Homework 2 (Coding Portion)

Muhammad Ahmed Chaudhry, Isaac Kleisle-Murphy, Kai Okada

2/24/2021

Public GitHub link here: https://github.com/isaackleislemurphy/Stanford-STATS-315B/tree/main/PS2

Problem 2 - Stepwise Linear Regression

Part (a)

The fitted intercept if $\mathbf{X}$ is mean-centered is $\frac{1}{N}\sum_{i=1}^{N} y_i$. When we mean-center the columns of X, this intercept will change, but the other coefficients will not. As proof, we can see that we originally obtain predictions based on $\hat{y}_i = \hat{\beta}_0 + \sum_{j=1}^{p} x_i j \hat{\beta}_j$. With centering, we can rearrange this as: $\hat{y}_i = (\hat{\beta}_0 + \sum_{i=1}^{p} \mu_j \hat{\beta}_j) + \sum_{j=1}^{p}(x_{ij} - \mu_j)\hat{\beta}_j$. Note that any values of $\hat{\beta}$ that could reduce the error further for the centered case would also reduce the error in the non-centered case. Therefore, the coefficient vector $\hat{\beta}$ remains consistent and the intercept shifts by the sum of the average for each feature multiplied by its coefficient.

Part (b) Determining Next Feature

If we take the product $Z^T r_s \in \mathbb{R}^{p-q}$, the elements of larger magnitude will correspond to features that are more highly correlated with the residuals, so adding them to the model will contribute most to fitting the model in the direction of the residuals, i.e. reducing error.

Part (c) Overall Strategy

At a high-level, our plan to perform forward stepwise regression breaks down as follows:
0.) Mean-center the features. This saves us intercept recalculations down the road, as no matter how many (mean-centered) features we add, this will remain the intercept.
1.) Initialise the "null" model to be the mean of the y. This mean, by the centering in 0.), will remain the intercept hereafter.
2.) Scan the remaining p features; for each, compute a univariate regression, and the univariate regression that maximizes $||(y - \bar{y}||_2^2 - RSS_{uv}$ most is the first feature included.
3.) Once this feature has been selected, and it's (univariate) coefficient saved, initialise a QR factorization process (i.e. a running Graham-Schmidt algorithm) with this first feature as the "jumping off point".
4.) Scan the remaining $p - 1$ features. For each feature, temporarily add it to the running Graham-Schmidt/QR decomposition. Then use the triangular structure of the QR decomposition to efficiently solve for the temporary/proposed coefficients of the next step.
5.) Evaluate the drops in RSS's of all the "temporary" Graham-Schmidt advancements (these are cheap – they just build off of the previous "permanent"/selected features); retain that which results in the largest drop permanently.
6.) Repeat 4-5 until all features have been used up.
7.) Now, Q is an $n \times p$ orthogonal matrix, and R is a $p \times p$ upper triangular matrix. If we wish to retrieve coefficients retroactive to a particular step number, we need only to backsolve via Q[1:step, 1:step], R[1:step, 1:step]. In this way, the QR/G-S decomposition allows us to preserve previous iterations of the model, for

ease of comparison.

