

# Lecture 5: Functions and Basic Classification Methods.

STAT GR5206 *Statistical Computing & Introduction to Data Science*

Linxi Liu  
Columbia University

October 5, 2018

- **Character Data in R.** Commands like `nchar()`, `paste()`, `strsplit()`, `substr()`, `grep()`.
- **Regular Expressions.** A grammar (lots of rules) to work with patterns of characters.
- **Web Scraping.** Data from the internet.

## Functions

## Why Functions?

- Data structures tie related values into one object.
- Functions tie related commands into one object.
- Both cases: easier to understand, work with, and to build into larger structures.

# Functions in R

## Basic Structure

```
function_name <- function(arg1, arg2, ... ) {  
  statements  
  return(object)  
}
```

# Functions in R

## Basic Structure

```
function_name <- function(arg1, arg2, ... ) {  
  statements  
  return(object)  
}
```

- A **function** is a group of instructions that takes inputs, uses them to compute other values, and returns a result.
- We can write and add our own functions in R.
- Functions:
  1. Have names.
  2. *Usually* take in arguments.
  3. Include body of code that does something.
  4. *Usually* return an object at the end.

# Example Function

## A Function to Check for Significance at $\alpha = 0.05$

```
> # Input x should be a single p-value in [0,1]
> significant <- function(x) {
+   if (x <= 0.05) { return(TRUE) }
+   else { return(FALSE) }
+ }
```

# Example Function

## A Function to Check for Significance at $\alpha = 0.05$

```
> # Input x should be a single p-value in [0,1]
> significant <- function(x) {
+   if (x <= 0.05) { return(TRUE) }
+   else { return(FALSE) }
+ }
```

- First, tell R to define a function named `significant`.
- Brackets `{` and `}` mark the start and close of the body.
- R tells you you're in the body of the function by using `+` as a prompt (instead of `>`).
- At the end, use the `return()` command.



# Example Function

## A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> # Inputs:  A vector of numbers (x)
> # Outputs: A loss vector with  $x^2$  for small elements,
> #           and  $2|x|-1$  for large ones
>
> res_loss <- function(x) {
+   loss_vec <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)
+   return(loss_vec)
+ }
```

Huber loss

# Example Function

## A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> # Inputs:  A vector of numbers (x)
> # Outputs: A loss vector with  $x^2$  for small elements,
> #           and  $2|x|-1$  for large ones
>
> res_loss <- function(x) {
+   loss_vec <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)
+   return(loss_vec)
+ }
```

Let's try it:

```
> vec <- c(-0.5, 0.9, -3, 4)
> res_loss(vec)
```

```
[1] 0.25 0.81 5.00 7.00
```

# Example Function

## A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> res_loss <- function(x) {  
+   loss_vec <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)  
+   return(loss_vec)  
+ }
```

## Break apart the function

What are the...

- Inputs?
- Outputs?
- Body Statements?

# Example Function

## A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> res_loss <- function(x) {  
+   loss_vec <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)  
+   return(loss_vec)  
+ }
```

## Break apart the function

What are the...

- Inputs? `x`
- Outputs? `loss_vec`
- Body Statements?

```
loss_vec <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)  
return(loss_vec)
```

# When Should We Make a Function?

- Things you will rerun.
- Chunks of code that are small parts of bigger analyses.

# Check Yourself

## Task

Write a function called `FiveTimesSum` that takes as input a vector of numerical values and returns 5 times the sum of those values. Test it on the vector `1:3`. Your output should be 30.

# Check Yourself

## Task

Write a function called `FiveTimesSum` that takes as input a vector of numerical values and returns 5 times the sum of those values. Test it on the vector `1:3`. Your output should be 30.

## Solution

```
> FiveTimesSum <- function(vec){  
+   return(5*sum(vec))  
+ }  
> FiveTimesSum(1:3)
```

```
[1] 30
```

# Named and Default Arguments

## A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> # Inputs: A vector of numbers (x),  
> #           crossover location (c > 0)  
> # Outputs: A loss vector with  $x^2$  for small elements,  
> #           and  $2\sqrt{c}|x| - c$  for large ones  
>  
> res_loss2 <- function(x, c = 1) {  
+   loss_vec <- ifelse(x^2 > c, 2*sqrt(c)*abs(x) - c, x^2)  
+   return(loss_vec)  
+ }
```



# Named and Default Arguments

## A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> # Inputs: A vector of numbers (x),  
> #           crossover location (c > 0)  
> # Outputs: A loss vector with  $x^2$  for small elements,  
> #           and  $2*\sqrt{c}*|x|-c$  for large ones  
>  
> res_loss2 <- function(x, c = 1) {  
+   loss_vec <- ifelse(x^2 > c, 2*sqrt(c)*abs(x) - c, x^2)  
+   return(loss_vec)  
+ }
```

Let's try it:

```
> identical(res_loss(vec), res_loss2(vec, c=1))
```

```
[1] TRUE
```

# Named and Default Arguments

## A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> # Inputs: A vector of numbers (x),  
> #           crossover location (c > 0)  
> # Outputs: A loss vector with  $x^2$  for small elements,  
> #           and  $2\sqrt{c}|x|-c$  for large ones  
>  
> res_loss2 <- function(x, c = 1) {  
+   loss_vec <- ifelse(x^2 > c, 2*sqrt(c)*abs(x) - c, x^2)  
+   return(loss_vec)  
+ }
```

Let's try it:

```
> identical(res_loss(vec), res_loss2(vec, c=2))
```

```
[1] FALSE
```

# Named and Default Arguments

Default values get used if names are missing:

```
> identical(res_loss2(vec, c=1), res_loss2(vec))
```

```
[1] TRUE
```

# Named and Default Arguments

Default values get used if names are missing:

```
> identical(res_loss2(vec, c=1), res_loss2(vec))
```

```
[1] TRUE
```

Named argument can go in any order when they are explicitly tagged:

```
> identical(res_loss2(x=vec, c=2), res_loss2(c=2, x=vec))
```

```
[1] TRUE
```

# Checking Arguments

Funny things can happen when arguments aren't as we expect:

```
> vec <- c(-0.5, 0.9, -3, 4)
> res_loss2(vec, c = c(1,1,1,5))
```

```
[1] 0.25000 0.81000 5.00000 12.88854
```

# Checking Arguments

Funny things can happen when arguments aren't as we expect:

```
> vec <- c(-0.5, 0.9, -3, 4)
> res_loss2(vec, c = c(1,1,1,5))
```

```
[1] 0.25000 0.81000 5.00000 12.88854
```

```
> res_loss2(vec, c = -1)
```

```
[1] NaN NaN NaN NaN
```

# Checking Arguments

Solution: Add some checks to your function.

## A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> res_loss2 <- function(x, c = 1) {  
+   # Scale should be a single positive number  
+   stopifnot(length(c) == 1, c > 0)  
+   loss_vec <- ifelse(x^2 > c, 2*sqrt(c)*abs(x) - c, x^2)  
+   return(loss_vec)  
+ }
```

# Checking Arguments

Solution: Add some checks to your function.

