Lab 2

Assessing Regression Model

Dr. Irene Vrbik

2023-09-19

Summary *This lab will outline some of the basics in R, as well as introduce R markdown as a file format for making dynamic documents for assignments. This lab assignment is* ***not*** *for marks. In the future, your assignments will involve creating Rmd files and submitting the resulting HTML outputs as your work. However,* ***for this lab, there's no need to submit anything.*** *If you are confident in your ability to achieve the learning outcomes, proceed directly to the exercises and confirm that you can complete them successfully.*

Learning outcomes

By the end of this lab students will be able to:

- Calculate MSE for both testing and training data in the Regression Setting
- gain an understanding of the bias-variance trade-off (and how they relate to the flexibility of the model)
- gain an understanding of the relationship and general trends of the MSE_{tr} vs. MSE_{te}
- Fit a simile linear regression model
- Fit a multiple linear regression model

Topics roughly map to lecture 3, 4, and 5

For reproducibility purpose, I have set a seed (`set.seed()`) for R's random number generator. If you follow this code for creating these simulations your random numbers (and therefore subsequent images and calculations) should be identical[1] to what I produced within this document.

Introduction

Recall our general model for statistical learning

$$Y = f(X) + \epsilon \tag{1}$$

where:

- X are our inputs
- Y is the numeric output (at least in this setting)
- ϵ is the error term (independent of X and with mean 0)
- f represents the systematic information X provides about Y.

Our *goal* is to find an $\hat{f}(X)$ that approximates the true function $f(x)$ as well as possible. In this lab we will be discussing metrics for assessing model accuracy in the supervised setting. This corresponds to Section 2.2 of the ISLR2 textbook. By the end of it you should gain an understanding of the

- bias-variance trade-off (and how they relate to the flexibility of the model)
- the relationship and general trends of the MSE_{tr} vs. MSE_{te}

Measuring the quality of fit

For a continuous response variable, in the supervised setting, a natural measure of quality is how close our predict $\hat{y} = \hat{f}(x)$ values compare to the "true" ys. In this context, the most commonly-used measure is the mean squared error (MSE). In words, the MSE is the average of all of the squared differences between the true values y_i and the the predicted values $\hat{f}(x_i)$ (the smaller the better!).

[1] R adjusted `set.seed()` and other random number generator defaults starting in R version 3.6.0. As such, please ensure the R installation you have is >= 3.6.0. — you can see what version is installed in the header at startup, or type the following into your console: `R.Version()$version.string`).

MSE for continuous response

When MSE is calculated using the training data $\mathbf{X}_{Tr} = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\} = \{x_i, y_i\}_1^n$ we call it the **training MSE**

$$MSE_{Tr} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2$$

When MSE is calculated using the testing data $\mathbf{X}_{Te} = \{(x_{n+1}, y_{n+1}), ..., (x_{n+m}, y_{n+m})\} = \{x_i, y_i\}_1^m$ we call it the **test MSE**:

$$MSE_{Te} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{f}(x_i))^2$$

where $\hat{f}(x_i) = \hat{y}_i$ is the prediction for the ith input x_i, and y_i is the response actually observed (ie "truth").

Recall from lecture that we are more interested on how our models perform on the *testing* set rather than the *training* set. In other words we would like our models to perform well on data it has never "seen" before. Hence desirable models are those which provide the *lowest* ***test*** *MSE*, as opposed to the lowest training MSE. If you have access to test data (i.e. a set of data that have not been used to train the model), we can simply evaluate MSE_{Te} for all methods and select the one which yields the lowest value.

Sometimes we are presented with the situation wherein no test data is available in which case we cannot calculate the MSE_{Te}. In this case you might be tempted to choose the model which minimizes the MSE_{Tr}. As discussed in lecture (and exemplified in ISLR 2.21, 2.12), the model which obtains the lowest MSE_{Te} is not necessarily the model which obtains the lowest MSE_{Tr}.

Side note: Later on in the course we will look at how we can use *cross validation* to fudge a testing data set as a means of providing an *estimate* the test MSE. In our considerations today, we will look at simulated data in which we can easily generate new testing observations and calculate MSE_{Te}

Bias-Variance Tradeoff

For a given test point x, it can be shown[2] that the expected test MSE, is given by:

$$\text{E}[(y - \hat{f}(x))^2] = \text{Var}(\hat{f}(x)) + (\text{Bias}[\hat{f}(x)])^2 + \text{Var}(\epsilon) \tag{2}$$

That is, the *reducible* error in the MSE can actually be decomposed into two competing forces:

[2] this post does a good job of explaining it, but it is heavier on the theory I expect you to follow for this course

