Class Prediction

Class Prediction

Here we give a brief introduction to the main task of machine learning: class prediction. In fact, many refer to class prediction as machine learning and we sometimes use the two terms interchangeably. We give a very brief introduction to this vast topic, focusing on some specific examples.

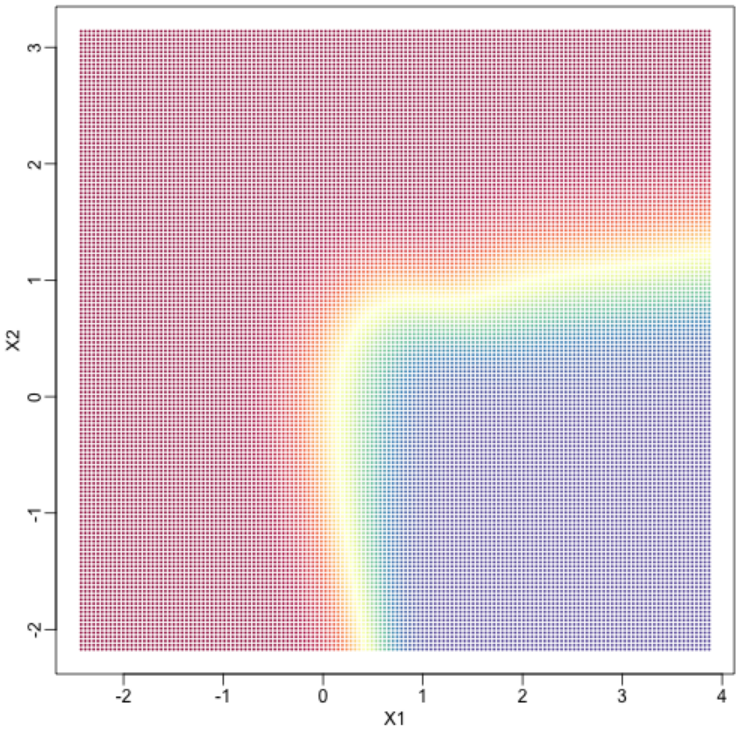
Some of the examples we give here are motivated by those in the excellent textbook *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, by Trevor Hastie, Robert Tibshirani and Jerome Friedman. A free PDF of this book can be found at the following URL:

http://statweb.stanford.edu/~tibs/ElemStatLearn/

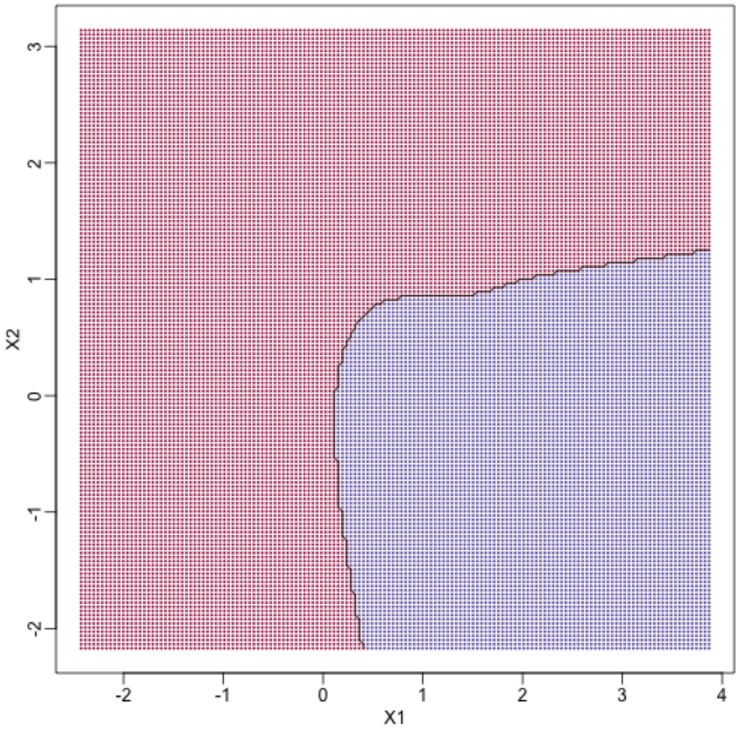
Similar to inference in the context of regression, Machine Learning (ML) studies the relationships between outcomes Y and covariates X. In ML, we call X the predictors or features. The main difference between ML and inference is that, in ML, we are interested mainly in predicting Y using X. Statistical models are used, but while in inference we estimate and interpret model parameters, in ML they are mainly a means to an end: predicting Y.

