Chapter 12

k-Nearest-Neighbors Classification

`In God We Trust; All Others Must Bring Data`
– *Willian Edwards Deming*

We will begin our exploration of machine learning algorithms today. We will begin with one of so called *lazy learning* algorithms - Classification using Nearest Neighbors. It is essentially a process of discovery - classifying data by placing it in the category with the most similar or *nearest* neighbors.

Despite the simple basic idea, nearest neighbor methods are extremely powerful and popular. Popular success examples like Netflix Challenge, Amazon Prediction Engine etc have used versions of this algorithm. This algorithm works best for classification tasks where relationships between features and target classes are complex and difficult to understand but the class types themselves are fairly homogeneous. If the groups are not well differentiated then nearest neighbor algorithm does not provide optimal results.

12.1 kNN Algorithm

kNN algorithm starts with a training dataset where records are already classified into target classes. We now attempt to classify a test dataset which is unclassified but otherwise has same features as the training dataset. For each record in the test dataset, kNN algorithm identifies `k` records in the training data that are "nearest" or closest in features - here `k` is an integer. The test record is assigned the class of the majority of the `k` nearest neighbors.

kNN algorithm treats dataset features as coordinates in a multidimensional feature space. Two dimensional feature spaces can be easily visualized as a scatter plot. Higher dimensional features spaces are difficult to visualize but the essential idea remains the same. When trying to categorize a new record, we need to calculate its distance to records in the training dataset. Distances can be calculated as the Cartesian coordinate distance - also known as Euclidean Distance.

12.2 Calculating Distance

If we have two records p and q; and p_i and q_i refer to the i^th feature of p and q respectively; then the distance between p and q can be calculated as:

$$dist(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2}$$

Once we have distances with different records, we can sort them in increasing order. If we were to use $k = 1$ then we will just assign the record to the same category as the record with the smallest distance. If we use $k = 3$ then we look at the closest three neighbors and assign the record to the category that is most applicable within the three neighbors.

12.3 Choosing An Appropriate k

Choosing an appropriate k should balance between overfitting and underfitting the training data. This problem is commonly known as the **bias-variance tradeoff**. Choosing a large k reduces the impact or variance caused by noisy data - but runs the risk of being biased against smaller but important patterns. Typically, smaller values allow for more complex decision boundaries that fit the training data better. In practice, choosing k depends upon the difficulty of the classification task and the number of records in the training data. Typically, k is set somewhere between 3 and 10. A common practice is to set k equal to the square root of the number of records in training dataset.

12.4 Rescaling Data for kNN

As we depend on Euclidean Distance, the scale of variables matter a lot. Before we can apply kNN, we must transform variable to comparable scale. Following are the common methods of feature rescaling:

Min-Max Normalization Convert all values on a 0 to 1 scale using the formula below to get normalized values.

$$X_{new} = \frac{X - min(X)}{max(X) - min(X)}$$

z-Score Standardization Subtract mean value of the feature with the feature value and then divide by the standard deviation of the feature.

$$X_{new} = \frac{X - Mean(X)}{StdDev(X)}$$

z-Score essentially provides the number of standard deviations from the mean. Z-scores have a mean of 0 and distribution of 1. Unlike the normalized values above, z-score values are unbounded with a range of $-\infty$ to ∞.

Dummy Coding For nominal values, we use dummy coding of 1 or 0. If the feature has several levels then it will require several dummy variables (one less than the number of features). Dummy coding has the advantage of being on the same scale as normalized data.