- $\text{Var}(\hat{f}(x))$ which indicates how much $\hat{f}$ changes from training set to training set (models that are very flexible will have high variance because they fit too closely to the data at hand <– overfitting)
- $\text{Bias}[\hat{f}(x)])$ which represents the difference between the true model and the average value of all predictions at x across all possible training sets (models that are too simple to explain the complex phenomenon will systemically be "off the mark" <– underfitting)

$$\text{Bias}[\hat{f}(x)]) = \text{E}[\hat{f}(x)] - f(x)$$

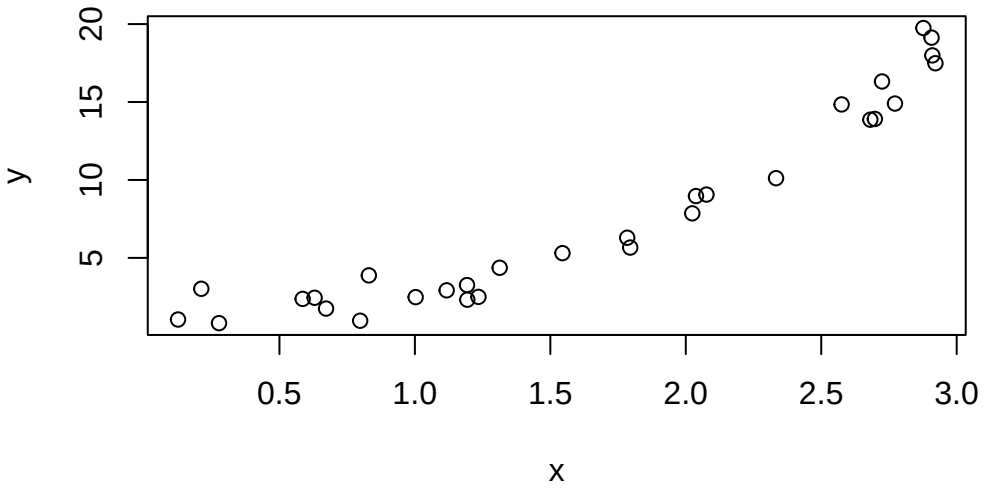
Simulation 1

To get a sense of what is referred to as the bias-variance trade-off, let's consider the contrived example in which we *know* what the true generating function f looks like. First, we'll simulate our training data by uniformly generating X values, and then assuming an exponential relationship between X and Y with standard normal (mean 0, variance 1) errors. In other "words",

$$Y = \exp(X) + \epsilon \tag{3}$$

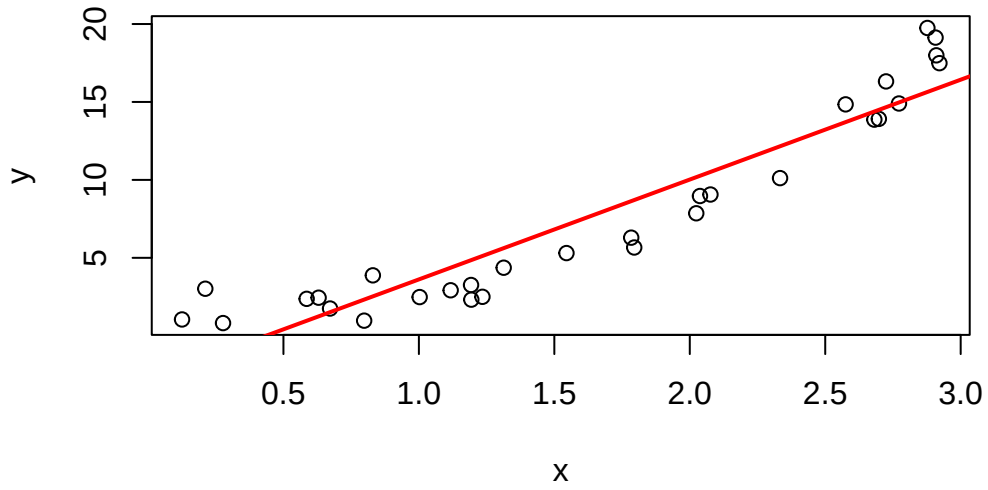
where $\epsilon \sim \text{Normal}(\mu = 0, \sigma = 1)$.

```
set.seed(23418)
x <- sort(runif(30,0,3))
y <- exp(x) + rnorm(length(x))
plot(x, y)
```



