# HW #2 (STAT 542, Fall 2017)

*October 6th, 2017*

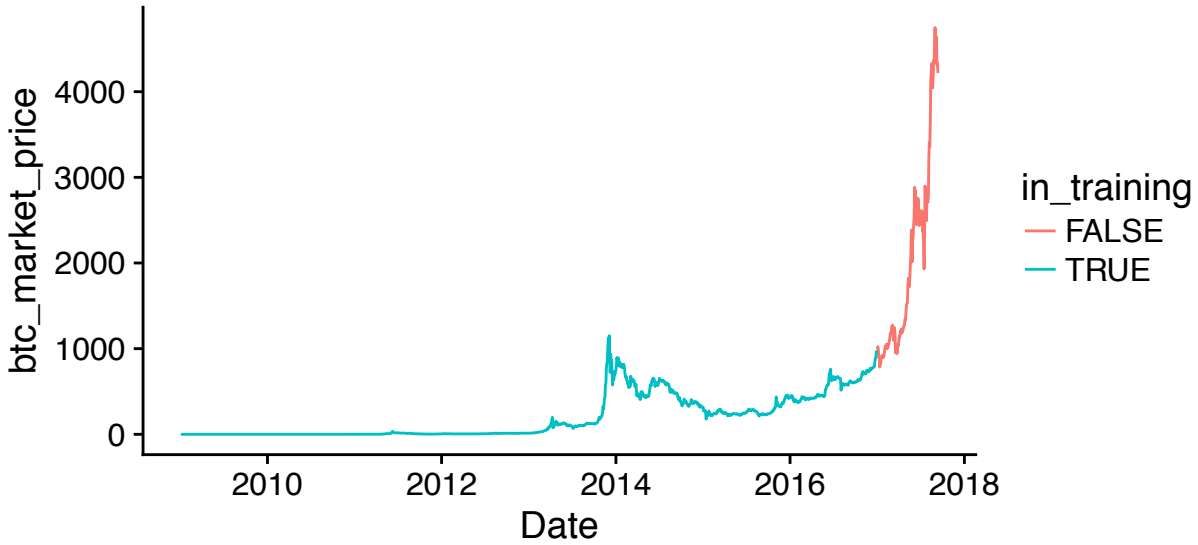
## Data Preparation

Before answering the questions for this homework, the bitcoin dataset needs to be put into a form appropriate for modeling. We start by loading the dataset into R and removing rows and columns that are irrelavant to the analysis:

```
# read the data from the csv on disk
bitcoin_raw <- readr::read_csv('../data/bitcoin_dataset.csv')

# split into train/test sets
#  - train: dates prior to 1/1/2017
#  - test: dates from 1/1/2017 to 9/12/2017
# and discard the remaining.
bitcoin <- bitcoin_raw %>%
  # remove rows we are not using
  filter(Date <= ymd('2017-09-12')) %>%
  # indicator for training and testing data
  mutate(in_training = Date < ymd('2017-1-1')) %>%
  # finally remove the 'btc_trade_volume' from the dataset
  select(-btc_trade_volume)
```

As always, we visualize the relationship we plan to model. Although we will not be modeling the data explicitely as a time-series, we are primarly interested in changes in `btc_market_price` over time. The following plot shows the full time-series as well as the difference between the training and testing portions of the time-series.

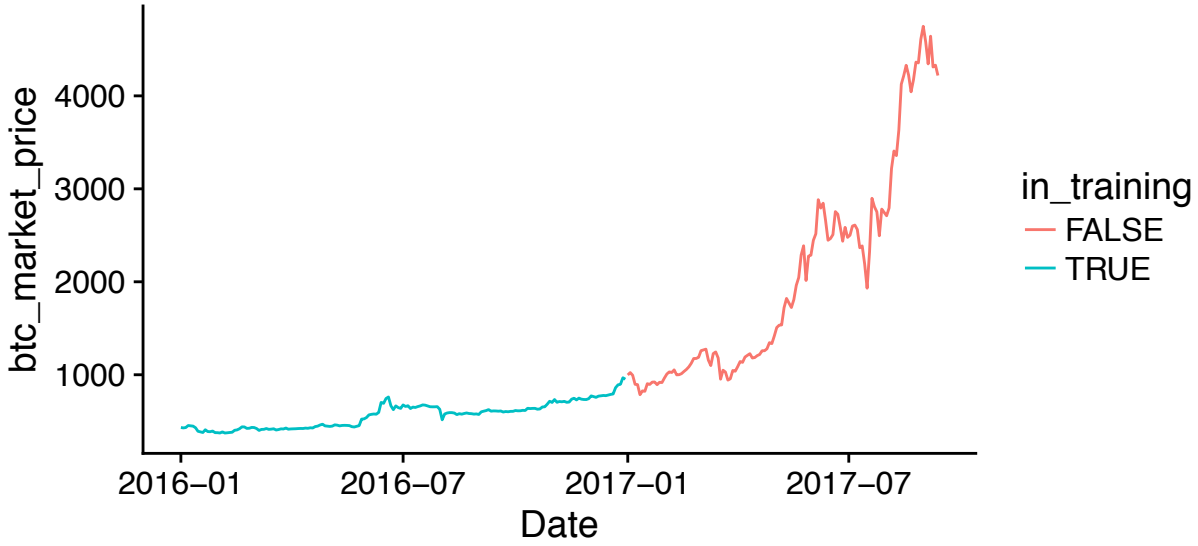
```
# visualize the target variable
ggplot(data = bitcoin, aes(x = Date, y = btc_market_price, color = in_training)) +
  geom_line()
```




