Machine Learning for Finance Applications - k-Nearest Neighbors

Brian Clark

2019

Contents

1 Introduction

This R-vignette file introduces the k-nearest neighbor (kNN) classifier. The kNN classifier is one of the most simplistic ML algorithms and as such, we will use it as an example to write the function to replicate the method from scratch. The goal is not to be able to write your own ML algorithms; rather to become more familiar with programming in R.

The kNN algorithm is rather simple in nature. The only parameter is k, the number of nearest neighbors. With k selected, the algorithm takes in a test observation x_0 and finds the k observations in the existing data that are most similar – or *nearest* – to x_0 and classifies x_0 using the Bayes rule. Th Bayes rule classifies x_0 into the class with the largest probability based on the set of k nearest neighbors.

The parameter k must be "tuned" by the user. The usual bias-variance tradeoff determines the optimal k. When k is very small, the algorithm will tend to overfit the data and the variance will be high (i.e., a small change in the data will have a large impact on the classifier). Conversely, as k increases the variance drops and the model bcomes more stable, but less accurate meaning that the bias increases.

1.1 Implementation in R

In this section, we will code the kNN function from scratch. Note that the following implementation is not the most efficient method for running the model but it captures the basic logic and is a good example for learning some basic programming tasks in R.

1.2 Data

For this example, we'll use R's built in `iris` data.

```
rm(list=ls())
dat <- iris
head(dat)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
dim(dat)
```

```
## [1] 150   5
```

```
names(dat)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

There are 150 rows and 5 variables. The first four are the features. "Species" is the response variable we are trying to predict. Let's summarize the data using and `apply()` function. R's `apply` suite of functions is essentially a different way of writing a for loop. Let's get the mean and variance of the data. `apply()` has three main inputs (type `help("apply")` to see the help file).

1. 'X': an array
2. 'MARGIN': a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where X has named dimnames, it can be a character vector selecting dimension names.
3. 'FUN': the function to be applied: see 'Details'. In the case of functions like +,

```
# Means using apply:
dat.means.apply <- apply(dat[,c(1:4)],2,mean)
print(dat.means.apply)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

We could have accomplished exactly the same thing using a for loop:

```
# Means using a for loop
dat.means.for <- c(NA)
for (i in 1:4){
  dat.means.for[i] <- mean(dat[,i])
}
print(dat.means.for)
```

```
## [1] 5.843333 3.057333 3.758000 1.199333
```

Do the same for the variance:

```
# Var using apply:
dat.var.apply <- apply(dat[,c(1:4)],2,var)
print(dat.var.apply)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##    0.6856935    0.1899794    3.1162779    0.5810063
```

Since the features are scaled, we may want to normalize the features. For example we could impose that they have mean = 0 and variance = 1. Otherwise, the scale of the inputs will have a huge impact on the kNN classifier. For this example, our normalization apporach is:

$$x_i' = \frac{x_i - min(X)}{max(X) - min(X)}$$

Let's write a function to normalize the data.

```
# Normalize the data:
myNorm <- function(x) {
  minX <- matrix(rep(apply(x,2,min),dim(x)[1]),byrow=T,ncol=dim(x)[2])
```

```
  maxX <- matrix(rep(apply(x,2,max),dim(x)[1]),byrow=T,ncol=dim(x)[2])
  num <- x - minX
  denom <- maxX - minX
  normdat <- num/denom
  return (normdat)
}

# Call the function
dat.norm <- myNorm(dat[,c(1:2)])

print(apply(dat.norm,2,mean))
```

```
## Sepal.Length  Sepal.Width
##    0.4287037    0.4405556
```

```
print(apply(dat.norm,2,var))
```

```
## Sepal.Length  Sepal.Width
##   0.05290845   0.03298254
```

Now we have the normalized data. Let's split it into training and test samples. Set the random number generator seed so you can replicate the results exactly. Also, note that we are only using two features to train the model. This is only so we can plot the results nicely.

```
# Set up training and test data:
set.seed(1234)

# Keep 2/3 of the data to train and 1/3 to test.
ind <- sample(2,nrow(dat.norm), replace=T, prob = c(0.67,0.33))

# Generate training data:
dat.norm.train <- dat.norm[which(ind==1),]

# Generate test data:
dat.norm.test <- dat.norm[which(ind==2),]

# Generate training Labels (response variable):
dat.norm.train.lab <- dat[which(ind==1),5]

# Generate test Labels:
dat.norm.test.lab <- dat[which(ind==2),5]
```

Let's first use a canned package to run the kNN algorithm for $k = 3$. We will use the `caret` library. We will come back to the `caret` package later in the workshop.

We can plot the decision boundary using Michael Hahsler's `decisionplot()` function which can be found **here**.

```
# Define the function:
decisionplot <- function(model, data, class = NULL, predict_type = "class",
  resolution = 100, showgrid = TRUE, ...) {

  if(!is.null(class)) cl <- data[,class] else cl <- 1
  data <- data[,1:2]
  k <- length(unique(cl))

  plot(data, col = as.integer(cl)+1L, pch = as.integer(cl)+1L, ...)
```

```
  # make grid
  r <- sapply(data, range, na.rm = TRUE)
  xs <- seq(r[1,1], r[2,1], length.out = resolution)
  ys <- seq(r[1,2], r[2,2], length.out = resolution)
  g <- cbind(rep(xs, each=resolution), rep(ys, time = resolution))
  colnames(g) <- colnames(r)
  g <- as.data.frame(g)

  ### guess how to get class labels from predict
  ### (unfortunately not very consistent between models)
  p <- predict(model, g, type = predict_type)
  # p <- predict(model,newdata=g)
  if(is.list(p)) p <- p$class
  p <- as.factor(p)

  if(showgrid) points(g, col = as.integer(p)+1L, pch = ".")

  z <- matrix(as.integer(p), nrow = resolution, byrow = TRUE)
  contour(xs, ys, z, add = TRUE, drawlabels = FALSE,
    lwd = 2, levels = (1:(k-1))+.5)

  invisible(z)
}
```

```
library("caret")
```

```
## Warning: package 'caret' was built under R version 3.6.2
```