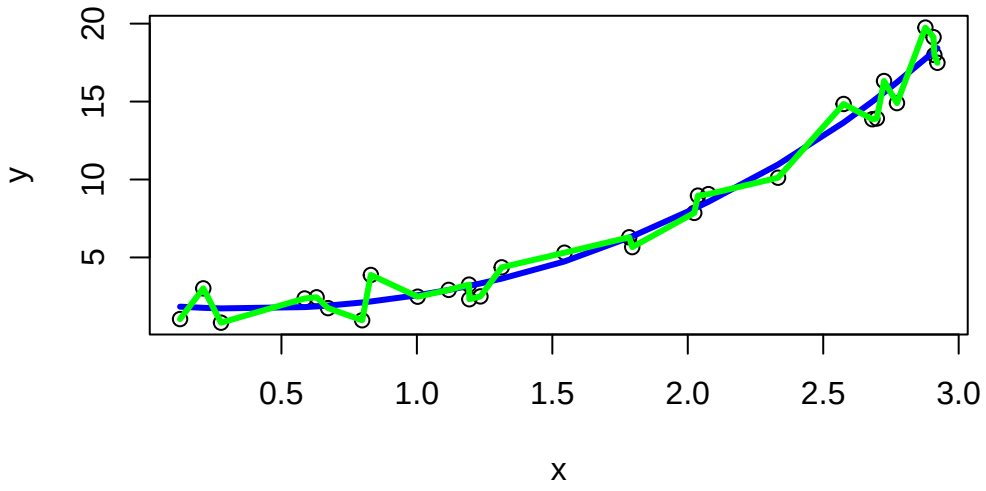
Now fit a standard linear model, and add the fitted line ($\hat{f}$) to the plot in red:

```r
linmod <- lm(y ~ x)
plot(x , y)
abline(linmod, col="red", lwd=2)
```



Next, we fit a *local polynomial* model using the `loess` function in R and add it in blue. We are unlikely to cover these in this course — just consider it a relatively flexible model. The `span` argument controls the degree of smoothing. We we fit a local polynomial with medium flexibility (store in `poly1`) and one with high flexibility (which we'll call `poly2`). The later approximately mimics the "connect-the-dots" scenario. N.B. you will get some errors when you fit this model (essentially warning us that this model is not reasonable) but just ignore them.

```r
poly1 <- loess(y~x, span=0.75)
poly2 <- loess(y~x, span=0.1)
plot(x, y)
lines(x, predict(poly1), col="blue", lwd=3)
lines(x, predict(poly2), col="green", lwd=3)
```



Now we can calculate the *training MSE* for each model by utilizing the `predict()` function. By default, `predict()` simply provides $\hat{y}$ for the previously observed predictors (aka, the training data). We'll use `round()` to print the MSE with 2 decimal places.

```
trmse_red <- mean((y - predict(linmod))^2)
trmse_blue <- mean((y - predict(poly1))^2)
trmse_green <- mean((y - predict(poly2))^2)
round(trmse_red, 2)
round(trmse_blue, 2)
round(trmse_green, 2)
```

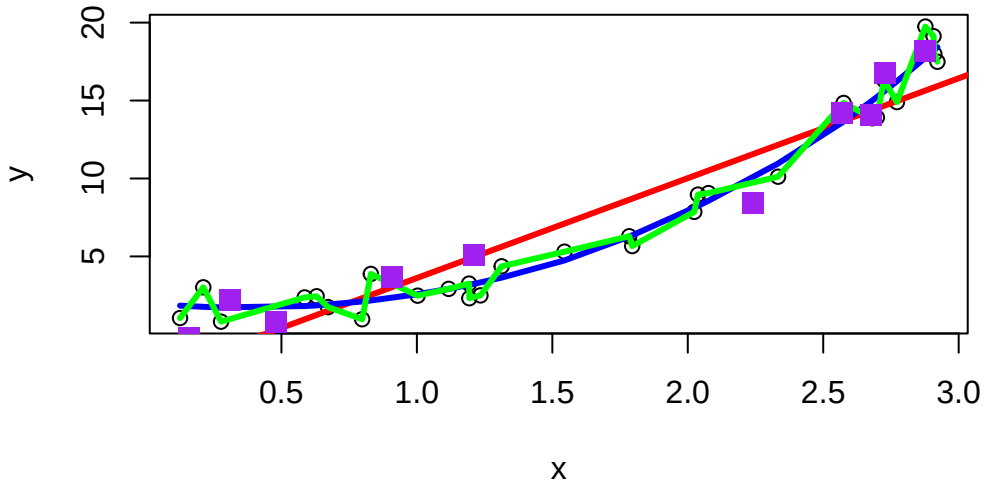
I have suppressed the output above and stored it in a table:

Model	MSE_{train}
linear (red)	4.37
med flexibility (blue)	0.83
connect-the-dots (green)	0