## A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> res_loss2 <- function(x, c = 1) {  
+   # Scale should be a single positive number  
+   stopifnot(length(c) == 1, c > 0)  
+   loss_vec <- ifelse(x^2 > c, 2*sqrt(c)*abs(x) - c, x^2)  
+   return(loss_vec)  
+ }
```

## stopifnot()

- Arguments are a series of expressions which should all be TRUE.
- Execution stops with error at first FALSE.



# Checking Arguments

Solution: Add some checks to your function.

## A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> res_loss2 <- function(x, c = 1) {  
+   # Scale should be a single positive number  
+   stopifnot(length(c) == 1, c > 0)  
+   loss_vec <- ifelse(x^2 > c, 2*sqrt(c)*abs(x) - c, x^2)  
+   return(loss_vec)  
+ }
```

Test it:

```
> # res_loss2(vec, c = c(1,1,1,5))  
> # res_loss2(vec, c = -1)
```

# Check Yourself

## Task

Write a function called `KTimesSum` that takes as input a vector of numerical values and a scalar value  $K$  (with a default value of 5). The function should return the sum of those values multiplied times the value  $K$ . Test it with the following: `KTimesSum(1:3)` and `KTimesSum(1:3, K = 10)`.

# Check Yourself

## Task

Write a function called `KTimesSum` that takes as input a vector of numerical values and a scalar value  $K$  (with a default value of 5). The function should return the sum of those values multiplied times the value  $K$ . Test it with the following: `KTimesSum(1:3)` and `KTimesSum(1:3, K = 10)`.

## Solution

```
> KTimesSum <- function(vec, K = 5){  
+   return(K*sum(vec))  
+ }  
> KTimesSum(1:3); KTimesSum(1:3, K = 10)
```

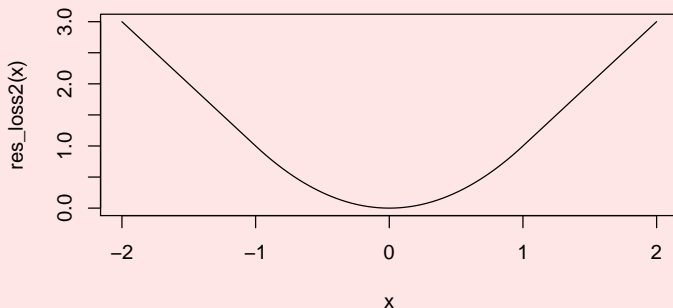
```
[1] 30
```

```
[1] 60
```

# Use Your Functions in Other Functions

- Use your own function in built-in R functions like `apply()`.
- Ex: `curve(expression, from = , to = )` plots a curve.

```
> curve(res_loss2, from = -2, to = 2)
```



# The R Environment

- The **global environment** (or the workspace) in R consists of the collection of your named objects.
- When you start an R session, a new environment is initialized (unless you load a saved environment).
- When a function is called, a new **local environment** is created within the body of the function.

## Clearing the Global Environment

Note the code `rm(list=ls())` clears the R global environment.

Code Example.

# The Function Environment

- Each function has its own (internal) **environment**.
- Names in the function environment override names from the **global environment**.
- Assignments in the internal environment don't change the global environment.
- Functions search for named variables (undefined in the function itself) in the environment in which the function was created (in our case, the global environment).

# The Function Environment

```
> x <- 7  
> y <- c("dog", "cat")  
> addition <- function(y) {x <- x + y; return(x)}  
> addition(1)
```

```
| [1] 8
```



# The Function Environment

```
> x <- 7  
> y <- c("dog", "cat")  
> addition <- function(y) {x <- x + y; return(x)}  
> addition(1)
```

```
[1] 8
```

```
> x
```

```
[1] 7
```

```
> y
```

```
[1] "dog" "cat"
```

# The Function Environment

```
> circle.area <- function(r) {return(pi*r^2)}  
> circle.area(1:3)
```

```
[1] 3.141593 12.566371 28.274334
```

# The Function Environment

```
> circle.area <- function(r) {return(pi*r^2)}  
> circle.area(1:3)
```

```
[1] 3.141593 12.566371 28.274334
```

```
> true.pi <- pi # Save the real value  
> pi      <- 3 # Assign a new value  
> circle.area(1:3)
```

```
[1] 3 12 27
```

# The Function Environment

```
> circle.area <- function(r) {return(pi*r^2)}  
> circle.area(1:3)
```

```
[1] 3.141593 12.566371 28.274334
```

```
> true.pi <- pi # Save the real value  
> pi <- 3 # Assign a new value  
> circle.area(1:3)
```

```
[1] 3 12 27
```

```
> pi <- true.pi # Restore the real value  
> circle.area(1:3)
```

```
[1] 3.141593 12.566371 28.274334
```

# Use Your Function Interfaces

The function **interfaces** are the places where the function interacts with the global environment: at the **inputs** and the **outputs**.

# Use Your Function Interfaces

The function **interfaces** are the places where the function interacts with the global environment: at the **inputs** and the **outputs**.

- Interact with the rest of the system only at the interfaces:
  - Arguments should give your function all the information it needs
  - Reduces the risk of bugs
  - Exception would be universal constants like `pi`.
- Output should be only through `return()`.

# Extended Example: Fitting a Model

# The Model

We study the following idea: bigger cities tend to produce more economically per capita.

Geoffrey West et al.

A proposed statistical model for this relationship:

$$Y = \beta_0 X^{\beta_1} + \epsilon,$$

where

- $Y$ : per-capita 'gross metropolitan product' of a city,
- $X$ : is the population of the city,
- $\beta_0, \beta_1$ : parameters,
- $\epsilon$ : noise.



