knn-project

Jordan Hilton

February 23, 2019

We're going to use the nearest neighbors method to predict blood donations. Let's first load our data, then drop the index column and the result column for our working data:

```
data<-read.csv("projectdata.csv")
instances<-data[-c(1,4,6)] ##drop the index column and the result column for working purposes
```

Now we'll assign weights and scale the data columns by the assigned weights. To start, we just scaled each column by the maximum value.

```
weights<-c(1/74,1/50,1/12500,1/98) ## make a vector of independent variable weights, scaling each by the
names(weights)<-c("monthssincelast", "numberdonations", "monthssincefirst")  ##name them so we don't for
scaledinstances<-cbind(instances[,1]*weights[1],instances[,2]*weights[2],instances[,3]*weights[3])
## make a scaled version of our instances so that the distances are equivalent
```

We'll now divide our working data into two buckets, one for training which we'll label "data", and one for calculating nearest neighbors to assign weights.

```
practicebucket<-scaledinstances[501:576,]
databucket<-scaledinstances[1:500,]
```

Here we built a little function to help us calculate distances between two different points. We're going to use the Euclidean distance for two reasons; one, it's what the prebuilt nearest-neighbors package uses, and two, some of our data are in numeric form instead of factors, and Euclidean distance is more appropriate for those.

```
distance<-function(x,y){
  result<-dist(rbind(x,y), method="euclidean") ## find the manhattan distance between two instances
  return(as.numeric(result)) #the result is weirdly vectorized so we just grab the value out of it
}
```

Here's an example of the calculation of the distance between one of our practice points and one of our data points:

```
databucket[1,]
```

```
## [1] 0.02702703 1.00000000 0.00784000
```

```
practicebucket[1,]
```

```
## [1] 0.1891892 0.0800000 0.0020800
```

```
distance(databucket[1,], practicebucket[1,])
```

```
## [1] 0.9342001
```

Now we'll create a table where we calculate the distances between each of our practice points and each of the data points. Each column, iterated by j, is a practice point and each row, iterated by j, is a data point.

```
distancesbyrow<-data.frame() #length(data) rows iterated by i for data, length(practice) columns iterat

for(i in 1:length(databucket[,1])){
  for(j in 1:length(practicebucket[,1])){
    distancesbyrow[i,j]<-distance(databucket[i,],practicebucket[j,])  # calculate the distance between
  }
}

head(distancesbyrow[,1:6])
```

```
##          V1         V2        V3        V4         V5        V6
## 1 0.9342001 0.96039578 0.9539049 0.7058296 0.96039533 0.8555182
## 2 0.2611371 0.22654330 0.2753050 0.2244211 0.22654337 0.2139992
## 3 0.2974264 0.28291995 0.3137885 0.2027305 0.28291984 0.2376202
## 4 0.3587463 0.36101510 0.3766965 0.2054242 0.36101484 0.2896493
## 5 0.4368965 0.44187970 0.4552823 0.2582409 0.44187903 0.3650570
## 6 0.1351466 0.04005404 0.1366139 0.2897079 0.04005828 0.1570829
```

Now we create a table where we tabulate the results: which point in the data is nearest the ith practice point, and what is the value of that point for our class variable (whether or not a donation was made). There is some additional fussing about the case where there are multiple nearest points, but as you can see there are no "ambiguous" values, where multiple nearest points have different values for the class variable.

```
distanceresults<-data.frame() #length(practice) rows, one for each test case. first column is minimum d

for(i in 1:length(practicebucket[,1])){ #traversing across our practice data
  distanceresults[i,1]<-min(distancesbyrow[,i]) #the minimum distance for each column of distancesbyrow
  distanceresults[i,2]<-sum(distanceresults[i,1]==distancesbyrow[,i]) #the number of occurences of this
  distanceresults[i,3]<-data[which.min(distancesbyrow[,i]),6] ## pulls the "donation" value for the firs
  for(j in 1:length(practicebucket[,1])){ ## traversing across the distances for one possible case
    if(distancesbyrow[j,i]==distanceresults[i,1] & data[j,6]!=distanceresults[i,3]){
     distanceresults[i,3]<-"amb"
    } ## if the distance for this case is minimum AND the playvalue is not equal to the first playvalue
  }
}

colnames(distanceresults)<-c("minimumdistance","occurrences", "donation")
head(distanceresults)
```

```
##   minimumdistance occurrences donation
## 1      0.00000000           1        0
## 2      0.00008000           1        0
## 3      0.00016000           1        1
## 4      0.02703283           1        0
## 5      0.00016000           1        0
## 6      0.00000000           1        1
```

```
sum(distanceresults$occurrences==1) # the number of possible classes that have a unique closest neighbo
```

```
## [1] 54
```

```
sum(distanceresults$donation=="amb") # the number of unambiguous results
```

```
## [1] 0
```