Part (d) Forward Stepwise Regression in R

```
classify <- function(y_hats) {
  sapply(y_hats, function(y) ifelse(y >= .5, 1, 0));
}

lm_step_construct <- function(X, y){
  #' Constructor function for lm_step object
  #' @param X: matrix. A matrix of features to train on.
  #' @param y: numeric. A vector of targets to train on
  #' @return : lm_step. a constructed lm_step object.

  Z = matrix(rep(0, nrow(X) * (ncol(X)+1)), ncol=ncol(X)+1)
  Z[, 1] = 1
  lm_init = list(
    n=nrow(X), # number of samples
    p=ncol(X), # max no. predictors
    k=0, # no features chosen yet
    X=X, # the data
    x_sel_idx=c(),
    X_sel=matrix(rep(NA, nrow(X)*ncol(X)), ncol=ncol(X)), # actual columns selected
    X_nonsel_idx=1:ncol(X), # indices of columns in X not selected
    y=y, # targets
    beta_step=matrix(rep(0, ncol(X)^2), ncol=ncol(X)), # log of betas
    r2_step=c(), # r-squared at each step
    rss_step=c(), # rss at each step
    misclass_step=c(), # misclassification error at each step
    tss =sum(sapply(y, function(y_i) (y_i - mean(y))^2)),
    # is_fit=F, # for checks
    j=1 # current predictor number
  )
  lm_init$Q = matrix(rep(0, (lm_init$p) * lm_init$n), ncol=lm_init$p)
  lm_init$R = matrix(rep(0, (lm_init$p)^2), ncol=lm_init$p)
  attr(lm_init, "class") = "lm_step"
  lm_init
}

advance_qr <- function(lm_step, new_col_idx){
  #' Performs G-S decomposition for one new column of X
  #' @param lm_step: lm_step. An lm_step object, partially fitted
  #' @param new_col_idx: integer. The index of the new column to add to the decomposition.
  #' @return : lm_step.

  # GRAM SCHMIDT STEP
  ai = lm_step$X[, new_col_idx]
  if (lm_step$j == 1 & is.null(lm_step$X_sel)){
    vi = ai
  }else{
    vi = ai + lapply(1:(lm_step$j - 1), function(ll)
      - as.numeric(t(ai) %*% lm_step$Q[, ll]) * lm_step$Q[, ll]
```

```
    ) %>%
      do.call("cbind", .) %>%
      rowSums()
  }

  ei = vi/as.numeric(sqrt(t(vi) %*% vi))
  lm_step$Q[, lm_step$j] = ei
  lm_step$x_sel_idx  = append(lm_step$x_sel_idx, new_col_idx)
  lm_step$X_nonsel_idx = setdiff(lm_step$X_nonsel_idx, new_col_idx)
  lm_step$X_sel[, lm_step$j] = lm_step$X[, new_col_idx]
  lm_step$R[1:(lm_step$j), lm_step$j] = sapply(1:lm_step$j, function(ll) t(ai)
                                               %*% lm_step$Q[, ll])
  lm_step$j = lm_step$j + 1
  lm_step
}

get_beta <- function(lm_step, k=NULL){
  #' Gets the coefficients (not the intercept) from an lm_step object,
  #' retroactive to a step number
  #' @param lm_step: lm_step. A (at least partially) fitted lm_step object
  #' @param k: integer. The stepnumber to get beta retroactive to.
  #'           If null, uses entirety of fit
  #' @return : numeric[k]. A vector of OLS coefficients
  y = lm_step$y
  k = ifelse(is.null(k), lm_step$j-1, k)
  Q = lm_step$Q[, 1:k]; R = lm_step$R[1:k, 1:k]
  backsolve(R, crossprod(Q, y)) %>%
    as.numeric() -> beta_current

  beta_current
}

# helper function to update lm_step with new fit
log_intermediates <- function(lm_step, beta_current) {

  lm_step$beta_step[lm_step$j-1,lm_step$x_sel_idx] = beta_current

  if (length(beta_current) == 1){
    rss_current = sum((lm_step$y - mean(lm_step$y) -
                         lm_step$X_sel[, 1:(lm_step$j-1)] * beta_current)^2)
    misclass_current = mean(lm_step$y != classify(mean(lm_step$y) +
                                                    lm_step$X_sel[, 1:(lm_step$j-1)] * beta_current))
  }else{
    rss_current = sum((lm_step$y - mean(lm_step$y) -
                         lm_step$X_sel[, 1:(lm_step$j-1)] %*% beta_current)^2)
    misclass_current = mean(lm_step$y != classify(mean(lm_step$y) +
                                                    lm_step$X_sel[, 1:(lm_step$j-1)] %*% beta_current))
  }
  lm_step$rss_step = append(lm_step$rss_step, rss_current)
  lm_step$r2_step = append(lm_step$r2_step, 1 - rss_current / lm_step$tss)
  lm_step$misclass_step = append(lm_step$misclass_step, misclass_current)
  lm_step
}
```

```
advance_selection <- function(lm_step){
  #' Identifies which "next feature" to add, and adds that feature to a
  #' copied version of the input
  #' @param lm_step: An partially fit lm_step object
  #' @return : lm_step. The lm_step, having been advanced to include the
  #'          next available feature that most reduces RSS

  beta_current = get_beta(lm_step);
  #beta_compare = coef(lm(lm_step$y ~ lm_step$X_sel[, 1:(lm_step$j-1)])) compare results to lm

  if (length(beta_current) == 1){
    rss_current = sum((lm_step$y - mean(lm_step$y) -
                         lm_step$X_sel[, 1:(lm_step$j-1)] * beta_current)^2)
  }else{
    rss_current = sum((lm_step$y - mean(lm_step$y) -
                         lm_step$X_sel[, 1:(lm_step$j-1)] %*% beta_current)^2)
  }

  candidates = lapply(lm_step$X_nonsel_idx, function(ll){
    lm_step_cand = advance_qr(lm_step, ll);
    beta_cand = get_beta(lm_step_cand);
    rss_delta = rss_current - sum((lm_step_cand$y - mean(lm_step_cand$y) -
                                     lm_step_cand$X_sel[, 1:(lm_step_cand$j-1)] %*% beta_cand)^2);
    list(lm_step_cand, rss_delta, beta_cand);
  })

  candidates[[which.max(sapply(candidates, function(x) x[[2]]))]][[1]]
}

stepwise.fit <- function(X, y, k=NULL){
  #' Fits OLS stepwise.
  #' @param X: numeric[n, p]. A matrix of features, mean centered
  #' @param y: numeric[n]. A vector of OLS targets
  #' @param k: integer. The optional number of steps to take. If NULL, takes p steps
  #' @return : lm_step. A fitted lm_step object.

  k = ifelse(is.null(k), ncol(X), k)
  lm_step = lm_step_construct(unname(as.matrix(X)), y)
  init_beta = which.min(sapply(1:ncol(lm_step$X), function(ll)
    sum((lm_step$y - mean(lm_step$y) -
           as.numeric((t(lm_step$X[, ll]) %*%
                         lm_step$y))/as.numeric(t(lm_step$X[, ll]) %*%
                                                  lm_step$X[, ll]) * lm_step$X[, ll])^2)
  ))
  lm_step = advance_qr(lm_step, init_beta)
  beta_current = get_beta(lm_step)
  lm_step = log_intermediates(lm_step, beta_current)
  for (iter in 2:k){
    lm_step = advance_selection(lm_step)
    beta_current = get_beta(lm_step)
    lm_step = log_intermediates(lm_step, beta_current)
  }
  lm_step
```

```
}
```