# The Data

```
> gmp <- read.table("gmp.txt", as.is = TRUE, header = TRUE)
> head(gmp)[1:3, ]
```

	city	gmp	pcgmp
1	Abilene, TX	3.8870e+09	24490
2	Akron, OH	2.2998e+10	32889
3	Albany, GA	3.9550e+09	24269

# The Data

```
> gmp <- read.table("gmp.txt", as.is = TRUE, header = TRUE)
> head(gmp)[1:3, ]
```

	city	gmp	pcgmp
1	Abilene, TX	3.8870e+09	24490
2	Akron, OH	2.2998e+10	32889
3	Albany, GA	3.9550e+09	24269

## Variables

- city
- gmp: gross metropolitan product (money the city makes)
- pcgmp: per-capita gross metropolitan product (money the city makes divided by the number of people in the city)

## The Data (cont.)

Our model uses the population, so let's create a variable for that in the dataset.

# The Data (cont.)

Our model uses the population, so let's create a variable for that in the dataset.

$$\text{pcgmp} = \frac{\text{gmp}}{\text{population}} \rightarrow \text{population} = \frac{\text{gmp}}{\text{pcgmp}}.$$

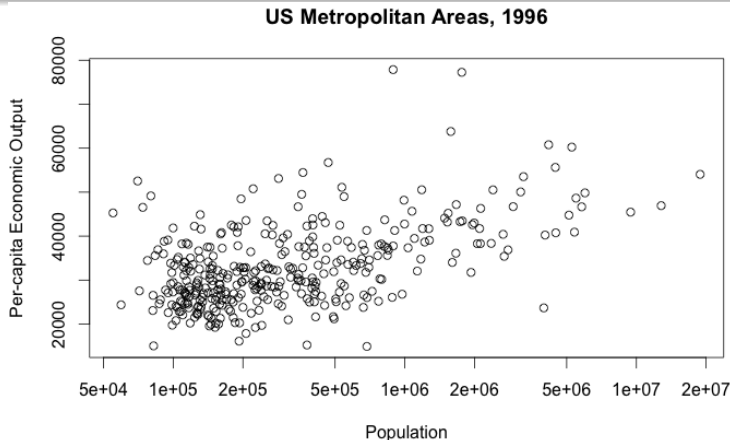
```
> gmp$pop <- gmp$gmp/gmp$pcgmp  
> head(gmp)[1:3, ]
```

	city	gmp	pcgmp	pop
1	Abilene, TX	3.8870e+09	24490	158717.8
2	Akron, OH	2.2998e+10	32889	699261.2
3	Albany, GA	3.9550e+09	24269	162965.1

# The Data (cont)

## Plotting Per-Capita GMP vs. Population

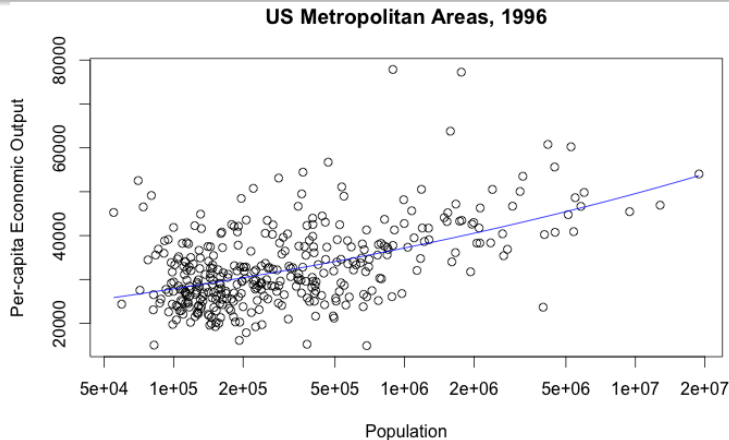
```
> plot(gmp$pop, gmp$pcgmp, log = "x", xlab = "Population",  
+      ylab = "Per-capita Economic Output")
```



# The Data (cont)

## Plotting Per-Capita GMP vs. Population

```
> # beta_0 = 6611; beta_1 = 1/8  
> curve(6611*x^{1/8}, add = TRUE, col = "blue")
```



# Fitting the Function

Want to fit the model:

$$Y = 6611X^{\beta_1} + \epsilon.$$

We're assuming  $\beta_0 = 6611$  and trying to estimate  $\beta_1$  using the data.

# Fitting the Function

Want to fit the model:

$$Y = 6611X^{\beta_1} + \epsilon.$$

We're assuming  $\beta_0 = 6611$  and trying to estimate  $\beta_1$  using the data.

## Strategy

Minimize the sum of the squares!

$$\min_{\beta} \sum_{i=1}^n (Y_i - 6611X_i^{\beta})^2$$

where  $n$  is the number of data points.

Note this is the **training mean square error** (times  $n$ ).



# Optimizing a Function

Want to find  $\min_x f(x)$ .

## Numerical Methods for Optimization

All numerical minimization methods perform roughly the same steps:

- Start with some point  $x_0$ .
- Find a sequence  $x_0, x_1, \dots, x_m$  such that  $f(x_m)$  is a minimum.
- At a given point  $x_n$ , compute properties of  $f$  (such as  $f'(x_n)$  and  $f''(x_n)$ ).
- Based on these values, choose the next point  $x_{n+1}$ .

# Optimizing a Function

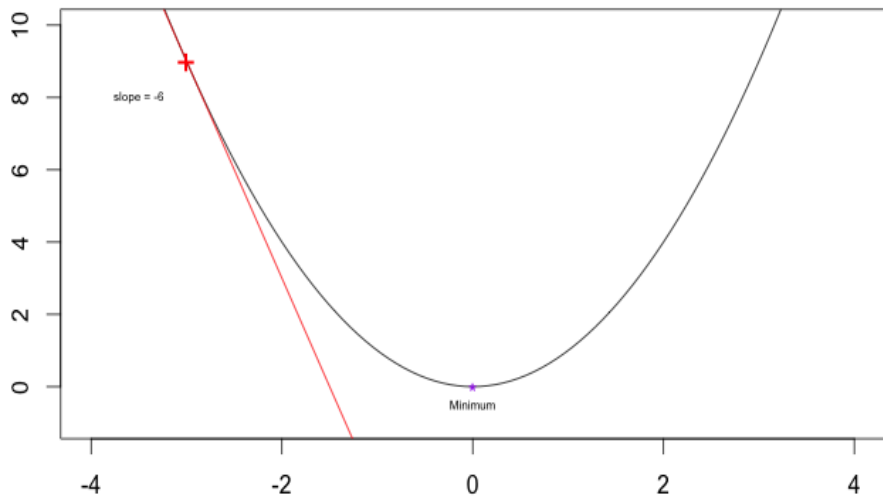
Gradient Descent is an iterative algorithm to find a (local) minimum of a function.

