8. Instance-Based Learning with kNN

8.1 Overview

The kNN algorithm is a supervised learning algorithm but it does not form a model of the input data. Instead, all the training observations are simply stored in memory. When a new observation needs to be evaluated, the algorithm compares it with the observations stored in memory, finding the closest k neighbors.

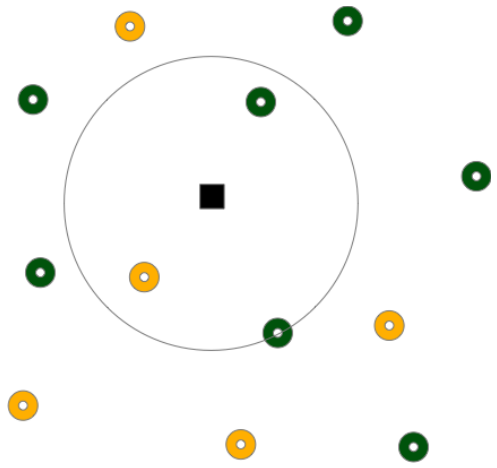

Figure 8.1: Finding k=3 Nearest Neighbors

As illustrated in Figure 8.1 the new observation is the black square. The three nearest neighbors were found and the observation will be classified as green because the majority of near neighbors were green. If this had been a regression task, the black square would be predicted to have a value that is the average target value of the nearest neighbors.

The kNN algorithm is sometimes called *instance-based learning* because it compares new instances with instances stored in memory.

8.2 Using kNN in R

Using the familiar iris data set we will run the kNN algorithm. Notice in Figure 8.2 that we have 3 classes. One of the nice things about the kNN algorithm is that it can predict class membership in a multi-class data set.

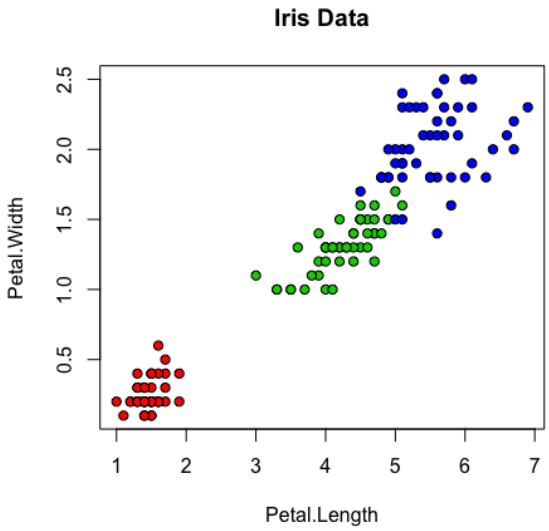

Figure 8.2: The Three Iris Classes

The code for the knn algorithm is shown below. In the notebook online, we have first randomly divided the 150 observations into 67% training and 33% test. We have also separated out the labels in both train and test into new vectors and removed the labels from train and test. We have supplied 4 arguments to the knn() function: the training data, the test data, the training data and the chosen value of k. Notice that we do not build a model, we are loading into memory and predicting on the test data all in one command.

Code 8.2.1 — kNN Classification. The Iris Data.

```
library(class)
iris_pred <- knn(train=iris.train, test=iris.test,
     cl=iris.trainLabels, k=3)
```

After running the above code, the iris_pred variable will be a vector of class labels: setosa, virginica or versicolor. An optional parameter for knn() is to set prob=TRUE which will return probabilities rather than class predictions. The algorithm achieved 98% prediction accuracy on the test data, but as we have seen before, and as you can observe in Figure 8.2 this is an easy data set to classify. The notebook code to compute accuracy was:

```
acc <- length(which(iris_pred == iris.testLabels)) /
       length(iris_pred)
```

R code can sometimes seem like those Russian nesting dolls so let's unpack from the inner function out. The `which(`*iris_pred == iris.testLabels*`)` returns a vector of indices for test items that were correct. The length() function surrounding this returned 49. This 49 was divided by the length of the predictions, 50, to get the 98% accuracy. The online notebook compares this result to performing one-versus-all classification with logistic regression, which got 100% accuracy.

Check Your Understanding 8.1 — kNN Classification on Wine Data. You can find the data set wine_all.csv in the github site. It is a 6497x13 data set of the chemical composition of red and white wines. This data set was edited from the wine data set on the UCI ML Repository. Your task is to use kNN to classify red/white wine.

