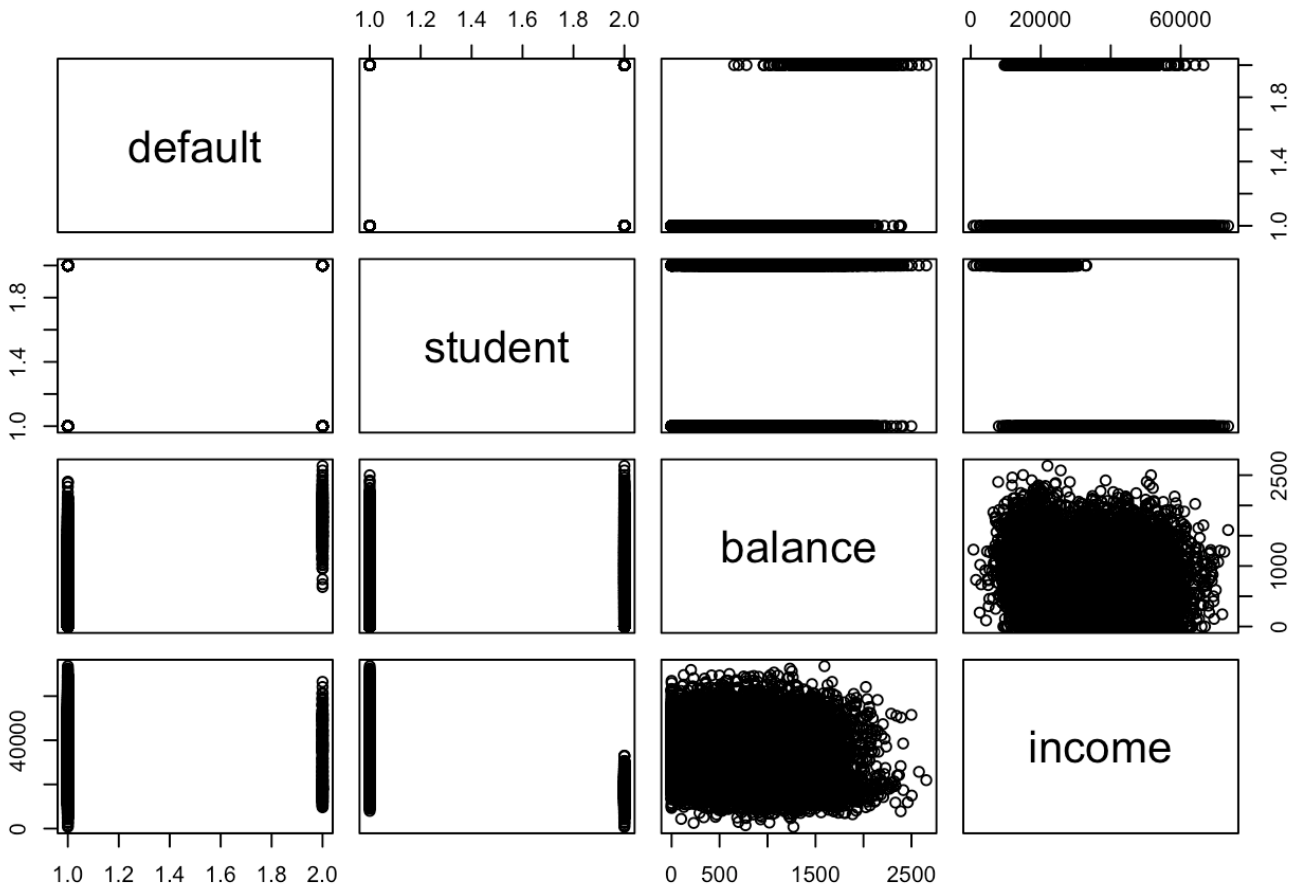
Problem 1

Chapter 5, Exercise 5 (Sec. 5.4, p. 198).

Part A




```
fit = glm(default ~ income + balance, family = 'binomial')
coef(fit)
```

