Assignment 2

Statistics and Data Science 365/565

Due: September 27 (midnight)

This homework treats classification and cross validation, and gives you more experience using R.

```
library(FNN)
```

```
## Warning: package 'FNN' was built under R version 3.4.4
```

```
train <- read.csv("spam_train.csv")
test <- read.csv("spam_test.csv")
```

Problem 1: Spam, wonderful spam! (30 points)

Background

The dataset consists of a collection of 57 features relating to about 4600 emails and a label of whether or not the email is considered spam. You have a training set containing about 70% of the data and a test set containing about 30% of the data. Your job is to build effective spam classification rules using the predictors.

A Note about Features

The column names (in the first row of each .csv file) are fairly self-explanatory.

- Some variables are named `word_freq_(word)`, which suggests a calculation of the frequency of how many times a specific word appears in the email, expressed as a percentage of total words in the email multiplied by 100.
- Some variables are named `char_freq_(number)`, which suggests a count of the frequency of the specific ensuing character, expressed as a percentage of total characters in the email multiplied by 100. Note, these characters are not valid column names in R, but you can view them in the raw .csv file.
- Some variables are named `capital_run_length_(number)` which suggests some information about the average (or maximum length of, or total) consecutive capital letters in the email.
- `spam`: This is the response variable, 0 = not spam, 1 = spam.

Missing Values

Unfortunately, the `capital_run_length_average` variable is corrupted and as a result, contains a fair number of missing values. These show up as `NA` (the default way of representing missing values in R.)

Your Task

Part 1 (20%)

Use k-nearest neighbors regression with $k = 15$ to **impute** the missing values in the `capital_run_length_average` column using the other predictors after standardizing (i.e. rescaling) them. You may use a package such as

FNN that has k-nearest neighbors regression as a built-in function. There is no penalty for using a built-in function.

When you are done with this part, you should have no more NA's in the capital_run_length_average column in either the training or the test set. Make sure you show all of your work.

```
#Standardize all the columns except the `capital_run_length_average` column.
#We also dont want to include and standardize the spam indicator in the train data.
means <- colMeans(train[-c(55,58)])
sds <- apply(train[-c(55,58)], 2, sd)
for (x in names(train[-c(55,58)])){
     train[x] <- (train[x]-means[x])/sds[x]
     test[x] <- (test[x]-means[x])/sds[x]
}

#Now we do knn regression to predict the missing values. On the test and the training data.
train_knnreg <- knn.reg(train=train[!is.na(train$capital_run_length_average),][-c(55,58)],test=train[is
test_knnreg <- knn.reg(train=test[!is.na(test$capital_run_length_average),][-55],test=test[is.na(test$ca

#Now we need to impute these prediction values into our original data.
train$capital_run_length_average[is.na(train$capital_run_length_average)] <- train_knnreg$pred
sum(is.na(train$capital_run_length_average))
```

```
## [1] 0
```

```
test$capital_run_length_average[is.na(test$capital_run_length_average)] <- test_knnreg$pred
sum(is.na(test$capital_run_length_average))
```

```
## [1] 0
```

Part 2 (20%)

Write a function named knnclass() that performs k-nearest neighbors classification. This function will be more sophisticated than the base function in the following way:

- The function should automatically do a split of the training data into a sub-training set (80%) and a validation set (20%) for selecting the optimal k. (More sophisticated cross-validation is not necessary.)
- The function should standardize each column: for a particular variable, say x_1, compute the mean and standard deviation of x_1 **using the training set only**, say $\bar{x}_1$ and s_1; then for each observed x_1 in the training set and test set, subtract $\bar{x}_1$, then divide by s_1.

Function skeletons:

In R, start with:

```
knnclass <- function(xtrain, xtest, ytrain)
```

Note: You can assume that all columns will be numeric and that Euclidean distance is the distance measure.

```
knnclass <- function(xtrain, xtest, ytrain) {
  set.seed(1)
  #Standardize the xtrain and xtest in general.
  means <- colMeans(xtrain)
  sds <- apply(xtrain, 2, sd)
  for (x in names(xtrain)) {
      xtrain[x] <- (xtrain[x]-means[x])/sds[x]
      xtest[x] <- (xtest[x]-means[x])/sds[x]
  }
```

```
  #Split training into 20% validation set and 80% subtraining set.
  sample_size <- round(nrow(xtrain)*.20,digits=0)
  validationrows <- sample(nrow(xtrain), sample_size,replace=FALSE)
  validation <- xtrain[validationrows,]
  subtrain <- xtrain[-validationrows,]
  subtrain_labels <- ytrain[-validationrows]

  #Standardize subtrain and validation data.
  means <- colMeans(subtrain)
  sds <- apply(subtrain, 2, sd)
  for (x in names(subtrain)) {
      subtrain[x] <- (subtrain[x]-means[x])/sds[x]
      validation[x] <- (validation[x]-means[x])/sds[x]
  }

  #Train model on the different values of k (here we look from 1 to 30).
  train_error <- rep(0,30)
  for (k in c(1:30)) {
    trainmod <- knn(train=subtrain,test=validation,cl=subtrain_labels,k=k)
    train_error[k] <- (sum(subtrain_labels != trainmod))/length(subtrain_labels)
  }

  #Choose k where the train_error is minimal, this is our optimal k to use for knn on the test data.
  k_optimal <- match(min(train_error),train_error)
  testmod <- knn(train=xtrain,test=xtest,cl=ytrain,k=k_optimal)

  #Let's observe a plot of the training k values to see our optimal k.
  #plot(train_error,type="o",xlab="k values",ylab="Misclassification Error",main="Choosing Optimal K fr

  #Return the predicted values on the test data. Print the optimal k.
  print(paste("Optimal K value:",k_optimal))
  return(testmod)
}
```

Part 3 (60%)

In this part, you will need to use a k-NN classifier to fit models on the actual dataset. If you weren't able to successfully write a k-NN classifier in Part 2, you're permitted to use a built-in package for it. If you take this route, you may need to write some code to standardize the variables and select k, which `knnclass()` from part 2 already does.

Now fit 4 models and produce 4 sets of predictions of `spam` on the test set:

1. `knnclass()` using all predictors except for `capital_run_length_average` (say, if we were distrustful of our imputation approach). Call these predictions `knn_pred1`.

```
set.seed(1)
knn_pred1 <- knnclass(train[-c(55,58)],test[-55],train$spam)
```

```
## [1] "Optimal K value: 2"
```

2. `knnclass()` using all predictors including `capital_run_length_average` with the imputed values. Call these predictions `knn_pred2`.

```
set.seed(1)
knn_pred2 <- knnclass(train[-58],test,train$spam)
```

```
## [1] "Optimal K value: 2"
```

3. logistic regression using all predictors except for `capital_run_length_average`. Call these predictions `logm_pred1`.

```
set.seed(1)
mod1 <- glm(spam ~ ., data=train[-55], family="binomial")
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
logm_probs1 <- predict(mod1,test)
logm_pred1 <- rep("0", nrow(test))
logm_pred1[logm_probs1  > .5] <-  "1"
```