Now we will generate 10 new points from the same simulation and plot them alongside our three fitted models:

```
set.seed(41368)
xnew <- sort(runif(10,0,3))
ynew <- exp(xnew) + rnorm(length(xnew))
plot(x, y)
abline(linmod, col="red", lwd=3)
```

```
lines(x, predict(poly1), col="blue", lwd=3)
lines(x, predict(poly2), col="green", lwd=3)
points(xnew, ynew, pch=15, col="purple", cex=1.5)
```



We can calculate the *test MSE* by giving the `predict` function the new data. Note that `predict` specifically wants new data as a `data.frame` structure, and will not accept a vector object for `newdata` — we will run into little issues like this throughout the course.

```
temse_red <- mean( (ynew - predict(linmod, data.frame(x = xnew)))^2 )
temse_blue <- mean( (ynew - predict(poly1, data.frame(x = xnew)))^2 )
temse_green <- mean((ynew - predict(poly2, data.frame(x = xnew)))^2 )
temse_red
temse_blue
temse_green
```

Model	MSE_{test}	MSE_{train}
linear (red)	3.32	4.37
med flexibility (blue)	1.56	0.83
connect-the-dots (green)	4.29	0

As noted in lecture, most of the results follow the general rule that testing MSE's will be larger than training MSE's, though by chance this is not currently holding for the linear model.

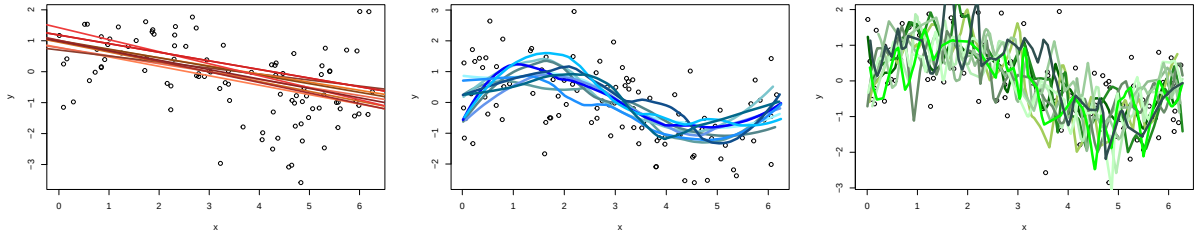
Exercise Rerun this code with a different seed and see how the results change with additional simulations!

Simulation 2

Next let's simulate data by uniformly generating X values between 0 and 2π, and generate Y values using:

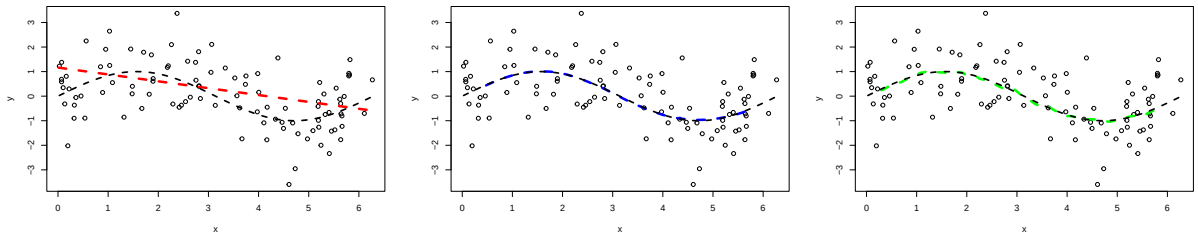
$$Y = \sin(X) + \epsilon \tag{4}$$

where $\epsilon \sim \text{Norm}(\mu = 0, \sigma = 1)$. To get a sense of what "bias" and "variance" mean in the context of our fitted models, let's look at fitted each model 100 different training sets. In the first figure we plot the first 10 fitted models, while the second figure plots the *average prediction* of each method over all of the 100 fitted models for a single test set X_{Te} (remember each fit was created using a different training set). Notice:



- The green model (which is the most flexible) has the highest variability. This model is overfitting, that is, it follows the observations too closely. Thus a change in training set causes the estimate $\hat{f}$ to change considerably. Notice that while the green model is highly variable, it has low bias. That is, even though a *single* fitted model does not close to our generating function f, *on average* the estimate is close the truth.
- In contrast, the red linear model (which is relatively inflexible) has low variance. That is to say, with each new training set it sees, the fitted model does not change much. While this model has low variance, it has high bias. Consequently, it will systematically underestimate/overestimate the true f (in black) for certain values of X. For example, we can see in the figure below that between the range of $x = 4$ to 5 model is not capturing the "dip", thus this model will systematically overestimate y in that region on average. This is a classic example of underfitting; no matter how much data this model learns from, it can never be persuaded away from a erroneous solution.
- The blue model strikes a nice balance in between. It simultaneously has low variance and low bias.