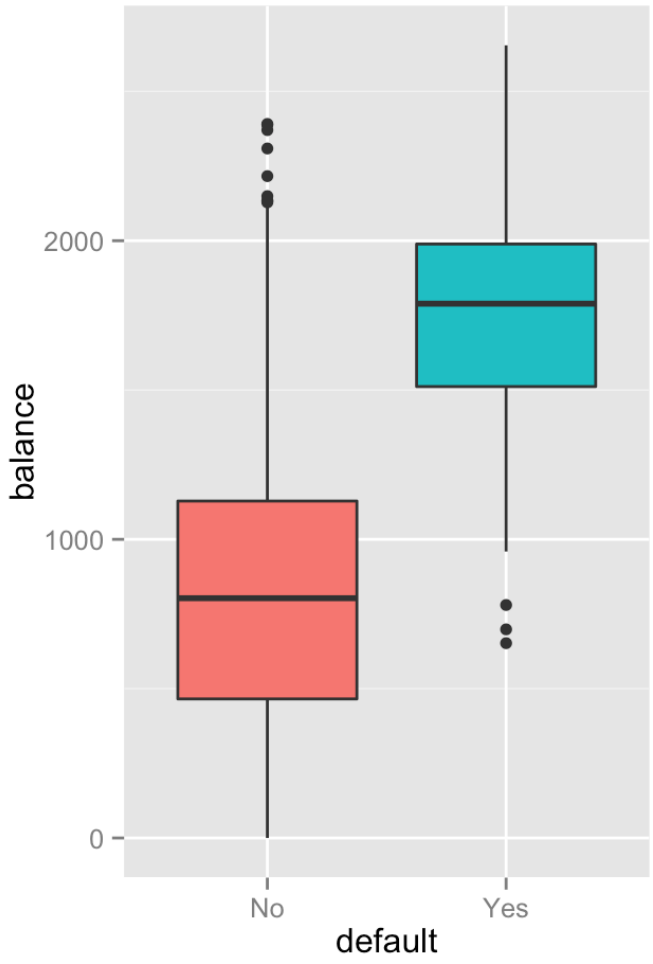
```
##   (Intercept)        income       balance
## -1.154047e+01  2.080898e-05  5.647103e-03
```

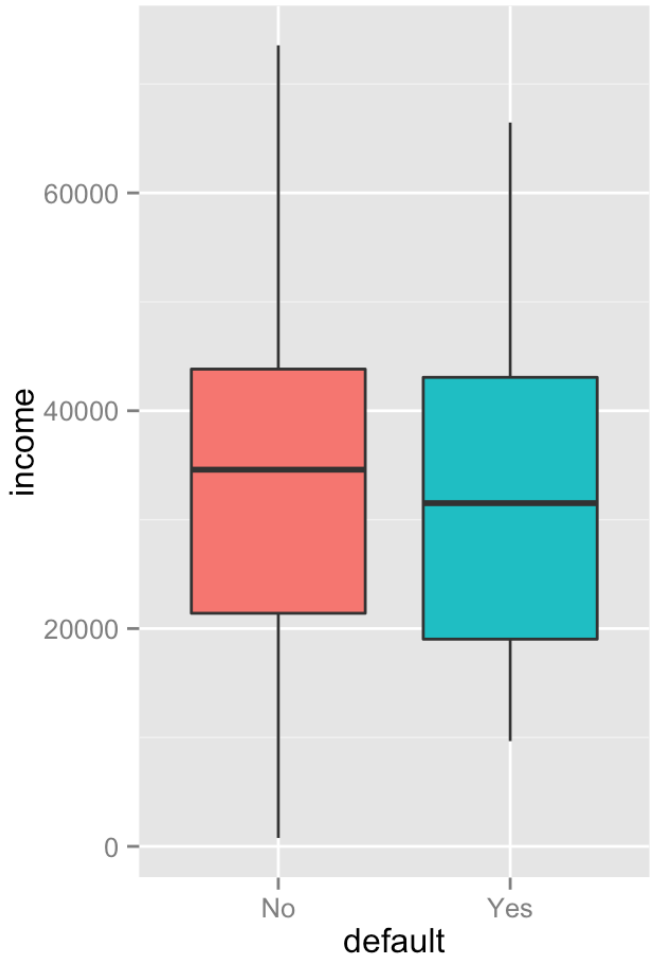
```
tmp = table(Default$default)
percent_defaults = (tmp[[2]]/tmp[[1]])*100
cat(percent_defaults, "percent of people default")
```

```
## 3.444709 percent of people default
```

```
# The following code is inspired by: rpubs.com/ryankelly/21379
x = qplot(x = balance, y = income, color = default, geom = 'point') + scale_shap
e(solid = FALSE)
y = qplot(x = default, y = balance, fill = default, geom = 'boxplot') + guides(fil
l = FALSE)
z = qplot(x = default, y = income, fill = default, geom = 'boxplot') + guides(fill
= FALSE)
x
```


```
grid.arrange(y, z, nrow=1)
```