Part (e) Prediction on new Matrix

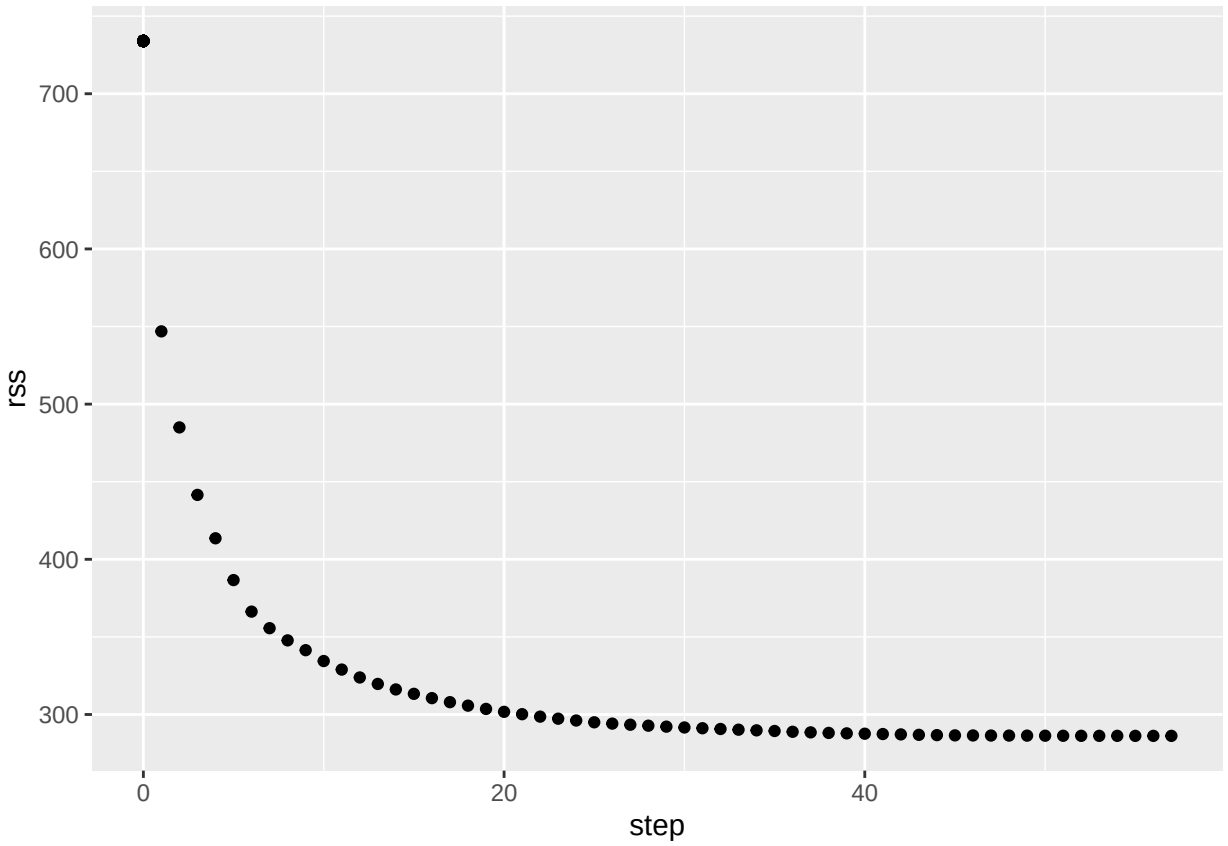
```
stepwise.predict <- function(X, lm_step, k=NULL){
  #' Predicts from a fitted lm_step.
  #' @param X: numeric[n1, p]. A matrix of features to predict.
  #'        Note columns must be ordered identical to original input.
  #' @param lm_step: lm_step. A fitted lm_step object
  #' @param k: integer. Number of features to retroactivelly predict.
  #'        If null, uses entirety of fit.
  #' @return : numeric[n1]. A vector of predictions.
  k = ifelse(is.null(k), lm_step$j-1, k)
  beta_hat = get_beta(lm_step, k=k)
  if(length(beta_hat) - 1){
    return(as.numeric(mean(lm_step$y) +
                        unname(as.matrix(X))[, lm_step$x_sel_idx[1:k]] %*% beta_hat))
  }else{
    return(as.numeric(mean(lm_step$y) +
                        unname(as.matrix(X))[, lm_step$x_sel_idx[1:k]] * beta_hat))
  }
}

allsteps.predict <- function(model, X_eval) {
  #' Predicts target based on each step of the forward stepwise model
  #' @param model: lm_step. An lm_step containing the model parameters
  #' @param X_eval: numeric[n1, p]. A matrix of features to predict.
  #' @return : Matrix of predictions based on each coefficient set

  as.matrix(lapply(1:length(model$x_sel_idx), function(z){
      yhat = stepwise.predict(X_eval, model, k=z);
  }))
}

coef.matrix <- function(model, k) {
  #' Returns formatted coefficient matrix for fitted model, with step
  #' and R^2 (for k steps)
  #' @param model: lm_step. An lm_step containing the model parameters
  #' @param k: integer number of steps to view
  #' @return : data.frame[k, k+2] of coefficient sets for model

  betas_df <- data.frame(model$beta_step)
  betas_df <- betas_df[1:k, colSums(betas_df, dims = 1L) != 0]
  betas_df$step <- 1:k
  betas_df$r2 <- overall_results$r2_step
  betas_df
}

prediction.score <- function(model, X_eval, y_eval) {
    #' Outputs the rss, mse, and misclassification error of a prediction
    #' @param model: lm_step. An lm_step containing the model parameters
    #' @param X_eval: numeric[n1, p]. A matrix of features to predict.
    #' @param y_eval: numeric[n1]. The target to evaluate against
```

```
    #' @return : data.frame. The RSS, MSE, and Misclassification error for each partial model
    scoring = lapply(1:length(model$x_sel_idx), function(z){
    yhat = stepwise.predict(X_eval, model, k=z);
    yhat_binary = classify(yhat)

    rss = sum((y_eval - yhat)^2);
    mse = mean((y_eval - yhat)^2);
    misclass = mean(y_eval != yhat_binary);
    c(rss, mse, misclass)
  }) %>%
    do.call("rbind", .) %>%
    data.frame() %>%
    `colnames<-`(c("RSS", "MSE", "MISCLASS"))
  scoring$nfeat = 1:length(model$x_sel_idx)
  scoring
}
```

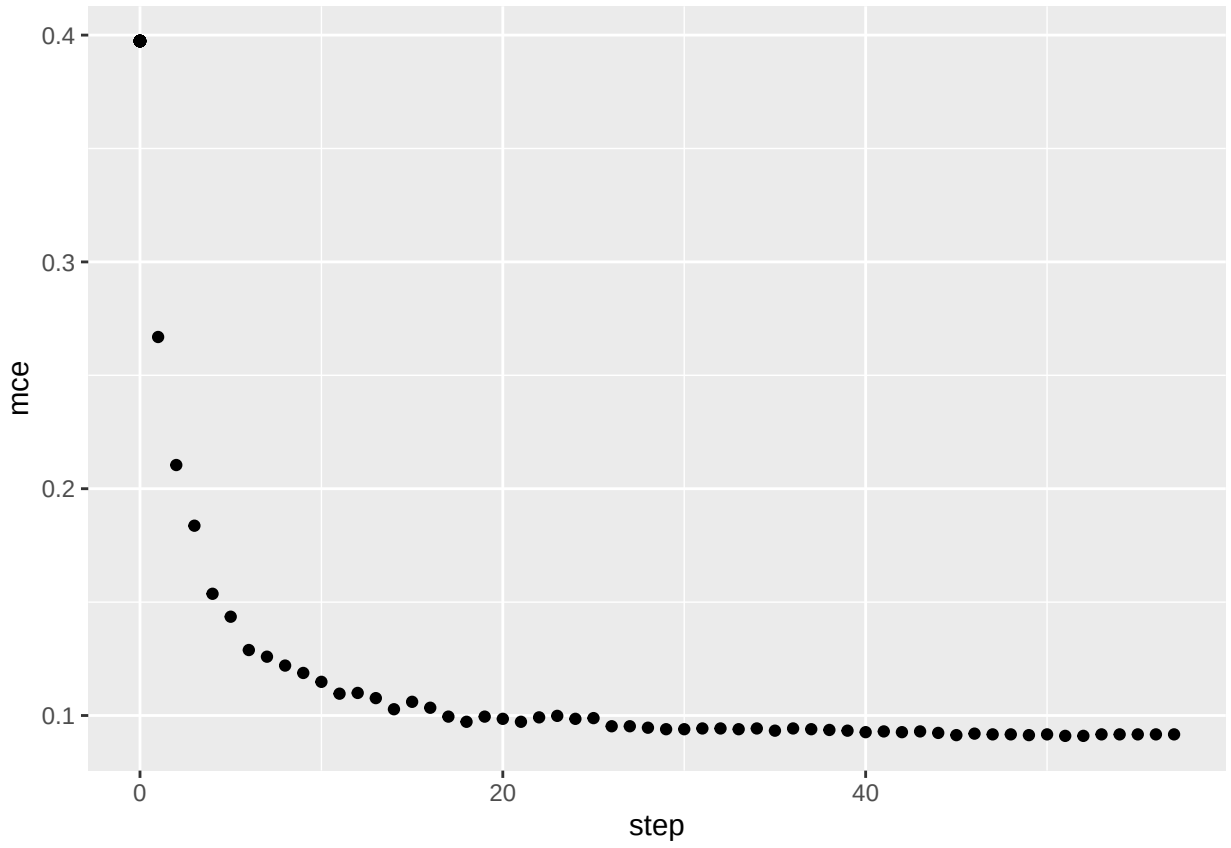
Part (f) Application to Spam Data

```
spam_data = read.csv("spamdata_indicated.csv", header = FALSE)
spam_data[, 55:57] = log(spam_data[, 55:57])

train_x_unscaled = spam_data[spam_data[, 59] == 0, 1:57];
train_y = spam_data[spam_data[, 59] == 0, 58];
test_x_unscaled = spam_data[spam_data[, 59] == 1, 1:57];
test_y = spam_data[spam_data[, 59] == 1, 58];

# centering
centers = train_x_unscaled %>%
  colMeans() %>%
  as.numeric()

train_x_scaled = lapply(1:nrow(train_x_unscaled), function(i)
  as.numeric(train_x_unscaled[i, 1:57]) - centers
) %>%
  do.call("rbind", .)

test_x_scaled = lapply(1:nrow(test_x_unscaled), function(i)
  as.numeric(test_x_unscaled[i, 1:57]) - centers
) %>%
  do.call("rbind", .)

# fit the full model
k = ncol(train_x_scaled)
full_model = stepwise.fit(train_x_scaled, train_y, k = k)
rss_df <- data.frame(step = 1:k, rss = full_model$rss_step) # rss
mce_df <- data.frame(step = 1:k, mce = full_model$misclass_step) # misclassification
```

```
# plot rss during the selection process
ggplot(rss_df, aes(x = step, y = rss)) +
  geom_point() +
  geom_point(aes(x = 0, y = sum((train_y - mean(train_y))^2 )))
```




```
# plot misclassification during the selection process
ggplot(mce_df, aes(x = step, y = mce)) +
  geom_point() +
  geom_point(aes(x = 0, y = mean(train_y != round(mean(train_y)) )))
```



