 0.6$ and $p = 0.9$, and everywhere or nowhere for $p = 0.5$ (because that's a flat line).

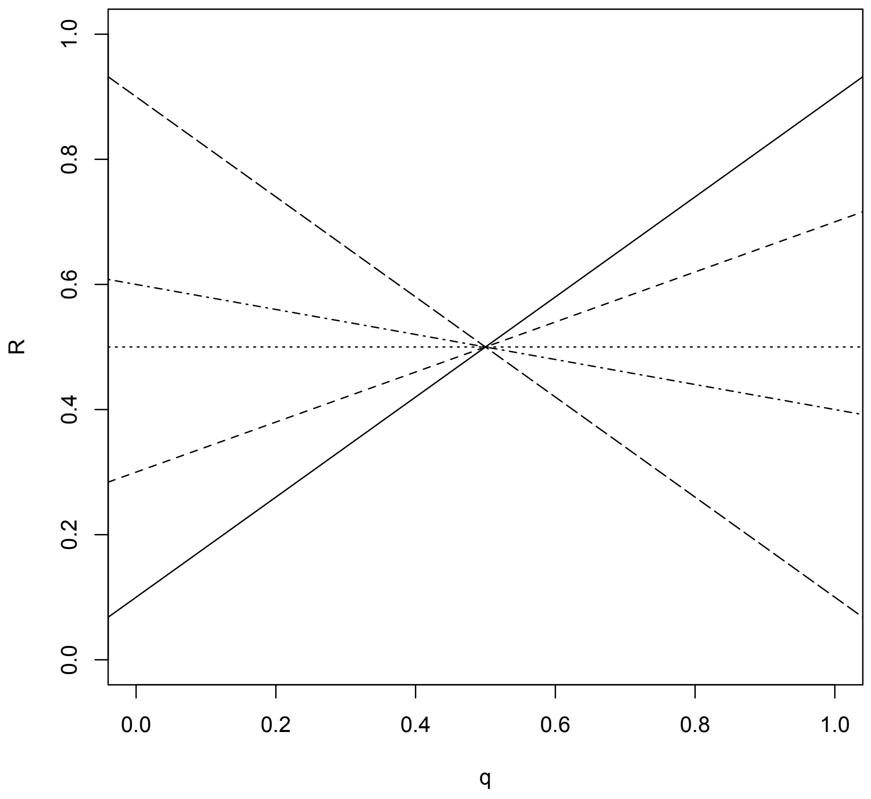
   (c) *Show that the derivative of R with respect to q is never zero, unless $p = 1/2$.*

   ANSWER:
$$\frac{\partial R}{\partial q} = \frac{\partial p}{\partial q} + \frac{\partial q}{\partial q} - \frac{\partial (2pq)}{\partial q} = 1 - 2p$$

   This is constant, independent of q; it is < 0 if $p > 1/2$, > 0 if $p < 1/2$, and it is $= 0$ if and only if $p = 1/2$.

   (d) *Show that R is minimized when $q = 1$ if $p > 0.5$, and when $q = 0$ if $p < 0.5$.*

   ANSWER: If $p \neq 0.5$, then $\partial R/\partial q \neq 0$ everywhere, and it always has the same sign. When a function's derivative always has the same sign in some region, its minimum must be at one boundary of the region. (So must its maximum.) If $p > 0.5$, the derivative is always negative, meaning that R can always be made smaller by increasing q, until



we reach the minimum at $q = 1$. Likewise, if $p < 0.5$, $\partial R/\partial q > 0$, so we minimize R by reducing q to its smallest possible value, $q = 0$. When $p = 1/2$, it does matter what we predict.