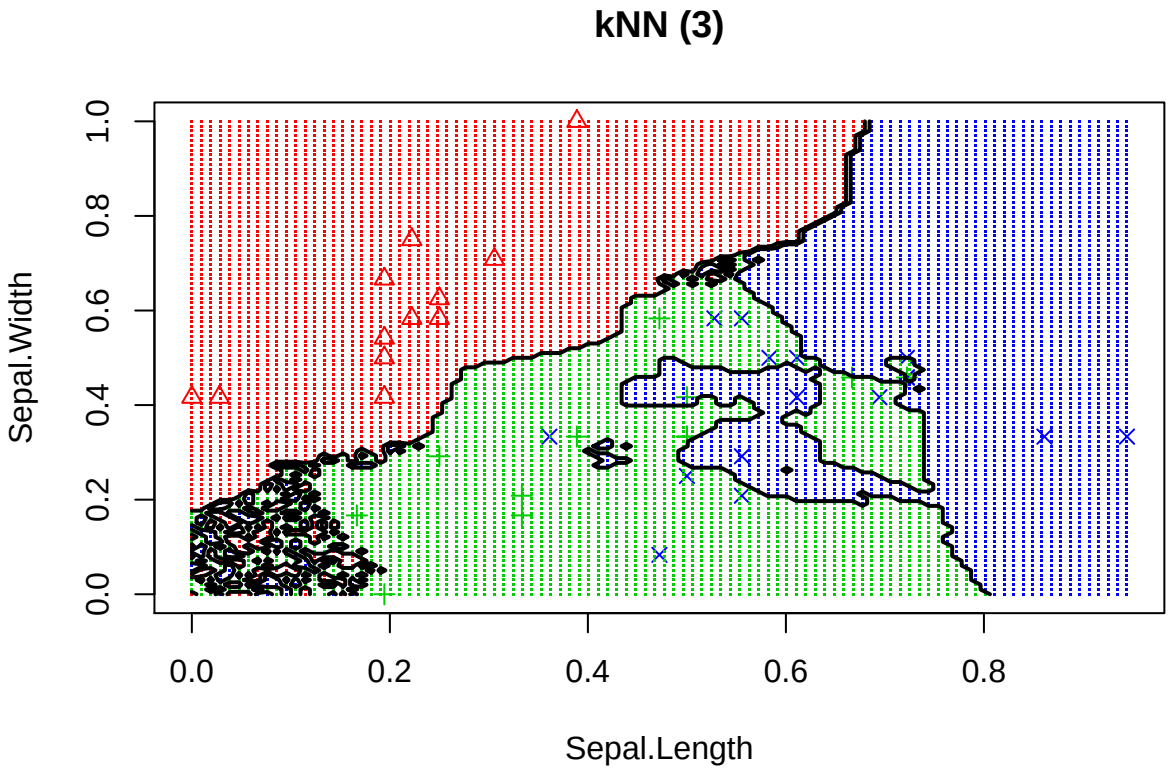
```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```
k <- 3

# Combine the data into a data frame:
dat.plot.train <- data.frame(dat.norm.train,dat.norm.train.lab)
names(dat.plot.train)[3] <- "Species"
head(dat.plot.train)
```

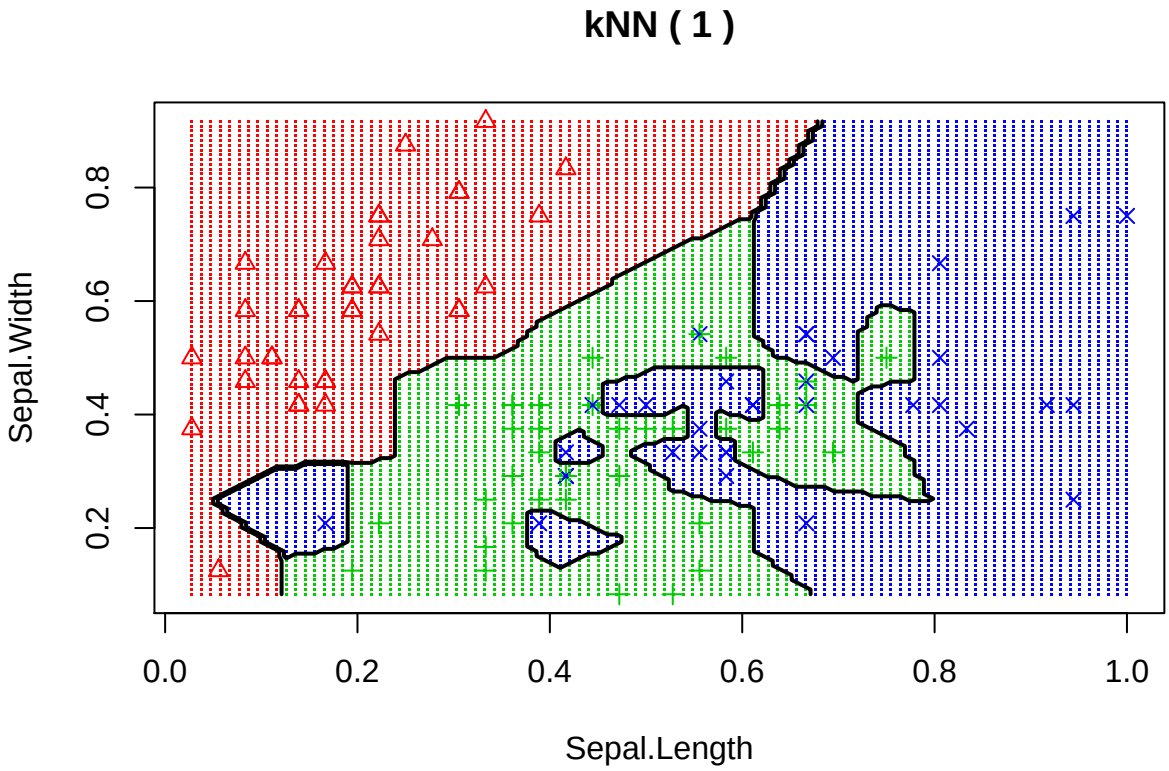
```
##   Sepal.Length Sepal.Width Species
## 1   0.22222222   0.6250000  setosa
## 2   0.16666667   0.4166667  setosa
## 3   0.11111111   0.5000000  setosa
## 4   0.08333333   0.4583333  setosa
## 6   0.30555556   0.7916667  setosa
## 7   0.08333333   0.5833333  setosa
```

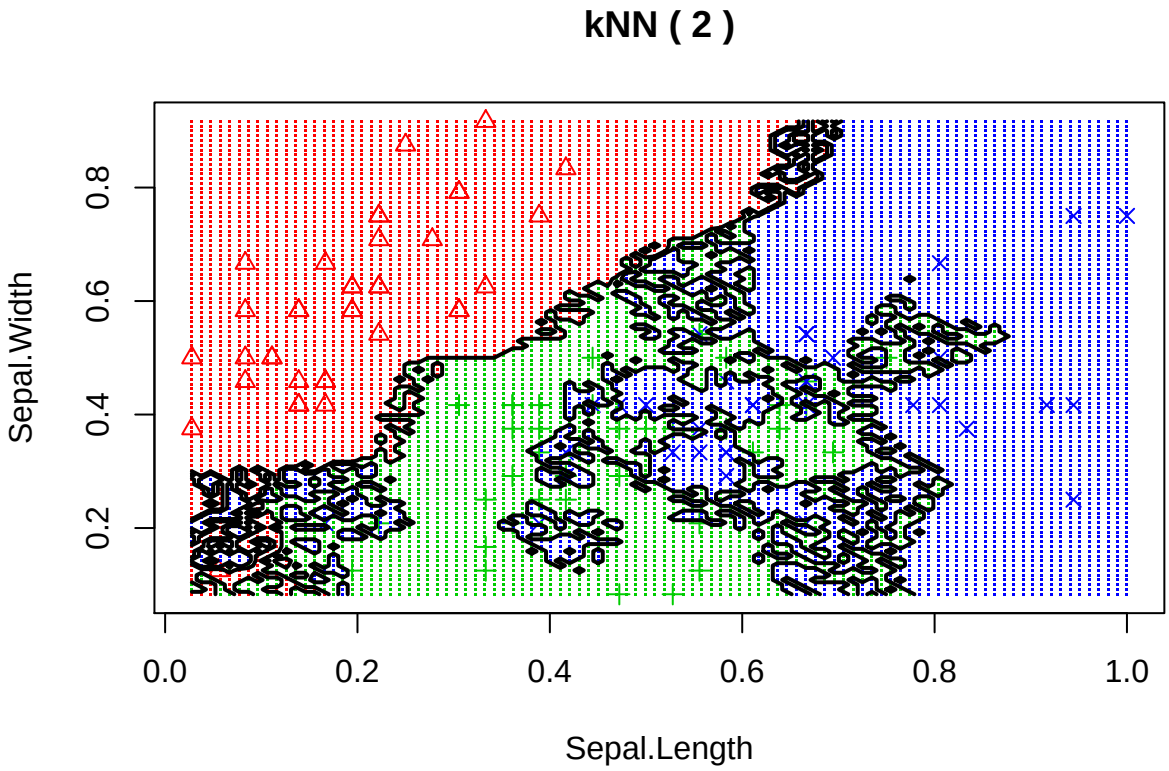
```
# Uncomment to fit the model to the whole dataset:
# dat.plot.train <- dat[,c(1,2,5)]

dat.plot.test <- data.frame(dat.norm.test,dat.norm.test.lab)
names(dat.plot.test)[3] <- "Species"

knn.model <- knn3(Species ~ ., data=dat.plot.train,
                  k = k,use.all=FALSE)
decisionplot(knn.model, data=dat.plot.test, class = "Species", main = "kNN (3)")
```