- Divide the data into train and test sets, setting a seed first for reproducibility.
- Use R commands to make sure that the train and test sets have distributions of white and red types similar to the overall data.
- For comparison, first build a logistic regression model predicting wine type (red or white) based on all other columns. What is your acccuracy?
- Run knn() with k=2 on the data and compare the accuracies of the two algorithms.

■

8.3 The Algorithm

In kNN learning, an observation is known by the company it keeps. For a given test observation x_i, the kNN classifier will identify the k closest points, the neighbors, and estimate the conditional probability for class j as the fraction of neighbors that have that class.

$$P(Y = j|X) = \frac{1}{k}\sum_i I(y_i = j) \tag{8.1}$$

where I() is an indicator function returning TRUE or FALSE.

For regression, an average of the neighbors' target value is taken to be the predicted value for an instance.

$$\hat{y} = \frac{1}{k}\sum_{i \in NB} y_i \tag{8.2}$$

where NB is the set of neighbors.

8.3.1 Choosing K

The choice of a value for k needs to be done before the algorithm is run. The choice of k is critical to the bias-variance tradeoff of the algorithm. If k is very small, the classifier will have low bias but high variance. As k grows, the algorithm becomes less flexible and bias increases while variance decreases. The optimal value for k is often found by cross validation.

A rule-of-thumb that is sometimes read says to choose a k that is the square root of the number of observations. We have not noticed this to be an effective heuristic. Cross-validation is much more reliable. If you are doing classification, it makes sense to let k be an odd number. If k is too small the algorithm will be susceptible to noise but if k is too large the computation time increases.

8.3.2 Curse of Dimensionality

The kNN algorithm works best when we have few predictors. If we have 3 predictors, it will be easier to find neighbors in this 3-dimensional space. If we have 20 predictors, it will be harder to find neighbors in this 20-dimensional space. This is referred to as the curse of dimensionality. Some algorithms, like Naive Bayes, do not suffer in high dimensions but kNN will bog down.

8.4 Mathematical Foundations

What does it mean for an instance to be near another? Often Euclidean distance is used:

$$dist = \sqrt{\sum_i (q_i - p_i)^2} \tag{8.3}$$

In computing the distance, the predictors need to be numeric. If we have a column that is a factor this may be appropriate for linear regression but you should convert it to an integer for kNN.

8.4.1 Scaling Data

The terms scaling and normalization are used inconsistently in the literature. Generally, *normalization* means applying transformations to the data so that it follows a normal distribution while *scaling* implies some linear transformation of the data that may or may not result in a normal distribution. The R scale() function, using the default settings, will transform data to have a mean of 0 and a standard deviation of 1. If the data is normally distributed this should be sufficient. You can easily look at the distribution with a hist() graph. If data is highly skewed then you might get better results with something like this:

```
normalize <- function(x) {
  return((x - min(x)) / (max(x) - min(x)))
 }
 # apply to all columns
 data_norm <- as.data.frame(lapply(df, normalize))
```

Another nice thing about the R scle() function is that there is an unscale() function to convert scaled data back to the original by doing the inverse operations that scaled it. The unscale() function uses information stored with the scaled data.

```
library(DMwR)
scaled <- scale(df)
predictions <- predict(...)
original <- unscale(predictions, scaled)
```

In the knn classification problem above on the iris data, we did not need to scale because all measurements are in the same units. As we will see in the next example, when predictors are not in the same units, scaling usually improves performance.

Check Your Understanding 8.2 — Scaling Data. Using the same wine data set as above:

- Scale the data and see if the knn performance improves.
- See if reducing the number of predictors (reducing the curse of dimensionality) improves the results of the knn algorithm.

▪

8.5 kNN Regression

Next we perform regression on the Auto data set in package `ISLR`. In the online notebook, first we do linear regression to predict mpg, miles per gallon, based on weight, year, and origin. The linear regression model gets a correlation of 0.9079 and an mse of 9.88. The rmse will be around 3.14 mpg. Let's see if kNN can do better.

The first kNN attempt in the online notebook did not scale the data and got a worse correlation of 0.813 and a worse mse of 19.29. THen we scaled the data as shown below. Running kNN on the scaled data resulted in a correlation of 0.922, which is better than the linear model, and an mse of 0.138. This is the best of the 3 models, the rmse tells us we are off an average of 0.37 mpg.