Here we introduce the main concepts needed to understand ML, along with two specific algorithms: regression and k nearest neighbors (kNN). Keep in mind that there are dozens of popular algorithms that we do not cover here.

In a previous section, we covered the very simple one-predictor case. However, most of ML is concerned with cases with more than one predictor. For illustration purposes, we move to a case in which X is two dimensional and Y is binary. We simulate a situation with a non-linear relationship using an example from the Hastie, Tibshirani and Friedman book. In the plot below, we show the actual values of $f(x_1, x_2) = E(Y \mid X_1 = x_1, X_2 = x_2)$ using colors. The following code is used to create a relatively complex conditional probability function. We create the test and train data we use later (code not shown). Here is the plot of $f(x_1, x_2)$ with red representing values close to 1, blue representing values close to 0, and yellow values in between.



If we show points for which $E(Y \mid X = x) > 0.5$ in red and the rest in blue, we see the boundary region that denotes the boundary in which we switch from predicting 0 to 1.

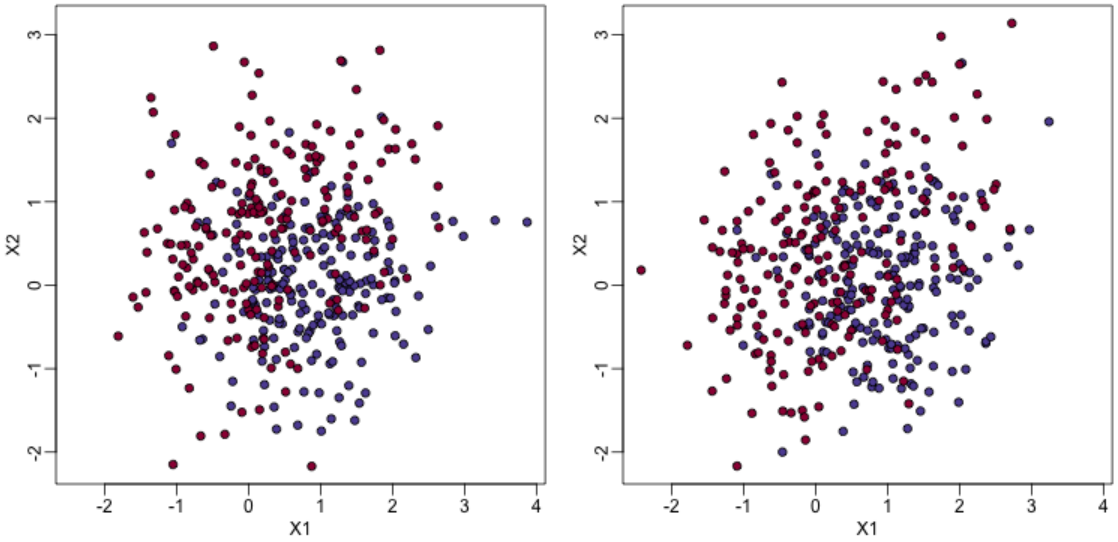


The above plots relate to the "truth" that we do not get to see. Most ML methodology is concerned with estimating $f(x)$. A typical first step in usually to consider a sample, referred to the training set, to estimate $f(x)$. We will review two specific ML techniques. First, we need to review the main concept we use to evaluate the performance of these methods.

Training and test sets

In the code (not shown) for the first plot in this chapter, we created a test and a training set. We plot them here:

```
#x, test, cols, and coltest were created in code that was not shown
#x is training x1 and x2, test is test x1 and x2
#cols (0=blue, 1=red) are training observations
#coltests are test observations
mypar(1,2)
plot(x,pch=21,bg=cols,xlab="X1",ylab="X2",xlim=XLIM,ylim=YLIM)
plot(test,pch=21,bg=colstest,xlab="X1",ylab="X2",xlim=XLIM,ylim=YLIM)
```



You will notice that the test and train set have similar global properties since they were generated by the same random variables (more blue towards the bottom right), but are, by construction, different. The reason we create test and training sets is to detect over-training by testing on a different data than the one used to fit models or train algorithms. We will see how important this is below.