Part (g) Cross Validation

```
stepwise.cv.fit <- function(X, y, nfolds=10, k=NULL, S=2020){
  #' Runs cross-validation on forward stepwise linear regression
  #' @param X: numeric[n1, p]. A matrix of features to predict.
  #'        Note columns must be ordered identical to original input.
  #' @param y: numeric[n1]. The target
  #' @param nfolds: integer. The number of cross-validation folds
  #' @param k: integer. Number of features to retroactivelly predict.
  #'        If null, uses entirety of fit.
  #' @param S: integer. The seed for CV sampling
  #' @return : numeric[n1]. A vector of predictions.
  set.seed(S)

  # cv preprocessing step
  fold_idx = sample(1:nrow(X), nrow(X), replace=F)
  folds = split(fold_idx, ceiling(seq_along(fold_idx) / ceiling(length(fold_idx)/nfolds)))
  cv_result = lapply(folds, function(fold){
    cat('-')
    X_train = X[setdiff(fold_idx, fold), ]; y_train = y[setdiff(fold_idx, fold)];
    X_dev = X[fold, ]; y_dev = y[fold];
    # center data
    centers = X_train %>% colMeans()
    # apply centers to training
    X_train = lapply(1:nrow(X_train), function(i)
      as.numeric(X_train[i, ]) - centers
```

```
    ) %>%
      do.call("rbind", .)
    X_dev = lapply(1:nrow(X_dev), function(i)
      as.numeric(X_dev[i, ]) - centers
    ) %>%
      do.call("rbind", .)

    model = stepwise.fit(X_train, y_train, k=k)
    scoring = prediction.score(model, X_dev, y_dev)
    scoring
  })

  cv_agg = do.call("rbind", cv_result) %>%
    group_by(nfeat) %>%
    summarise_all(., mean)

}
```

```
# note that CV automatically centers, so use unscaled here
cv_results = stepwise.cv.fit(train_x_unscaled, train_y, nfolds=10, S=2020)
```

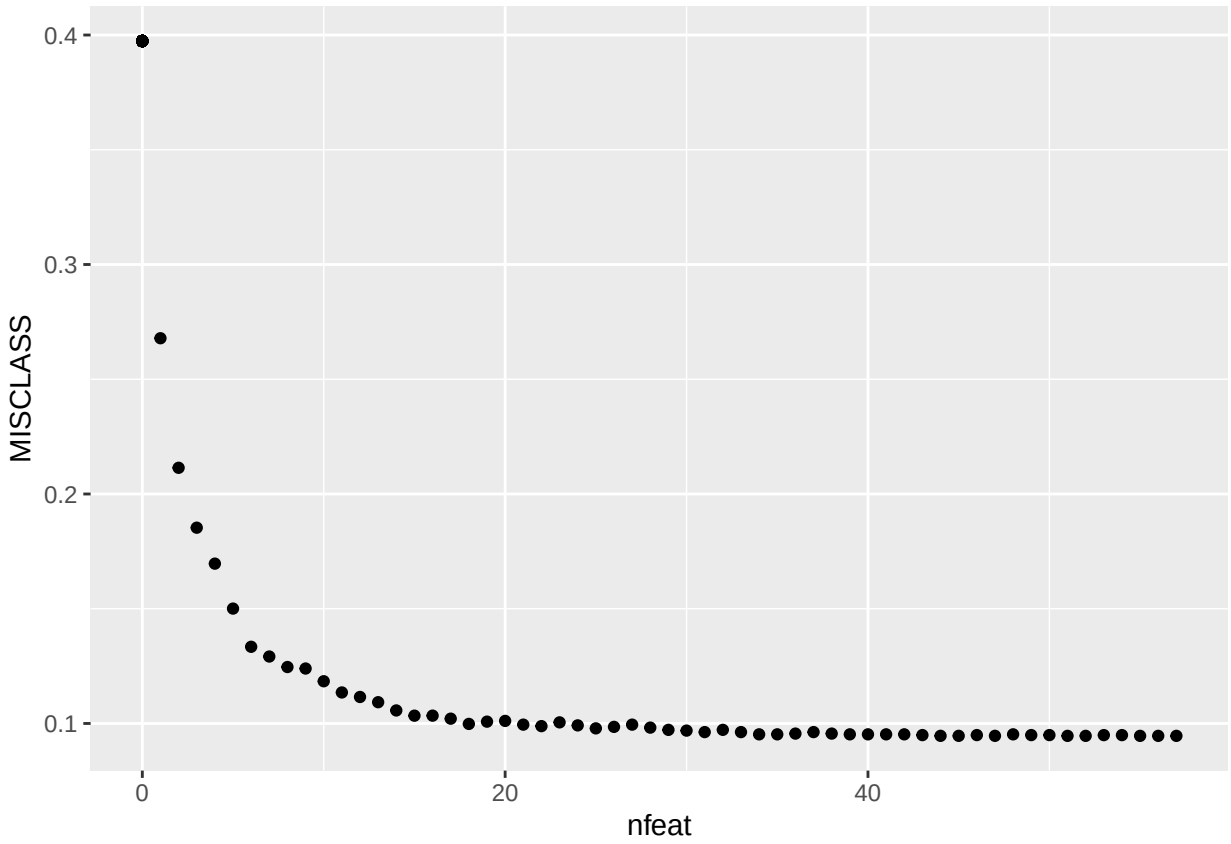
```
## ----------
```

```
# Minimum misclassification error achieved at 51 featues
k_cv= which.min(cv_results$MISCLASS)
k_cv
```

```
## [1] 51
```