## Gradient Descent

Procedure:

- Start at a point  $x_0$
- Calculate the derivative at the point,  $f'(x_0)$ .
- Take a step in the opposite direction of the derivative to find  $x_1$ .
- Repeat.
- Stop when the derivative is small enough or at some set number of iterations.

# Optimizing a Function



# Fitting Our Function

## Strategy

Minimize the sum of the squares!

$$\min_{\beta} \sum_{i=1}^n (Y_i - 6611X_i^{\beta})^2.$$

In our example,

Initial point  $\beta_0$ . Approximate the derivative w.r.t.  $\beta$  and move in the opposite direction:

$$SSE(\beta_0) = \sum_{i=1}^n (Y_i - 6611X_i^{\beta_0})^2$$

$$SSE'(\beta_0) \approx \frac{SSE(\beta_0 + h) - SSE(\beta_0)}{h} \quad h \text{ is small}$$

$$\beta_1 \approx \beta_0 - c * SSE'(\beta_0) \quad c \text{ scales our step size}$$

# Fitting Our Function

$$SSE(\beta_0) = \sum_{i=1}^n (Y_i - 6611X_i^{\beta_0})^2$$

$$SSE'(\beta_0) \approx \frac{SSE(\beta_0 + h) - SSE(\beta_0)}{h} \quad \text{h is small}$$

$$\beta_1 \approx \beta_0 - c * SSE'(\beta_0) \quad \text{c scales our step size}$$

A first attempt at code:

```
> # Parameters
> max.iter      <- 100      # How long we run the alg.
> stop.deriv    <- 1/100    # If derivative is small, stop
> derivative.step <- 1/1000  # This is h
> step.scale    <- 1e-15    # This is c
> # Initializations
> iter          <- 0        # Compare to max.iteration
> deriv         <- Inf      # Compare to stop.deriv
> beta         <- 0.15
```

## A first attempt at code:

```
> max.iter      <- 100      # How long we run the alg.
> stop.deriv    <- 1/100    # If derivative is small, stop
> deriv.step    <- 1/1000  # This is h
> step.scale    <- 1e-15   # This is c
> iter          <- 0       # Iteration counter
> deriv         <- Inf
> beta         <- 0.15
> while((iter < max.iter) & (deriv > stop.deriv)) {
+   iter <- iter + 1
+   sse.1 <- sum((gmp$pcgmp - 6611*gmp$pop^beta)^2)
+   sse.2 <- sum((gmp$pcgmp
+                 - 6611*gmp$pop^(beta + deriv.step))^2)
+   deriv <- (sse.2 - sse.1)/deriv.step
+   beta  <- beta - step.scale*deriv
+ }
> list(beta = beta, iteration = iter,
+       converged = (iter < max.iter))
```

## A first attempt at code:

```
> list(beta = beta, iteration = iter,  
+       converged = (iter < max.iter))
```

```
$beta
```

```
[1] 0.1258166
```

```
$iteration
```

```
[1] 100
```

```
$converged
```

```
[1] FALSE
```

# What's Wrong With the Previous Attempt?

- **Not encapsulated:** Re-run by copying and pasting. Note easy to fit into bigger project.
- **Inflexible:** To edit initializations must copy, paste, re-run
- **Error-prone:** If you change the dataset, not sure if it would still work...
- **Hard to fix:** Should stop when *absolute value* of derivative is small.



## First Fix

```
> est.scaling.exponent <- function(beta) {
+   max.iter      <- 100      # How long we run the alg.
+   stop.deriv    <- 1/100    # If derivative is small, stop
+   deriv.step    <- 1/1000   # This is h
+   step.scale    <- 1e-15    # This is c
+   iter          <- 0
+   deriv         <- Inf
+   while((iter < max.iter) & (abs(deriv) > stop.deriv)) {
+     iter <- iter + 1
+     sse.1 <- sum((gmp$pcgmp - 6611*gmp$pop^beta)^2)
+     sse.2 <- sum((gmp$pcgmp
+                  - 6611*gmp$pop^(beta + deriv.step))^2)
+     deriv <- (sse.2 - sse.1)/deriv.step
+     beta  <- beta - step.scale*deriv
+   }
+   fit <- list(beta = beta, iteration = iter,
+               converged = (iter < max.iter))
+   return(fit)
+ }
```

## Second Fix

**Problem:** Have to rerun if we want to change defined parameters.

**Solution:** Let's make them arguments (with default values) of the function.

## Second Fix

```
> est.scaling.exponent <- function(beta, beta_0 = 6611,
+   max.iter = 100, stop.deriv = .01, deriv.step = .001,
+   step.scale = 1e-15) {
+
+   iter <- 0
+   deriv <- Inf
+
+   while((iter < max.iter) & (abs(deriv) > stop.deriv)) {
+     iter <- iter + 1
+     sse.1 <- sum((gmp$pcgmp - beta_0*gmp$pop^beta)^2)
+     sse.2 <- sum((gmp$pcgmp
+                   - beta_0*gmp$pop^(beta + deriv.step))^2)
+     deriv <- (sse.2 - sse.1)/deriv.step
+     beta <- beta - step.scale*deriv
+   }
+   fit <- list(beta = beta, iteration = iter,
+               converged = (iter < max.iter))
+   return(fit)
+ }
```

### Third Fix

**Problem:** Don't need to write out the SSE calculations twice in the body of the function.

**Solution:** Write a `SSE()` function.

## Third Fix

```
> est.scaling.exponent <- function(beta, beta_0 = 6611,
+   max.iter = 100, stop.deriv = .01, deriv.step = .001,
+   step.scale = 1e-15) {
+
+   iter <- 0
+   deriv <- Inf
+
+   sse <- function(b) {sum((gmp$pcgmp - beta_0*gmp$pop^b)^2)}
+
+   while((iter < max.iter) & (abs(deriv) > stop.deriv)) {
+     iter <- iter + 1
+     deriv <- (sse(beta + deriv.step) - sse(beta))/deriv.step
+     beta <- beta - step.scale*deriv
+   }
+   fit <- list(beta = beta, iteration = iter,
+               converged = (iter < max.iter))
+   return(fit)
+ }
```

`sse()` is declared inside the function so it's not added to the global environment.