Part B

```
set.seed(1)

total = nrow(Default)
num   = floor(0.9 * total)

sampled       = Default[sample(total), ]
Default.train = sampled[1:num, ]
Default.test  = sampled[(num + 1):total, ]
fit = multinom(default ~ income + balance, data = Default.train, family = 'binomia
l')
```

```
## # weights:  4 (3 variable)
## initial  value 6238.324625
## iter  10 value 716.944726
## final  value 716.865607
## converged
```

```
print(summary(fit))
```

```
## Call:
## multinom(formula = default ~ income + balance, data = Default.train,
##     family = "binomial")
##
## Coefficients:
##                     Values    Std. Err.
## (Intercept) -1.152515e+01 4.576857e-08
## income       2.027518e-05 4.415690e-06
## balance      5.668073e-03 9.492261e-05
##
## Residual Deviance: 1433.731
## AIC: 1439.731
```

```
pred = predict(fit, Default.test)
print(confusionMatrix(pred, Default.test$default)$table)
```

```
##           Reference
## Prediction  No Yes
##        No  965  19
##        Yes   6  10
```

Part C

```
##
## ====== Cross validation run # 2 ========================
## # weights:  4 (3 variable)
## initial  value 6238.324625
## iter  10 value 685.892190
## final  value 685.885544
## converged
##           Reference
## Prediction  No Yes
##        No  956  29
##        Yes   2  13
## MSE = 0.0475
## ====== Cross validation run # 3 ========================
## # weights:  4 (3 variable)
## initial  value 6238.324625
## iter  10 value 702.235743
## final  value 702.233405
## converged
##           Reference
## Prediction  No Yes
##        No  966  22
##        Yes   4   8
## MSE = 0.0443
## ====== Cross validation run # 4 ========================
## # weights:  4 (3 variable)
## initial  value 6238.324625
## iter  10 value 691.160350
## final  value 691.157602
## converged
##           Reference
## Prediction  No Yes
##        No  960  29
##        Yes   4   7
## MSE = 0.0433
```

Each of the 4 runs gave in similar results with just a little bit of variation:

- The 0-1 loss for each run was `25/1000`, `31/1000`, `26/1000`, and `33/1000` respectively.
- Of these errors, the respective ratios of false positives to false negatives were `6:19`, `2:29`, `4:22`, and `4:29`.
- We missed `10/(10 + 19) = .35`, `13/(29 + 13) = .31`, `8/(8 + 22) = .27`, and `7/(7 + 29) = .19` of defaults.
- We missed `6/(6 + 965) = .0062`, `2/(2 + 956) = .0021`, `4/(4 + 966) = .0041`, and `4/(960 + 4) = .0041` of non-defaults.

Our model does a pretty good job at categorizing non-defaults correctly (missing < 1%), but it fails miserably on actual defaults (missing upwards of 20% and as bad as 35%).

Part D

```
##
## ====== Cross validation run # 1 ========================
## # weights:  5 (4 variable)
## initial  value 6238.324625
## iter  10 value 712.052175
## final  value 711.984415
## converged
## MSE = 0.0465
## ====== Cross validation run # 2 ========================
## # weights:  5 (4 variable)
## initial  value 6238.324625
## iter  10 value 683.842024
## final  value 683.604493
## converged
## MSE = 0.0467
## ====== Cross validation run # 3 ========================
## # weights:  5 (4 variable)
## initial  value 6238.324625
## iter  10 value 697.797874
## final  value 697.507099
## converged
## MSE = 0.0443
## ====== Cross validation run # 4 ========================
## # weights:  5 (4 variable)
## initial  value 6238.324625
## iter  10 value 687.959598
## final  value 687.757482
## converged
## MSE = 0.0433
```

Including the student variable didn't affect our MSE significantly. Since it has no notable effect on our results, it's best to just remove the student variable from our analysis entirely.

Problem 2

Chapter 5, Exercise 6 (Sec. 5.4, p. 199)

Part A

```
fit = glm(default ~ income + balance, family = 'binomial')
summary(fit)$coef
```

```
##                   Estimate   Std. Error    z value      Pr(>|z|)
## (Intercept) -1.154047e+01 4.347564e-01 -26.544680 2.958355e-155
## income       2.080898e-05 4.985167e-06   4.174178  2.990638e-05
## balance      5.647103e-03 2.273731e-04  24.836280 3.638120e-136
```

Part B

```
boot.fn = function(d, i)
  return( coef(lm(default ~ income + balance, data = d, subset = i)) )
```