Code 8.5.1 — kNN Regression. Auto Data.

```
library(caret)
df <- data.frame(scale(Auto[,c(1, 5,7,8)])) # mpg, weight, year, origin
train <- df[i,]
test <- df[-i,]
fit <- knnreg(train[,2:4],train[,1],k=3)
predictions <- predict(fit, test[,2:4])
cor(predictions, test$mpg)
mse <- mean((predictions - test$mpg)^2)
```

Check Your Understanding 8.3 — kNN Regression. Using the same wine data set as above, we now try to predict the quality.

- Remove the red/wine column from the scaled train and test sets.
- Run knnreg() on the data.
- What is the cor() and mse?
- Create a linear regression model.
- Compare the linear regression and knn model performance on the test set.

■

8.6 Find the Best K

The results with k=3 were good, but how do we know if a different k would produce even better results? We can try various values for k to find out. First we fill two vectors with 20 zeroes. These will hold our results for each level of k. Then we let k be 1, 3, 5, ..., 39. At each iteration in the for loop we store the correlation and mse and also print them out.

Code 8.6.1 — Find the Best K. Auto Data

```
cor_k <- rep(0, 20)
mse_k <- rep(0, 20)
i <- 1
for (k in seq(1, 39, 2)){
  fit_k <- knnreg(train[,2:4],train[,1], k=k)
  pred_k <- predict(fit_k, test[,2:4])
  cor_k[i] <- cor(pred_k, test$mpg)
  mse_k[i] <- mean((pred_k - test$mpg)^2)
  print(paste("k=", k, cor_k[i], mse_k[i]))
  i <- i + 1
}
```

We can either visually go down the printed values to find the minimum mse and the maximum correlation or we can use the following commands at the console:

```
> which.min(mse_k)
[1] 8
```

```
> which.max(cor_k)
[1] 8
```

It looks like the 8th element is best. That corresponds to k=15. Let's plot this to get a visual understanding. The plot in Figure 8.3 confirms that k=15 gives the best results.

Code 8.6.2 — Plot mse and cor. For various k.

```
plot(1:20, cor_k, lwd=2, col='red')
par(new=TRUE)
plot(1:20, mse_k, lwd=2, col='blue', labels=FALSE, ylab="", yaxt='n')
```

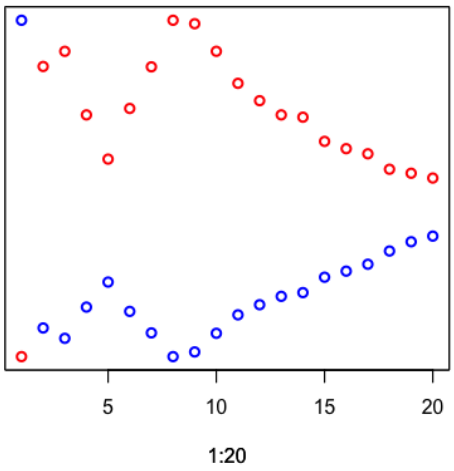

Figure 8.3: MSE (blue) and Correlation (red) for Various K

Check Your Understanding 8.4 — Finding the Best k. Using the same regression task as above, predicting quality from other variables, try the following:

- Find the best k for your model.
- What is the cor() and mse for this model? How does it compare to the linear regression model?

8.7 k-fold Cross Validation

The code sample in the last section ran through several values of k to find the optimal k. Running the model many times gave us more information about the data that running only once with k=3 would have. Techniques where algorithms are run different times to find the best or the most accurate parameters fall in the category of resampling methods. In effect the kNN algorithm is "sampling" neighbors. As seen in Figure 8.3 above, when k=3 we got good results but not for k= 4, 5, 6, or 7. Then results improved again at k=8. This shows how kNN is sensitive to the data. If we sampled the data many times to get different test sets, the graphs would vary quite a bit from sample to sample. There are many statistical approaches to sampling. In this section we will talk about k-fold cross validation.

8.7.1 Creating k Folds

Figure 8.4 illustrates k-fold cross validation with k=10. In k-fold cross validation, the entire data set is divided into k (10 in this case) equal portions. For k iterations, the algorithm is trained on all but k-1 portions of the data, leaving the held-out data for test. The test metrics for the k runs are then averaged together to get an overall estimate of test error. Common values of k are 5 and 10.