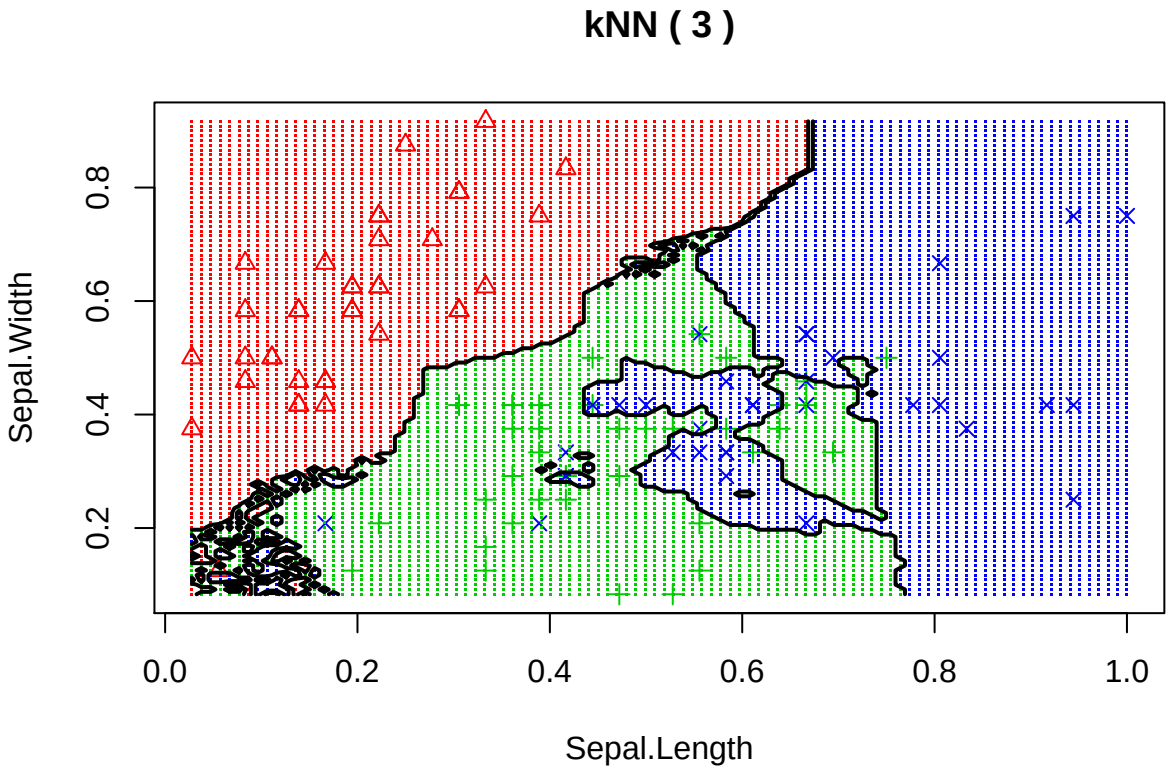


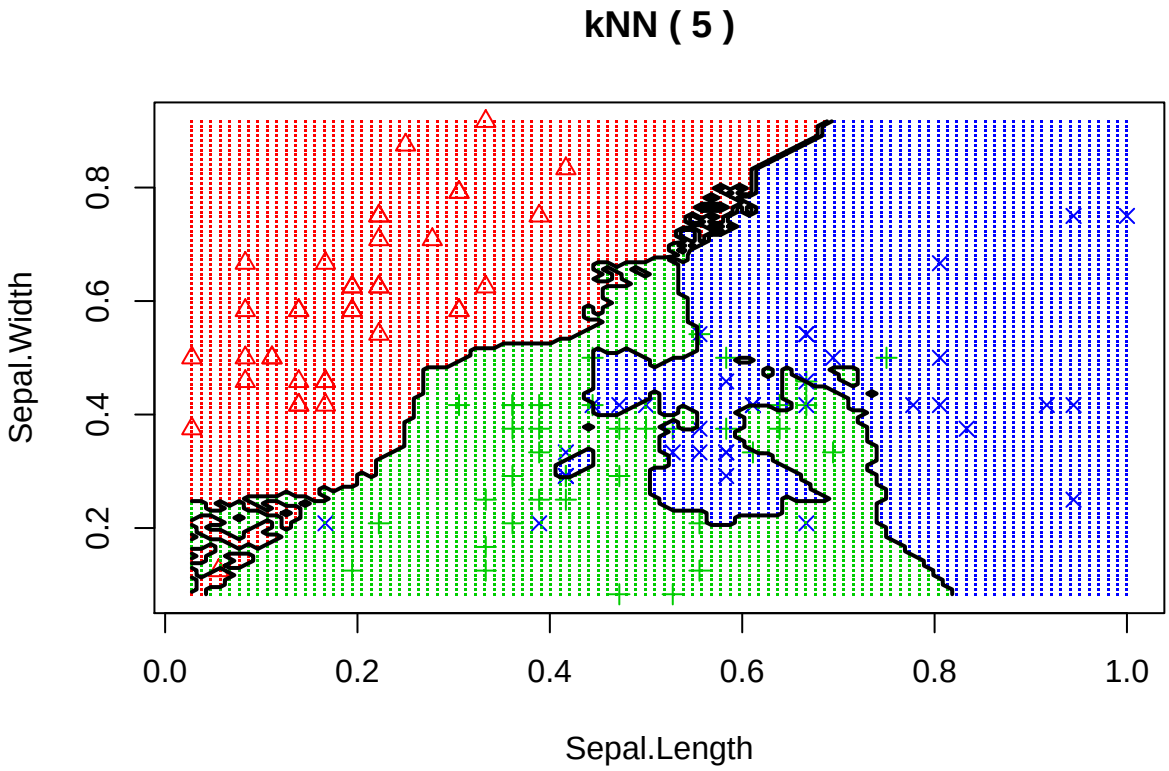
We could repeat this for various values of k.

