Caravans: KNN

Trevor Hastie and Robert Tibshirani

Here, I am adapting part of the lab associated with Chapter 4 of the textbook.

K-Nearest Neighbors

The library we need is `class`.

```
library(ISLR2)
library(class)
```

We will now perform KNN using the `knn()` function, which is part of the `class` library. This function works rather differently from the other model-fitting functions that we have encountered thus far. Rather than a two-step approach in which we first fit the model and then we use the model to make predictions, `knn()` forms predictions using a single command. The function requires four inputs.

- A matrix containing the predictors associated with the training data, labeled `train.X` below.
- A matrix containing the predictors associated with the data for which we wish to make predictions, labeled `test.X` below.
- A vector containing the class labels for the training observations, labeled `train.Y` below.
- A value for K, the number of nearest neighbors to be used by the classifier.

As an example we will apply the KNN approach to the `Insurance` data set, which is part of the `ISLR2` library. This data set includes 85 predictors that measure demographic characteristics for 5822 individuals. The response variable is `Purchase`, which indicates whether or not a given individual purchases a caravan insurance policy.

```
#data(Caravan)
dim(Caravan)
```

```
## [1] 5822   86
```

```
attach(Caravan)
#Purchase
summary(Purchase)
```

```
##   No  Yes
## 5474  348
```

```
summary(Purchase)[2]/(summary(Purchase)[1]+summary(Purchase)[2])
```

```
##        Yes
## 0.05977327
```

As we can see, in this data set, only 6% of people purchased caravan insurance.

Because the KNN classifier predicts the class of a given test observation by identifying the observations that are nearest to it, the scale of the variables matters. Variables that are on a large scale will have a much larger effect on the *distance* between the observations, and hence on the KNN classifier, than variables that are on a small scale. For instance, imagine a data set that contains two variables, `salary` and `age` (measured in dollars and years, respectively). As far as KNN is concerned, a difference of $\$1,000$ in salary is enormous compared to a difference of 50 years in age. Consequently, `salary` will drive the KNN classification results, and `age` will have almost no effect. This is contrary to our intuition that a salary difference of $\$1,000$ is quite small compared to an age difference of 50 years. Furthermore, the importance of scale to the KNN classifier leads to another issue: if we measured `salary` in Japanese yen, or if we measured `age` in minutes, then we'd get quite different classification results from what we get if these two variables are measured in dollars and years.

A good way to handle this problem is to *standardize* the data so that all variables are given a mean of zero and a standard deviation of one. Then all variables will be on a comparable scale. The `scale()` function does just this. In standardizing the data, we exclude column 86, because that is the qualitative `Purchase` variable.

```
standardized.X <- scale(Caravan[, -86])
mean(Caravan[, 1])
```

```
## [1] 24.25335
```

```
mean(Caravan[, 2])
```

```
## [1] 1.110615
```

```
mean(standardized.X[, 1])
```

```
## [1] 8.051119e-17
```

```
mean(standardized.X[, 2])
```

```
## [1] 3.567893e-16
```

```
var(Caravan[, 1])
```

```
## [1] 165.0378
```

```
var(Caravan[, 2])
```

```
## [1] 0.1647078
```

```
var(standardized.X[, 1])
```

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

```
var(standardized.X[, 2])
```

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

Now every column of `standardized.X` has a standard deviation of one and a mean of zero.