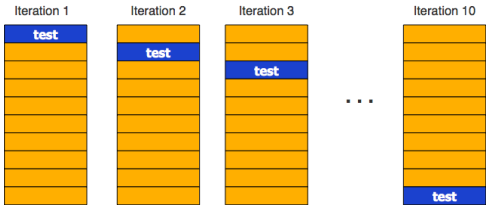

Figure 8.4: 10-fold Cross Validation

Since k-fold cross validation is computationally expensive, why do it? There are a few of situations where it will be useful. One is if you have a very small data set and your algorithm is prone to high variance. By running the algorithm many times on different subsets of test data you will get a better idea of how the algorithm will generalize to new data. With just one sampling of the test data you may have gotten lucky or unlucky with the selection of observations in the test set.

Another reason to use cross validation is to select model parameters. In the case of kNN we have to select a value for k. In the last section we selected this value based on how well it performed on this test set. Will that value of k also be good for other random draws of test data? Cross validation can answer that question. This is sometimes called *parameter tuning*.

Cross validation is a technique that can be used with any algorithm. For example, we could use it in linear regression by trying different polynomial regression lines to find the model that best fits the data. This is sometimes called *model selection*.

Next we use 10-fold cross validation on the knn regression problem we discussed earlier. This is in a separate notebook in the github. After loading the Auto data and subsetting it to just mpg, weight, year, and origin we need to divide the data into folds. We could easily write code for that ourselves but the `caret` package already has a nice function for that, `createFolds`. The first argument to createFolds() is our target column and the second tells it we want 10 folds. Since there are 392 observations in Auto, we expect a little less than 40 indices in each fold. We confirm that with sapply().

Code 8.7.1 — Cross Validation. Divide Data into Folds.

```
library(caret)
set.seed(1234)
folds <- createFolds(df$mpg, k=10)
sapply(folds, length)
Fold01 Fold02 Fold03 Fold04 Fold05 Fold06 Fold07 Fold08 Fold09 Fold10
    39     41     37     40     39     39     39     39     40     39
```

You can look at the indices in the first fold with `folds[[i]]`. Recall that double square brackets are used with lists.

```
> folds[[3]]
 [1]   13   15   24   42   54   55   57   60   73   76   84   91 113 126 131 138
       143 144 157 159
[21] 164 173 191 197 204 206 207 218 225 230 270 278 298 310 339 351
       362 366
```

8.7.2 Run knnreg() on each Fold

Now that we have the folds, we can run knnreg() on each fold and average the results. For now we just let k=3.

Code 8.7.2 — Cross Validation. Run 10 times.

```
test_mse <- rep(0, 10)
test_cor <- rep(0, 10)
for (i in 1:10){
  fit <- knnreg(df[-folds[[i]], 2:4], df$mpg[-folds[[i]]], k=3)
  pred <- predict(fit, df[folds[[i]], 2:4])
  test_cor[i] <- cor(pred, df$mpg[folds[[i]]])
  test_mse[i] <- mean((pred - df$mpg[folds[[i]]])^2)
}
print(paste("Average correlation is ", round(mean(test_cor), 2)))
print(paste("range is ", range(test_cor)))
print(paste("Average mse is ", round(mean(test_mse), 2)))
print(paste("range is ", range(test_mse)))
```

```
[1] "Average correlation is  0.93"
[1] "range is  0.90883537818507"  "range is  0.946753085315825"
[1] "Average mse is  0.15"
[1] "range is  0.111756182643287" "range is  0.201818162667268"
```

You can see that the average correlation and mse are good and also that there is a sizeable range in both vectors. And this is with holding k steady at 3. Let's see what the values will be if we try various values of k. This will involve rewriting the code above.

8.7.3 Cross Validation with Various K

In the code section below we use sapply() to try different values of k on an anonymous function that does the 10-fold cross validation similarly to the previous code segment.

Code 8.7.3 — Cross Validation. For k=1,3,5...

```
# try various values for k
k_values <- seq(1, 39, 2)
results <- sapply(k_values, function(k){
  mse_k <- rep(0, 10)
  cor_k <- rep(0, 10)
  for (i in 1:10){
    fit <- knnreg(df[-folds[[i]], 2:4], df$mpg[-folds[[i]]], k=k)
    pred <- predict(fit, df[folds[[i]], 2:4])
    cor_k[i] <- cor(pred, df$mpg[folds[[i]]])
    mse_k[i] <- mean((pred - df$mpg[folds[[i]]])^2)
  }
  list(mean(cor_k), mean(mse_k))
})
# reshape results into matrix
m <- matrix(results, nrow=20, ncol=2, byrow=TRUE)
```

The output of sapply() is stored in variable results. This will be a list consisting of the 40 output values, alternating the correlation and mse values. We'd like those separated so we coerce the list

into a mstrix m. Then column 1 of m will contain all the correlations and column 2 will contain all the mse values.