4. logistic regression using all predictors including `capital_run_length_average` with the imputed values. Call these predictions `logm_pred2`.

```
set.seed(1)
mod2 <- glm(spam ~ ., data=train, family="binomial")
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
summary(mod2)
```

```
##
## Call:
## glm(formula = spam ~ ., family = "binomial", data = train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -4.0919  -0.2125   0.0000   0.1275   4.5985
##
## Coefficients:
##                            Estimate Std. Error z value Pr(>|z|)
## (Intercept)                -9.55717    2.18916  -4.366 1.27e-05 ***
## word_freq_make             -0.08826    0.07905  -1.116 0.264240
## word_freq_address          -0.20537    0.11524  -1.782 0.074736 .
## word_freq_all               0.09903    0.06713   1.475 0.140190
## word_freq_3d                3.23638    2.76732   1.169 0.242203
## word_freq_our               0.38683    0.08514   4.543 5.53e-06 ***
## word_freq_over              0.24582    0.08401   2.926 0.003434 **
## word_freq_remove            0.73547    0.13607   5.405 6.47e-08 ***
## word_freq_internet          0.24765    0.09728   2.546 0.010907 *
## word_freq_order             0.12480    0.08861   1.408 0.159022
## word_freq_mail              0.16103    0.06283   2.563 0.010380 *
## word_freq_receive          -0.05729    0.06843  -0.837 0.402474
## word_freq_will             -0.07315    0.07335  -0.997 0.318613
## word_freq_people           -0.04017    0.08291  -0.484 0.628068
## word_freq_report            0.05190    0.05141   1.010 0.312724
## word_freq_addresses         0.21989    0.18562   1.185 0.236162
## word_freq_free              0.74210    0.14138   5.249 1.53e-07 ***
## word_freq_business          0.43366    0.12478   3.475 0.000510 ***
## word_freq_email             0.11317    0.07811   1.449 0.147354
## word_freq_you               0.12436    0.07594   1.638 0.101522
## word_freq_credit            0.55114    0.35199   1.566 0.117400
## word_freq_your              0.35968    0.07789   4.618 3.88e-06 ***
```

```
## word_freq_font               0.23514    0.19743   1.191 0.233650
## word_freq_000                0.85339    0.20319   4.200 2.67e-05 ***
## word_freq_money              0.35326    0.14755   2.394 0.016660 *
## word_freq_hp                -2.81409    0.54157  -5.196 2.03e-07 ***
## word_freq_hpl               -0.84526    0.40398  -2.092 0.036408 *
## word_freq_george           -28.41121    6.78436  -4.188 2.82e-05 ***
## word_freq_650                0.16506    0.12559   1.314 0.188738
## word_freq_lab               -1.20400    0.76890  -1.566 0.117375
## word_freq_labs              -0.13200    0.15052  -0.877 0.380495
## word_freq_telnet            -0.05538    0.52588  -0.105 0.916126
## word_freq_857                0.92741    0.84306   1.100 0.271306
## word_freq_data              -0.32578    0.19495  -1.671 0.094706 .
## word_freq_415                0.13600    0.46827   0.290 0.771488
## word_freq_85                -0.78614    0.37353  -2.105 0.035325 *
## word_freq_technology         0.36624    0.15386   2.380 0.017299 *
## word_freq_1999               0.02500    0.08999   0.278 0.781132
## word_freq_parts             -0.11136    0.09649  -1.154 0.248430
## word_freq_pm                -0.52477    0.22907  -2.291 0.021972 *
## word_freq_direct             0.14191    0.28597   0.496 0.619735
## word_freq_cs               -15.29477   12.79930  -1.195 0.232099
## word_freq_meeting           -1.82306    0.64867  -2.810 0.004947 **
## word_freq_original          -0.17794    0.18021  -0.987 0.323462
## word_freq_project           -0.92403    0.42019  -2.199 0.027871 *
## word_freq_re                -0.82652    0.17966  -4.600 4.22e-06 ***
## word_freq_edu               -1.31571    0.29338  -4.485 7.30e-06 ***
## word_freq_table             -0.08289    0.14233  -0.582 0.560312
## word_freq_conference        -1.27322    0.61628  -2.066 0.038831 *
## char_freq_.                 -0.29885    0.12891  -2.318 0.020429 *
## char_freq_..1                0.03284    0.08490   0.387 0.698854
## char_freq_..2               -0.10160    0.14189  -0.716 0.473933
## char_freq_..3                0.21720    0.05790   3.752 0.000176 ***
## char_freq_..4                0.95349    0.16001   5.959 2.54e-09 ***
## char_freq_..5                0.86349    0.42638   2.025 0.042850 *
## capital_run_length_average   0.05218    0.04081   1.278 0.201076
## capital_run_length_longest   2.18974    0.57909   3.781 0.000156 ***
## capital_run_length_total     0.31736    0.15076   2.105 0.035291 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 4316.6  on 3219  degrees of freedom
## Residual deviance: 1295.2  on 3162  degrees of freedom
## AIC: 1411.2
##
## Number of Fisher Scoring iterations: 13
```

```
logm_probs2 <- predict(mod2,test)
logm_pred2 <- rep("0", nrow(test))
logm_pred2[logm_probs2  > .5] <-  "1"
```

In 3-4 sentences, provide a quick summary of your second logistic regression model (model 4). Which predictors appeared to be most significant? Are there any surprises in the predictors that ended up being significant or not significant?

Comments: We see that the characters 3 and 4 (! and the dollar sign respectively) are highly significant predictors of spam mail, as most scams are involving exciting fake investments. Similarly the word "free" is a significant predictor, trying to fool people into a fake free prize. Additionally, the longest run of capital letters is a significant predictor, which could be indicative of human grammatical errors (as scammers intentionally try to mimimic human errors to make them appear more real. However, "recieve" and "credit" are not significant and "money" is only significant to the 0.5 level, which are surprising because they do not seem much different than "free" and the dollar sign.

Submit a .csv file called `assn2_NETID_results.csv` that contains 5 columns:

- `capital_run_length_average`: the predictor in your test set that now contains the imputed values (so that we can check your work on imputation).
- `knn_pred1`
- `knn_pred2`
- `logm_pred1`
- `logm_pred2`

Make sure that row 1 here corresponds to row 1 of the test set, row 2 corresponds to row 2 of the test set, and so on.

```
assn2_my354_results <- data.frame(c(1:nrow(test)))
assn2_my354_results$capital_run_length_average <- test$capital_run_length_average
assn2_my354_results$knn_pred1 <- knn_pred1[1:nrow(test)]
assn2_my354_results$knn_pred2 <- knn_pred2[1:nrow(test)]
assn2_my354_results$logm_pred1 <- logm_pred1
assn2_my354_results$logm_pred2 <- logm_pred2
assn2_my354_results <- assn2_my354_results[-1]
write.csv(assn2_my354_results,"assn2_my354_results.csv")
```

Problem 2: Gradient descent (25 points)

Stochastic gradient descent is a variation on "ordinary" or "batch" gradient descent. They both are based on the fact that the gradient of a function points in the direction of greatest increase, and therefore the negative gradient gives us the direction of greatest decrease. For this problem you get some practice using both forms of gradient descent for logistic regression.