12.5 Illustrated Example: Breast Cancer Diagnosis

We will illustrate kNN using "Breast Cancer Wisconsin Diagnostic" dataset from the UCI Machine Learning Repository. The dataset includes 569 examples of cancer biopsies each with 32 features. Of these 32, one is identification number and another is the cancer diagnosis (the target classification). Rest 30 are the features we need to use to figure out kNN classification.

```
#Import Data
wbcd <- read.csv("wisc_bc_data.csv", stringsAsFactors = FALSE)
#str(wbcd) # Lets check what we have in this dataset
# Str command commented for brevity in output
```

As the first variable *id* is only a unique identifier, it is not likely to add any value to our model, so we can safely drop it from the dataset.

```
wbcd <- wbcd[-1]
```

The second variable, *diagnosis* is of significant interest as this is the target classification - in this case a variable with two values *B* for Benign and *M* for Malignant. We can check details of this variable. We will also convert the variable as a factor to recognize its essential nature.

```
table(wbcd$diagnosis)
##
##   B   M
## 357 212
wbcd$diagnosis <- factor(wbcd$diagnosis, levels = c("B", "M"),
                         labels = c("Benign", "Malignant"))
round(prop.table(table(wbcd$diagnosis)) * 100, 1)
##
##    Benign Malignant
##      62.7      37.3
```

Rescaling the Data

We need to now figure whether our data has a problem of differing scales. Exploring three variables here:

```
summary(wbcd[c("radius_mean", "area_mean", "smoothness_mean")])
##   radius_mean       area_mean      smoothness_mean
##  Min.   : 6.981   Min.   : 143.5   Min.   :0.05263
##  1st Qu.:11.700   1st Qu.: 420.3   1st Qu.:0.08637
##  Median :13.370   Median : 551.1   Median :0.09587
##  Mean   :14.127   Mean   : 654.9   Mean   :0.09636
##  3rd Qu.:15.780   3rd Qu.: 782.7   3rd Qu.:0.10530
##  Max.   :28.110   Max.   :2501.0   Max.   :0.16340
```

We can clearly see that different variables differ significantly in their scale. So we will need to normalize our data so that we rescale variables to a standard range of values. We can write our own function for normalizing and then use *lapply* to normalize our dataset.

```
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
wbcd_n <- as.data.frame(lapply(wbcd[2:31], normalize))
summary(wbcd_n[c("radius_mean", "area_mean", "smoothness_mean")])
##   radius_mean        area_mean       smoothness_mean
##  Min.   :0.0000   Min.   :0.0000   Min.   :0.0000
##  1st Qu.:0.2233   1st Qu.:0.1174   1st Qu.:0.3046
##  Median :0.3024   Median :0.1729   Median :0.3904
##  Mean   :0.3382   Mean   :0.2169   Mean   :0.3948
##  3rd Qu.:0.4164   3rd Qu.:0.2711   3rd Qu.:0.4755
##  Max.   :1.0000   Max.   :1.0000   Max.   :1.0000
```

As we can now see, our three variables in question are now distributed on a standard scale; and are comparable to each other as far as euclidean distance is concerned.

Dividing Data Into Training and Testing Sets

Alright - now we need to train our model. Of course, we shouldn't use the entire dataset to train as then we will have no way to test the model. Lets keep 100 observations as the test dataset and use the rest for training.

```
wbcd_train <- wbcd_n[1:469, ]
wbcd_test <- wbcd_n[470:569, ]
```

We would also want to extract the target variable in factors. These factors would be useful for us in a little bit when we train and test out model.

```
wbcd_train_labels <- wbcd[1:469, 1]
wbcd_test_labels <- wbcd[470:569, 1]
```

Running the Model

Now is the time to train our model on the training data. We will need a package named *class*. Class package includes several classification algorithms.

```
#install.packages("class") #Commented
library(class)
## Warning:  package 'class' was built under R version 3.3.1
```

We will use the *knn*() function. The function call is pretty simple. It needs a training dataset, a testing dataset, a factor vector with the classification for each row in training data and an integer value corresponding to the number of nearest neighbors k. Since we have 469 observations in the training data, we might want to start with $k = 21$ as it is closest odd number to square root of 469. The function returns a factor vector of predicted values for each record in the test dataset.

```
wbcd_test_pred <- knn(train = wbcd_train, test = wbcd_test,
                      cl = wbcd_train_labels, k=21)
```