Predicting with regression

A first naive approach to this ML problem is to fit a two variable linear regression model:

```
##x and y were created in the code (not shown) for the first plot
#y is outcome for the training set
X1 <- x[,1] ##these are the covariates
X2 <- x[,2]
fit1 <- lm(y~X1+X2)
```

Once we the have fitted values, we can estimate $f(x_1, x_2)$ with $\hat{f}(x_1, x_2) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2$. To provide an actual prediction, we simply predict 1 when $\hat{f}(x_1, x_2) > 0.5$. We now examine the error rates in the test and training sets and also plot the boundary region:

```
##prediction on train
yhat <- predict(fit1)
yhat <- as.numeric(yhat>0.5)
cat("Linear regression prediction error in train:",1-mean(yhat==y),"\n")
```

```
## Linear regression prediction error in train: 0.295
```

We can quickly obtain predicted values for any set of values using the `predict` function:

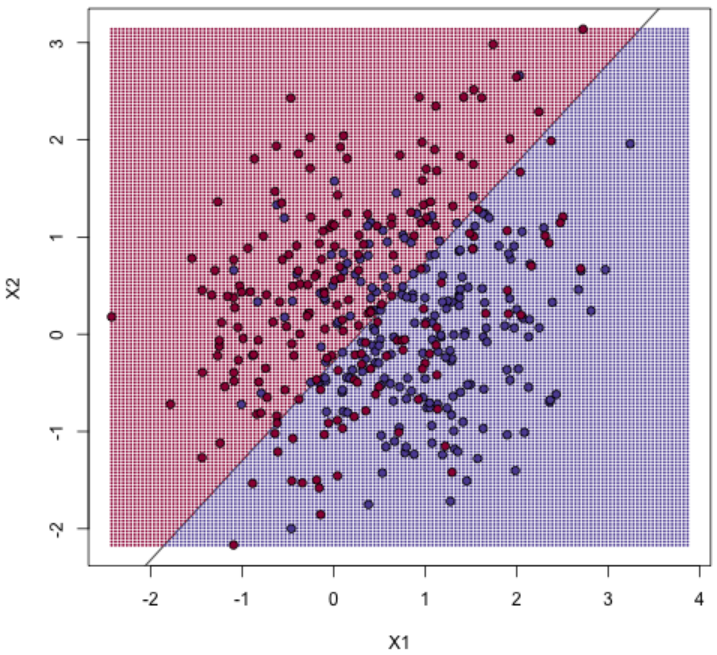
```
yhat <- predict(fit1,newdata=data.frame(X1=newx[,1],X2=newx[,2]))
```

Now we can create a plot showing where we predict 1s and where we predict 0s, as well as the boundary. We can also use the `predict` function to obtain predicted values for our test set. Note that nowhere do we fit the model on the test set:

```
colshat <- yhat
colshat[yhat>=0.5] <- mycols[2]
colshat[yhat<0.5] <- mycols[1]
m <- -fit1$coef[2]/fit1$coef[3] #boundary slope
b <- (0.5 - fit1$coef[1])/fit1$coef[3] #boundary intercept

##prediction on test
yhat <- predict(fit1,newdata=data.frame(X1=test[,1],X2=test[,2]))
yhat <- as.numeric(yhat>0.5)
cat("Linear regression prediction error in test:",1-mean(yhat==ytest),"\n")
```

```
## Linear regression prediction error in test: 0.3075
```

```
plot(test,type="n",xlab="X1",ylab="X2",xlim=XLIM,ylim=YLIM)
abline(b,m)
points(newx,col=colshat,pch=16,cex=0.35)

##test was created in the code (not shown) for the first plot
points(test,bg=cols,pch=21)
```



The error rates in the test and train sets are quite similar. Thus, we do not seem to be over-training. This is not surprising as we are fitting a 2 parameter model to 400 data points. However, note that the boundary is a line. Because we are fitting a plane to the data, there is no other option here. The linear regression method is too rigid. The rigidity makes it stable and avoids over training, but it also keeps the model from adapting to the non-linear relationship between Y and X. We saw this before in the smoothing section. The next ML technique we consider is similar to the smoothing techniques described before.