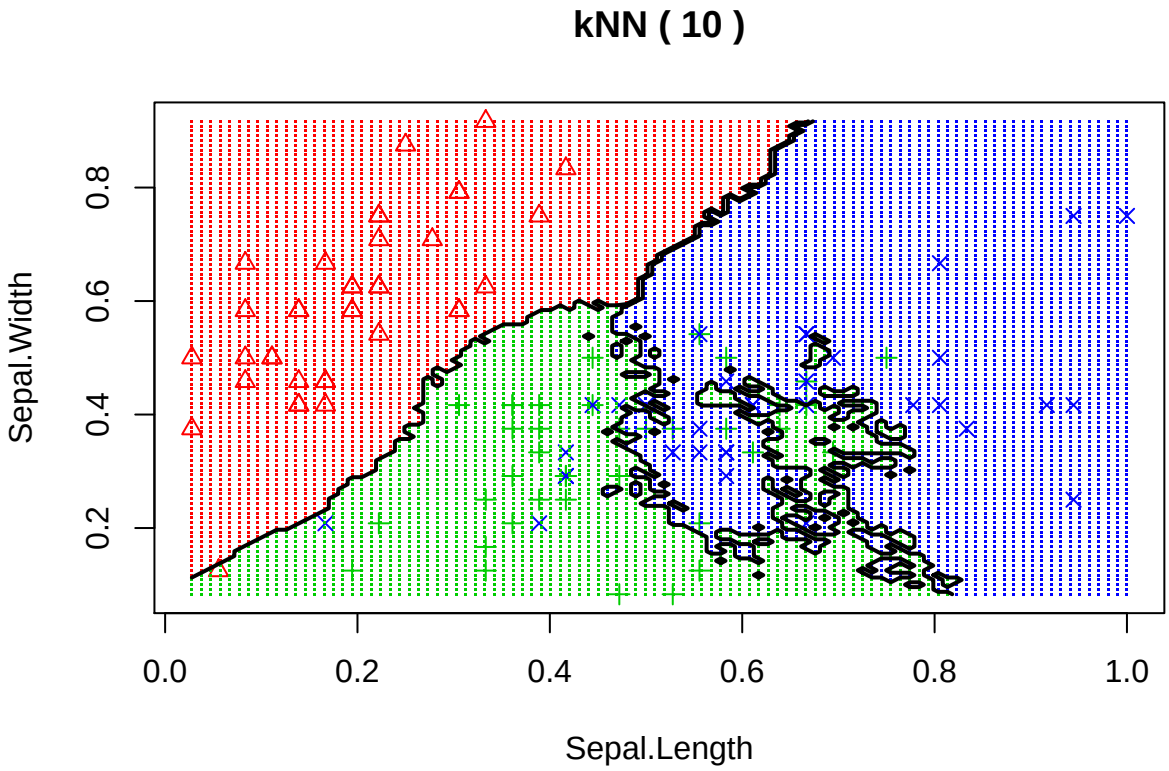
```
k <- c(1,2,3,5,10,50)
knn.model <- list(NA)
my.title <- c(NA)
for (i in 1:length(k)){
  knn.model[[i]] <- knn3(Species ~ ., data=dat.plot.train, k = k[i],use.all=FALSE)
  my.title[i] <- paste("kNN (",k[i],")")
}


# par(mfrow=c(3,2))
for (i in 1:length(k)){
  decisionplot(knn.model[[i]], data=dat.plot.train, class = "Species", main = my.title[i])
}
```

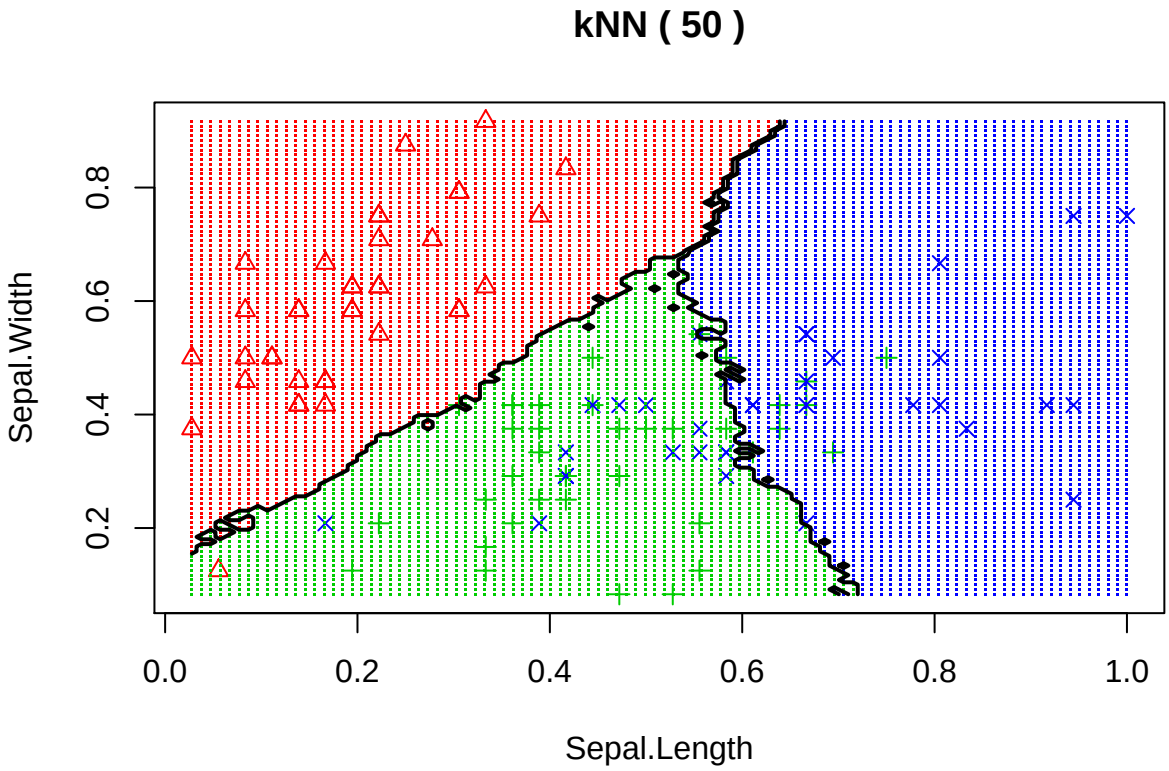

kNN (1)
Sepal.Width
0.2
0.4
0.6
0.8
0.0
0.2
0.4
0.6
0.8
1.0
Sepal.Length


kNN (2)
Sepal.Width
0.2
0.4
0.6
0.8
0.0
0.2
0.4
0.6
0.8
1.0
Sepal.Length


kNN (3)
Sepal.Width
0.2
0.4
0.6
0.8
0.0
0.2
0.4
0.6
0.8
1.0
Sepal.Length


kNN (5)
Sepal.Width
0.2
0.4
0.6
0.8
0.0
0.2
0.4
0.6
0.8
1.0
Sepal.Length


kNN (10)
Sepal.Width
0.2
0.4
0.6
0.8
0.0
0.2
0.4
0.6
0.8
1.0
Sepal.Length



It is clear from the above plots that increasing k effectively smooths the decision boundary. The small values of k will thus overfit the data. We could choose the value of k that maximizes some output statistic such as prediction accuracy.