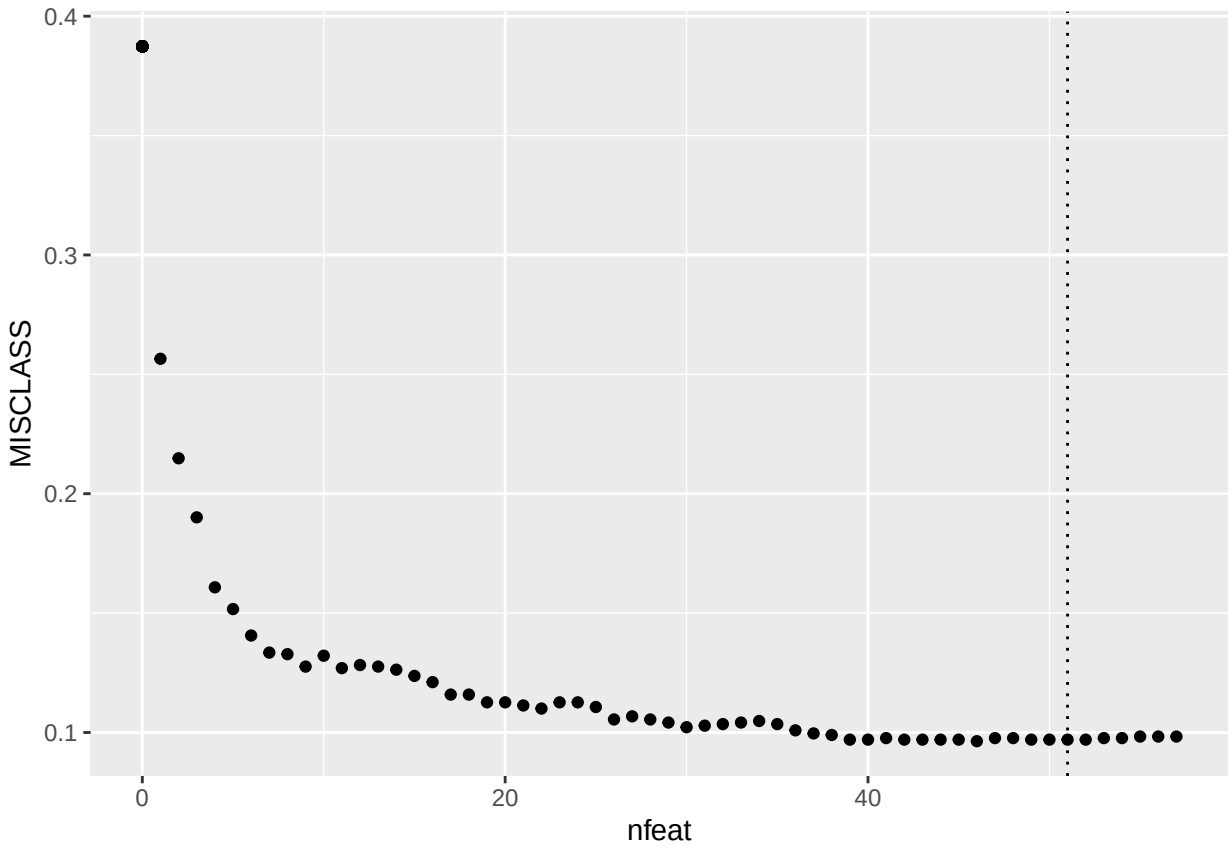
We took the number of steps that produced the minimum misclassification error from cross-validation, but the misclassification appears to level off much earlier. In a practical setting, we would consider using some kind of threshold or the one-standard-error rule to obtain the simplest model within an appropriate range. However, this does not seem to be a problem for the test data in part (h) below.

```
# plot misclassification error as a function of step in CV.
ggplot(cv_results, aes(x = nfeat, y = MISCLASS)) +
  geom_point() +
  geom_point(aes(x = 0, y = mean(train_y != round(mean(train_y)))))
```



Part (h) Predictions on Test Data

```
# predict results on test
full_model = stepwise.fit(train_x_scaled, train_y)
test_results <- prediction.score(full_model, test_x_scaled, test_y)
ggplot(test_results, aes(x = nfeat, y = MISCLASS)) +
  geom_point() +
  geom_vline(xintercept=k_cv, linetype="dotted")  +
  geom_point(aes(x =0, y = mean(test_y != round(mean(test_y))))) # add step 0
```