## Fourth Fix

**Problem:** Locked into using specific columns of `gmp`: if we want to use a different data set, have to rewrite the function.

**Solution:** Make them arguments.

## Fourth Fix

```
> est.scaling.exponent <- function(beta, beta_0 = 6611,
+   response = gmp$pcgmp, predictor = gmp$pop,
+   max.iter = 100, stop.deriv = .01, deriv.step = .001,
+   step.scale = 1e-15) {
+
+   iter <- 0
+   deriv <- Inf
+
+   sse <- function(b) {sum((response - beta_0*predictor^b)^2)}
+
+   while((iter < max.iter) & (abs(deriv) > stop.deriv)){
+     iter <- iter + 1
+     deriv <- (sse(beta + deriv.step) - sse(beta))/deriv.step
+     beta <- beta - step.scale*deriv
+   }
+   fit <- list(beta = beta, iteration = iter,
+               converged = (iter < max.iter))
+   return(fit)
+ }
```

## Fifth Fix

**Problem:** Want to make it easy for humans to read.

**Solution:** Change the `while` loop to `for` loop with a `break()` command.



## Fifth Fix

```
> est.scaling.exponent <- function(beta, beta_0 = 6611,
+   response = gmp$pcgmp, predictor = gmp$pop,
+   max.iter = 100, stop.deriv = .01, deriv.step = .001,
+   step.scale = 1e-15) {
+
+   iter <- 0
+   deriv <- Inf
+
+   sse <- function(b) {sum((response - beta_0*predictor^b)^2)}
+
+   for (i in 1:max.iter) {
+     iter <- iter + 1
+     deriv <- (sse(beta + deriv.step) - sse(beta))/deriv.step
+     beta <- beta - step.scale*deriv
+     if (abs(deriv) < stop.deriv) {break()}
+   }
+   fit <- list(beta = beta, iteration = iter,
+     converged = (iter < max.iter))
+   return(fit)
+ }
```

Final code is shorter, more flexible, easier to understand, and more re-usable!

- *Exercise:* Run the code with the default values to get an estimate of  $\beta$ . Plot the curve with the data points to check out the fit.
- *Exercise:* Randomly remove one data point – how much does the estimate change?
- *Exercise:* Run the code from different starting points – how much does the estimate change?

# Assessing Model Accuracy

# Selecting a Model

Let's recall the grocery store example from Lecture 2 on multiple linear regression. How do we know if the model we selected is a good fit?

# Selecting a Model

Let's recall the grocery store example from Lecture 2 on multiple linear regression. How do we know if the model we selected is a good fit?

## Example

A large national grocery retailer tracks productivity and costs of its facilities closely. Consider a data set obtained from a single distribution center for a one-year period. Each data point for each variable represents one week of activity. The variables included are number of cases shipped in thousands ( $X_1$ ), the indirect costs of labor as a percentage of total costs ( $X_2$ ), a qualitative predictor called holiday that is coded 1 if the week has a holiday and 0 otherwise ( $X_3$ ), and total labor hours ( $Y$ ).

# The Model

## Example

```
> Grocery <- read.table("Kutner_6_9.txt", header=T)
> lm0      <- lm(Y ~ X1 + X2 + X3, data = Grocery)
> lm0
```

Call:

```
lm(formula = Y ~ X1 + X2 + X3, data = Grocery)
```

Coefficients:

(Intercept)	X1	X2	X3
4149.8872	0.7871	-13.1660	623.5545

$$\widehat{\text{LaborHours}} = 4149.89 + 0.79 \times \text{NumCases} \\ - 13.17 \times \text{IndirectCosts} + 623.55 \times \text{Holiday}$$

# Selecting a Model

## Some Questions

- The coefficient for number of cases was nearly 0, should this be included as a predictor in the model?
- Suppose we have additional information like weather data, should it be included in the model?
- Is linear regression the best predictive model to use with this data?

# Selecting a Model

## Some Questions

- The coefficient for number of cases was nearly 0, should this be included as a predictor in the model?
- Suppose we have additional information like weather data, should it be included in the model?
- Is linear regression the best predictive model to use with this data?

Often we must compare multiple models. How do we know which is best?



# Quality of Fit

To evaluate the performance of a method on a given data set, we need to measure how well its predictions match the observed data.

- Linear regression uses **mean squared error** (MSE):

$$\frac{1}{n} \sum_{i=1}^n (Y_i - \hat{\beta}_0 - \hat{\beta}_1 X_{i1} - \dots - \hat{\beta}_p X_{ip})^2.$$

Called the **training MSE** since it's calculated on the training data.

# Quality of Fit

To evaluate the performance of a method on a given data set, we need to measure how well its predictions match the observed data.

- Linear regression uses **mean squared error** (MSE):

$$\frac{1}{n} \sum_{i=1}^n (Y_i - \hat{\beta}_0 - \hat{\beta}_1 X_{i1} - \dots - \hat{\beta}_p X_{ip})^2.$$

Called the **training MSE** since it's calculated on the training data.

- But what we actually want to minimize is the **test MSE**:

$$\text{Ave}(Y_{\text{test}} - \hat{\beta}_0 - \hat{\beta}_1 X_{\text{test},1} - \dots - \hat{\beta}_p X_{\text{test},p})^2,$$

using **test** data that was not used to fit the model.

We are interested in the accuracy of the predictions that we obtain when we apply our method to previously unseen **test data**.

How do we minimize the test MSE?

- Sometimes have a **test** dataset (that wasn't used to train the model.)

How do we minimize the test MSE?

- Sometimes have a **test** dataset (that wasn't used to train the model.)
- What if no **test** data is available?

How do we minimize the test MSE?

- Sometimes have a **test** dataset (that wasn't used to train the model.)
- What if no **test** data is available?
- Idea: Use the model with the lowest **training** MSE.
  - Unfortunately, no guarantee that the method with the lowest **training** MSE will also have the lowest **test** MSE.
- This week we'll learn a strategy for estimating the **test** MSE when no **test** data is available: *Cross-validation*.

# Classification

# What is Classification?

- We studied linear regression for use when we want to predict a quantitative response variable.
- What if we have a categorical response variable?
- The study of predicting categorical response variables is called **classification**.

# What is Classification?

- We studied linear regression for use when we want to predict a quantitative response variable.
- What if we have a categorical response variable?
- The study of predicting categorical response variables is called **classification**.