1.2.1 Bias versus Variance Tradeoff

A fundamental tradeoff in ML algorithms involves the need to tradeoff bias for variance. Bias refers to the error introduced by approximating a real-life problem with a simple model. Variance refers to the amount our prediction would change if we estimated it using a different training set (i.e., a difference sample).

MSE (mean-square error) is a combination of the two:

$$MSE = BIAS^2 + VARIANCE \tag{1}$$

Optimal "tuning" parameters balance the tradeoff and seek to find the minimum MSE. At one extreme, the model tends to overfit (low bias, high variance). At the other extreme, the model is too simple (high bias, low variance).

To demonstrate the tradeoff, let's run a simple experiment using the IRIS data.

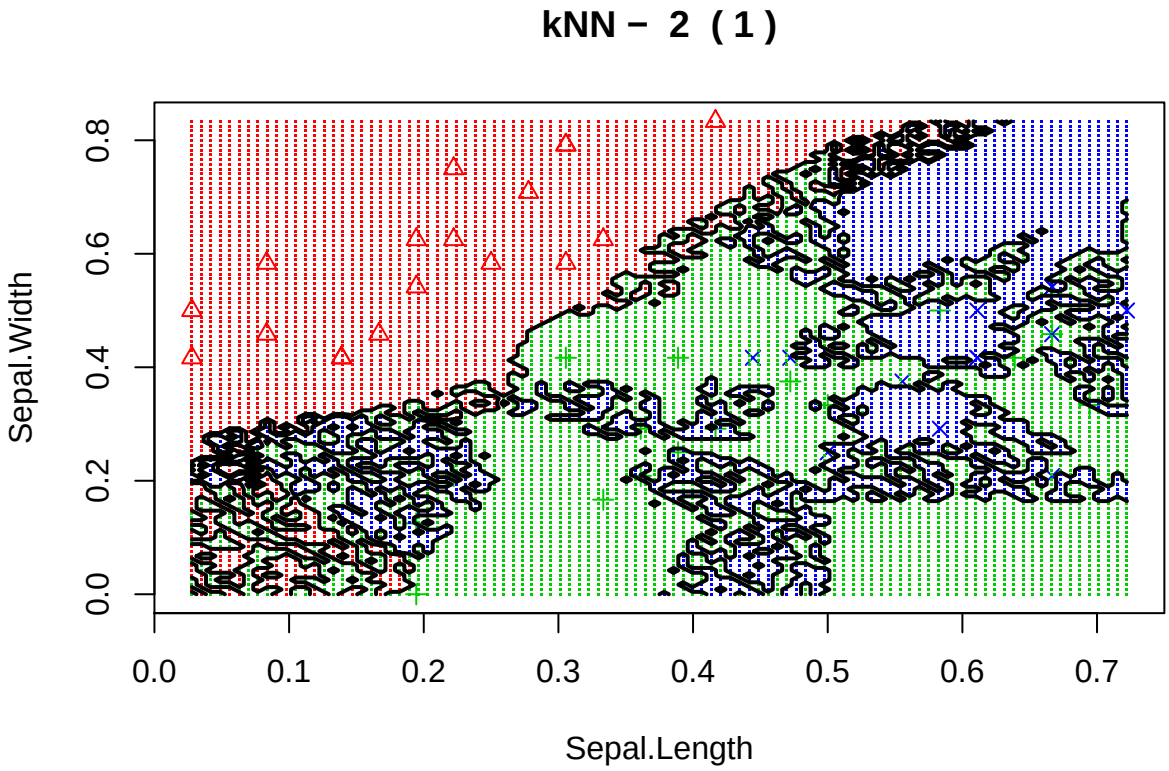
First, let's take three random samples of our data.

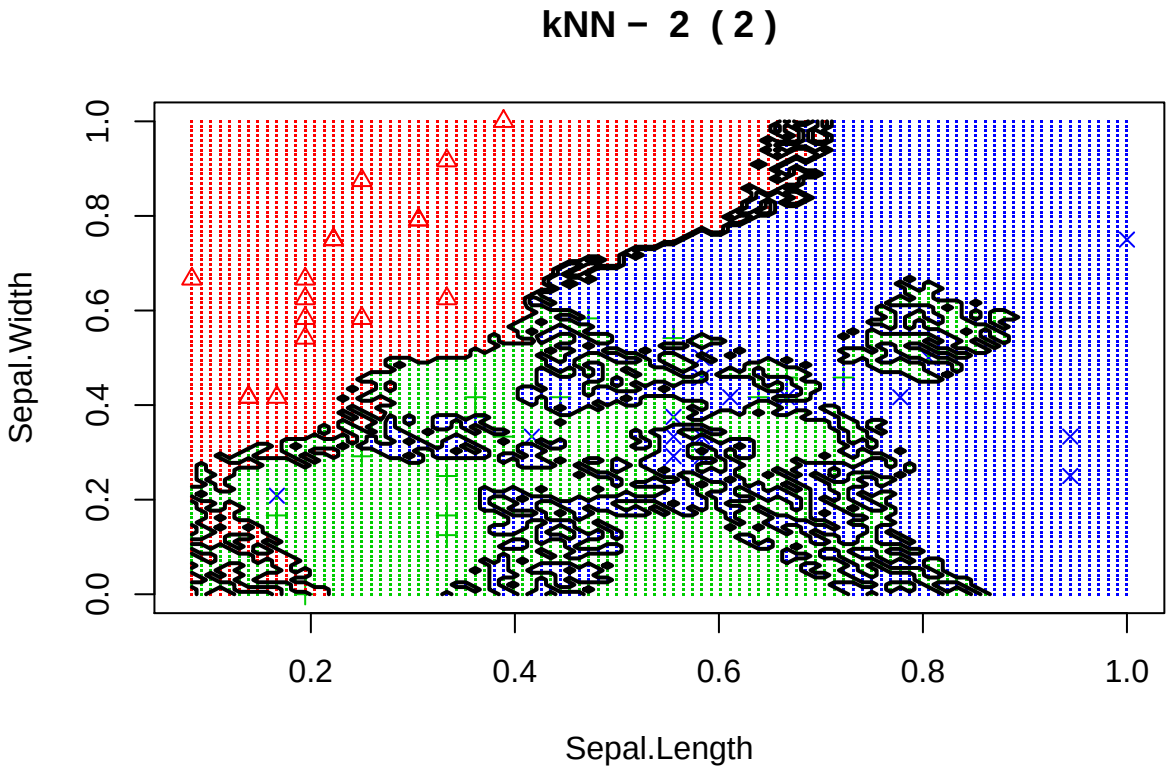
```
my.seeds <- c(1,2,1234)
my.ind <- my.dat.norm.train <- my.dat.norm.test <- my.dat.norm.train.lab <- my.dat.norm.test.lab <- lis
for (i in 1:length(my.seeds)){
  # Set up training and test data:
  set.seed(my.seeds[i])
```