2. *Three Classifiers*

   The easiest way to load the data is

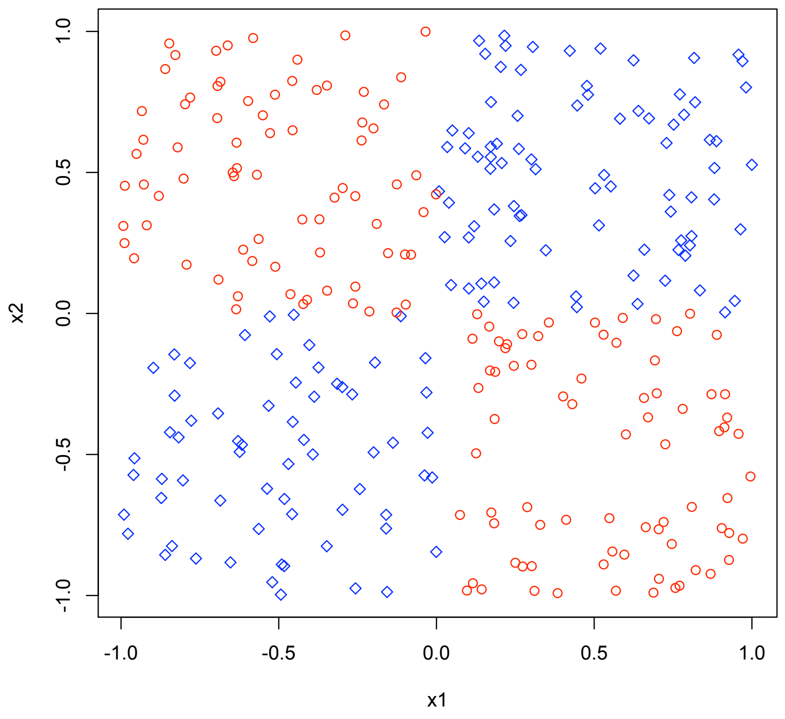
```
foobar = read.table("foobar",header=TRUE)
```

   (a) *Plot the data. Use different colors (via the `col` argument) or point-shapes (via the `pch` argument) for the two classes. If you use different colors, make sure they look distinct when you print them out!*

   Answer:

```
plot(foobar[,"x1"],foobar[,"x2"],
     col=ifelse(foobar[,"y"]=="foo","red","blue"),
     pch=ifelse(foobar[,"y"]=="foo",21,23),
     xlab="x1",ylab="x2")
```

This plots points with the label `foo` as red circles, and others as blue diamonds.



X_1 and X_2 were independently and uniformly distributed on $[-1, 1]$. Y was `foo` if one coordinate was negative and the other positive, otherwise, if both X_1 and X_2 had the same sign, Y was `bar`.

(b) *Divide the data set at random into two equal haves, one for training and one for testing. Include your code. Include a check that the two halves have the right size, and that they do not overlap.*

Answer: Here's one way. It uses two R conveniences: selecting multiple rows (or columns) by giving a vector of indices, and removing rows or columns by giving negated indices.

```
dim(foobar)
training.rows = sample(1:nrow(foobar),nrow(foobar)/2,replace=FALSE)
training.data = foobar[training.rows,]
testing.data = foobar[-training.rows,]
dim(training.data)
dim(testing.data)
intersect(rownames(training.data),rownames(testing.data))
```

`foobar` is a 300×3 array, so both the training and the testing sets should be 150×3 arrays each, and they are. The command `intersect` returns the (unique) common elements of two vectors — just like set intersections; run here it returns `character(0)`, meaning the empty set. (You should check that `training.data` and `testing.data` both inherit their row names from `foobar`. When I run this, for example, `rownames(training.data)` begins `"281" "238" "40"` , and so on for 147 more entries.)