For $i = 1, 2, ..., n$ we have data points $X_i = (1, X_{i1}, X_{i2})^T \in \mathbb{R}^3$ with binary outcomes $Y_i \in \{0, 1\}$. For logistic regression, we attempt to classify the ith data point as either 0 or 1 based on X_i. To do so, we need to minimize the logistic loss function:

$$\ell(\beta) = \ell(\beta_0, \beta_1, \beta_2) = \sum_{i=1}^{n} \left\{ -Y_i X_i^T \beta + \log\left(1 + e^{X_i^T \beta}\right) \right\}.$$

where $X_i^T \beta = \beta_0 + X_{i1}\beta_1 + X_{i2}\beta_2$, so that an intercept is included.

Part (a)

Calculate the gradient of $\ell(\beta)$). Show that $\ell(\beta)$ is a convex function of β, and therefore has a unique minimum.

Solution:

$$\nabla \ell(\beta) = \sum_{i=1}^{n} \left(-Y_i X_i^T + \frac{1}{1 + e^{X_i^T \beta}} * e^{X_i^T \beta} * X_i^T \right)$$

We know that sum of convex functions is a convex function. So to prove that $\ell(\beta)$ is convex, we just need to prove that each part is convex. First we know that $-Y_i X_i^T \beta$ is convex because it is just a linear function. So we just need to prove that $\log\left(1 + e^{X_i^T \beta}\right)$ is a convex function. We can also prove that any function is convex by showing that its second derivative is positive. Let's observe the seocnd derivative of $\log\left(1 + e^{X_i^T \beta}\right)$:

$$f(\beta) = \log\left(1 + e^{X_i^T \beta}\right) \; f'(\beta) = \frac{e^{X_i^T \beta}}{1 + e^{X_i^T \beta}} * X_i^T \; f''(\beta) = \frac{e^{X_i^T \beta}}{1 + e^{X_i^T \beta}} * X_i^T - \frac{(e^{X_i^T \beta})^2}{(1 + e^{X_i^T \beta})^2} * X_i^T X_i$$

Here we can see that $\frac{e^{X_i^T \beta}}{1 + e^{X_i^T \beta}} = \pi_i$ where π_i is the probability of classifying $Y_i = 1$. So we can view this as $f''(\beta) = \pi_i(1 - \pi_i) X_i^T X_i$. We can see that $\pi_i(1 - \pi_i)$ is just the Bernoulli variance with probability of succes π_i, which is positive. Also, $X_i^T X_i$ is a positive definite square matrix where X_i has full column rank, because for any vector v, $v_T X_i^T X_i v = (X_i v)^T (X v) = \|X_i v\|_2^2 \geq 0$. Thus the second derivative is positive and $\log\left(1 + e^{X_i^T \beta}\right)$ is convex and therefore the enitre function $\ell(\beta)$ is convex.

Part (b)

For this part of the problem you will generate data and compute the maximum likelihood estimator using the standard approach, which is Newton's algorithm. Simulate data for this problem as follows:

```
n = 1000
x1 = runif(n, min=-5, max=5)
x2 = runif(n, min=3, max=9)
x = data.frame(one=1, x1, x2)

beta = c(2.5, 1.0, -.5);
p = exp(as.matrix(x) %*% beta) / (1 + exp(as.matrix(x) %*% beta))
y = sapply(p, function(p) {rbinom(1,1,p)})
```