Next we have the code to create plots, which are shown in Figure 8.5. The graphs in the figure indicate that k=3 gives the highest correlation and the lowest mse.

Code 8.7.4 — Cross Validation. Plot Results

```
par(mfrow=c(2, 1))
plot(1:20, unlist(m[,1]), lwd=2, type="o", col='red', ylab="Correlation")
plot(1:20, unlist(m[,2]), lwd=2, type="o", col='blue', ylab="MSE")
```

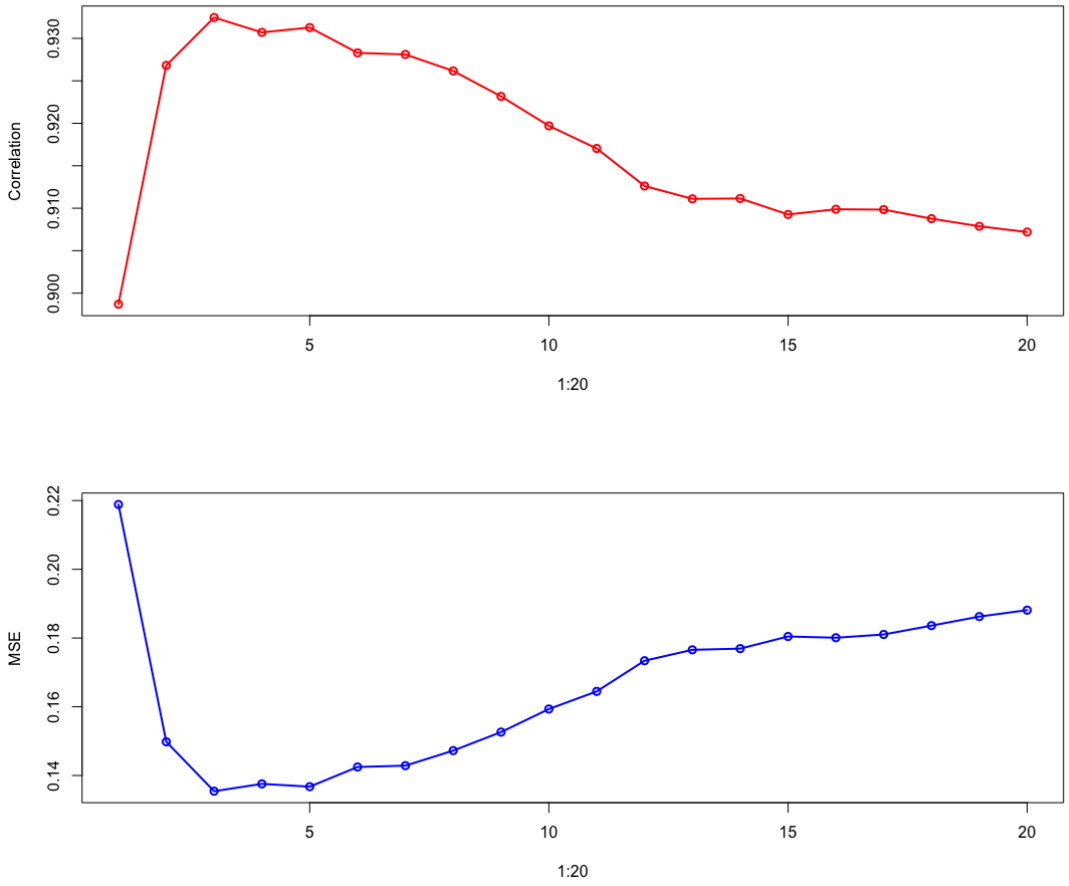

Figure 8.5: Cross Fold Validation for various K

8.7.4 Applying CV to Other Algorithms

The technique demonstrated in this section can be applied to any algorithm. However, this approach becomes significantly more time intensive with larger data. For example, if you have an algorithm on a large data set that takes 30 minutes to run, and you do 10-fold cross-validation, that will take 5 hours. Of course, if you have a large amount of data there is less of a need to use the cross-validation approach. The catch-22 with small data is that your small test set is likely to result in variance and

yet if you put more of your limited data into the test set then you have less for training which could make your algorithm perform poorly. Cross-validation enabled us to have more confidence in our test set metrics by compensating for a relatively small data set. Researchers generally recommend values of k=5 or k=10 since these have been demonstrated empirically to create a good balance between bias and variance.