We now split the observations into a test set, containing the first 1,000 observations, and a training set, containing the remaining observations. We fit a KNN model on the training data using $K = 1$, and evaluate its performance on the test data.

```
test <- 1:1000
train.X <- standardized.X[-test, ]
test.X <- standardized.X[test, ]
train.Y <- Purchase[-test]
test.Y <- Purchase[test]

set.seed(1)
knn.pred <- knn(train.X, test.X, train.Y, k = 1)
mean(test.Y != knn.pred)
```

```
## [1] 0.118
```

```r
mean(test.Y != "No")
```

```
## [1] 0.059
```

The vector `test` is numeric, with values from 1 through $1,000$. Typing `standardized.X[test, ]` yields the submatrix of the data containing the observations whose indices range from 1 to $1,000$, whereas typing `standardized.X[-test, ]` yields the submatrix containing the observations whose indices do *not* range from 1 to $1,000$. The KNN error rate on the 1,000 test observations is just under 12%. At first glance, this may appear to be fairly good. However, since only 6% of customers purchased insurance, we could get the error rate down to 6% by always predicting `No` regardless of the values of the predictors!

Suppose that there is some non-trivial cost to trying to sell insurance to a given individual. For instance, perhaps a salesperson must visit each potential customer. If the company tries to sell insurance to a random selection of customers, then the success rate will be only 6%, which may be far too low given the costs involved. Instead, the company would like to try to sell insurance only to customers who are likely to buy it. So the overall error rate is not of interest. Instead, the fraction of individuals that are correctly predicted to buy insurance is of interest.

It turns out that KNN with $K = 1$ does far better than random guessing among the customers that are predicted to buy insurance. Among 77 such customers, 9, or 11.7%, actually do purchase insurance. This is double the rate that one would obtain from random guessing.

```r
(tab=table(knn.pred, test.Y))
```

```
##         test.Y
## knn.pred  No Yes
##      No  873  50
##      Yes  68   9
```

```r
tab[2,2]/(tab[2,1]+tab[2,2])
```

```
## [1] 0.1168831
```

Using $K = 3$, the success rate increases to 19%, and with $K = 5$ the rate is 26.7%. This is over four times the rate that results from random guessing. It appears that KNN is finding some real patterns in a difficult data set!

```r
knn.pred <- knn(train.X, test.X, train.Y, k = 3)
(tab=table(knn.pred, test.Y))
```

```
##         test.Y
## knn.pred  No Yes
##      No  920  54
##      Yes  21   5
```

```r
tab[2,2]/(tab[2,1]+tab[2,2])
```

```
## [1] 0.1923077
```

```r
knn.pred <- knn(train.X, test.X, train.Y, k = 5)
(tab=table(knn.pred, test.Y))
```

```
##         test.Y
## knn.pred  No Yes
##      No  930  55
##      Yes  11   4
```

```r
tab[2,2]/(tab[2,1]+tab[2,2])
```

```
## [1] 0.2666667
```

However, while this strategy is cost-effective, it is worth noting that only 15 customers are predicted to purchase insurance using KNN with $K = 5$. In practice, the insurance company may wish to expend resources on convincing more than just 15 potential customers to buy insurance.

As a comparison, we can also fit a logistic regression model to the data. If we use 0.5 as the predicted probability cut-off for the classifier, then we have a problem: only seven of the test observations are predicted to purchase insurance. Even worse, we are wrong about all of these! However, we are not required to use a cut-off of 0.5. If we instead predict a purchase any time the predicted probability of purchase exceeds 0.25, we get much better results: we predict that 33 people will purchase insurance, and we are correct for about 33% of these people. This is over five times better than random guessing!

```
glm.fits <- glm(Purchase ~ ., data = Caravan,
    family = binomial, subset = -test)
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
glm.probs <- predict(glm.fits, Caravan[test, ],
    type = "response")
glm.pred <- rep("No", 1000)
glm.pred[glm.probs > .5] <- "Yes"
table(glm.pred, test.Y)
```

```
##         test.Y
## glm.pred  No Yes
##      No  934  59
##      Yes   7   0
```

```
glm.pred <- rep("No", 1000)
glm.pred[glm.probs > .25] <- "Yes"
(tab=table(glm.pred, test.Y))
```

```
##         test.Y
## glm.pred  No Yes
##      No  919  48
##      Yes  22  11
```

```
tab[2,2]/(tab[2,1]+tab[2,2])
```

```
## [1] 0.3333333
```