In particular, we can see major changes in the behavior of the time-series around the beginning of 2014, 2015, 2016 and again around mid-2017. Due to these major changes in the `btc_market_price` distribution, I'm

going to filter the data down even more to dates after 2016 (we are of course missing yearly cycles in this case, but price is so volatile I doubt any cycle could be extrapolated to the future at this point).

```
bitcoin <- bitcoin %>%
  filter(Date >= ymd('2016-01-01'))

ggplot(data = bitcoin, aes(x = Date, y = btc_market_price, color = in_training)) +
  geom_line()
```




The result is a time-series that looks roughly linear on a logarithmic scale, so we may have a chance at modeling it with a linear model.

Now that we've loaded the data, we can add a few predictors which are occasionally helpful for time-series data. In particular, I added seasonal dummy variables in order to pick up any seasonal cycles. For the cycles it is usually a good idea to have multiple cycles for extrapolation, so I only coded `day_of_month` and `day_of_week` variables (as opposed to months and years). In addition, we will convert the date variable to a numeric (unix time-stamp) in order to pick up on any linear trends in the data. Note we could explicitely integer encode the date variable; however, the use of unix time-stamps allows use to easily convert back to the actual dates if necessary. Plus we'll be standardizing everything anyway. In addition, normally I would encode the seasonal dummies as categorical variables; however, that would require extra care in the lasso code. Since there is a natural ordering for `day_of_week` and `day_of_month` it should be okay to leave them as numeric. The following code performs this engineering in R:

```
bitcoin_dummies <- bitcoin %>%
  # ordinal encoding of day of week (Mon., Tues., etc.)
  mutate(day_of_week = wday(Date)) %>%
  # ordinal encoding of day of month (1st, 15th, etc.)
  mutate(day_of_month = mday(Date)) %>%
  # convert the date variable to a numeric scale (unix time stamps)
  # and re-name it.
  mutate(date = as.numeric(Date)) %>%
  # remove the old date variable
  select(-Date)
```

With all the variables included in the `bitcoin_dummies` dataset we can now split the data-set into training and testing sets for modeling.

```
# now we need to perform a train/test split
training <- filter(bitcoin_dummies, in_training == TRUE)
```

```r
# select the target variable
y_train <- select(training, btc_market_price)

# select all predictors (remove target)
X_train <- select(training, -c(btc_market_price, in_training))

# same thing for test data
testing <- filter(bitcoin_dummies, in_training == FALSE)
y_test <- select(testing, btc_market_price)
X_test <- select(testing, -c(btc_market_price, in_training))

# re-assemble our datasets
bitcoin_train <- as.tibble(cbind(X_train, y_train))
bitcoin_test <- as.tibble(cbind(X_test, y_test))
```

# Question 1

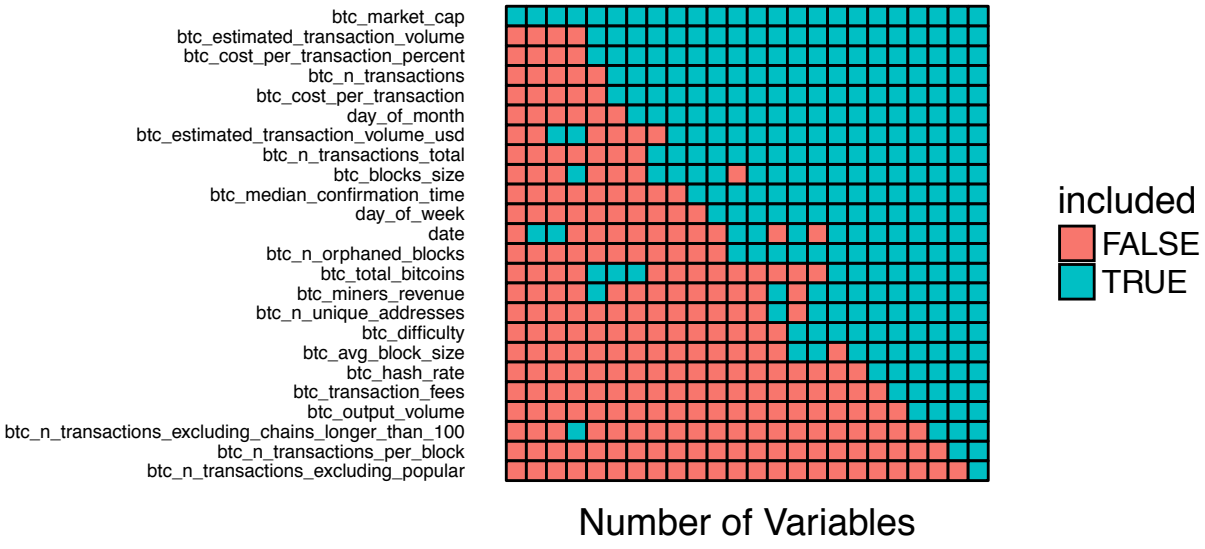
a) Fit the best subset selection to the dataset and report the best model of each size.