Part C

```
boot.fn(Default, 1:nrow(Default))
```

```
##  (Intercept)       income      balance
## 9.077603e-01 4.604568e-07 1.318050e-04
```

```
boot(Default, boot.fn, 1000)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Default, statistic = boot.fn, R = 1000)
##
##
## Bootstrap Statistics :
##         original        bias     std. error
## t1* 9.077603e-01  6.463132e-05 6.451908e-03
## t2* 4.604568e-07 -3.352490e-10 1.283030e-07
## t3* 1.318050e-04 -7.879041e-09 6.585405e-06
```

Part D

The estimated standard errors obtained using the `glm()` function and the bootstrap function were within a fairly small range of each other. `glm()` resulted in standard errors of `[4.3e-1, 4.9e-6, 2.3e-4]`, while bootstrap resulted in `[6.5e-3, 1.3e-7, 6.6e-6]`, both of which equate to very small errors. As a result, we can see that the two methods are interchangeable in this situation.

Problem 3

Chapter 5, Exercise 8 (Sec. 5.4, p. 200)

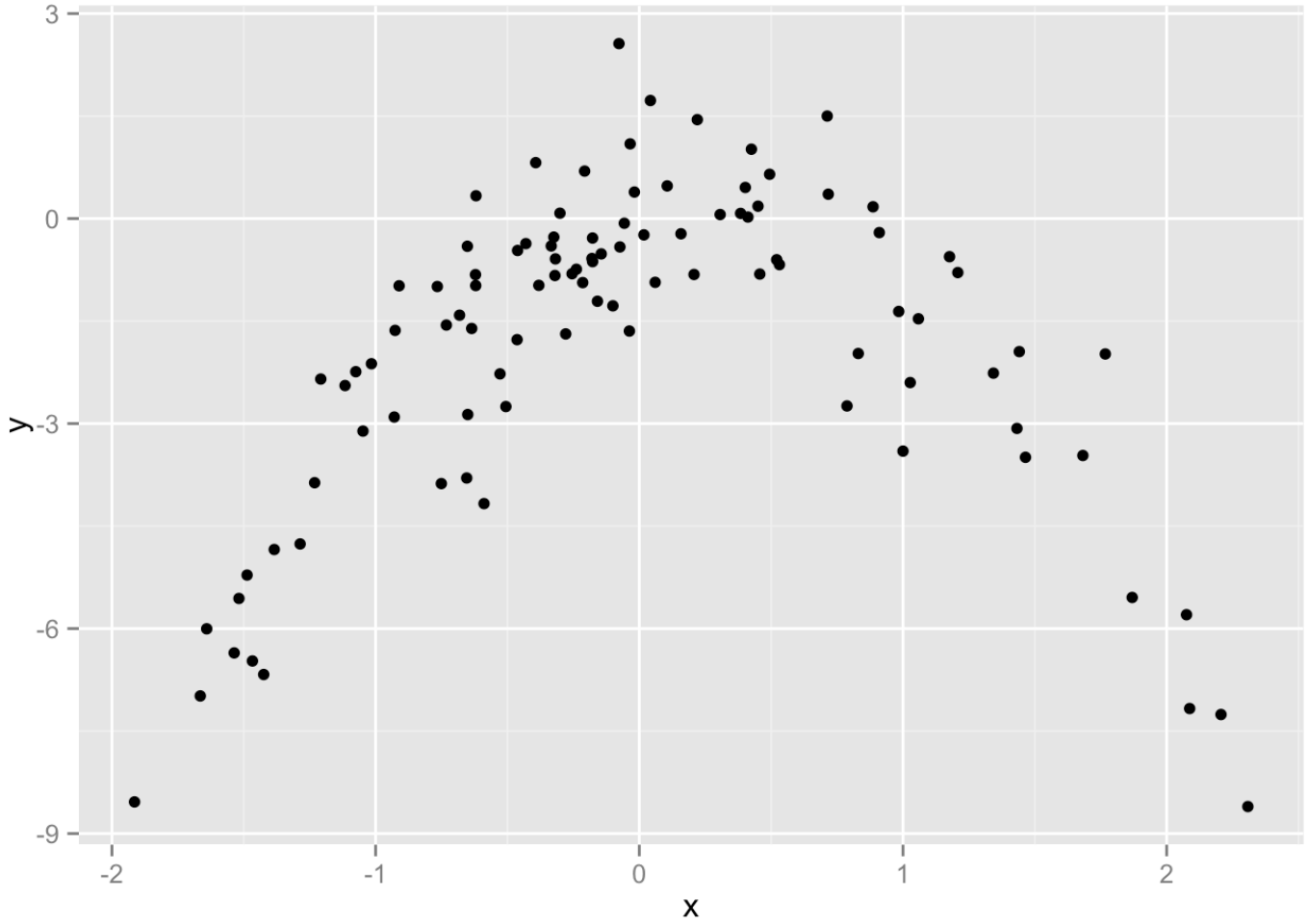
Part A

```
set.seed(1)
y = rnorm(100)
x = rnorm(100)
y = x - 2*x^2 + rnorm(100)
```

```
n = 100  # number of samples
p = 2    # number of dimensions
```

Part B

```
qplot(x, y)
```



There is clearly a non-random relationship between `x` and `y`. In particular, `y` takes on a parabolic shape centered around 0 as `x` varies. However, there is a range of variance of a bit over 1, creating a band of values rather than a neat line of points.