Next fit the logistic regression model using the built-in `glm` function:

```
model <- glm(y ~ x1+x2, family=binomial(link='logit'), data=x)
beta_op <- model$coefficients
print(summary(model))
```

```
##
## Call:
## glm(formula = y ~ x1 + x2, family = binomial(link = "logit"),
##     data = x)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -3.02464  -0.40866  -0.09312   0.38487   2.88985
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  2.70509    0.38589   7.010 2.38e-12 ***
## x1           1.02461    0.06317  16.219  < 2e-16 ***
## x2          -0.52568    0.06406  -8.207 2.28e-16 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1377.45  on 999  degrees of freedom
## Residual deviance:  618.87  on 997  degrees of freedom
## AIC: 624.87
##
## Number of Fisher Scoring iterations: 6
```

```
#Here we actually generate our data and run the model 1000 times.
#We save our coefficients so later we can observe the mean coefficients and calculate the standard erro
set.seed(1)
#Figure out how to properly store these coefficients? rbind() to a new dataframe?
coefficientsdf <- data.frame("Intercept"=as.numeric(),"Beta1"=as.numeric(),"Beta2"=as.numeric())
for (i in c(1:1000)) {
  n = 1000
  x1 = runif(n, min=-5, max=5)
  x2 = runif(n, min=3, max=9)
  x = data.frame(one=1, x1, x2)

  beta = c(2.5, 1.0, -.5);
  p = exp(as.matrix(x) %*% beta) / (1 + exp(as.matrix(x) %*% beta))
  y = sapply(p, function(p) {rbinom(1,1,p)})

  newmod <- glm(y ~ x1+x2, family=binomial(link='logit'), data=x)
  coefficientsdf <- rbind(c(newmod$coefficients[[1]],newmod$coefficients[[2]],newmod$coefficients[[3]])
}
names(coefficientsdf)[1] <- "Intercept (Beta0)"
names(coefficientsdf)[2] <- "Beta1"
names(coefficientsdf)[3] <- "Beta2"
meancoef <- colMeans(coefficientsdf)
#Standard error is sd/sqrt(n), however this doesn't give a similar stanrard error.
#However, just the standard deviation gives a very similar answer.
sderrors <- apply(coefficientsdf, 2, sd)
print("Mean Coefficients:")
```

```
## [1] "Mean Coefficients:"
```

```
meancoef
```

```
## Intercept (Beta0)             Beta1             Beta2
##         2.5083362         1.0097516        -0.5020647
```

```
print(paste("Sum of Squared of Differences:",sum((beta_op-meancoef)^2)))
```

```
## [1] "Sum of Squared of Differences: 0.0394916575139669"
```

```
print("Standard Errors")
```

```
## [1] "Standard Errors"
```

```
sderrors
```

```
## Intercept (Beta0)             Beta1             Beta2
##        0.39163557        0.06155288        0.06471885
```

Do you recover the true coefficients? Note that this uses "Fisher scoring" which is the same as Newton's

algorithm. Run this simulation many times, keeping track of the coefficients $\widehat{\beta} = (\widehat{\beta}_0, \widehat{\beta}_1, \widehat{\beta}_2)$. Using these values, estimate the standard errors of the coefficients. Do they agree with the standard errors shown by `summary(model)`? Comment on your findings.

Comments: After generating data and fitting the model 1000 times, my mean coefficients are very similar to the true beta (the sum of squared differences between the true beta and the mean coefficients is 0.11). We are essentially recovering the true coefficients. Additionally, the standard errors are very similar.

Part (c)

The simulated data $\{(X_i, Y_i)\}_{i=1}^{1000}$ defines a loss function $\ell(\beta)$ that we'll now minimize using *batch* gradient descent, where the full data set is used to compute the gradient. Starting with a uniform model, with $\beta = (\beta_0, \beta_1, \beta_2) = (0, 0, 0)$, compute the negative gradient evaluated at point, adjust your estimate accordingly, and repeat the process until it (hopefully) converges. You will need to pick a step size, η_t, the form of which you are free to decide on (either constant or varying with step t). Comment on your choice of η_t. It is also up to you to decide when ℓ is approximately minimized. Does it converge? If so, how do you assess convergence? How long does it take to converge? Do you recover the true model? Comment on your findings.

```
#Here let's regenerate our data with a specific seed so our findings for the rest of problem 2 are repr
set.seed(1)
n = 1000
x1 = runif(n, min=-5, max=5)
x2 = runif(n, min=3, max=9)
x = data.frame(one=1, x1, x2)

beta = c(2.5, 1.0, -.5);
p = exp(as.matrix(x) %*% beta) / (1 + exp(as.matrix(x) %*% beta))
y = sapply(p, function(p) {rbinom(1,1,p)})
```

```
#Define loss function.
loss <- function(beta,x,y) {
  lossvalue <- 0
  for (i in c(1:nrow(x))) {
    ichange <- -y[i]%*%as.matrix(x[i,])%*%beta + log(1+exp(as.matrix(x[i,])%*%beta))
    lossvalue <- lossvalue + ichange
  }
  return(lossvalue)
}

#Define a function that calculates the gradient for a given beta.
grad <- function(beta,x,y) {
  gradient = as.matrix(c(0,0,0), ncol=1)
  for (i in c(1:nrow(x))) {
    ichange <- as.matrix(c(-y[i]%*%as.matrix(x[i,]) + drop(exp(as.matrix(x[i,])%*%beta)/(1+exp(as.matri
    gradient <- gradient+ichange
  }
  return(gradient)
}
```

```
beta <- as.matrix(c(0,0,0),ncol=1)
t <- 1
C <- .01
repeat{
  prevbeta <- beta
  step_size <- C/sqrt(t)
```

```
  beta <- beta - step_size*grad(beta,x,y)
  t <- t+1
  #We can check is the change in beta is minimal as convergence.
  #But this essentially is just capping the number of iterations, because step_size decreases each time
  #Let's just make sure our loss function is no longer decreasing much for our convergence criteria.
  if(abs(loss(beta,x,y)-loss(prevbeta,x,y)) < 0.025) {
    break
  }
}
beta_bgd <- beta
t
```

```
## [1] 45
```

```
beta_bgd
```

```
##            [,1]
## [1,]  2.2206151
## [2,]  0.9977901
## [3,] -0.4444177
```