We already have the data in a form ready for modeling, so we can go ahead and fit the best-subset selection algorithm.

```r
# fit the best-subset selection algorithm.
# the maximum variables to use is all of the variables in the dataset.
subset_fit <- regsubsets(btc_market_price ~ ., data = bitcoin_train,
                         nvmax = ncol(bitcoin_train) - 1)
```

In order to visualize the best model at each iteration I created a `subset_waffleplot` function in the accompanying `HW2_plots.R` script. The result is a `p x n_subsets` grid indicating what variables were chosen in each iteration. To determine the variables chosen for a given subset you can find the subset column you are interested in and read off the variables colored in blue. In addition, the variables or ordered by the number of times they are chosen for inclusion in a subset.

```r
subset_waffleplot(subset_fit)
```




We can see that `btc_market_cap` and `btc_estimated_transaction_volume` are very important predictors.

This is not surprising since `btc_market_price` = `btc_market_cap / btc_market_volume`.

b) Use $C_p$, AIC and BIC criteria to select the best model and report the results from each. Apply the fitted models to the testing dataset and report the prediction error.

To do this we need a few utility functions. First we need a way to retrieve the variables contained in each subset. This done by the following function:

```
#' Extracts variables contained in a given subset
which_variables <- function(subset_fit, n_variables = 1, intercept.rm = TRUE) {
  sumsubset <- summary(subset_fit)
  selected_variables <- names(which(sumsubset$which[n_variables, ]))

  # remove intercept
  if (intercept.rm) {
    selected_variables <- selected_variables[2:length(selected_variables)]
  }

  selected_variables
}
```

In addition, leaps stores the value for $C_p$ and BIC, but not AIC. Therefore, we need a function to calculate a models AIC:

```
#' AIC calculation from a leaps summary
#'
#' @param sumsubset summary of a regsubsets model
#' @param n_samples number of data samples used to fit the model
#' @returns value of aic for each model size.
subset_aic <- function(sumsubset, n_samples) {
  model_size <- apply(sumsubset$which, 1, sum)
  n_samples * log(sumsubset$rss / n_samples) + 2 * model_size
}
```

With this we can determine the best subset for a given criteria:

```
#' Determine the best subset based on a given criteria
#'
#' The best variable subset can be chosen using a variety of criteria.
#' This utility function determines the best subset based on
#' Mallow's Cp, AIC, or BIC.
#'
#' @param subset_fit a regsubset model
#' @param n_samples number of samples used to choose the subests.
#'    Only nececessary for AIC.
#' @param criteria Criteria used to choose the best subset. Can be one of
#'    AIC, BIC, or Cp. Default is AIC.
#' @returns The best number of variables determined by the given criteria.
best_subset <- function(subset_fit, n_samples = NULL, criteria = 'AIC') {
  sumsubset <- summary(subset_fit)

  # split on the chosen criteria and choose the argmin
  # as the best number of variables.
  if (criteria == 'AIC') {
    if (is.null(n_samples)) {
      stop('n_samples necessary for AIC criteria.')
    }
```

```
    n_variables <- which.min(subset_aic(sumsubset, n_samples))
  }
  else if (criteria == 'Cp') {
    n_variables <- which.min(sumsubset$cp)
  }
  else if (criteria == 'BIC') {
    n_variables <- which.min(sumsubset$bic)
  } else {
      stop('Unrecoginized criteria. Must be one of AIC, BIC, Cp.')
  }

  n_variables
}
```

Finally, we can combine these functions into a method that fits a linear model based on the best subset for a given criteria.

```
#' Fit a linear model based using best subset selection
bestsubset_lm <- function(subset_fit, target_var, data,
                          criteria = 'AIC') {

  # determine the best variables based on the given criteria
  n_variables <- best_subset(subset_fit,
                             n_samples = nrow(data),
                             criteria = criteria)

  # actually extract the predictor names from the subset
  predictors <- which_variables(subset_fit, n_variables = n_variables)

  # create a formula string used to fit the linear model
  target_str <- paste(target_var, "~", sep = " ")
  predictor_str <- paste(predictors, collapse = " + ")
  formula_str <- paste(target_str, predictor_str, sep = " ")
  formula <- as.formula(formula_str)

  # fit the linear model
  lm(formula, data = data)
}
```

Since we need to report the predictors used by each model, I also defined a function that extracts the predictors from a fitted linear regression model. I hid it from the report but you can find it in the accompying markdown.

Putting this all together we can fit 4 models: the model using all of the predictors and the best model according AIC, BIC and Mallow's $C_p$.

```
# model with all variables
model_full <- lm(btc_market_price ~ ., data = bitcoin_train)

# AIC
model_aic <- bestsubset_lm(subset_fit, target_var = 'btc_market_price',
                           data = bitcoin_train,
                           criteria = "AIC")

print_predictors(model_aic)
```

```
## 8 variables used in this model:
##  btc_market_cap
##  btc_blocks_size
##  btc_cost_per_transaction_percent
##  btc_cost_per_transaction
##  btc_n_transactions
##  btc_n_transactions_total
##  btc_estimated_transaction_volume
##  day_of_month
```

```
# BIC
model_bic <- bestsubset_lm(subset_fit, target_var = 'btc_market_price',
                           data = bitcoin_train,
                           criteria = "BIC")

print_predictors(model_bic)
```