The `caret` package contains many functions related to sampling. This section demonstrated a regression example. If we are performing classification, we need to be concerned about the distribution of observations in the train and test sets. The downSample() function will randomly sample each class to be the size of the smallest class. This results in smaller data which is fine if you have a lot of data to start with. The upSample() function will randomly sample with replacement so that the smaller class becomes as large as the majority class. These are called *subsampling* techniques, and more can be read about them in the caret documentation.

8.8 Summary

The kNN algorithm is an example of an instance-based approach. It does not create a model of the data and by extension is a non-parametric algorithm. If k is small, the algorithm tends towards low bias and high variance. As k gets larger, the algorithm tends towards high bias and low variance. This algorithm can be used for either classification or regression.

Advantages:

- Makes no assumptions about the shape of the data
- Performs well in low dimensions

Disadvantages:

- Bogs down in high dimensions
- K must be chosen
- Data should be scaled for best performance
- Difficult to interpret

8.8.1 Quick Reference

Reference 8.8.1 kNN Classification

```
library(class)
predictions <- knn(train, test, cl=trainLabels, k=3)
```

Reference 8.8.2 kNN Regression

```
library(caret)
fit <- knnreg(train, labels, k=3)
predictions <- knnreg(fit, test)
```

Reference 8.8.3 Scaling and Unscaling

```
scaled <- scale(df) # scale is built-in
# unscaled in package DMwR
predictions <- ...
unscaled_predictions <- unscale(predictions, scaled)
```

8.8.2 Lab

Problem 8.1 — Practice on the Abalone Data - kNN Regression.. Try the following:

1. Download the Abalone data from the UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml/datasets/Abalone`.
2. The data does not have column names so you will have to create them. Use meaningful names based on your reading about the data on the UCI site.
3. Check if there are missing values.
4. Divide the data into 80-20 train-test, setting a seed for reproducibility.
5. Normalize the data.
6. Perform knn regression with k=3. What is the cor and mse? Are these good results? Why or why not?
7. Try a range of k values to find the best k. Did the metrics improve?

Problem 8.2 — Practice on the Abalone Data - kNN Classification.. Start with the same Abalone data as the previous problem, but add a size column as we did in an earlier lab.

1. Examine the rings column with range(), median(), and hist() to determine where you would like to split the data into two classes: large and small.
2. Create a new factor column for binary large/small based on the rings column and your cut-off decision.
3. Divide the data into 80-20 train-test, setting a seed for reproducibility.
4. Normalize the data.
5. Perform knn classification with all predictors except rings, using k=3. What is the accuracy of the model? Do you think these are good results? Why or why not?
6. Try a range of k values to find the best k. Did the accuracy improve?

Problem 8.3 — Practice on the Heart Data. Try the following:

1. Download the Heart data from the UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml/datasets/Heart+Disease`.
2. The data does not have column names so you will have to create them. Use meaningful names based on your reading about the data on the UCI site.
3. Make sure class is a factor and that all the other columns are numeric or integer.
4. Remove columns with large numbers of NAs because the knn algorithm can't handle them.
5. After removing those columns (slope, ca, and thal), reduce the heart data to only complete cases.
6. Divide the data into 80-20 train-test, setting a seed for reproducibility.
7. Normalize the predictor data.
8. Run knn() with k=3 to create predictions on the test data. What is the accuracy? Do you think this is a good result? Why or why not?
9. Try a range of k values to find the best k. What is the accuracy?

8.8.3 Exploring Concepts

Problem 8.4 Based on your experience with the knn algorithm in R, does removing predictors necessarily improve performance? Discuss possible reasons for your answer.

Problem 8.5 If you found that for a kNN regression problem, the optimal value for k using mse differed from the optimal value using cor. Which one would you prefer? Justify your answer.

Problem 8.6 As k increases, do you think the fit to the data is more or less flexible? Justify your answer.

Problem 8.7 As k increases, do you think the fit to the data tends toward higher bias or higher variance? Justify your answer.

8.8.4 Going Further

The kNN algorithm has wide application. Here is an interesting paper that uses kNN.

Li, Fang, et. al use a modified kNN algorithm for network anomaly detection. Their paper was published by the ACM in 2007 as is available here: https://www.cs.bgu.ac.il