Part C

```
data   = data.frame(y = y, x = x)
```

Linear:

```
model = glm(y ~ x, data = data)
model$coef
```

```
## (Intercept)           x
##  -1.8185184   0.2430443
```

```
## standard k-fold CV estimate = 5.890979
##      bias-corrected version = 5.888812
```

Squared:

```
model = glm(y ~ poly(x, degree = 2), data = data)
model$coef
```

```
##          (Intercept) poly(x, degree = 2)1 poly(x, degree = 2)2
##            -1.827707             2.316401           -21.058587
```

```
## standard k-fold CV estimate = 1.086596
##      bias-corrected version = 1.086326
```

Cubic:

```
model = glm(y ~ poly(x, degree = 3), data = data)
model$coef
```

```
##          (Intercept) poly(x, degree = 3)1 poly(x, degree = 3)2
##           -1.8277074            2.3164010          -21.0585869
## poly(x, degree = 3)3
##           -0.3048398
```

```
## standard k-fold CV estimate = 1.102585
##      bias-corrected version = 1.102227
```

Quadratic:

```
model = glm(y ~ poly(x, degree = 4), data = data)
model$coef
```

```
##          (Intercept) poly(x, degree = 4)1 poly(x, degree = 4)2
##           -1.8277074            2.3164010          -21.0585869
## poly(x, degree = 4)3 poly(x, degree = 4)4
##           -0.3048398           -0.4926249
```

```
## standard k-fold CV estimate = 1.114772
##      bias-corrected version = 1.114334
```

Part D

```
set.seed(5)
y = rnorm(100)
x = rnorm(100)
y = x - 2*x^2 + rnorm(100)

data   = data.frame(y = y, x = x)
```

Linear:

```
model = glm(y ~ x, data = data)
```

```
## standard k-fold CV estimate = 10.32995
##      bias-corrected version = 10.32529
```

Squared:

```
model = glm(y ~ poly(x, degree = 2), data = data)
```

```
## standard k-fold CV estimate = 0.9586209
##      bias-corrected version = 0.9583222
```

Cubic:

```
model = glm(y ~ poly(x, degree = 3), data = data)
```

```
## standard k-fold CV estimate = 0.9867481
##      bias-corrected version = 0.9862594
```

Quadratic:

```
model = glm(y ~ poly(x, degree = 4), data = data)
```

```
## standard k-fold CV estimate = 1.335795
##      bias-corrected version = 1.332331
```

Both runs use the same data generator function, but the `rnorm(...)` creates variance between runs. This variance results in slightly different fits resulting from a linear regression model.

Part E

The squared model had the smallest error. This is what I expected, since the original function is based off the square of `x`.