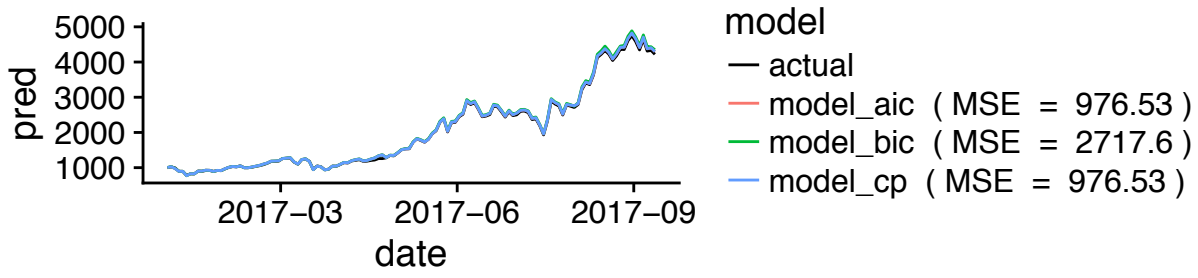
```
## 5 variables used in this model:
##  btc_total_bitcoins
##  btc_market_cap
##  btc_miners_revenue
##  btc_cost_per_transaction_percent
##  btc_estimated_transaction_volume
```

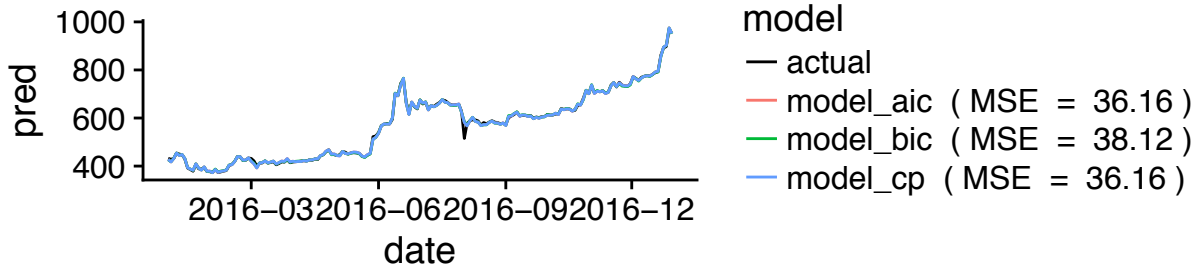
```
# Mallow's Cp
model_cp <- bestsubset_lm(subset_fit, target_var = 'btc_market_price',
                          data = bitcoin_train,
                          criteria = "Cp")

print_predictors(model_cp)
```

```
## 8 variables used in this model:
##  btc_market_cap
##  btc_blocks_size
##  btc_cost_per_transaction_percent
##  btc_cost_per_transaction
##  btc_n_transactions
##  btc_n_transactions_total
##  btc_estimated_transaction_volume
##  day_of_month
```

We can determine the performance by plotting the train/test curves over time. You can find the code used to generate this performance plot in the attached `HW2_plots.R` script. The plot contains the predictions over time fore each model as well as the actual values in the dataset. The legend indicates the overall mean squared error on each data partition. The top plot is the test set and the bottom plot is the training set.

```
train_test_curves(train = bitcoin_train, test = bitcoin_test,
                  target_var = 'btc_market_price', date_var = 'date',
                  metric = "MSE",
                  model_aic, model_bic, model_cp)
```

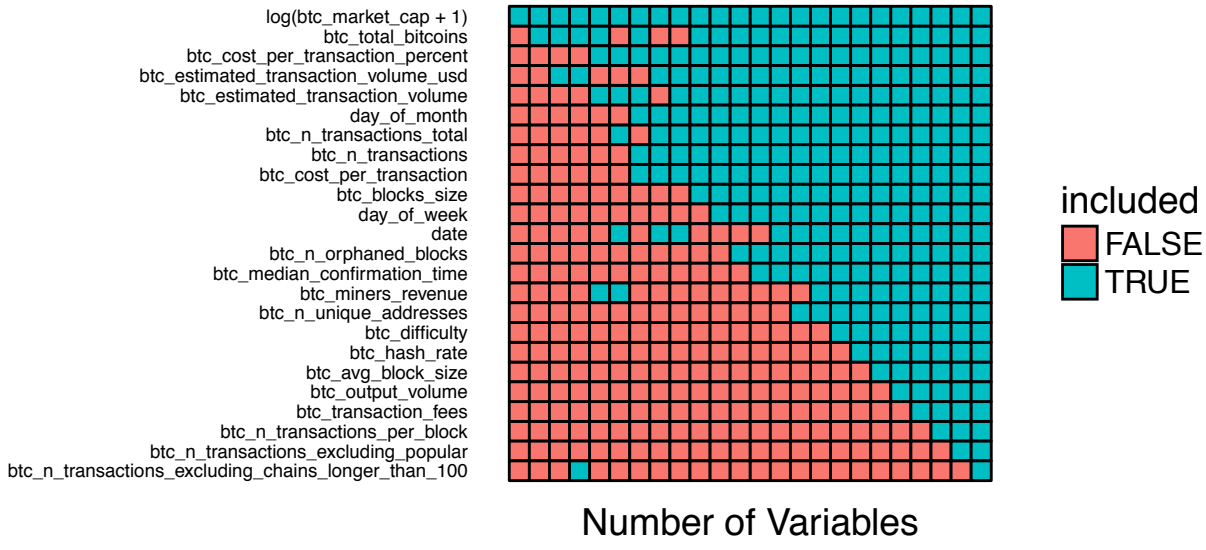



c) Redo a) and b) using log(1 + Y) as the outcome. Report the best models. Then for prediction, transform the predicted values into the original scale and report the prediction error of each model.