Model Accuracy

Alright - so we have run a model - is it any good? We should evaluate model performance. In this case it is easy since we already know what should have been the classification. The *CrossTable*() function in the *gmodels* package is a good choice to do this calculation.

```
#install.packages("gmodels") #Comments
library(gmodels)
## Warning:  package 'gmodels' was built under R version 3.3.1
CrossTable(x = wbcd_test_labels, y = wbcd_test_pred,
           prop.chisq=FALSE)
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##                  | wbcd_test_pred
## wbcd_test_labels |    Benign | Malignant | Row Total |
## -----------------|-----------|-----------|-----------|
##           Benign |        61 |         0 |        61 |
##                  |     1.000 |     0.000 |     0.610 |
##                  |     0.968 |     0.000 |           |
##                  |     0.610 |     0.000 |           |
## -----------------|-----------|-----------|-----------|
##        Malignant |         2 |        37 |        39 |
##                  |     0.051 |     0.949 |     0.390 |
##                  |     0.032 |     1.000 |           |
##                  |     0.020 |     0.370 |           |
## -----------------|-----------|-----------|-----------|
##     Column Total |        63 |        37 |       100 |
##                  |     0.630 |     0.370 |           |
## -----------------|-----------|-----------|-----------|
##
##
```

As we can see, there were 69 true negative cases - the mass was benign - and the kNN algorithm correctly identified each of the cases. There were 37 true positive cases where the model and the test data agree on identification of malignant cases. The model presents 2 false negatives and 0 false positives.

So we have a 98% accuracy - only two cases were wrongly classified.

Improving the Model with z-Scores

We have two ways to see if we can improve this model - we can try different values of k and try a different rescaling technique. Lets start by using the z-score as the rescaling method. We can use the *scale*() function for this.

```
wbcd_z <- as.data.frame(scale(wbcd[-1]))
```

We follow the same process as before for creating training and testing data.

```
wbcd_train <- wbcd_z[1:469, ]; wbcd_test <- wbcd_z[470:569, ]
```

We are ready to run a new model now.

```
wbcd_test_pred <- knn(train = wbcd_train, test = wbcd_test,
                      cl = wbcd_train_labels, k=21)
```

Let us see how we did this time.

```
CrossTable(x = wbcd_test_labels, y = wbcd_test_pred,
           prop.chisq=FALSE)
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##                  | wbcd_test_pred
## wbcd_test_labels |    Benign | Malignant | Row Total |
## -----------------|-----------|-----------|-----------|
##           Benign |        61 |         0 |        61 |
##                  |     1.000 |     0.000 |     0.610 |
##                  |     0.924 |     0.000 |           |
##                  |     0.610 |     0.000 |           |
## -----------------|-----------|-----------|-----------|
##        Malignant |         5 |        34 |        39 |
##                  |     0.128 |     0.872 |     0.390 |
##                  |     0.076 |     1.000 |           |
##                  |     0.050 |     0.340 |           |
## -----------------|-----------|-----------|-----------|
##     Column Total |        66 |        34 |       100 |
##                  |     0.660 |     0.340 |           |
## -----------------|-----------|-----------|-----------|
##
##
```

Well - the new model is actually a little bit worse. We now have 95% accuracy. False negatives actually increased - and that is actually a worse mistake to make than a false positive - so all in all - not a change that worked for us.

Trying Different k Values

Okay - we still have the option of trying different *k* values. Following are the results for different *k* values using normalized training data.

k value	False Positives	False Negatives	Accuracy Percentage
1	1	3	96%
5	2	0	98%
11	3	0	97%
15	3	0	97%
21	2	0	98%
27	4	0	96%

Figure 12.1: Different k Values

What do you think is the best model. $k = 1$ is better for avoiding false negatives. Original $k = 21$ was better as far as overall accuracy is concerned.