Comments: I set my step size constant through trial and error. If the step size constant was too large, then the Beta would have too large an update and the loss would be extremely high. It would be osicallting around the minimum, not approaching it. If too small, then we are running too many iterations. I use the aggressive approach to changing the step size by dividing our constant by sqrt(t), where t is the number of iterations. My convergance criteria was to see if the absolute change in the loss function was less than 0.025. Each iteration takes a long time because we are calculating the entire gradient and the loss function for beta and our previous beta. It only take 44 iterations to converge under this criteria. **However, if I decrease my convergence criteria to the change being less than 0.01 or change my step size factor to t instead of sqrt(t), then after around 46 iterations my Beta starts to get further away from the true Beta and the loss begins to grow larger. I get a lot oscillation which is weird since my step size is still decreasing with each iteration.**

Part (d)

Now repeat the above, but use *stochastic* gradient descent, where you compute the gradient using only a *single* data point in each step. Like before, you will need to pick a step size, η_t, the form of which you are free to decide on (either constant or varying with step t). How does the step size η_t that you choose differ from that used for batch gradient descent? Now how do you assess convergence? Do you recover the true model? Comment on the speed of convergence and computation required for both batch and stochastic gradient descent.

```
#Funtion to choose optimal constant to minimize loss function.
#Found that there was lots of differences in convergence rate for SGD depending on C.
#So let's choose the C that minimizes loss over 100 iterations (thus assuming it converges faster).
set.seed(1)
constants <- (seq(0.1,0.5,.01))
losses <- rep(0,length(constants))
for (j in c(1:length(constants))) {
  beta <- as.matrix(c(0,0,0),ncol=1)
  C <- constants[j]
  for (l in c(1:100)) {
    prevbeta <- beta
    row <- sample(1:nrow(x),1)
    step_size <- C/sqrt(l)
    beta <- beta-step_size*grad(beta,x[row,],y[row])
```

```
  }
  losses[j] <- loss(beta,x,y)
}
C_minloss <- constants[match(min(losses),losses)]
C_minloss
```

```
## [1] 0.31
```

```
set.seed(1)
beta <- as.matrix(c(0,0,0),ncol=1)
t <- 1
repeat{
  j <- sample(1:nrow(x),1)
  prevbeta <- beta
  #We want to reduce step size at each iteration.
  step_size <- C_minloss/sqrt(t)
  #Calculate new beta in the direction of a random partial derivative.
  beta <- beta-step_size*grad(beta,x[j,],y[j])
  #print(t)
  t <- t+1
  #Here let's set convergence as the change in beta being less than 10^-9.
  #Because we are decreasing step size each time, this is effectively setting a limit on the number of i
  #We could check is the loss is no longer decreasing like in part(c), but this is not computationally e
  if(sum((beta-prevbeta)^2) < 10^-(11)) {
    break
  }
}
beta_sgd <- beta
t
```

```
## [1] 589995
```

```
beta_sgd
```

```
##            [,1]
## [1,]  2.6182233
## [2,]  1.0270880
## [3,] -0.5385105
```

Comments: After running our first chunk of code, we find the optimal constant to begin the step size with is 0.31 (compared to 0.01 in batch gradient descent). We then use that as our initial step size, and then we continue to decrease the step size by dividing the constant by the square root of the number of iterations. Here, I assess convergence by the sum of squared differences in beta being less than 10^-11. This effectively sets our convergence criteria as the number of steps to occur. After 589,995 iterations, our beta converges to the above. Each iterations computes significantly faster than in batch gradient descent, however since we are doing so many iterations it takes several minutes to run. Our beta converges much closer to the true beta than our batch gradient descent alogrithm above.