We start be re-fitting the best-subset selection algorithm to the log-transformed response. However, I am also going to log-transform `btc_market_cap` since we want to preserve what was already a roughly linear relationship. Again I also visualized the variables selected in each subset.

```
subset_fit <- regsubsets(
    log(btc_market_price + 1) ~ . + log(btc_market_cap + 1) - btc_market_cap,
    data = bitcoin_train,
    nvmax = ncol(bitcoin_train) - 1)

subset_waffleplot(subset_fit)
```




Now we can re-fit the best models according to each selection criteria:

```
# AIC
log_model_aic <- bestsubset_lm(subset_fit, target_var = 'log(btc_market_price + 1)',
                               data = bitcoin_train,
                               criteria = "AIC")
```

```r
print_predictors(log_model_aic)
```

```
## 9 variables used in this model:
##  btc_cost_per_transaction_percent
##  btc_cost_per_transaction
##  btc_n_transactions
##  btc_n_transactions_total
##  btc_estimated_transaction_volume
##  btc_estimated_transaction_volume_usd
##  day_of_month
##  date
##  log(btc_market_cap + 1)
```

```r
# BIC
log_model_bic <- bestsubset_lm(subset_fit, target_var = 'log(btc_market_price + 1)',
                               data = bitcoin_train,
                               criteria = "BIC")

print_predictors(log_model_bic)
```

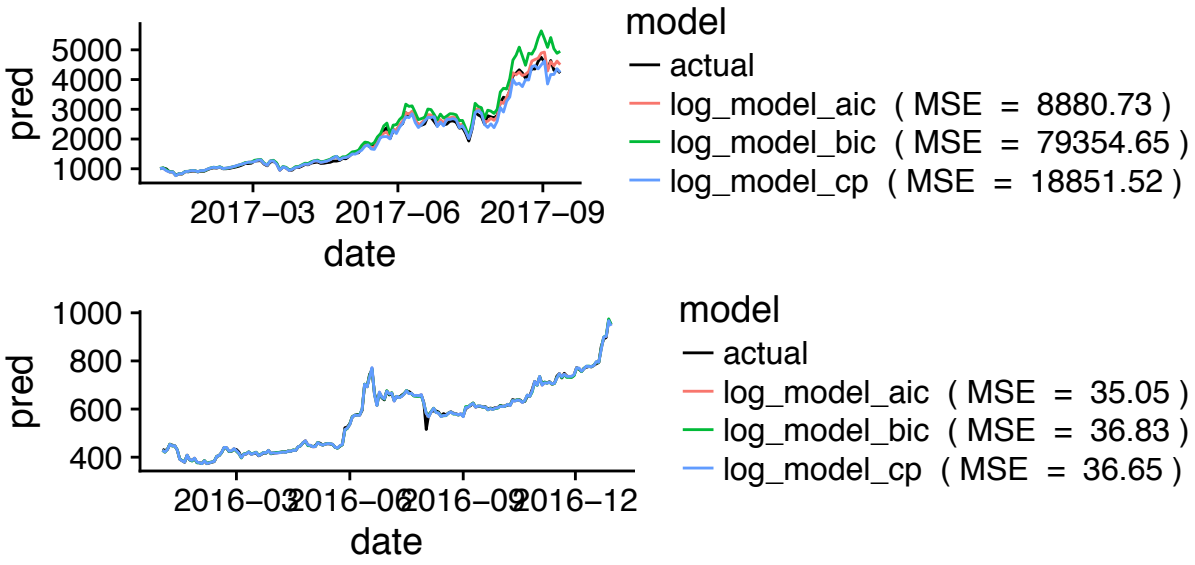
```
## 6 variables used in this model:
##  btc_miners_revenue
##  btc_cost_per_transaction_percent
##  btc_n_transactions_total
##  btc_estimated_transaction_volume
##  date
##  log(btc_market_cap + 1)
```

```r
# Mallow's Cp
log_model_cp <- bestsubset_lm(subset_fit, target_var = 'log(btc_market_price + 1)',
                              data = bitcoin_train,
                              criteria = "Cp")

print_predictors(log_model_cp)
```

```
## 8 variables used in this model:
##  btc_cost_per_transaction_percent
##  btc_cost_per_transaction
##  btc_n_transactions
##  btc_n_transactions_total
##  btc_estimated_transaction_volume_usd
##  day_of_month
##  date
##  log(btc_market_cap + 1)
```

Once again we plot the performance curves.

```r
train_test_curves(train = bitcoin_train, test = bitcoin_test,
                  target_var = 'btc_market_price',
                  y_transform = function(x) { exp(x) - 1 },
                  date_var = 'date', metric = 'MSE',
                  log_model_aic, log_model_bic, log_model_cp)
```







The performance is definitely worse for the log-response compared to the previous fit. The most likely reason is that a few predictors are proxies for the response (we are fitting this series way to well), so the logarithmic transform is destroying their linear relationship. This was the reason why I also log-transformed `btc_market_cap`.

## Question 2

a) Complete the Lasso fitting code.