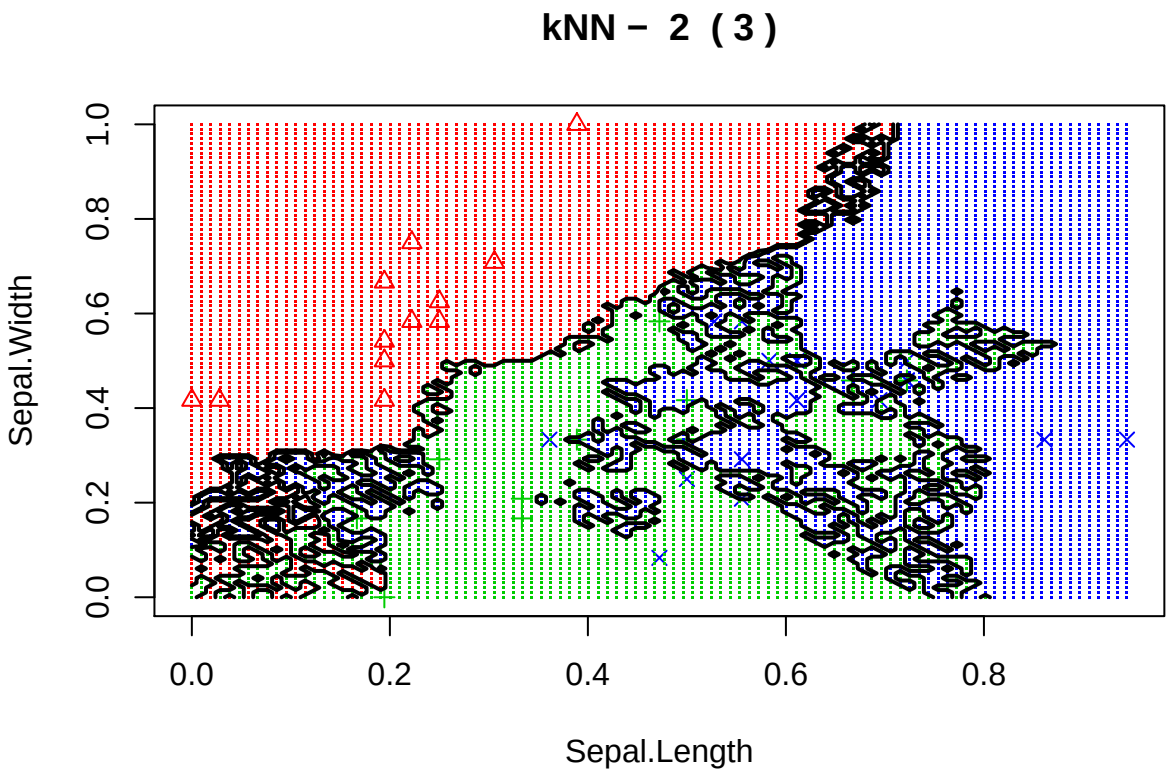
```
  # Keep 2/3 of the data to train and 1/3 to test.
  my.ind[[i]] <- sample(2,nrow(dat.norm), replace=T, prob = c(0.67,0.33))

  # Generate training data:
  my.dat.norm.train[[i]] <- dat.norm[which(my.ind[[i]]==1),]

  # Generate test data:
  my.dat.norm.test[[i]] <- dat.norm[which(my.ind[[i]]==2),]

  # Generate training Labels (response variable):
  my.dat.norm.train.lab[[i]] <- dat[which(my.ind[[i]]==1),5]

  # Generate test Labels:
  my.dat.norm.test.lab[[i]] <- dat[which(my.ind[[i]]==2),5]
}
```

Now, let's run the kNN algorithm for each data set for a low value and a high value of k and compare the differences.

```
# First, let k = 2:
k <- 2

for (i in 1:3){
  # Combine the data into a data frame:
  dat.plot.train <- data.frame(my.dat.norm.train[[i]],my.dat.norm.train.lab[[i]])
  names(dat.plot.train)[3] <- "Species"
  # head(dat.plot.train)

  # Uncomment to fit the model to the whole dataset:
  # dat.plot.train <- dat[,c(1,2,5)]

  dat.plot.test <- data.frame(my.dat.norm.test[[i]],my.dat.norm.test.lab[[i]])
  names(dat.plot.test)[3] <- "Species"

  my.title[i] <- paste("kNN - ",k," (",i,")")
  knn.model[[i]] <- knn3(Species ~ ., data=dat.plot.train,
                    k = k,use.all=FALSE)
  decisionplot(knn.model[[i]], data=dat.plot.test, class = "Species", main = my.title[i])
}
```

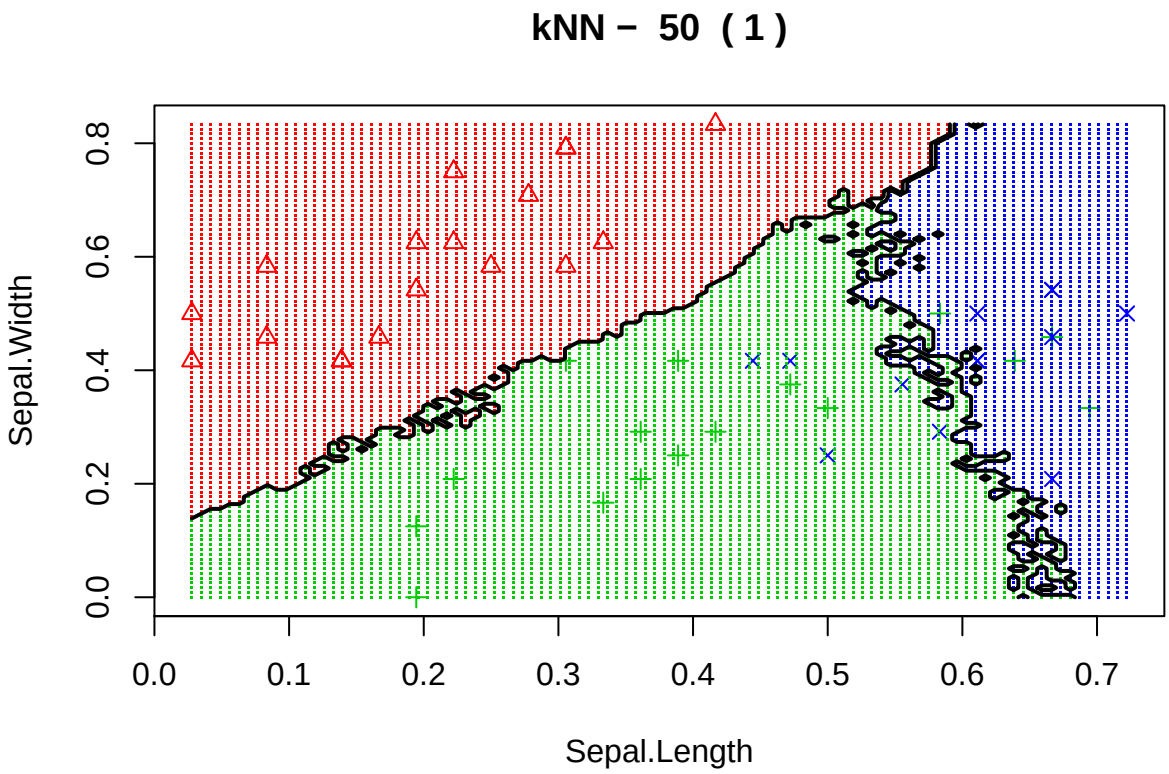

kNN − 2 (1)
Sepal.Width
Sepal.Length
0.0
0.2
0.4
0.6
0.8
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7