Part F

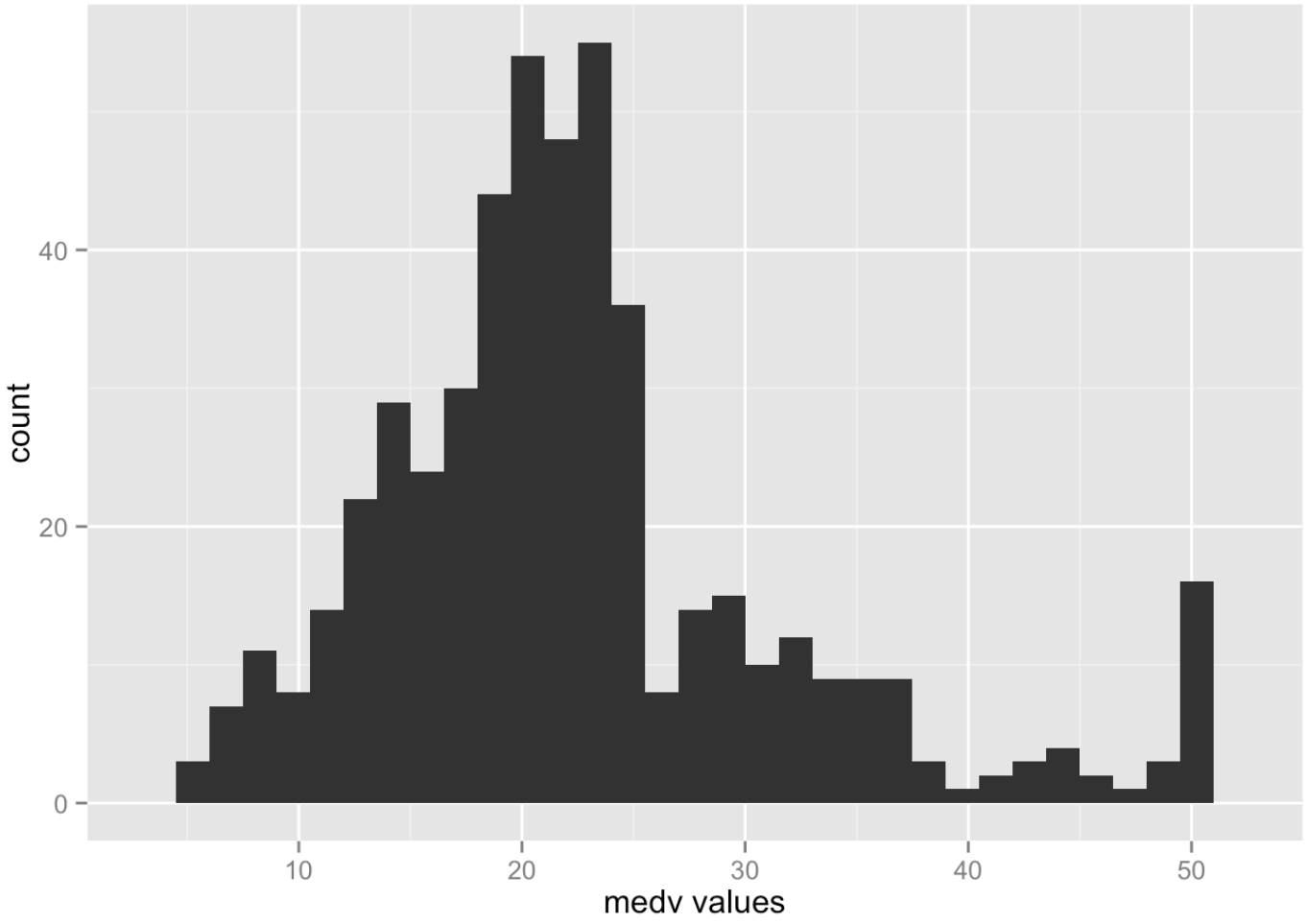
Our original function was `y = x - 2*x^2 + rnorm(100)`, which means the correct coefficients ought to be: `B0 = 0, B1 = 1, B2 = -2`.

Instead, we got `[-1.82, 0.24]`, `[-1.83, 2.32, -21.06]`, `[-1.83, 2.31, -21.06, -0.35]`, and `[-1.83, 2.32, -21.06, -0.31, -0.49]` for the linear, squared, cubed, and quadratic fits respectively. These are wayyyy off, even for the best fit (squared, with coefficients `[-1.83, 2.32, -21.06]`). This does not agree with the conclusions drawn based on the cross-validation results, which implied that the squared fit was fairly accurate.

Problem 4

Chapter 5, Exercise 9 (Sec. 5.4, p. 201)

```
m = Boston$medv
qplot(m, xlab = 'medv values')
```



Part A

```
m.mean = mean(m)
m.mean
```

```
## [1] 22.53281
```

Part B

```
m.std_err = sd(m)/sqrt(nrow(Boston))
m.std_err
```

```
## [1] 0.4088611
```

This standard error is fairly low, though it could be better.

Part C

```
# Inspiration for the following code comes from:
#   stats.stackexchange.com/questions/22472/use-of-standard-error-
#   of-bootstrap-distribution

m.bs = boot(m, function(d, i) mean(d[i]), 1000)

# The standard error from bootstrap is simply the standard deviation
# of the bootstrap distribution:
sd(m.bs$t)
```

```
## [1] 0.4033504
```

The bootstrap method's standard error is very similar to that of part (b).

Part D

```
boot.ci(m.bs, type='bca')
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = m.bs, type = "bca")
##
## Intervals :
## Level       BCa
## 95%   (21.79, 23.37 )
## Calculations and Intervals on Original Scale
```

```
t.test(Boston$medv)
```

```
##
##  One Sample t-test
##
## data:  Boston$medv
## t = 55.111, df = 505, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  21.72953 23.33608
## sample estimates:
## mean of x
##  22.53281
```

These result in nearly identical confidence intervals, with a difference of < 0.1 on each end.