Going forward we will be solving the following optimization problem for the lasso:

$$f(\beta, \beta_0) = \frac{1}{2n}||\mathbf{y} - \beta_0 - \mathbf{X}\beta||_2^2 + \lambda \sum_{j=1}^{p} |\beta_j|.$$

To start out we need the following soft-thresholding function:

```
soft_threshold <- function(beta, lambda) {
  sign(beta) * max(abs(beta) - lambda, 0)
}
```

We will be solving the lasso problem using the pathwise coordinate-descent algorithm discussed in class. The first function we need is a coordinate descent solver for the lasso problem for a single value of $\lambda$. This function assumes that the design-matrix $\mathbf{X}$ has been centered and scaled and that the response vector $\mathbf{y}$ has been centered. The coordinate-descent algorithm is implemented in the following function:

```
lasso_cd <- function(x, y, lambda, beta = NULL, tol = 1e-10, max_iter = 500,
                     patience = 10) {
  # if the initial beta is null, then start at zero
  if(is.null(beta)) {
    beta = rep(0, ncol(x))
  }

  # value of the loss function (MSE) at each iteration
  loss <- rep(0, max_iter)
```

```
  for (k in 1:max_iter) {
    # the full residual used to check the value of the loss function
    residual <- y - x %*% beta
    loss[k] <- mean(residual * residual)

    for (j in 1:ncol(x)) {
      # partial residual (effect of all the other co-variates)
      residual <- residual + x[, j] * beta[j]

      # single variable OLS estimate
      beta_ols_j <- mean(residual * x[, j])

      # soft-threshold the result
      beta[j] <- soft_threshold(beta_ols_j, lambda)

      # restore the residual
      # (although adding back the effect of the updated beta)
      residual <- residual - x[, j] * beta[j]
    }
    # end-of coefficient loop

    if (k > patience) {
      # I changed the early stopping criterion to compare the mean loss change
      # across a window length of one half the patience (5 by default).
      # The one provided was not stopping early (even when converged),
      # so the algorithm was taking ages...
      if (should_stop(loss, k = k, window = patience, tol = tol)) {
        break
      }
    }
  }
  # end-of iteration loop

  beta
}
```

One thing worth noting is that I changed the stopping criteria for the algorithm. I was finding that the version provided in the example code would never exit early (even after convergence), so the code would take a very long time. I created a stricter version of the stopping criteria, which compares the mean change in the objective over a window and stops if the new window loss is not less then a given tolerance of the previous window. You can find the `should_stop` function defined in the attached `lasso.R`.

Now that we have solved the lasso problem for a single value of $\lambda$, we need an outer iteration that fits the lasso for a sequence of $\lambda$'s, i.e. the lasso path. This is done by the following function

```
lasso <- function(x, y, lambda = NULL, n_lambda = 100, lambda_ratio = 1e-4,
                  max_iter = 500, tol = 1e-10, patience = 10) {
  # now we need to standardize our predictors
  # and scale the response.
  x <- scale(x)
  y <- scale(y, scale = FALSE)

  # if not supplied use a sequence of decreasing lambdas,
  # such that the largest lambda has only one non-zero coefficient.
  if (is.null(lambda)) {
```

```
    max_lambda <- (1 / nrow(x)) * max(abs(t(x) %*% y))
    lambda <- exp(seq(log10(max_lambda), log10(max_lambda * lambda_ratio),
                      length.out = n_lambda))
  }

  # the coefficients for each lambda value
  beta_all <- matrix(NA, nrow = ncol(x), ncol = length(lambda))
  rownames(beta_all) <- colnames(x)

  # the intercept for each lambda value
  beta0_all <- rep(NA, length(lambda))

  # initial beta value for the largest lambda
  bhat <- rep(0, ncol(x))

  for (i in 1:length(lambda)) # loop from the largest lambda value
  {
    # solve the lasso objective using coordinate-descent for a given lambda
    bhat <- lasso_cd(x, y, lambda[i], beta = bhat, tol = tol,
                     max_iter = max_iter, patience = patience)

    # to get the unscaled / uncentered version we need to apply the scale to
    # the coefficients
    # Note: x_new = x_old * center + scale => beta_new = beta_old / center.
    #       From SLR theory.
    x_scale <- attr(x, 'scaled:scale')
    beta_all[, i] <-  bhat / x_scale

    # We can always calculate the intercept from the other coefficients
    # Note: beta0 = ybar - sum(xbar_i * beta_i). From MLR theory.
    y_mean <- attr(y, 'scaled:center')
    x_mean <- attr(x, 'scaled:center')
    beta0_all[i] <- y_mean - sum(x_mean * beta_all[, i])
  }

  # create a lasso object (mostly to make my life easier)
  lasso_init(intercept=beta0_all, coef=beta_all, lambda=lambda)
}
```

Note that I changed the initial $\lambda$ value so that it was the largest value that would result in a non-empty model (before it was zero for a few iterations). To see this we use the fact that the initial betas are zero $\beta = 0 \implies \mathbf{r} = \mathbf{y} - X\beta = \mathbf{y}$ along with the condition that the features are standardized so that $\beta_i^{ols} = \langle \mathbf{x}_i, \mathbf{y} \rangle$. Thus the soft-thresholding function is non-zero if and only if $\frac{1}{N} |\langle \mathbf{x}_i, \mathbf{y} \rangle| > \lambda$. In order for this constraint to hold only for one coefficient, we can set $\lambda_0 = \frac{1}{N} \max_i |\langle \mathbf{x}_i, \mathbf{y} \rangle|$, which is what is done in the code above.