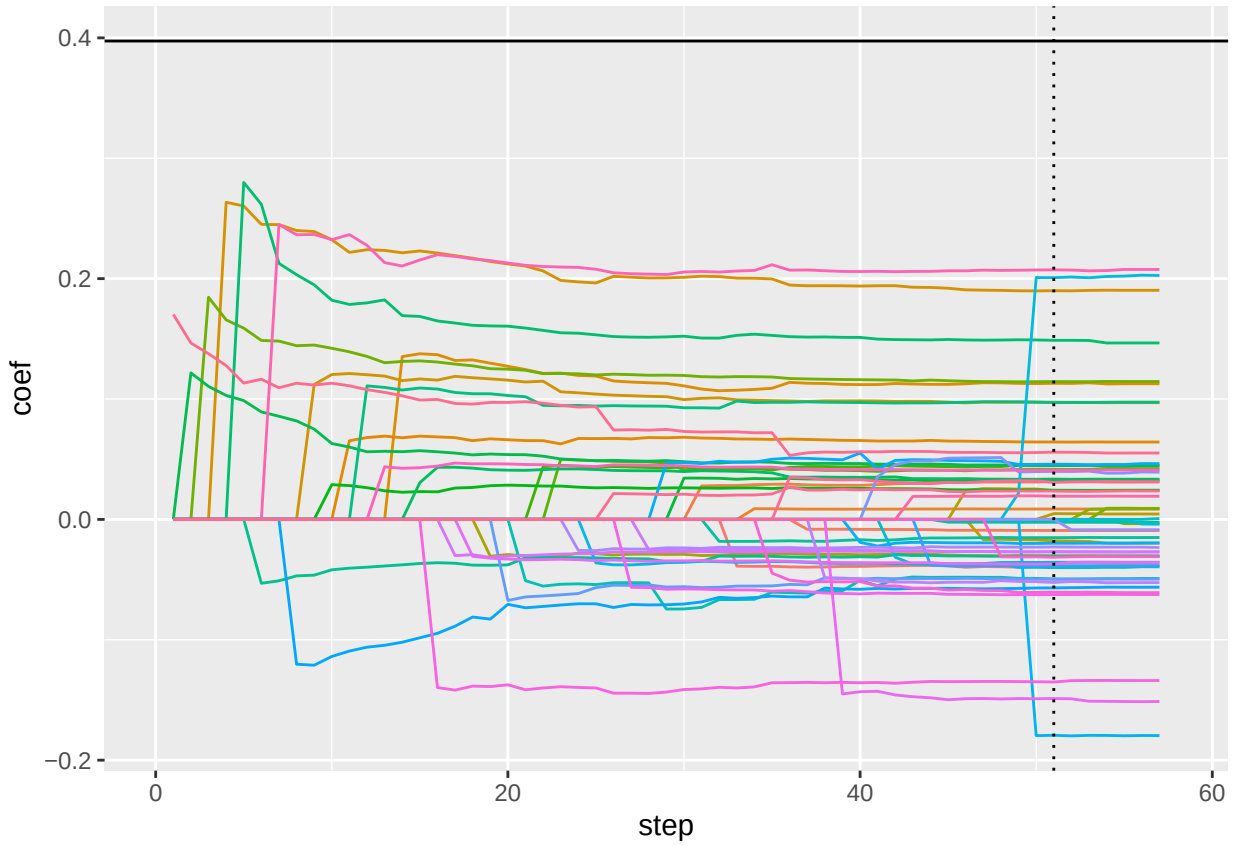


The vertical line corresponds the number of steps selected by CV.

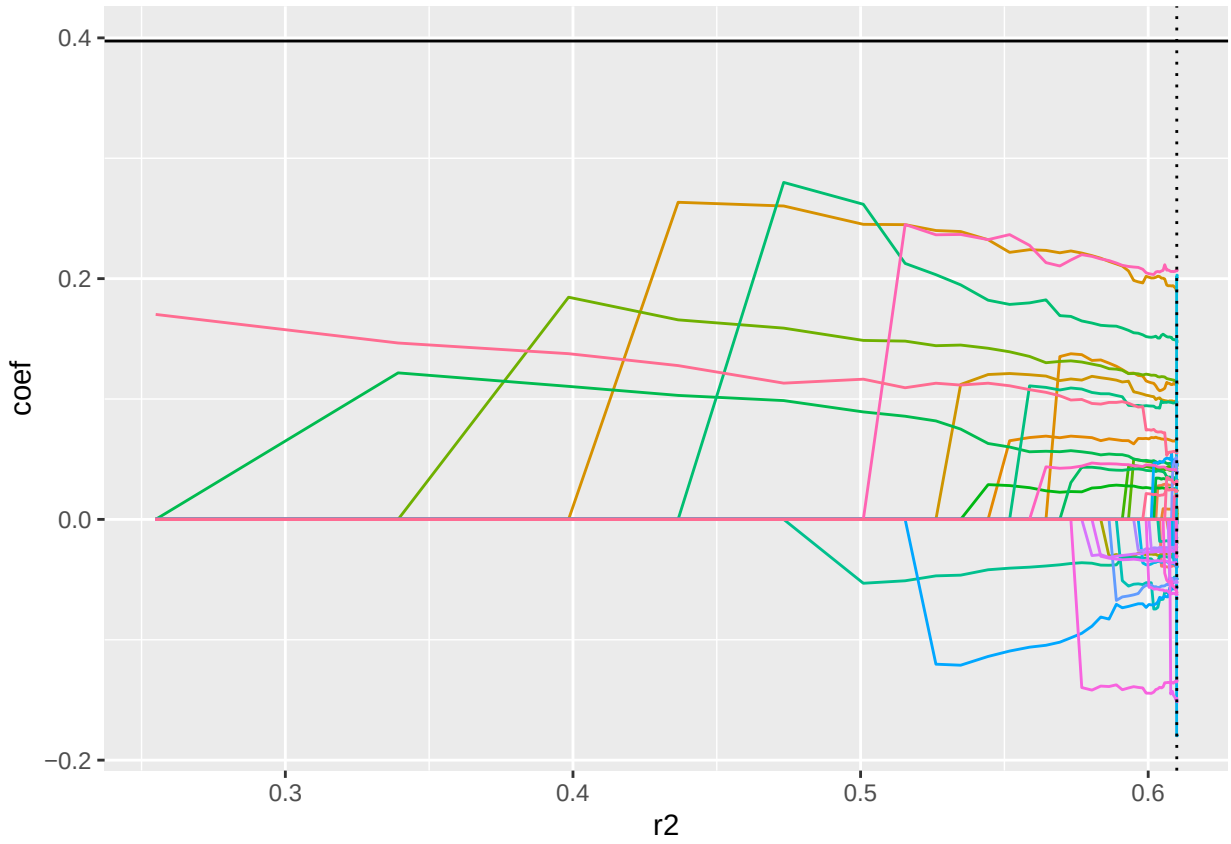
Part (i) Plots of fit per step

```
# plot betas as a function of step and R^2 on all the training data
k = ncol(train_x_scaled)
overall_results = stepwise.fit(train_x_scaled, train_y, k=k)
betas_df <- coef.matrix(overall_results, k)
betas_dfStep <- melt(subset(betas_df, select=-c(r2)) ,
                     id.vars = "step",
                     variable.name = "beta",
                     value.name = "coef")
betas_dfR2 <- melt(subset(betas_df, select=-c(step)) ,
                   id.vars = "r2",
                   variable.name = "beta",
                   value.name = "coef")
```

```
# plot coefficient path as a function of step and R^2
ggplot(betas_dfStep, aes(x = step, y = coef, group_by(beta))) +
  geom_path(aes(colour = beta), show.legend=FALSE) +
  xlim(c(0,k+1)) +
  geom_vline(xintercept=k_cv, linetype="dotted") +
  geom_hline(yintercept=mean(train_y)) # add beta_0
```




```
ggplot(betas_dfR2, aes(x = r2, y = coef, group_by(beta))) +
  geom_path(aes(colour = beta), show.legend=FALSE) +
  geom_vline(xintercept=betas_df$r2[k_cv], linetype="dotted") +
  geom_hline(yintercept=mean(train_y)) # add beta_0
```




```
# first 10 features selected
first_ten.feats <- colnames(spam_data)[overall_results$x_sel_idx[1:10]]
first_ten.coefs <- sapply(1:10, function(i) {
  overall_results$beta_step[i,overall_results$x_sel_idx[i]]
})
# show table for clarity
data.frame(feature = first_ten.feats, coef1 = first_ten.coefs)
```

```
##    feature       coef1
## 1      V56  0.17026498
## 2      V21  0.12171339
## 3      V16  0.18454562
## 4       V7  0.26341833
## 5      V23  0.27993126
## 6      V25 -0.05306591
## 7      V53  0.24494795
## 8      V37 -0.12030062
## 9       V8  0.11209760
## 10     V19  0.02885550
```