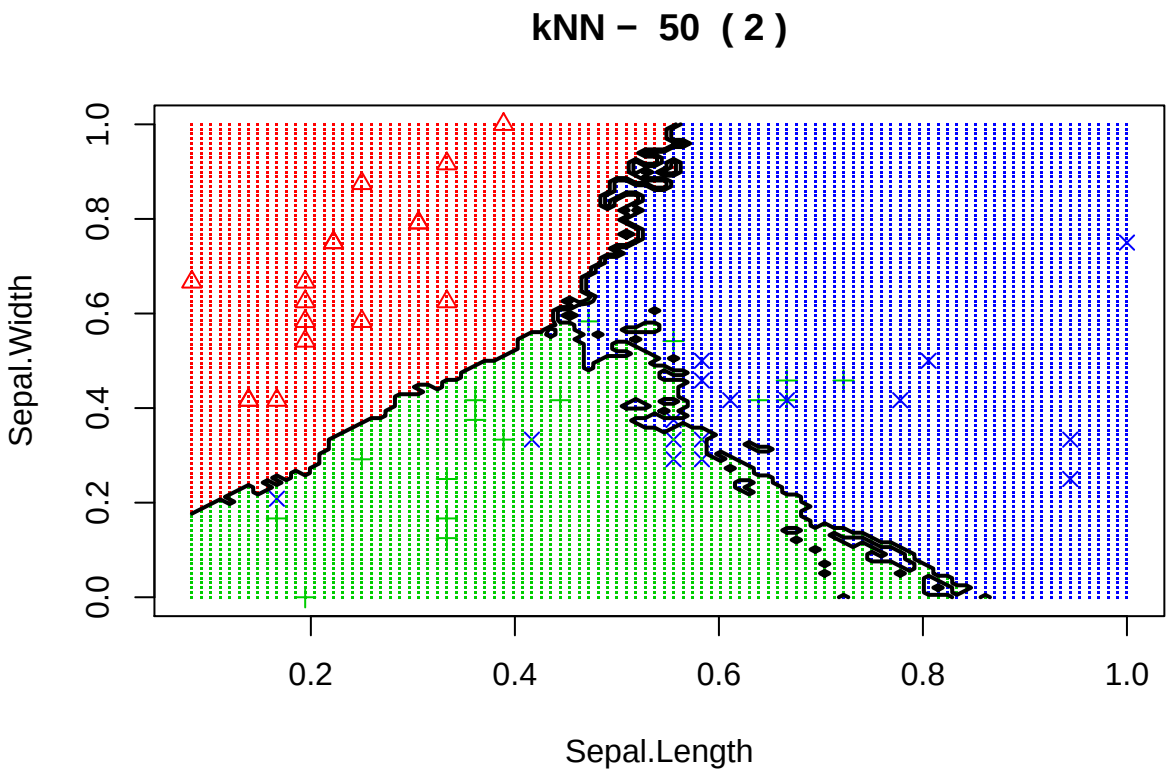

kNN − 2 (2)
Sepal.Width
0.0
0.2
0.4
0.6
0.8
1.0
0.2
0.4
0.6
0.8
1.0
Sepal.Length

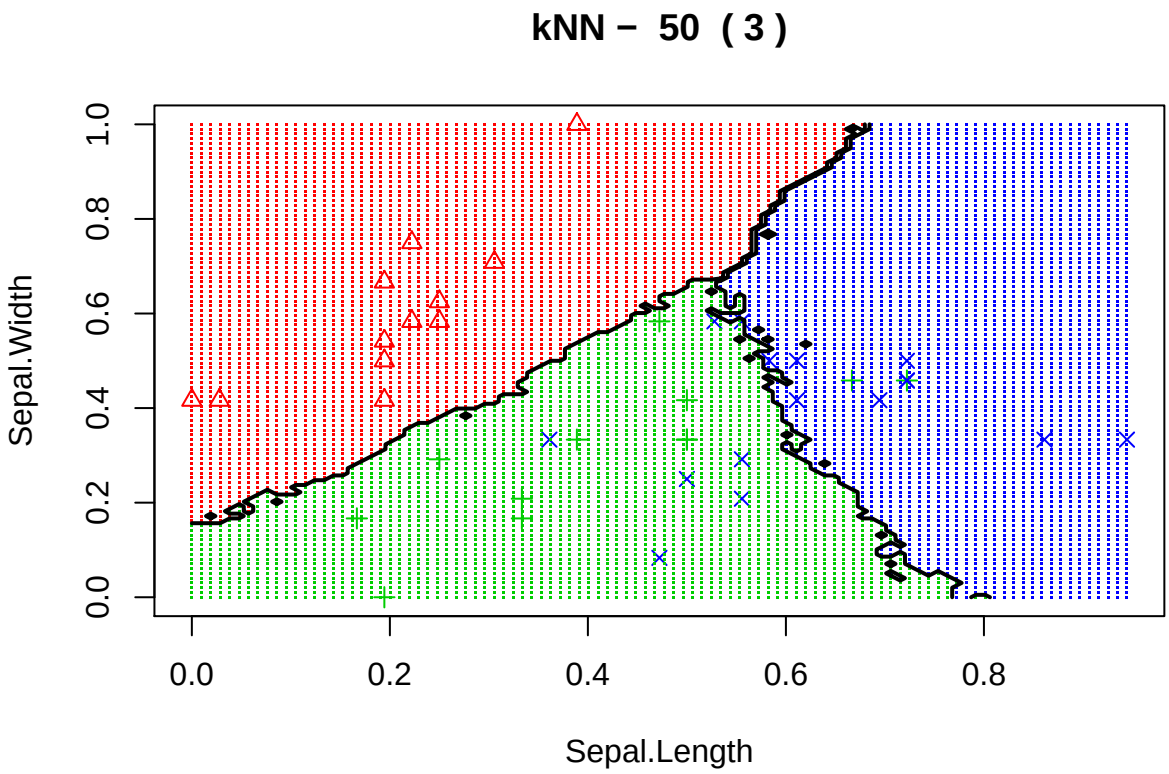


Notice the substantial differences across the plots. This suggests that the variance is very high. Let's next repeat the experiment for a larger value of k = 50.

```
# First, let k = 1:
k <- 50

for (i in 1:3){
  # Combine the data into a data frame:
  dat.plot.train <- data.frame(my.dat.norm.train[[i]],my.dat.norm.train.lab[[i]])
  names(dat.plot.train)[3] <- "Species"
  # head(dat.plot.train)

  # Uncomment to fit the model to the whole dataset:
  # dat.plot.train <- dat[,c(1,2,5)]

  dat.plot.test <- data.frame(my.dat.norm.test[[i]],my.dat.norm.test.lab[[i]])
  names(dat.plot.test)[3] <- "Species"

  my.title[i] <- paste("kNN - ",k," (",i,")")
  knn.model[[i]] <- knn3(Species ~ ., data=dat.plot.train,
                         k = k,use.all=FALSE)
  decisionplot(knn.model[[i]], data=dat.plot.test, class = "Species", main = my.title[i])
}
```


kNN − 50 (1)
Sepal.Width
Sepal.Length
0.0
0.2
0.4
0.6
0.8
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7


kNN − 50 (2)
Sepal.Width
0.0
0.2
0.4
0.6
0.8
1.0
0.2
0.4
0.6
0.8
1.0
Sepal.Length