## Examples

- A person has symptoms that could possibly be attributed to one of three medical conditions. Which condition does he have?
- A bank must be able to determine whether an online transaction is fraudulent, on the basis of the user's IP address, past transaction history, etc.
- Using DNA sequence data for patients with and without a disease, a biologist would like to figure out which DNA mutations cause diseases and which do not.



# Types of Classifiers

## Many methods of classification:

- Logistic Regression.
- Linear Discriminant Analysis.
- K Nearest Neighbors.
- Trees and Random Forests.
- Support Vector Machines

# Types of Classifiers

## Many methods of classification:

- Logistic Regression.
- Linear Discriminant Analysis.
- K Nearest Neighbors.
- Trees and Random Forests.
- Support Vector Machines

## Set-up

- Just as in regression, we have a set of training observations (data):

$(X_1, Y_1), (X_2, Y_2), \dots, (X_n, Y_n), \quad Y_1, Y_2, \dots, Y_n \text{ categorical.}$

- Use the data to build the classifier.
- We want our classifier to perform well not only on the training data, but also on test observations that were not used to build the classifier.

# Assessing the Accuracy

Recall, for linear regression we calculated to **training** mean squared error:

$$\frac{1}{n} \sum_{i=1}^n (Y_i - \hat{\beta}_0 - \hat{\beta}_1 X_{i1} - \dots - \hat{\beta}_p X_{ip})^2$$

# Assessing the Accuracy

Recall, for linear regression we calculated to **training** mean squared error:

$$\frac{1}{n} \sum_{i=1}^n (Y_i - \hat{\beta}_0 - \hat{\beta}_1 X_{i1} - \dots - \hat{\beta}_p X_{ip})^2$$

For classification, we study the **error rate**:

$$\frac{1}{n} \sum_{i=1}^n \mathbb{I}[Y_i \neq \hat{Y}_i].$$

- $\hat{Y}_i$  is the predicted classification for the  $i^{th}$  observation.
- $\mathbb{I}[Y_i \neq \hat{Y}_i]$  is an *indicator variable* that equals 1 if  $Y_i \neq \hat{Y}_i$  and 0 if  $Y_i = \hat{Y}_i$ .
- If  $\mathbb{I}[Y_i \neq \hat{Y}_i] = 0$  then the  $i^{th}$  observation was classified correctly.
- The *error rate* computed the fraction of incorrect classifications.

- Similarly to the regression case, the following is referred to as the **training** error rate:

$$\frac{1}{n} \sum_{i=1}^n \mathbb{I}[Y_i \neq \hat{Y}_i].$$

- The **test** error rate associated with a set of **test** observations  $(X_{test}, Y_{test})$  is given by

$$\text{Ave}(\mathbb{I}[Y_{test} \neq \hat{Y}_{test}]).$$

- A good classifier is one for which the **test** error rate is the smallest.

The best classifier (in terms of **test** error) *assigns each observation to the most likely class*, given its predictor values.

The best classifier (in terms of **test** error) *assigns each observation to the most likely class*, given its predictor values.

- Assign test observation with predictor vector  $X_{test}$  to the class  $j$  for which

$$Pr(Y = j|X = X_{test})$$

is largest.

- This is a *conditional probability*. Recall 'Bayes Rule':

$$Pr(A|B) = \frac{Pr(A \text{ and } B)}{Pr(B)}$$

Consider a two-class problem, meaning  $Y_{test}$  is either class  $A$  or class  $B$ .

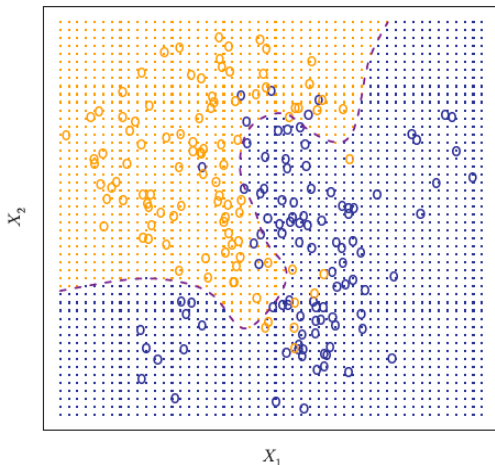
The Bayes classifier predicts:

$$\begin{cases} \text{Class A} & \text{if } Pr(Y = A|X = X_{test}) > 0.5, \\ \text{Class B} & \text{otherwise.} \end{cases} \quad (1)$$



# Bayes Classifier Example <sup>1</sup>

Simulated data of 100 observations. Response  $Y$  either blue or orange, with two predictors  $X_1$  and  $X_2$ . The purple line represents the **Bayes decision boundary**.



# $K$ Nearest Neighbors

In theory would like to always use the Bayes classifier. In practice, don't know  $Pr(Y|X)$ !

## $K$ Nearest Neighbors (KNN)

- Estimates  $Pr(Y|X)$  and then classifies observations to the class with highest estimated probability.

# K Nearest Neighbors

In theory would like to always use the Bayes classifier. In practice, don't know  $Pr(Y|X)$ !

## K Nearest Neighbors (KNN)

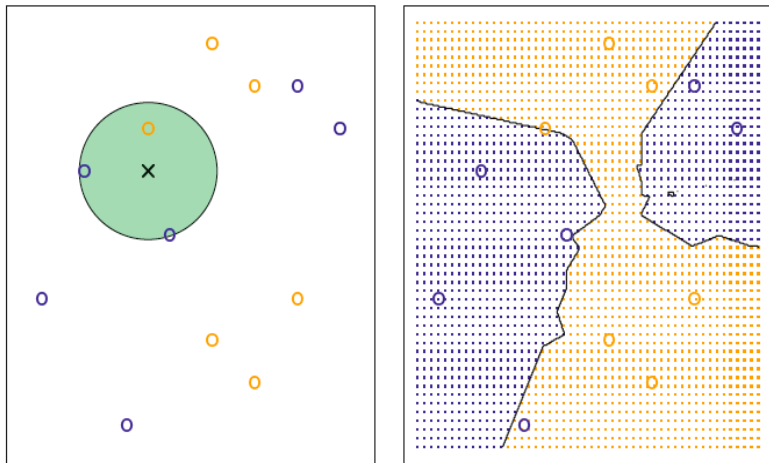
- Estimates  $Pr(Y|X)$  and then classifies observations to the class with highest estimated probability.
- Given a positive integer  $K$  and a test observation  $X_{test}$ :
  - Identify  $K$  points in training data closest to  $X_{test}$ . Label  $\mathcal{N}_{test}$ .
  - Estimate conditional probability for class  $j$  as fraction of points in  $\mathcal{N}_{test}$  whose response values equal  $j$ :

$$Pr(Y = j|X = X_{test}) = \frac{1}{K} \sum_{i \in \mathcal{N}_{test}} \mathbb{I}(Y_i = j).$$

- Classify the test observation to class with the largest probability.