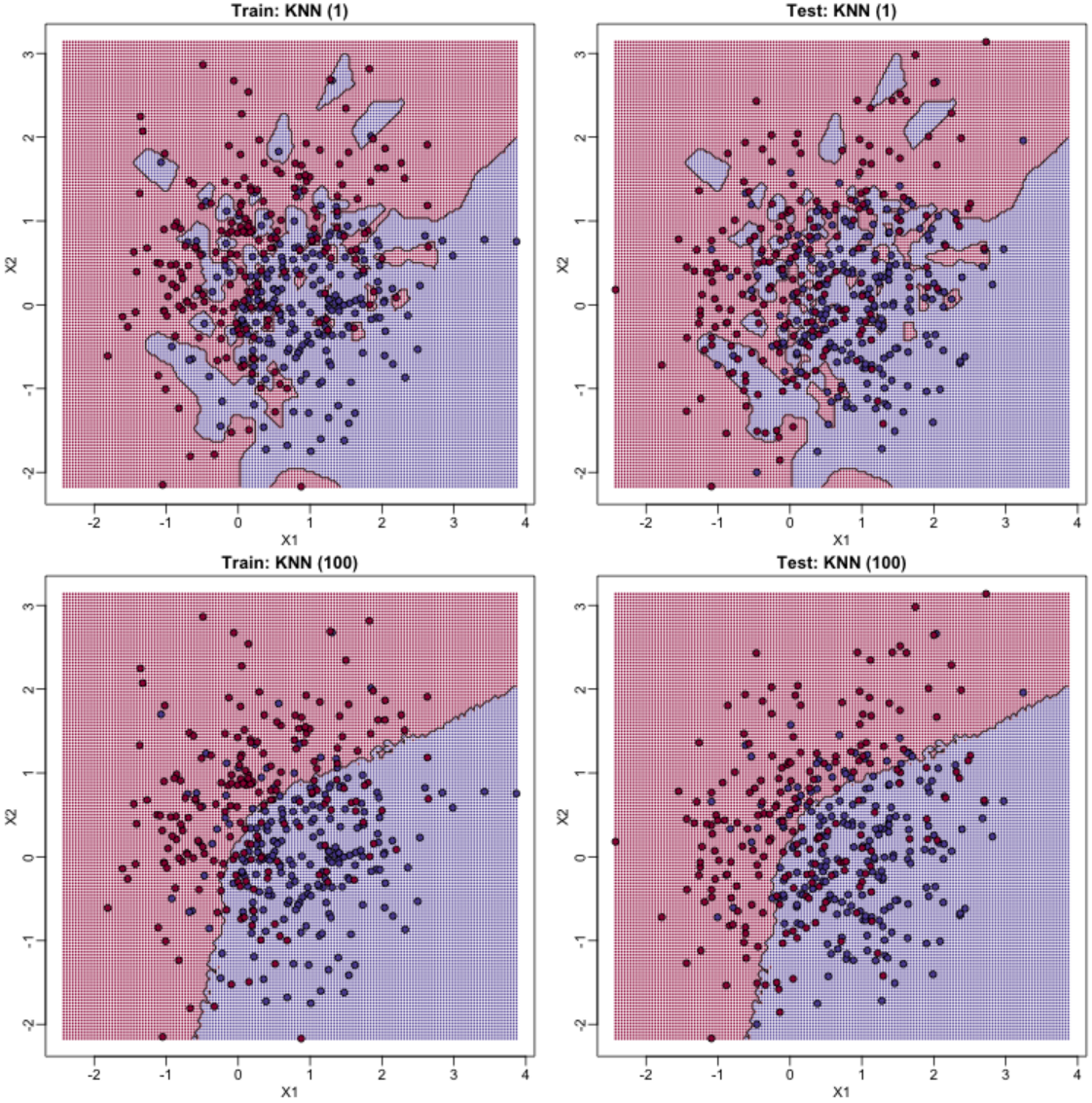
K-nearest neighbor

K-nearest neighbors (kNN) is similar to bin smoothing, but it is easier to adapt to multiple dimensions. Basically, for any point x for which we want an estimate, we look for the k nearest points and then take an average of these points. This gives us an estimate of $f(x_1, x_2)$, just like the bin smoother gave us an estimate of a curve. We can now control flexibility through k. Here we compare $k = 1$ and $k = 100$.

```
library(class)
mypar(2,2)
for(k in c(1,100)){
  ##predict on train
  yhat <- knn(x,x,y,k=k)
  cat("KNN prediction error in train:",1-mean((as.numeric(yhat)-1)==y),"\n")
  ##make plot
  yhat <- knn(x,test,y,k=k)
  cat("KNN prediction error in test:",1-mean((as.numeric(yhat)-1)==ytest),"\n")
}
```

```
## KNN prediction error in train: 0
## KNN prediction error in test: 0.375
## KNN prediction error in train: 0.2425
## KNN prediction error in test: 0.2825
```