We observe that the coefficients do not change sign after entering the model, and that the magnitude of each coefficient tends to decrease as more features are added. We also notice that entering the model at a later step does not preclude the variables from having large coefficient magnitudes - this may just be due to the scale of the particular variable.

Problem 7

Preprocessing

```
# Read in and preprocess data
zz_train = gzfile('zip.train.gz', 'rt')
zz_test  = gzfile('zip.test.gz', 'rt')
num_features = 256
raw_train = as_tibble(read.csv(zz_train, header=F))
raw_test = as_tibble(read.csv(zz_test,header=F))
raw_train %>% separate(col = 1, sep = ' ',
                       into=as.character(c("digit", 1:num_features+1)),
                       convert = TRUE) -> dataset.train
```

```
## Warning: Expected 257 pieces. Additional pieces discarded in 7291 rows [1, 2, 3,
## 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].
```

```
raw_test %>% separate(col = 1, sep = ' ',
                      into = as.character(c("digit", 1:num_features+1)),
                      convert = TRUE) -> dataset.test
```

Part (i)

```
# Compare results of linear regression, LDA, and
# multiclass linear logistic regression
# helper function to obtain class prediction from regression outputs
lr_to_pred <- function(y) {round(min(max(y,0), 9))}
# helper function to obtain error rate based on predicted vs. actual
err_rate <- function(pred.y, test.y) {mean(pred.y != test.y)}

# Linear Regression
model.lr <- lm(digit ~ ., data = dataset.train)
output.lr <- lapply(predict(model.lr, newdata = dataset.test[-1]), lr_to_pred)
err_rate.lr <- err_rate(output.lr, dataset.test$digit)
err_rate.lr # 74.5% test error
```

```
## [1] 0.7453911
```

```
# LDA
model.lda <- lda(digit ~ ., data = dataset.train)
output.lda <- predict(model.lda, newdata = dataset.test)$class
err_rate.lda <- err_rate(output.lda, dataset.test$digit)
err_rate.lda # 11.5% test error
```

```
## [1] 0.1145989
```

```
# Multiclass Logistic Regression
model.mlr <- multinom(digit ~ ., data = dataset.train, MaxNWts = 4000, maxit=500)
```

```
## # weights:  2580 (2313 variable)
## initial  value 16788.147913
## iter  10 value 2598.959017
## iter  20 value 1494.978090
## iter  30 value 903.291402
## iter  40 value 443.785686
## iter  50 value 260.626756
## iter  60 value 190.835491
```

```
## iter  70 value 160.773160
## iter  80 value 114.048146
## iter  90 value 88.746976
## iter 100 value 76.302570
## iter 110 value 63.400188
## iter 120 value 54.375215
## iter 130 value 46.291174
## iter 140 value 38.303473
## iter 150 value 28.822812
## iter 160 value 17.888650
## iter 170 value 9.531259
## iter 180 value 2.985635
## iter 190 value 0.715017
## iter 200 value 0.209663
## iter 210 value 0.066709
## iter 220 value 0.030412
## iter 230 value 0.014034
## iter 240 value 0.006701
## iter 250 value 0.004145
## iter 260 value 0.001842
## iter 270 value 0.001125
## iter 280 value 0.000743
## iter 290 value 0.000464
## iter 300 value 0.000307
## iter 310 value 0.000265
## iter 320 value 0.000214
## final  value 0.000083
## converged
```

```
output.mlr <- predict(model.mlr, newdata = dataset.test[-1])
err_rate.mlr <- err_rate(output.mlr, dataset.test$digit)
err_rate.mlr # 16.6% test error
```

```
## [1] 0.1664175
```

Linear Regression fares much worse than LDA and Logistic Regression, likely due to masking effects. LDA is ultimately the best classifier in the absence of shrinkage.

Part (ii)

```
# Helper function for obtaining predicted class from logistic regression
f.extract_mlm <- function(x) { which.max(x) - 1}

glm_dev_vs_err <- function(train.x, train.y, test.x, test.y,
                           alpha, family, f.extract) {
  #' Get data frame of % deviance vs. test error from glmnet model
  #' @param train.x the training inputs as a matrix
  #' @param train.y the training outputs (list or matrix)
  #' @param test.x the test inputs as a matrix
  #' @param test.y the test outputs (list or matrix)
  #' @param alpha hyperparameter for L1 vs. L2 penalty
  #' @param family the (GLM) model family to fit
  #' @param f.extract a function to extract predictions from the model
  #' @return a data frame of the % deviance vs. test error for each lambda
  train_coded.y <- model.matrix(~ as.factor(train.y) + 0)
```