# KNN Classifier Example <sup>2</sup>

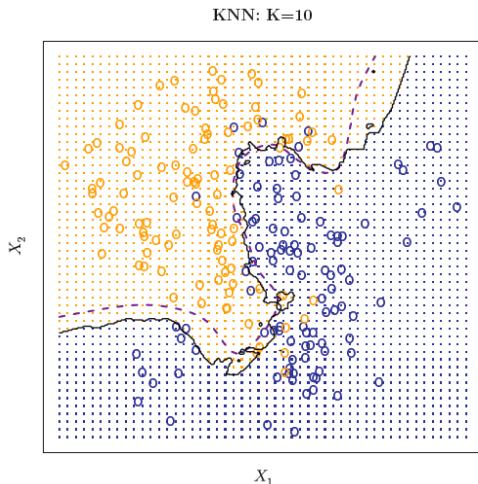
Training data with six blue and six orange observations. Use  $K = 3$ . **KNN decision boundary** shown in black.



<sup>1</sup>Image from 'Introduction to Statistical Learning'.

# KNN Classifier Example <sup>3</sup>

Though simple, KNN can be surprisingly close to Bayes optimal classifier!

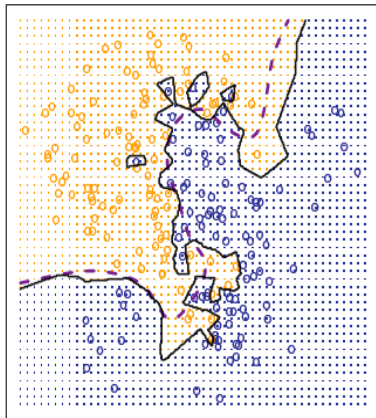


<sup>1</sup>Image from 'Introduction to Statistical Learning'.

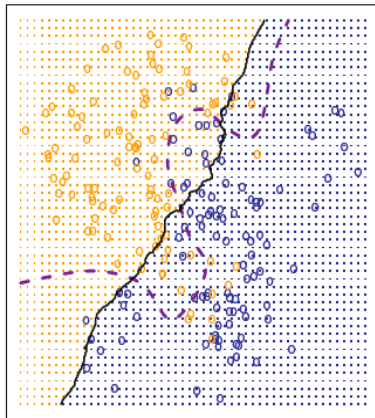
# KNN Classifier Example <sup>4</sup>

Choice of  $K$  matters!

KNN:  $K=1$



KNN:  $K=100$



<sup>1</sup>Image from 'Introduction to Statistical Learning'.

As with regression, **training** error not a good predictor of **test** error.  
(Previous example!)

# Classification: An Example

Can we predict the direction of the stock market today using the previous two days' movements?



# Classification: An Example

Can we predict the direction of the stock market today using the previous two days' movements?

```
> # install.packages("ISLR")  
> library(ISLR)  
> head(Smarket, 3)
```

	Year	Lag1	Lag2	Lag3	Lag4	Lag5	Volume	Today
1	2001	0.381	-0.192	-2.624	-1.055	5.010	1.1913	0.959
2	2001	0.959	0.381	-0.192	-2.624	-1.055	1.2965	1.032
3	2001	1.032	0.959	0.381	-0.192	-2.624	1.4112	-0.623

Direction

1	Up
2	Up
3	Down

# Classification: An Example

## Variables

- **Year:** The year that the observation was recorded
- **LagX:** Percentage return X days ago.
- **Volume:** Volume of shares traded (number of daily shares traded in billions)
- **Today:** Percentage return for today
- **Direction:** A factor with levels Down and Up indicating whether the market had a positive or negative return on a given day.

# Classification: An Example

## Variables

- **Year:** The year that the observation was recorded
- **LagX:** Percentage return X days ago.
- **Volume:** Volume of shares traded (number of daily shares traded in billions)
- **Today:** Percentage return for today
- **Direction:** A factor with levels Down and Up indicating whether the market had a positive or negative return on a given day.

```
> mean(Smarket$Lag1[Smarket$Direction == "Up"])
```

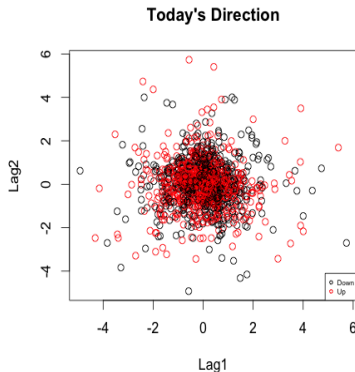
```
[1] -0.03969136
```

```
> mean(Smarket$Lag1[Smarket$Direction == "Down"])
```

```
[1] 0.05068605
```

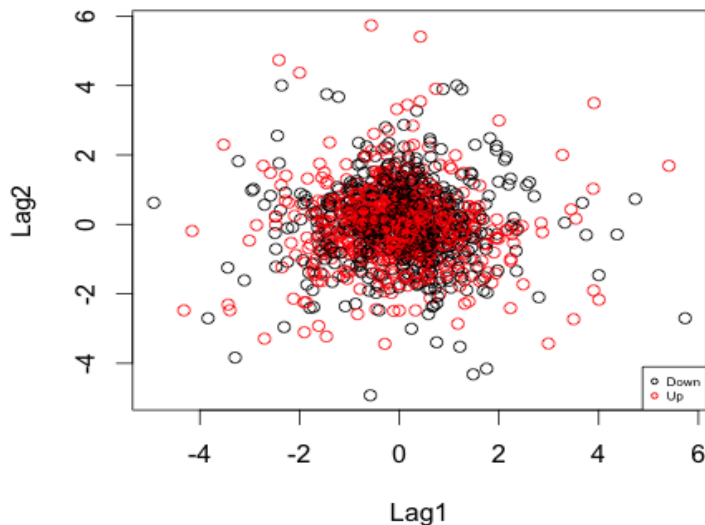
# Classification: An Example

```
> plot(Smarket$Lag1, Smarket$Lag2, col = Smarket$Direction,  
+       xlab="Lag1", ylab="Lag2", main="Today's Direction")  
> legend("bottomright", legend = levels(Smarket$Direction),  
+       col=1:length(levels(Smarket$Direction)), pch=1)
```



# Classification: An Example

## Today's Direction



# Classification: An Example

## Procedure

For a new point  $(Lag1_{new}, Lag2_{new})$ ,

- Calculate the Euclidean distance between the new point and all data points. For a data point  $(L1, L2)$ ,

$$\text{dist}^2 = (Lag1_{new} - L1)^2 + (Lag2_{new} - L2)^2.$$

- Create the set  $\mathcal{N}_{new}$  containing the  $K$  closest points.
- Determine the number of 'UPs' and 'DOWNs' in  $\mathcal{N}_{new}$  and classify the new point.