Problem 3: Cross validation (20 points)

(a) Generate a simulated data set as follows:

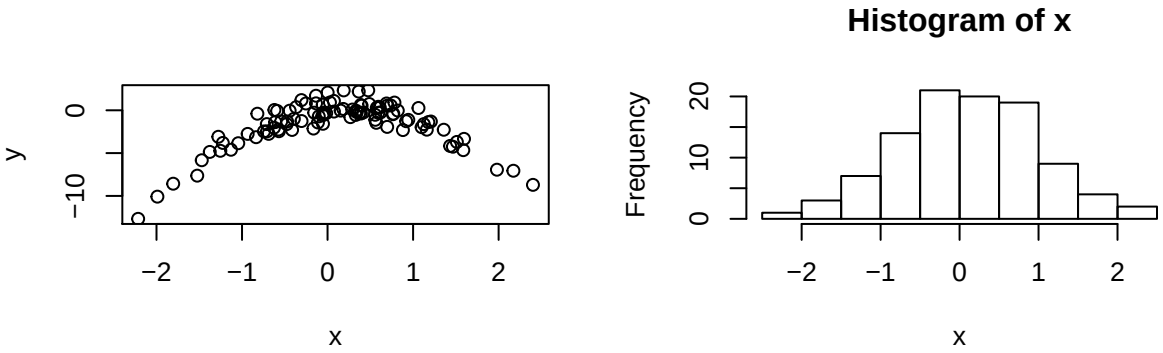
```
set.seed(1)
x <- rnorm(100)
y <- x - 2*x^2 + rnorm(100)
```

In this data set, what is n and what is p? Write out the model used to generate the data in equation form.

Solution: n=100 because we have n data points/observations and p=2 because we have 2 predictor columns (x and xˆ2, there is no intercept and the epsilon term are just normally distributed errors). $Y = f(X) = X - 2X^2 + \epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$.

(b) Create a scatterplot of X against Y. Comment on what you find.

```
par(mfrow=c(2,2))
plot(x,y)
hist(x,breaks=10)
```



Solution: The scatterplot looks like a concave down quadratic function. However, the points are not clearly fit along a single line, there is error around what appears to be a quadratic shape. The points themselves appear normally distributed across x. A histogram plot of x confirms this. This makes sense because we are selecting x values from the rnorm(100) function which chooses 100 random points from the standard normal distribution.

(c) Set a random seed, and then compute the Leave-One-Out Cross-Validation (LOOCV) errors that result from fitting the following four models using least squares:

i. $Y = \beta_0 + \beta_1 X + \epsilon$
ii. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \epsilon$
iii. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \epsilon$
iv. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \beta_4 X^4 + \epsilon$

Note you may find it helpful to use the data.frame() function to create a single data set containing both X and Y. For linear regression, the LOOCV error can be computed via the following short-cut formula:

$$\text{LOOCV Error} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{Y_i - \widehat{Y}_i}{1 - H_{ii}}\right)^2$$

where H_{ii} is the i^{th} diagonal entry of the projection matrix $H = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$, and $\mathbf{X}$ is a matrix of predictors (the design matrix). This formula is an alternative to actually carrying out the $n = 100$ regressions you would otherwise need for LOOCV.

```
set.seed(1)
df <- data.frame("Y"=y, "Intercept"=rep(1,100), "X"=x, "X2"=x^2, "X3"=x^3, "X4"=x^4)

#Let's define a function that takes a model and the model the data was fit on and outputs the LOOCV erro
LOOCVerror <- function(model,data) {
    Xmat <- as.matrix(data[-1])
    H <- Xmat%*%solve(t(Xmat)%*%Xmat)%*%t(Xmat)
    Y <- data[1]
    Yhat <- predict(model,data[-1])
    error <- 1/nrow(data)*sum(((Y - Yhat)/(1-diag(H)))^2)
    return(error)
```

```
}
mod1 <- lm(Y ~ X, data=df)
LOOCVerror(mod1,df[1:3])
```

```
## [1] 7.288162
```

```
mod2 <- lm(Y ~ X + X2, data=df)
LOOCVerror(mod2,df[1:4])
```

```
## [1] 0.9374236
```

```
mod3 <- lm(Y ~ X + X2 + X3, data=df)
LOOCVerror(mod3,df[1:5])
```

```
## [1] 0.9566218
```

```
mod4 <- lm(Y ~ X + X2 + X3 + X4, data=df)
LOOCVerror(mod4,df[1:6])
```

```
## [1] 0.9539049
```