Furthermore, I created a `lasso` object at the end of the code. This is mostly to make my life easier, since it groups the coefficients and intercepts together and a we can use the normal `plot` and `predict` functions. You can find the definition at the end of `lasso.R`.

To demonstrate that the lasso code is correct, we generate a synthetic dataset and compare it with the model obtained using `glmnet`.

```
# include the script for the lasso
set.seed(1)
N = 400
```

```
P = 20

Beta = c(1:5/5, rep(0, P-5))
Beta0 = 0.5

# genrate X
V = matrix(0.5, P, P)
diag(V) = 1

X = as.matrix(mvrnorm(N, mu = 3*runif(P)-1, Sigma = V))

# create artifical scale of X
X = sweep(X, 2, 1:10/5, "*")

# genrate Y
y = Beta0 + X %*% Beta + rnorm(N)

# fit glmnet
glmnet_fit <- glmnet(X, y)

# use the same lasso path as glmnet
my_fit <- lasso(X, y, lambda = glmnet_fit$lambda)

# these should be small if the code is correct
print(max(abs(my_fit$coef - glmnet_fit$beta)))
```

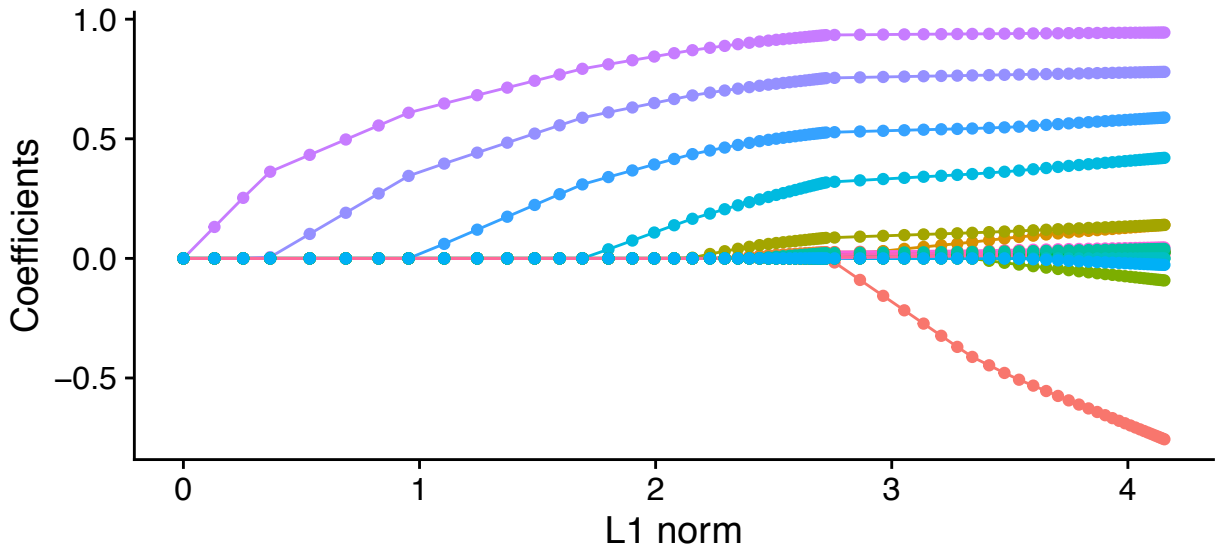
```
## [1] 0.00580583
```

```
print(max(abs(my_fit$intercept - glmnet_fit$a0)))
```

```
## [1] 0.002722373
```

In addition, we can take a look at the lasso path, which should be a continous path with coefficients growing away from zero.

```
plot(my_fit)
```




b) Fit your Lasso model to the `bitcoin` dataset. Report the test errors for each lambda as well as choose

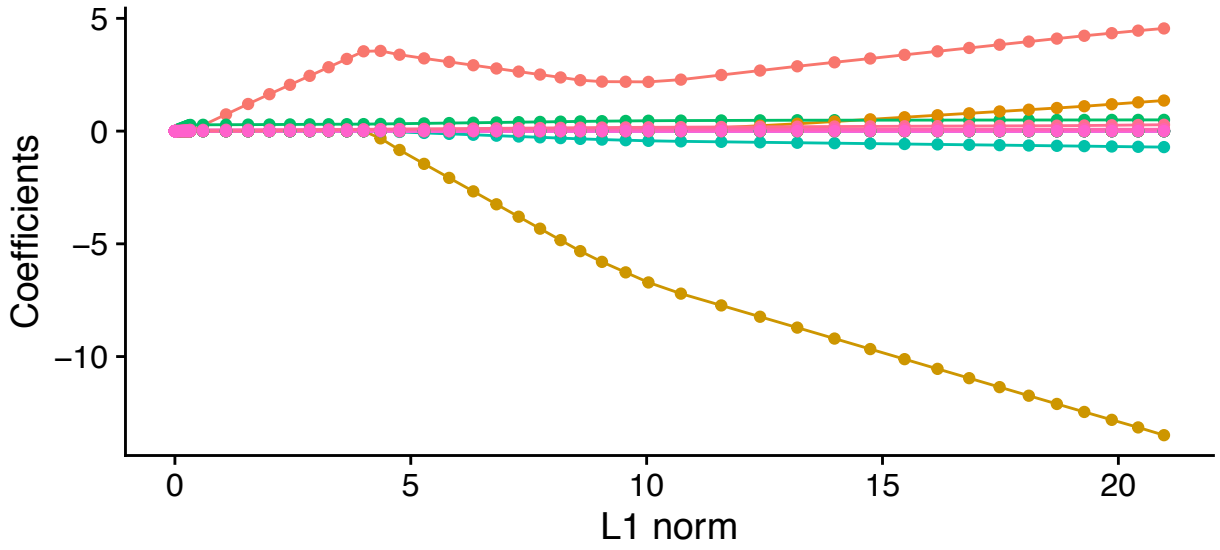
the best model based on the lasso path.