(c) *Fit a prototype classifier to the training data and evaluate it on the test data. Report the error rate.*

ANSWER: You wrote a prototype classifier for HW #2. This modifies the prototype function in the solutions to that problem set so that it can calculate the prototypes once, and then classify multiple vectors.[1] It calls the `nearest.points` function from the first problem set. It also strips out the pre-processing for bags of words.

```
prototype.classifier <- function(newdata,examples.inputs,examples.labels) {
  class.prototypes = aggregate(examples.inputs,
                       list(class.labels=examples.labels),
                       mean)
  label.set = class.prototypes$class.labels
  matches = nearest.points(newdata,class.prototypes[,-1])$which
  label.predictions = label.set[matches]
  return(label.predictions)
}
```

And here's how to count the errors:

```
prototype.predictions = prototype.classifier(newdata=testing.data[,-1],
                          examples.inputs = training.data[,-1],
                          examples.labels = training.data[,"y"])
sum(prototype.predictions != testing.data[,"y"])/nrow(testing.data)
```

I get an error rate of 49%; your error rate will depend on the random training/testing split, but should be around 50%, which is what you'd get by tossing a coin.

Comment: The prototype method always draws linear boundaries between classes. With only two classes, this means it assumes they can be separated by a *single* straight line. This problem is a simple example of classification problems which cannot be solved by any linear classifier.

[1]You could use the unmodified version, but that's much slower. An even better approach would be to define a new type of object for prototype classifiers, and then write separate fitting and `predict` functions.

(d) *Do the same with a nearest-neighbor classifier.*

ANSWER: Again, writing a nearest-neighbor classifier was part of HW # 2. Here I modify the solution code to classify multiple vectors at once. Again, it calls `nearest.points` from the first problem set.

```
my.nn.multiple = function(newdata, examples.inputs, examples.labels) {
  matches = nearest.points(newdata,examples.inputs)$which
  label.predictions = examples.labels[matches]
  return(label.predictions)
}
```

Evaluating the error in the same way as for the prototype classifier, I get a rate of 5%.

(e) *Do the same with a classification tree. Include a picture of the tree, annotated with the actual splits.*

ANSWER: The tree can be fit with

```
library(tree)
my.tree = tree(y ~ x1 + x2, data=training.data)
```