(d) Repeat (c) using another random seed, and report your results. Are your results the same as what you got in (c)? Why?

```
set.seed(2)
x <- rnorm(100)
y <- x - 2*x^2 + rnorm(100)
df <- data.frame("Y"=y, "Intercept"=rep(1,100), "X"=x, "X2"=x^2, "X3"=x^3, "X4"=x^4)

newmod1 <- lm(Y ~ X, data=df)
LOOCVerror(mod1,df[1:3])
```

```
## [1] 11.11186
```

```
newmod2 <- lm(Y ~ X + X2, data=df)
LOOCVerror(newmod2,df[1:4])
```

```
## [1] 1.00441
```

```
newmod3 <- lm(Y ~ X + X2 + X3, data=df)
LOOCVerror(newmod3,df[1:5])
```

```
## [1] 1.01803
```

```
newmod4 <- lm(Y ~ X + X2 + X3 + X4, data=df)
LOOCVerror(newmod4,df[1:6])
```

```
## [1] 1.035601
```

*Solution: My results are similar to those above. This makes sense because we are just regenerating the data with a different random seed, but modeling the data in the same way.

(e) Which of the models in (c) had the smallest LOOCV error? Is this what you expected? Explain your answer.

Solution: We have the highest error rate for the linear model. We have the lowest error rate for the quadratic model. Our error rate for the higher order polynomial models (p=3 and p=4) are lower than that of the linear model, but slightly higher than that of the quadratic model. This makes sense because again the the data is fit by the equation: $Y = f(X) = X - 2X^2 + \epsilon$.

(f) Comment on the statistical significance of the coefficient estimates that results from fitting each of the models in (c) using least squares. Do these results agree with the conclusions drawn based on the cross-validation results?

```r
summary(mod1)
```

```
##
## Call:
## lm(formula = Y ~ X, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -9.5161 -0.6800  0.6812  1.5491  3.8183
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -1.6254     0.2619  -6.205 1.31e-08 ***
## X             0.6925     0.2909   2.380   0.0192 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.6 on 98 degrees of freedom
## Multiple R-squared:  0.05465,    Adjusted R-squared:  0.045
## F-statistic: 5.665 on 1 and 98 DF,  p-value: 0.01924
```

```r
summary(mod2)
```

```
##
## Call:
## lm(formula = Y ~ X + X2, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.9650 -0.6254 -0.1288  0.5803  2.2700
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.05672    0.11766   0.482    0.631
## X            1.01716    0.10798   9.420  2.4e-15 ***
## X2          -2.11892    0.08477 -24.997  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.958 on 97 degrees of freedom
## Multiple R-squared:  0.873,  Adjusted R-squared:  0.8704
## F-statistic: 333.3 on 2 and 97 DF,  p-value: < 2.2e-16
```

```r
summary(mod3)
```

```
##
## Call:
## lm(formula = Y ~ X + X2 + X3, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.9765 -0.6302 -0.1227  0.5545  2.2843
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.06151    0.11950   0.515    0.608
```

```
## X            0.97528    0.18728   5.208 1.09e-06 ***
## X2          -2.12379    0.08700 -24.411  < 2e-16 ***
## X3           0.01764    0.06429   0.274    0.784
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9626 on 96 degrees of freedom
## Multiple R-squared:  0.8731, Adjusted R-squared:  0.8691
## F-statistic: 220.1 on 3 and 96 DF,  p-value: < 2.2e-16
```

```
summary(mod4)
```

```
##
## Call:
## lm(formula = Y ~ X + X2 + X3 + X4, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.0550 -0.6212 -0.1567  0.5952  2.2267
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.156703   0.139462   1.124    0.264
## X            1.030826   0.191337   5.387 5.17e-07 ***
## X2          -2.409898   0.234855 -10.261  < 2e-16 ***
## X3          -0.009133   0.067229  -0.136    0.892
## X4           0.069785   0.053240   1.311    0.193
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9591 on 95 degrees of freedom
## Multiple R-squared:  0.8753, Adjusted R-squared:  0.8701
## F-statistic: 166.7 on 4 and 95 DF,  p-value: < 2.2e-16
```

Solution: In the polynomial models (models 2-4), only the X and X^2 terms are statistically significant. Again this makes sense because our data is modeled with a quadratic equation. This is in agreement with our cross-validation results, which suggest the quadratic model is the best because it gives the lowest error rate. Adding more statistically insignifcant predictors ends up increasing our error rate (models 3 and 4). However, in model 1, we are missing the X^2 predictor. So all the intercept and X predictors are both statistically significant, but our cross-validation error rate is still too high, because we are trying to fit our data with too simple a model (linear model on quadratic data).