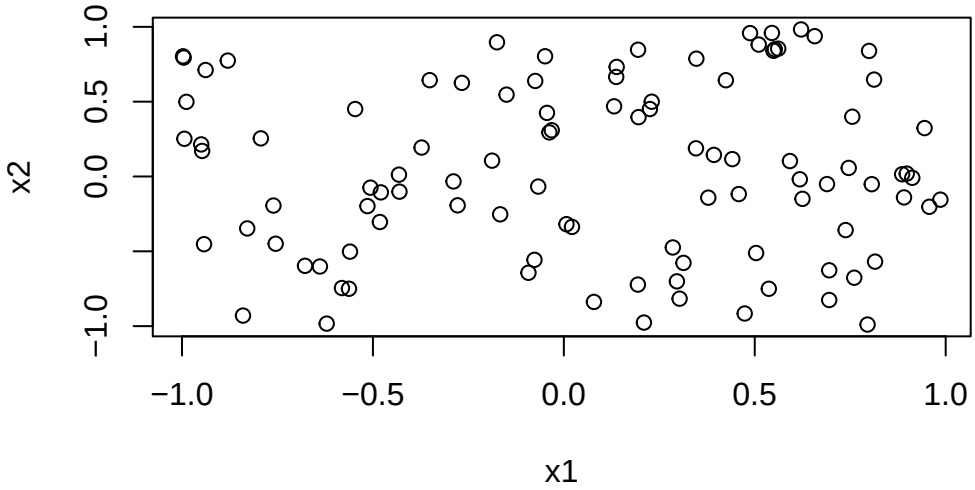
Question Which of these models would you expect to have the lowest test MSE?



Assessing Models: Classification

Let's now recreate the classification example shown in Lecture 3. Here we'll simulate an underlying classification model where the continuous variables are uniformly distributed...

```
set.seed(4623)
x1 <- runif(100, -1, 1)
x2 <- runif(100, -1, 1)
plot(x1, x2)
```



and the group classification probabilities are associated with the X2 variable. Specifically, the probability of being in "Class 1" is equal to the observation's value at X2 or 0 when X2 is negative. Note that loops can, and should, generally be avoided in R. We will often break that rule in lab in the interest of simplicity, plus the fact that most of you will be more comfortable with loops rather than R-specific solutions. First, let's initialize a classification vector of NAs (missing values)

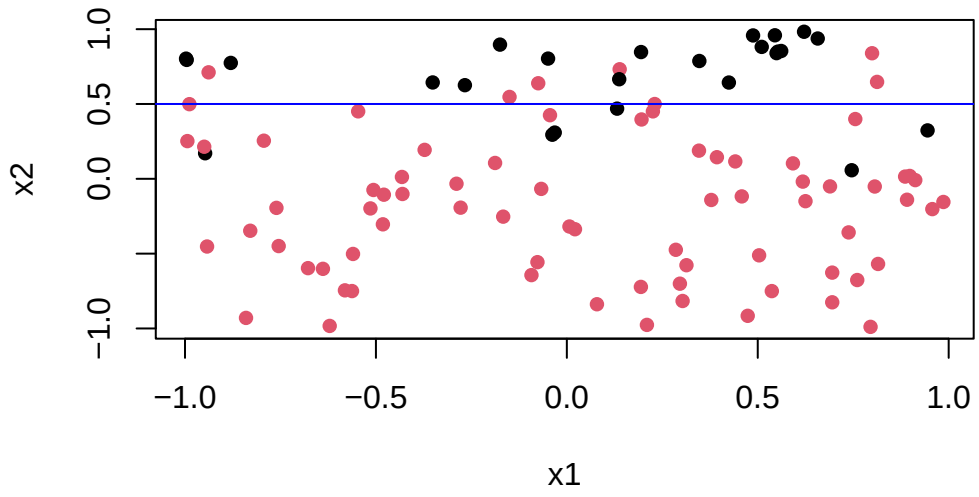
```
clas <- rep(NA, length(x2))
```

Now we will replace each element of that vector with an "observed" classification sampled from the probability scheme mentioned above.

```
for(i in 1:length(x2)){
  clas[i] <- sample( c(1, 2) , size = 1, prob =c(max(0, x2[i]), min(1 - x2[i], 1)))
}
```

Now we can plot the points according to their observed classification, and include the Bayes Classifier boundary at X2 = 0.5 in blue.

```
plot(x1, x2, col = clas, pch=16)
abline(h = 0.5, col="blue")
```