The commands

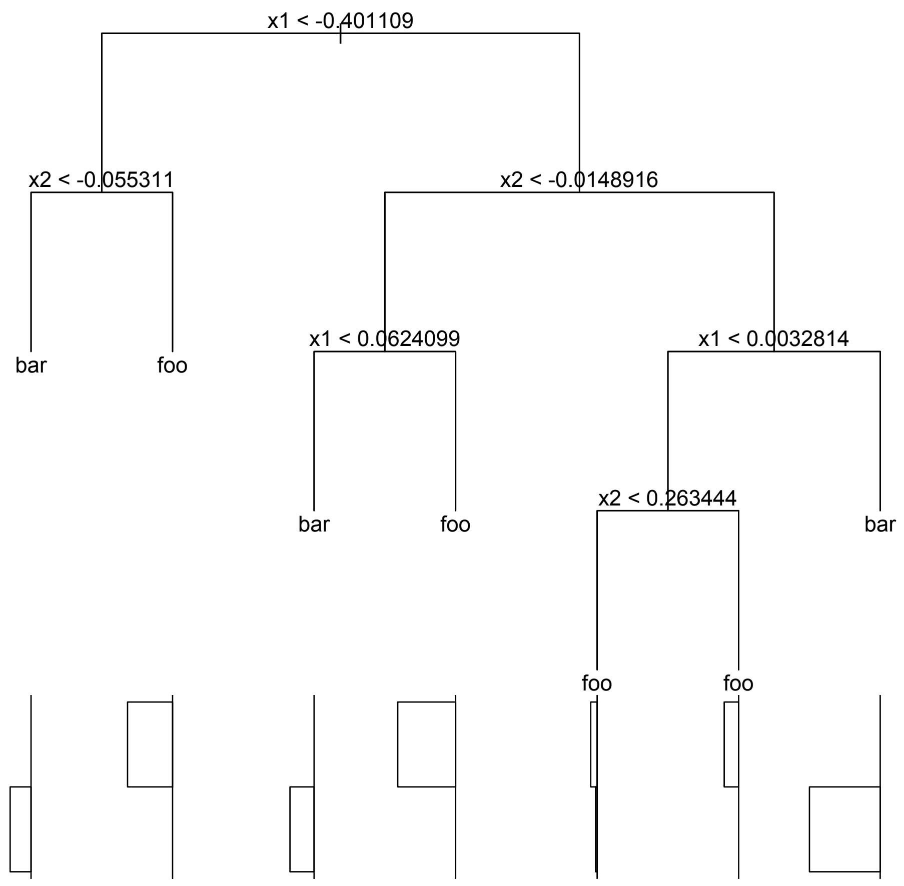
```
plot(my.tree)
text(my.tree)
```

draw and label the tree.

A fancier version is

```
tree.screens()
plot(my.tree)
text(my.tree)
tile.tree(my.tree,trainig.data[,"y"])
```

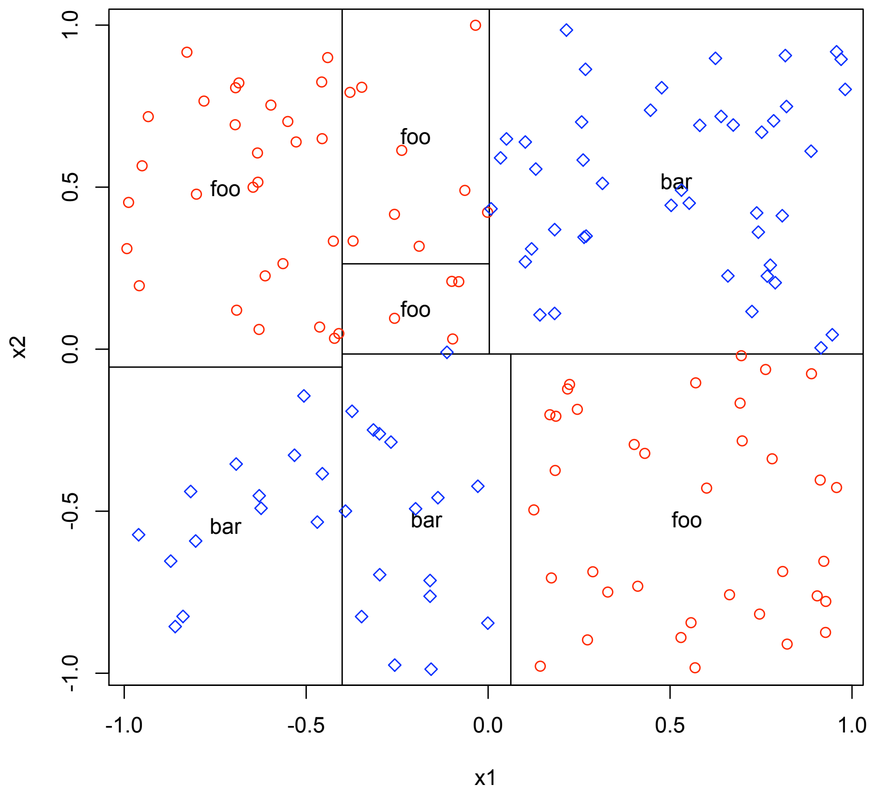
This plots the tree as above, but then adds a bar-chart underneath each leaf showing the distribution of the classes for that leaf.



One can also plot the actual partition:

```
partition.tree(my.tree)
points(training.data[,"x1"],training.data[,"x2"],
       pch=ifelse(training.data[,"y"]=="foo",21,23),
       col=ifelse(training.data[,"y"]=="foo","red","blue"))
```

The first command draws the boundaries and labels them; the second adds the training data (where again `foo`==red==circles).



We get predicted class labels for the testing data as

```
tree.predictions = predict(my.tree,newdata=testing.data,type="class")
sum(tree.predictions != testing.data[,"y"])/nrow(testing.data)
```

This gives me an error rate of 2.6%. (Notice that the boundaries in the plot aren't *quite* on the axes.)