Part E

```
m.median = median(m)
m.median
```

```
## [1] 21.2
```

Part F

```
m.bs = boot(m, function(d, i) median(d[i]), 1000)
sd(m.bs$t)
```

```
## [1] 0.3724187
```

The standard error of the median using the bootstrap (`0.385`) is somewhat lower than that of the mean (`0.419`).

Part G

```
u10 = quantile(m, c(.10))
```

Part H

```
m.bs = boot(m, function(d, i) quantile(d[i], c(.10)), 1000)
sd(m.bs$t)
```

```
## [1] 0.5010284
```

Our standard error increases significantly when we try to predict the 10th percentile. This is understandable; since our data basically takes the form of a normal distribution, the bulk of the data is in the middle, while there are far fewer examples to draw upon on the edges.

Problem 5

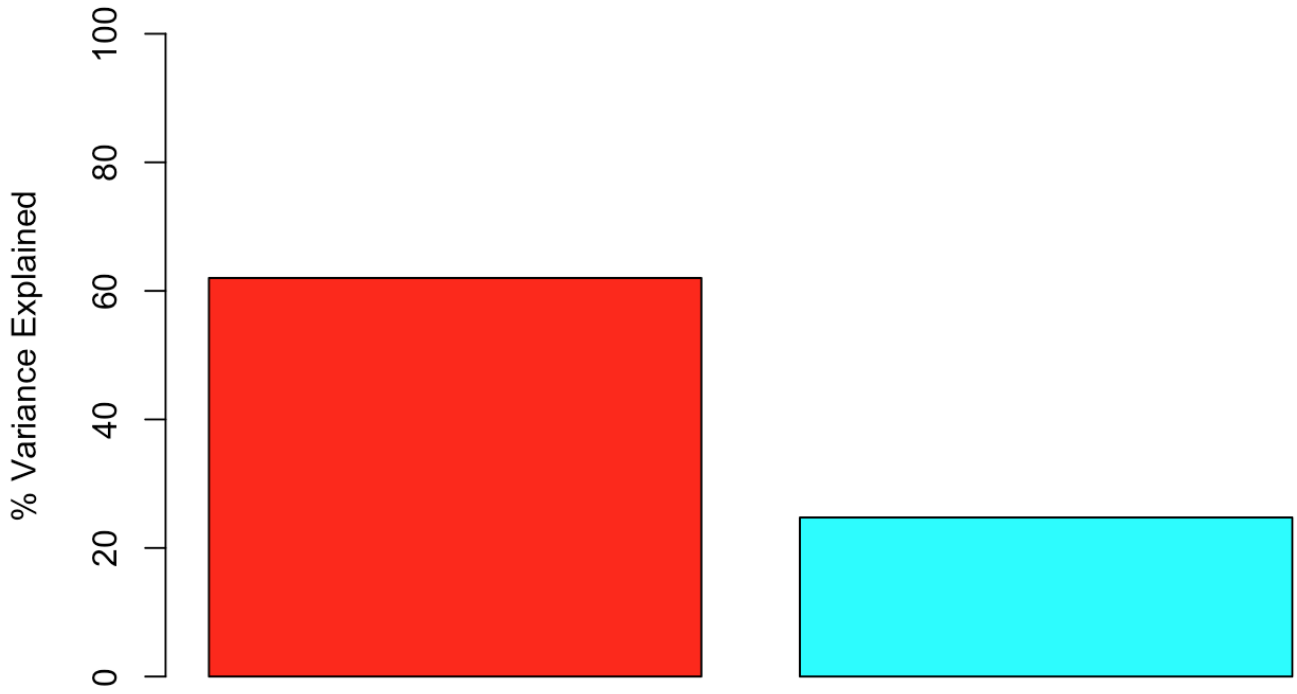
Notes from lecture

- resampling rows of our data matrix (a.k.a. “bootstrap replicates”)
- extract principal components of each sampled matrix
- what’s the range of loading values on each sampled matrix?

Part 1

```
set.seed(1)
# Calculate matrix of % variance explained.
boot.fn = function(d, i) {
  pr.out = prcomp(d[i,], scale = TRUE)
  pr.var = pr.out$sdev^2
  return( (pr.var / sum(pr.var))[1:2] )
}
pca = boot.fn(USArrests)
bs  = boot(USArrests, boot.fn, R = 1000)

barplot(100 * pca, xlab='', ylab='% Variance Explained', ylim = c(0, 100), col = r
ainbow(2))
```



Part 2

```
sd(bs$t)
```

```
## [1] 0.1909424
```

```
boot.ci(bs, type='bca')
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = bs, type = "bca")
##
## Intervals :
## Level       BCa
## 95%   ( 0.5415,  0.7064 )
## Calculations and Intervals on Original Scale
```

Part 3

The loading values for resulting from PCA can have a sign flip. If we naively place it into our plot, it makes our spread look huge.