NOw note that there is much less variation across the plots. The takeaway is that as we reduce the bias ($k \to 1$), the variance increases. This is known as over-fitting. At the other other extreme, we ignore a lot of details of the data and tend to increase the bias. However, our model becomes much more stable. This is an example of the bias-variance tradeoff.

1.3 Writing a k-NN Function

The k-NN algorithm is probably the most simplistic of the ML algorithms so we will code it ourselves as a practice case. The kNN algorithm requires three inputs:

1. k: a user-defined paramter
2. x_0: the test data observation
3. X: the training data.

Note that the we could introduce additional parameters such as a user-defined metric for calculating distance or a user-defined version of the Bayes classfication rule. However, we will keep it simple for now.

As for outputs, we definitely need the predictions. We could also output other measures such as the probability of classification or aggregated metrics of the distance between each test instance and the k nearest neighbors.

Our function will be of the form `my.knn <- function(k,train,x0){...run the algorithm; return(pred)}`. We will test our function against the `caret` package.

We will use the Euclidian distance to find the nearest neighbor. It is defined as the two norm:

$$d(x, x') = \sqrt{\sum_{i=1}^{n} (x_i - x_i')^2} \tag{2}$$

After defining the distance between our data point and all observations, we select the k closest and classify the observation.

```
# Write the distance function (assume Euclidian):
myknn <- function(this_obs,train,train.lab,k){
  datin <- train
  # d <- matrix(NA,nrow=dim(datin)[1],ncol=1)
  ties <- matrix(NA,ncol=1,nrow=dim(train)[1])

  # Compute the distances using Euclidian distance:
  # for (i in 1:dim(datin)[1]){
  #   d[i] <- norm(datin[i,] - this_obs,type="2")
  # }
  this_obs.mat <- matrix(this_obs,
                         ncol=length(this_obs),
                         nrow=dim(datin)[1],
                         byrow=T)

  d <- as.matrix(pdist(datin,this_obs))

  # Find k nearest neighbors:
  neighbors <- order(d)[1:k]

  # Count the number of min distances:
  ties <- length(which(d==min(d)))

  # Make the Prediction:
  getmode <- function(v) {
    uniqv <- unique(v)
    uniqv[which.max(tabulate(match(v, uniqv)))]
  }
  pred <- getmode(train.lab[c(neighbors)])
  # print(pred[1])
  out <- list("pred"=pred,"ties"=ties)
  return(out)
}
```

Call the function to see how it works.

```
library("pdist")
k <- c(1,2,3,5,10,50)

# Initalize a matrix for the predicitons
pred <- matrix(NA,nrow=length(dat.norm.test.lab),
               ncol=length(k)*2 + 1)
pred.names <- rep(NA,length=length(k)*2 + 1)
my.ties <- ties <- matrix(NA,ncol=length(k), nrow=length(dat.norm.test.lab))

# Populate the first columns with the actual classes
pred[,1] <- dat.norm.test.lab

# Call the function
for (j in 1:length(k)){
  k.tmp <- k[j]
  pred.names[2 + 2*(j-1)] <- paste("myknn.",k[j],sep="")
```

```
  pred.names[3 + 2*(j-1)] <- paste("knn.",k[j],sep="")
  for (i in 1:length(dat.norm.test.lab)){
    tmp.pred <- myknn(this_obs=dat.norm.test[i,],
                      train=dat.norm.train,
                      train.lab=dat.norm.train.lab,
                      k=k.tmp)
    pred[i,(2 + 2*(j-1))] <- tmp.pred$pred
    my.ties[i,j] <- tmp.pred$ties
  }
  # Get the predictions for the caret knn3 models:
  tmp <- predict(knn.model[[j]],newdata=dat.plot.test)

   pred[,(3 + 2*(j-1))] <- matrix(apply(tmp,1,which.max),ncol=1)

   # Index the instances where there was a tie:
   ties[,j] <- apply(tmp,1,function(x){length(which(x==max(x)))})
}

pred <- as.data.frame(pred)
names(pred) <- pred.names
```

We have the results for the predictions for each model stored in `pred`. Since we didn't specify how to handle ties, we will compare the results for observations both with and without ties. The "ties" can come from one of two sources: 1) there is an equal number of votes for each class (e.g., 2 votes for each class when k=2) or 2) there can be observations that are the same distance from x_0. The first class of ties is stored in `ties` and the second in `my.ties`. The first step is to convert each of those matrices into binary matrices where 1 means there is a tie and 0 means there is not.

```
# Replace the values > 1 with 1 and == 1 with 0:
for (i in 1:length(k)){
  ties[which(ties[,i]==1),i] <- 0
  ties[which(ties[,i] > 1),i] <- 1
}

for (i in 1:length(k)){
  my.ties[which(my.ties[,i]==1),i] <- 0
  my.ties[which(my.ties[,i] > 1),i] <- 1
}
```

Next, compute a matrix of instances where the methods produce the same results.

```
# Check for consistency:
pred.isequal <- matrix(0,ncol=dim(my.ties)[2],
                       nrow=dim(my.ties)[1])

for (i in 1:length(k)){
  tmp <- which(pred[,(2 + 2*(i-1))]==pred[,(3 + 2*(i-1))])
  pred.isequal[tmp,i] <- 1
}
```

Finally, we can print the proprtion of instances where both functions (`knn3()` and `myknn()`) are consistent.

```
# Print the proportion of consistent predictions:
print(apply(pred.isequal,2,mean))
```

```
## [1] 0.650 0.600 0.725 0.900 0.950 1.000
```

Notice that there are many instances where the predicitons diverge. The reason the methods differ is because of the way ties are handled. See the `knn3` help file for how that method works. We essentially ignored the issue so there could be some differences. So let's confirm that for all observations where there are not ties, the functions are consistent.

```
# Check the proportion of equal predicitons in the cases
# where there were no ties
I.ties <- pmax(my.ties,ties)
for (i in 1:length(k)){
  tmp <- which(I.ties[,i] == 0)
  print(mean(pred.isequal[tmp,i]))
}
```

```
## [1] 0.7407407
## [1] 0.7307692
## [1] 0.8518519
## [1] 1
## [1] 1
## [1] 1
```

#References