# Classification: An Example

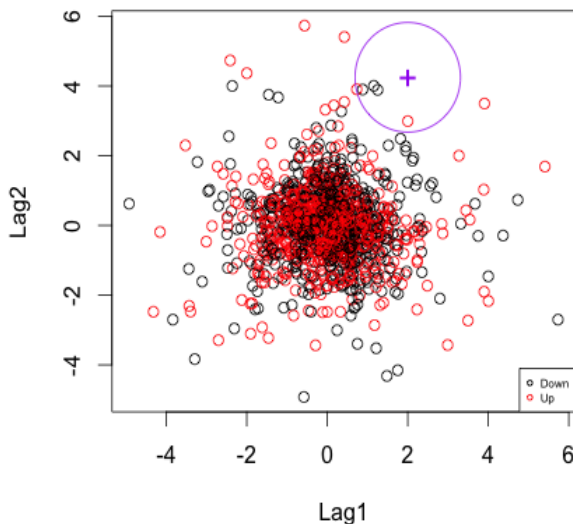
## Coding The Procedure

```
> K          <- 5
> Lag1.new   <- 2
> Lag2.new   <- 4.25
> # K = 5 and new point (2, 4.25).
>
> dists       <- sqrt((Smarket$Lag1 - Lag1.new)^2
+                   + (Smarket$Lag2 - Lag2.new)^2)
> neighbors   <- order(dists)[1:K]
> neighb.dir  <- Smarket$Direction[neighbors]
> choice      <- names(which.max(table(neighb.dir)))
> choice
```

```
[1] "Down"
```

# Classification: An Example

## Today's Direction





## Task

Write a function called `KNN.decision` that returns the class decision for any new point  $(Lag1_{new}, Lag2_{new})$  and any choice of  $K$  (with  $K = 5$ ) as default.

# Check Yourself

## One Solution:

```
> KNN.decision <- function(Lag1.new, Lag2.new, K = 5,  
+                           Lag1 = Smarket$Lag1,  
+                           Lag2 = Smarket$Lag2,  
+                           Dir = Smarket$Direction) {  
+   n <- length(Lag1)  
+   stopifnot(length(Lag2) == n, length(Lag1.new) == 1,  
+           length(Lag2.new) == 1, K <= n)  
+  
+   dists <- sqrt((Lag1-Lag1.new)^2 + (Lag2-Lag2.new)^2)  
+  
+   neighbors <- order(dists)[1:K]  
+   neighb.dir <- Dir[neighbors]  
+   choice <- names(which.max(table(neighb.dir)))  
+   return(choice)  
+ }
```

# Testing Our Model

Let's build our model using data from 2001 - 2004 and use the 2005 data as a test. Can we predict market direction better than a random guess?

# Testing Our Model

Let's build our model using data from 2001 - 2004 and use the 2005 data as a test. Can we predict market direction better than a random guess?

```
> test  <- Smarket[Smarket$Year == 2005, ]
> train <- Smarket[Smarket$Year != 2005, ]
> n.test <- nrow(test)
> predictions <- rep(NA, n.test)
> for (i in 1:n.test){
+   predictions[i] <- KNN.decision(test$Lag1[i],test$Lag2[i],
+                                 Lag1 = train$Lag1,Lag2 = train$Lag2,
+                                 Dir = train$Direction)
+ }
> test.error <- sum(predictions != test$Direction)/n.test
> test.error
```

```
[1] 0.515873
```

# Testing Our Model: Another Try

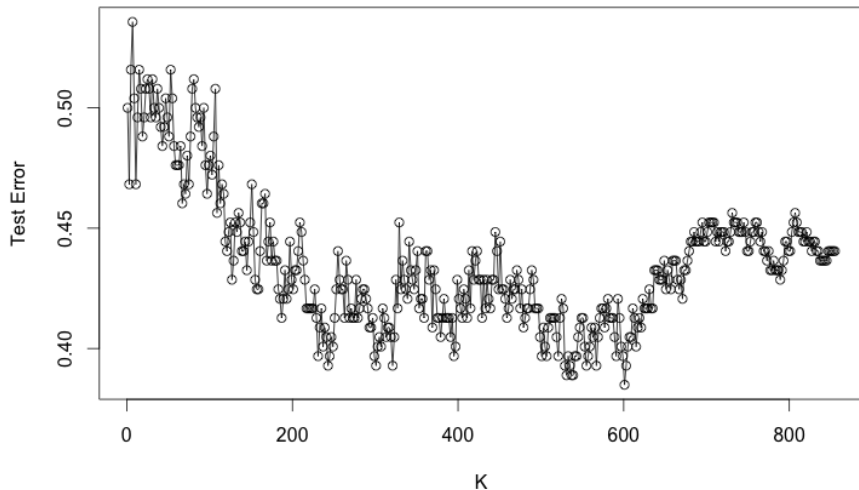
Let's build our model using data from 2001 - 2004 and use the 2005 data as a test. Can we predict market direction better than a random guess?

```
> test <- Smarket[Smarket$Year == 2005, ]
> train <- Smarket[Smarket$Year != 2005, ]
> n.test <- nrow(test)
> predictions <- rep(NA, n.test)
> for (i in 1:n.test){
+   predictions[i] <- KNN.decision(test$Lag1[i], test$Lag2[i],
+                                   K = 7,
+                                   Lag1 = train$Lag1, Lag2 = train$Lag2,
+                                   Dir = train$Direction)
+ }
> test.error <- sum(predictions != test$Direction)/n.test
> test.error
```

```
[1] 0.5357143
```

# Classification: An Example

**Predicting Market Direction**



- Chapter 4 (Classification) and Chapter 5 (Cross-Validation) in An Introduction to Statistical Learning.
- Chapter 10 (Writing Your Own Functions) in An Introduction to R