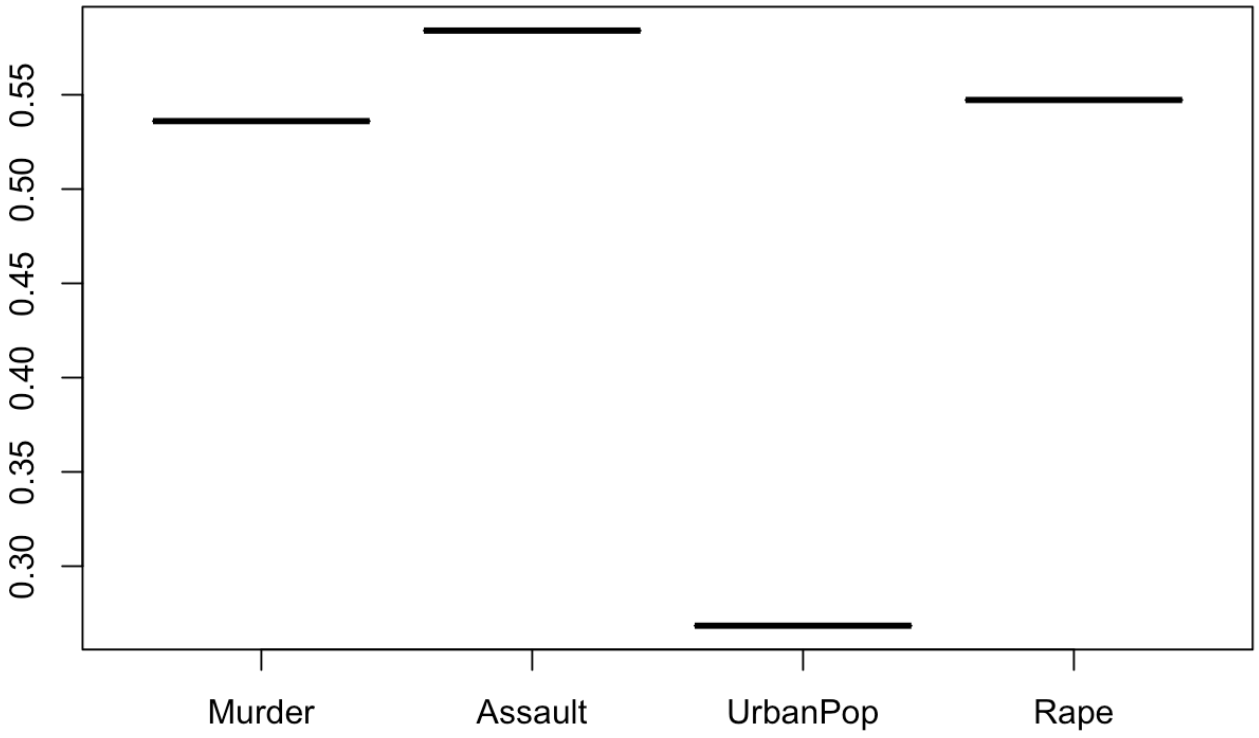
Part 4

```
# boot.fn = function(data, i) {
#   pr.out = prcomp(data[i,], scale=TRUE)
#   v = pr.out$rotation[,1]
#   i = which.max(abs(v))
#   return(v)
# }
# bs = boot(USArrests, boot.fn, R = 1000)
# boxplot(bs)

boot.fn = function(data, i) {
  sampled = sample(nrow(data[i, ]), 1000, replace = TRUE)
  d       = data[i, ][sampled, ]
  pca.out = prcomp(d, scale=T)
  pca.v   = pca.out$rotation[,1]
  i       = which.max(abs(pca.v))
  pca.vf  = pca.v * sign(pca.v[i])
  return(t(data.frame(pca.vf)))
}
```

Part 5

```
boxplot(boot.fn(USArrests))
```



Part 6

We solved our problem in part 3 by multiplying the principal component by the sign of its largest element. This method relies on the assumption that the correct boxplot doesn't range over both negative and positive values, overlapping the `y = 0` line. This particularly becomes a problem because the last principal components tend to have higher variance than the first, resulting in a taller box for those components in the box plot, which in turn makes it more likely that the box will overlap both positive and negative values.