Furthermore, we can calculate what the observed classification error would be for the Bayes Classifier (that is, the correct classification model). That model would tell us that if X2 > 0.5, we should classify as group "1" (black) and if X2 < 0.5 we should classify as group "2" (red). So we can tabulate that

```
table(x2 > 0.5, clas)
```

```
       clas
         1  2
  FALSE  6 69
  TRUE  19  6
```

And easily pull the misclassifications from that table by looking at the diagonal (in this particular case).

```
sum(diag(table(x2 > 0.5, clas))) / length(x2)
```

```
[1] 0.12
```

This suggests that even using the correct model would result in 12% misclassification on this particular training set. Note that some models can beat this misclassification rate on the training set by overfitting. Let's explore k-nearest neighbours (KNN) classification. First, we need to install another package. Enter the following in the command-line.

```
install.packages("class")
```

Now, we'll fit KNN for $k = 15$, $k = 10$, and $k = 1$. Take a look at the help file for `knn` by entering `?knn` in the console. You can see some discussion surrounding ties in there (which we covered a little bit in lecture). The `knn` function takes both a training set and a testing set. We'll give it `x1` and `x2` for both the training set and the testing set to look at MSE for training.

```
library("class")
mod15 <- knn(cbind(x1,x2), cbind(x1, x2), clas, k=15, prob=TRUE)
table(clas, mod15)
```

```
    mod15
clas  1  2
   1 16  9
   2  5 70
```

```
#now misclassifications are on the off diagonal, so...
(length(clas)-sum(diag(table(clas, mod15))))/length(clas)
```

```
[1] 0.14
```

```
mod10 <- knn(cbind(x1,x2), cbind(x1, x2), clas, k=10, prob=TRUE)
(length(clas)-sum(diag(table(clas, mod10))))/length(clas)
```

```
[1] 0.15
```

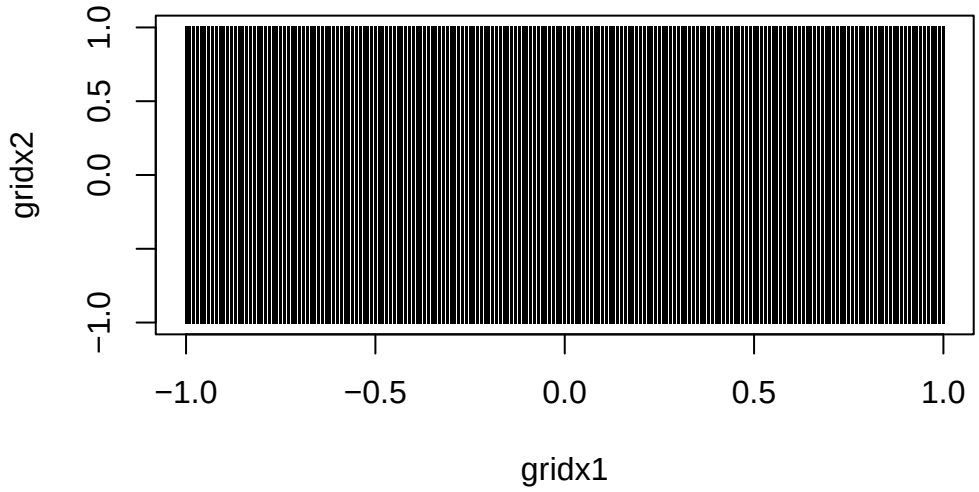
```
mod1 <- knn(cbind(x1,x2), cbind(x1, x2), clas, k=1, prob=TRUE)
(length(clas)-sum(diag(table(clas, mod1))))/length(clas)
```

```
[1] 0
```

So misclassification rates of 14%, 15%, and 0% on the training data. Compare to Bayes Classifier! Clearly overfitting at $k = 1$.