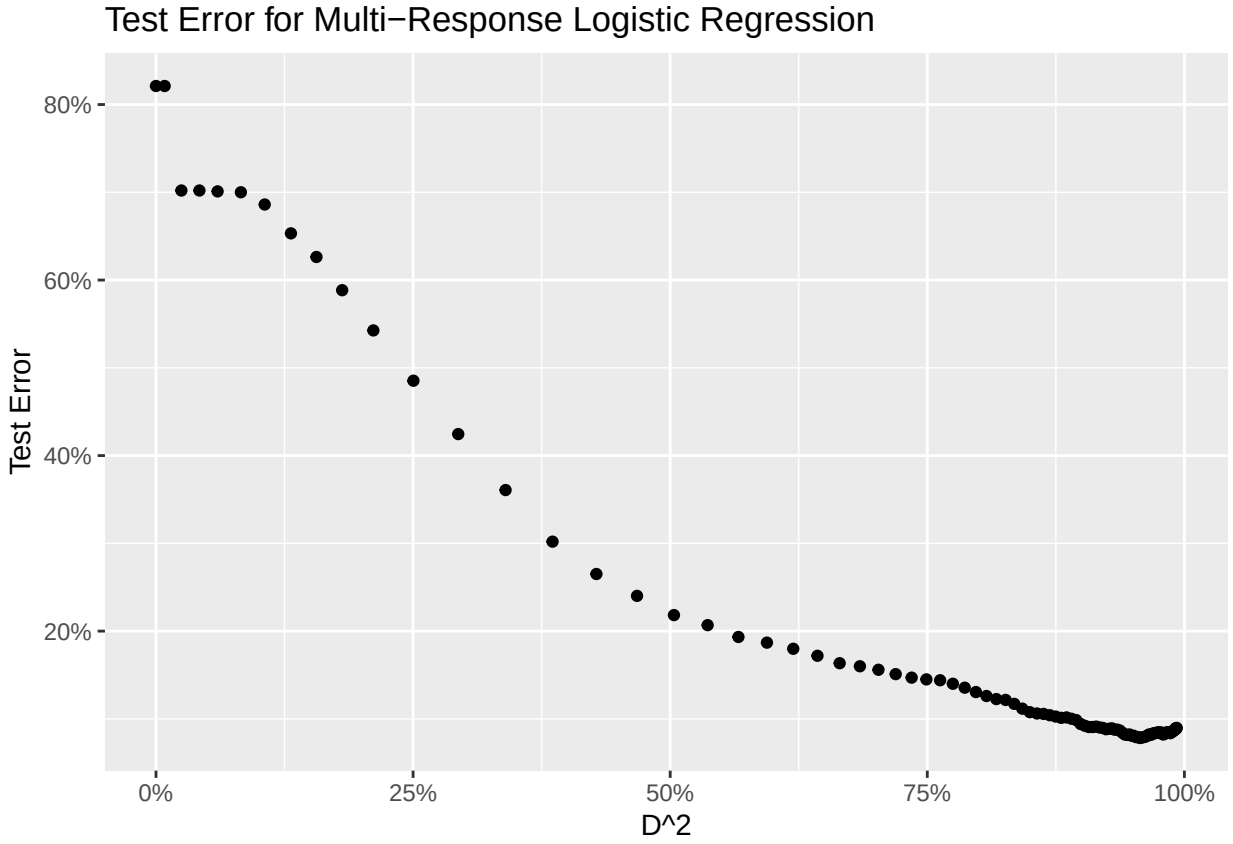
```
  fit <- glmnet(train.x, train_coded.y, alpha = alpha, family = family)

  # apply model to the test data
  result <- lapply(fit$lambda,
                   function(s) predict(fit, type = "response",
                                       newx = test.x, s = s))
  # obtain predicted classes from results
  ghats <- lapply(result, function(t) apply(t, MARGIN = 1, FUN = f.extract))

  # obtain error rates for each lambda
  err_rates <- lapply(ghats, err_rate, test.y = test.y)
  data.frame(dev = fit$dev.ratio, test_error = unlist(err_rates))
}
# adapt train and test data to glmnet
train.y <- dataset.train[[1]]
train.x <- as.matrix(dplyr::select(dataset.train, -c("digit")))
test.y <- dataset.test[[1]]
test.x <- as.matrix(dplyr::select(dataset.test, -c("digit")))
```

Plot for Linear Regression

```
# Linear Regression
dev_to_err.glmlin <- glm_dev_vs_err(train.x, train.y, test.x, test.y,
                                    0.3, "mgaussian", f.extract_mlm)
ggplot(dev_to_err.glmlin, aes(x = dev, y = test_error)) +
  geom_point() +
  labs(title="Test Error for Multi-Response Linear Regression", x ="R^2", y = "Test Error") +
  scale_x_continuous(labels = scales::percent) +
  scale_y_continuous(labels = scales::percent)
```

Plot for Multi-class Logistic Regression

```
# Logistic Regression
# Note: should y be a N x k response matrix? I think it's ok as
dev_to_err.glmlog <- glm_dev_vs_err(train.x, train.y, test.x, test.y,
                                    0.3, "multinomial", f.extract_mlm)
ggplot(dev_to_err.glmlog, aes(x = dev, y = test_error)) +
  geom_point() +
  labs(title="Test Error for Multi-Response Logistic Regression", x ="D^2", y = "Test Error") +
  scale_x_continuous(labels = scales::percent) +
  scale_y_continuous(labels = scales::percent)
```



Part (iii)

The optimization problem being solved in Linear Regression is: $\hat{\beta} = \text{argmin}_{(\beta,\beta_0)}[\frac{1}{2N}\sum_{i=1}^{N}||y_i - \beta_0 - x_i^T\beta||_F^2 + \lambda[(1-\alpha)\frac{1}{2}||\beta||_F^2 + \alpha\sum_{j=1}^{p}||\beta_j||_2]]$

In the multiresponse case, β is a matrix composed of coefficient vectors per response, so the operation that corresponds to the L2-norm in the vector case is the Frobenius Norm $||.||_F$, which takes the the sum of squares for each element in the matrix. Meanwhile, the operation that corresponds to the L1-norm is the sum of the magnitudes of the feature vectors (this is L1 in the feature dimension, while the Frobenius norm is L2 in the feature dimension.)

The optimization problem being solved in Multinomial Logistic Regression is: $\hat{\beta} = \text{argmin}_{(\beta,\beta_0)}[-l(\beta) - \lambda[(1-\alpha)\frac{1}{2}||\beta||_F^2 + \alpha\sum_{j=1}^{p}||\beta_j||_2]]$, where $l(\beta)$ is the likelihood function:

$l(\beta) = \frac{1}{N}\sum_{i=1}^{N}(\sum_{l=1}^{K} I(g_i = l)\log\mathbb{P}(G = k|X = x_i)) = \frac{1}{N}\sum_{i=1}^{N}[\sum_{l=1}^{K} I(g_i = l)(\beta_{0k} + \beta_k^T x_i) - \log(\sum_{l=1}^{K}\exp(\beta_{0k} + x_i^T\beta_k))]$