To visualize why we make no errors in the train set and many errors in the test set when $k = 1$ and obtain more stable results from $k = 100$, we show the prediction regions (code not shown):



When $k = 1$, we make no mistakes in the training test since every point is its closest neighbor and it is equal to itself. However, we see some islands of blue in the red area that, once we move to the test set, are more error prone. In the case $k = 100$, we do not have this problem and we also see that we improve the error rate over linear regression. We can also see that our estimate of $f(x_1, x_2)$ is closer to the truth.

Bayes rule

Here we include a comparison of the test and train set errors for various values of k. We also include the error rate that we would make if we actually knew $\mathrm{E}(Y \mid X_1 = x1, X_2 = x_2)$ referred to as *Bayes Rule*.

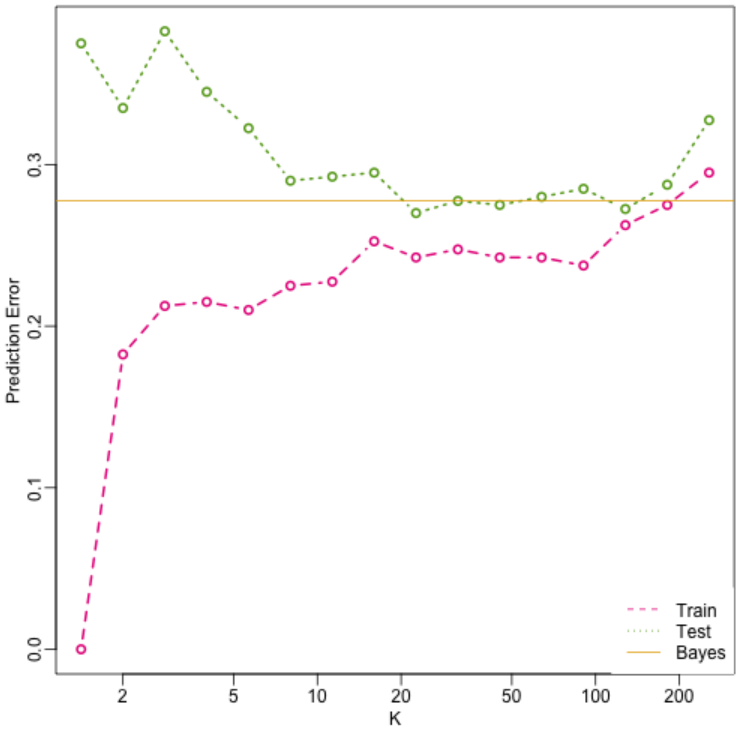
We start by computing the error rates…

```
###Bayes Rule
yhat <- apply(test,1,p)
cat("Bayes rule prediction error in train",1-mean(round(yhat)==y),"\n")

## Bayes rule prediction error in train 0.2775

bayes.error=1-mean(round(yhat)==y)
train.error <- rep(0,16)
```

```
test.error <- rep(0,16)
for(k in seq(along=train.error)){
  ##predict on train
  yhat <- knn(x,x,y,k=2^(k/2))
  train.error[k] <- 1-mean((as.numeric(yhat)-1)==y)
  ##prediction on test
  yhat <- knn(x,test,y,k=2^(k/2))
  test.error[k] <- 1-mean((as.numeric(yhat)-1)==y)
}
```

… and then plot the error rates against values of k. We also show the Bayes rules error rate as a horizontal line.

```
ks <- 2^(seq(along=train.error)/2)
mypar()
plot(ks,train.error,type="n",xlab="K",ylab="Prediction Error",log="x",ylim=range(c(test.error,train.error)))
lines(ks,train.error,type="b",col=4,lty=2,lwd=2)
lines(ks,test.error,type="b",col=5,lty=3,lwd=2)
abline(h=bayes.error,col=6)
legend("bottomright",c("Train","Test","Bayes"),col=c(4,5,6),lty=c(2,3,1),box.lwd=0)
```



Note that these error rates are random variables and have standard errors. In the next section we describe cross-validation which helps reduce some of this variability. However, even with this variability, the plot clearly shows the problem over over-fitting when using values lower than 20 and under-fitting with values above 100.