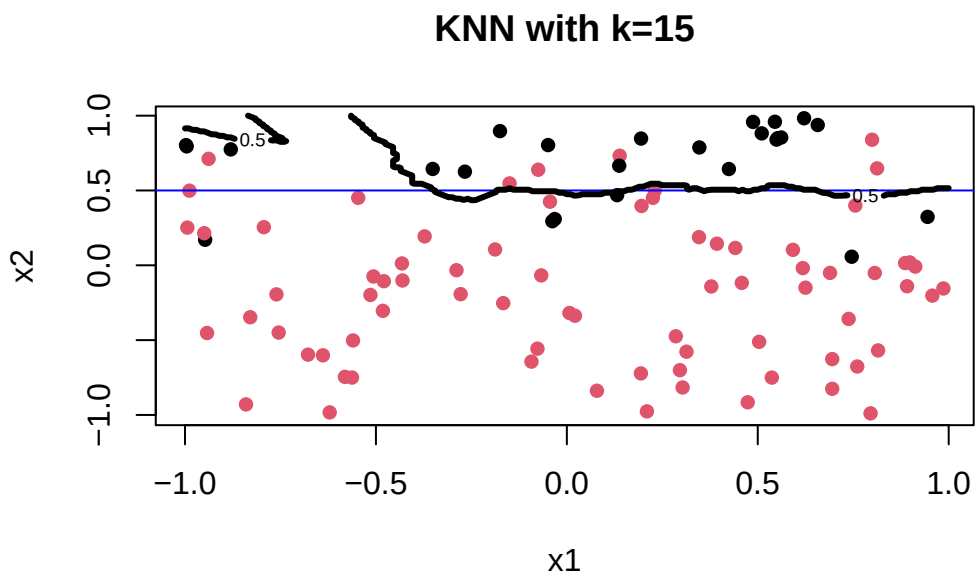
Now let's give a big grid of values for the testing set in order to eventually recreate the contours shown in lecture.

```
gridseq <- seq(-1, 1, 0.01)
gridx1 <- rep(gridseq, each=length(gridseq))
gridx2 <- rep(gridseq, length(gridseq))
#We are putting a point at each 0.01 increment in both x1 and x2
plot(gridx1, gridx2, pch=".")
```

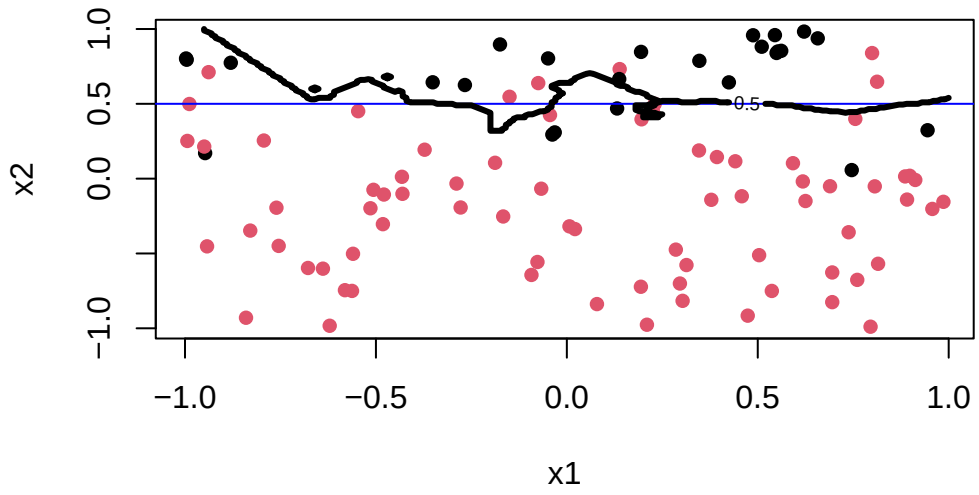



```
mod15 <- knn(cbind(x1,x2), cbind(gridx1, gridx2), clas, k=15, prob=TRUE)
mod10 <- knn(cbind(x1,x2), cbind(gridx1, gridx2), clas, k=10, prob=TRUE)
mod1 <- knn(cbind(x1,x2), cbind(gridx1, gridx2), clas, k=1, prob=TRUE)
```