Let's compare the results of our training against the real donation values:

```
predictedresults<-distanceresults[,3]
originaldata<-data[501:576,6]
correctanswers<-sum(predictedresults==originaldata)
errorrate<-1-correctanswers/length(originaldata)
errorrate
```

```
## [1] 0.1973684
```

We have an error rate of 20%. Ordinarily, the next thing to do would be to run several adjustments to the weight vector, trying to minimize the error rate of the calculated nearest neighbors. Let's instead graduate to using a real k-nearest neighbors package, in the "class" library.

```
train<-databucket
test<-practicebucket
cl<-data[1:500,6]
libraryresults<-knn(train, test, cl, k=3, prob=TRUE)
libraryresults
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
## [36] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [71] 0 0 0 0 0 0
## attr(,"prob")
##  [1] 0.5000000 1.0000000 0.6666667 1.0000000 1.0000000 0.6666667 1.0000000
##  [8] 1.0000000 0.7500000 1.0000000 1.0000000 0.6666667 0.8333333 0.8333333
## [15] 0.8333333 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
## [22] 1.0000000 0.6666667 1.0000000 1.0000000 1.0000000 0.6666667 1.0000000
## [29] 0.7500000 1.0000000 0.6666667 0.6666667 1.0000000 1.0000000 1.0000000
## [36] 1.0000000 0.6666667 0.6666667 0.6666667 0.6666667 1.0000000 1.0000000
## [43] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
## [50] 1.0000000 0.6666667 1.0000000 0.6666667 1.0000000 0.6666667 1.0000000
## [57] 0.6666667 0.6666667 0.9090909 0.9090909 0.6666667 0.9090909 0.9090909
## [64] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
## [71] 1.0000000 1.0000000 1.0000000 1.0000000 0.8888889 0.6666667
## Levels: 0 1
```

Note that we've told R to look at the 3 nearest neighbors, and the library returns (in the first result) the winning classification for each test row, and also the probability of that classification based on the nearest neighbors. Let's calculate the error rate here:

```
predictedresults<-as.numeric(libraryresults)-1
originaldata<-data[501:576,6]
correctanswers<-sum(predictedresults==originaldata)
errorrate<-1-correctanswers/length(originaldata)
errorrate
```

```
## [1] 0.09210526
```

Using the same code, we checked the error rate considering the 1, 2, 3, 4 nearest neighbors:

```
k<-c(1,2,3,4)
error<-c("18.4%", "13.1%", "9.2%",  "10.5%")
pander(cbind(k,error))
```

k	error
1	18.4%
2	13.1%
3	9.2%
4	10.5%

We should clearly consider the 3 nearest neighbors, since error rate actually goes up at $k = 4$, and we'd like to avoid overfitting. Note that the built-in package has slightly smaller error than our manual work even at $k = 1$, and that at $k = 3$ the error rate is half of what it was in our original work. As a conclusion here are the predictions that our method makes for the test data the contest gives us.

```
testdata<-read.csv("project test data.csv")
testdata<-testdata[-c(1,4,6)]
testdata<-cbind(testdata[,1]*weights[1],testdata[,2]*weights[2],testdata[,3]*weights[3])
train<-scaledinstances
test<-testdata
cl<-data[,6]
libraryfullresults<-knn(train, test, cl, k=, prob=TRUE)
head(libraryfullresults)
```

```
## [1] 1 0 0 0 0 1
## Levels: 0 1
```