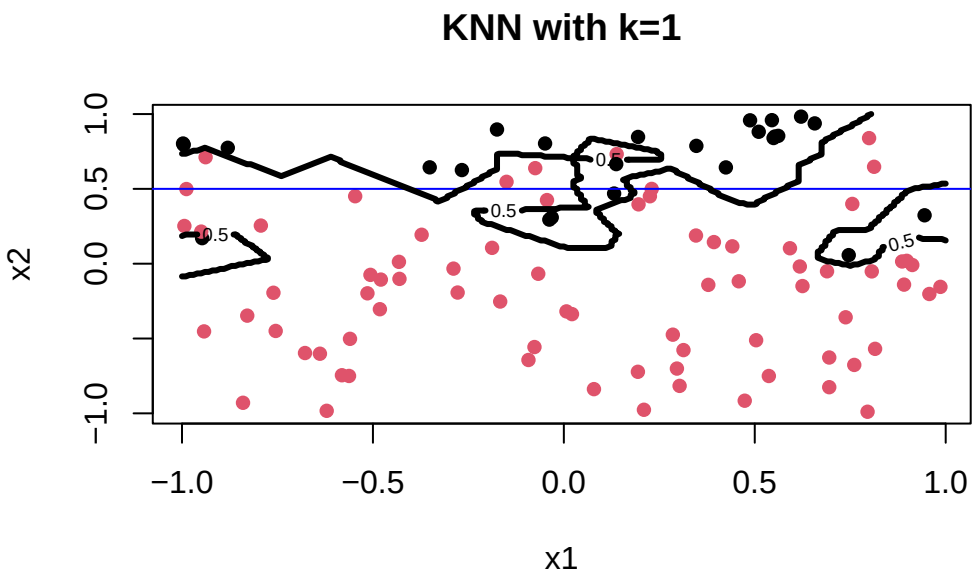
```
plot(x1, x2, col = clas, pch=16, main="KNN with k=15")
abline(h = 0.5, col="blue")
conprob <- attr(mod15, "prob")
conpoints <- ifelse(mod15==1, conprob, 1-conprob)
conmat <- matrix(conpoints, length(gridseq), length(gridseq))
contour(gridseq, gridseq, t(conmat), levels=0.5, nlevels=1, add=TRUE, col="black", lwd=3)
```




```
plot(x1, x2, col = clas, pch=16, main="KNN with k=10")
abline(h = 0.5, col="blue")
conprob <- attr(mod10, "prob")
conpoints <- ifelse(mod10==1, conprob, 1-conprob)
conmat <- matrix(conpoints, length(gridseq), length(gridseq))
contour(gridseq, gridseq, t(conmat), levels=0.5, nlevels=1, add=TRUE, col="black", lwd=3)
```

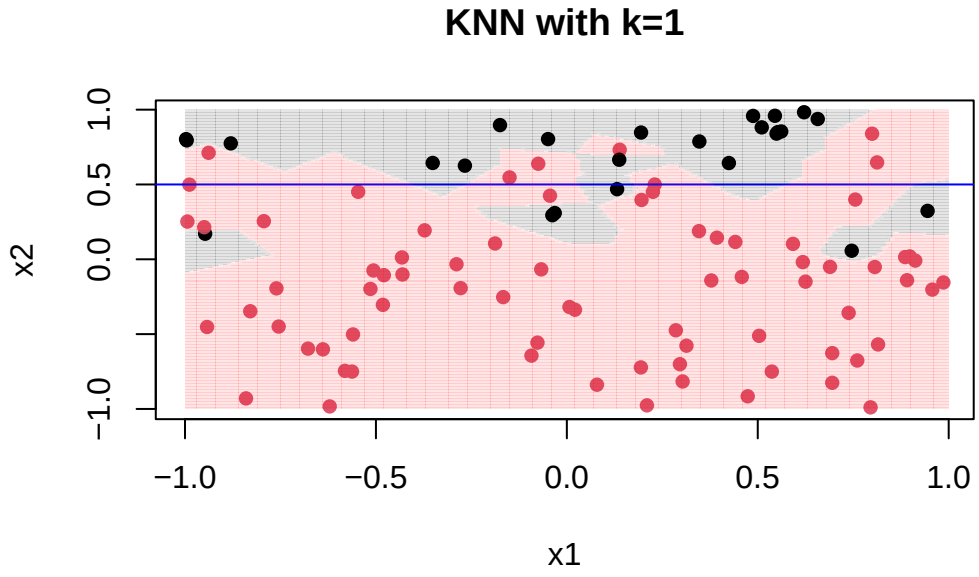


```
plot(x1, x2, col = clas, pch=16, main="KNN with k=1")
abline(h = 0.5, col="blue")
conprob <- attr(mod1, "prob")
conpoints <- ifelse(mod1==1, conprob, 1-conprob)
conmat <- matrix(conpoints, length(gridseq), length(gridseq))
contour(gridseq, gridseq, t(conmat), levels=0.5, nlevels=1, add=TRUE, col="black", lwd=3)
```



Those show the plots given in the lecture slides, however I updated to a more visually appealing and understandable version in the interactive part of the lecture. The commands to get that version of the visualization require the `scales` package to play with the opacity of the colours (so again, you'll need `install.packages("scales")` on your machine prior to calling this code chunk)...

```
library(scales)
plot(x1, x2, col = clas, pch=16, main="KNN with k=1")
abline(h = 0.5, col="blue")
conprob <- attr(mod1, "prob")
conpoints <- ifelse(mod1==1, conprob, 1-conprob)
conmat <- matrix(conpoints, length(gridseq), length(gridseq))
.filled.contour(gridseq, gridseq, t(conmat), levels=c(0,0.50,1), col=alpha(c("red","black"
```


KNN with k=1
x2
1.0
0.5
0.0
−1.0
−1.0
−0.5
0.0
0.5
1.0
x1