Since the `lasso` function expects matrix input, we start by converting the dataframes to matrix objects:

```
# training matrices
y_train <- as.matrix(dplyr::select(bitcoin_train, btc_market_price))
X_train <- as.matrix(dplyr::select(bitcoin_train, -btc_market_price))

# testing matrices
y_test <- as.matrix(dplyr::select(bitcoin_test, btc_market_price))
X_test <- as.matrix(dplyr::select(bitcoin_test, -btc_market_price))
```

The `lasso` code takes care of standardizing and scaling our predictors, so we are ready for modeling. Fitting the model and visualizing the lasso path:

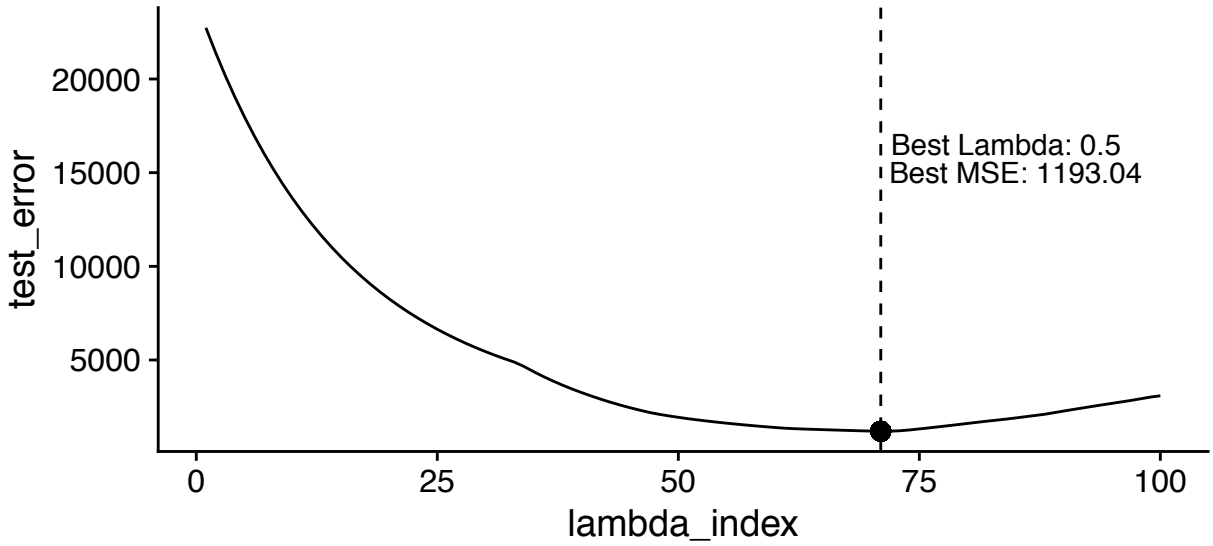
```
lasso_model <- lasso(X_train, y_train)

plot(lasso_model)
```




Now that we've fit the lasso path, we need to determine the best lambda value based on the testing error.

```
# This is a matrix of shape [n_samples, n_lambda]
# where each column is the predictions for a particular lambda
preds <- predict(lasso_model, X_test)

# returns the mse error on the test set
test_error <- function(y_pred) {
  mean((y_test - y_pred)^2)
}

# apply the test_error function across the columns of preds
# (i.e. preds for given lambda)
errors <- apply(preds, 2, test_error)

# the best lambda index has the lowest test error
best.index <- which.min(errors)
best.lambda <- lasso_model$lambda[best.index]
min.error <- min(errors)
```

```r
# Visualization of the testing error as a function of the lambda index
# Note higher indices indicate lower lambda values.
data <- tibble(
  lambda_index = 1:length(errors),
  test_error = errors,
  lambda = lasso_model$lambda
)

ggplot(data, aes(x = lambda_index, y = test_error)) +
  geom_line() +
  geom_vline(xintercept = best.index, linetype = 'dashed') +
  geom_point(aes(x = best.index, y = min.error), size = 3) +
  annotate("text", x = 85, y = 15000,
           label = paste("Best MSE:", round(min.error, 2))) +
  annotate("text", x = 84, y = 16500,
           label = paste("Best Lambda:", round(best.lambda, 4)))
```




We see that the best $\lambda$ is 0.5 with a testing error of 1193.04.

In addition, we'd like to visualize the prediction curves as well as determine the non-zero coefficients of the best model. We can easily determine the non-zero coefficients and their values:

```r
best_coef <- lasso_model$coef[, best.index]
nonzero_coef <- best_coef[abs(best_coef) > 0]
(coef_sum <- tibble(variable_name = names(nonzero_coef), value = unname(nonzero_coef)))
```

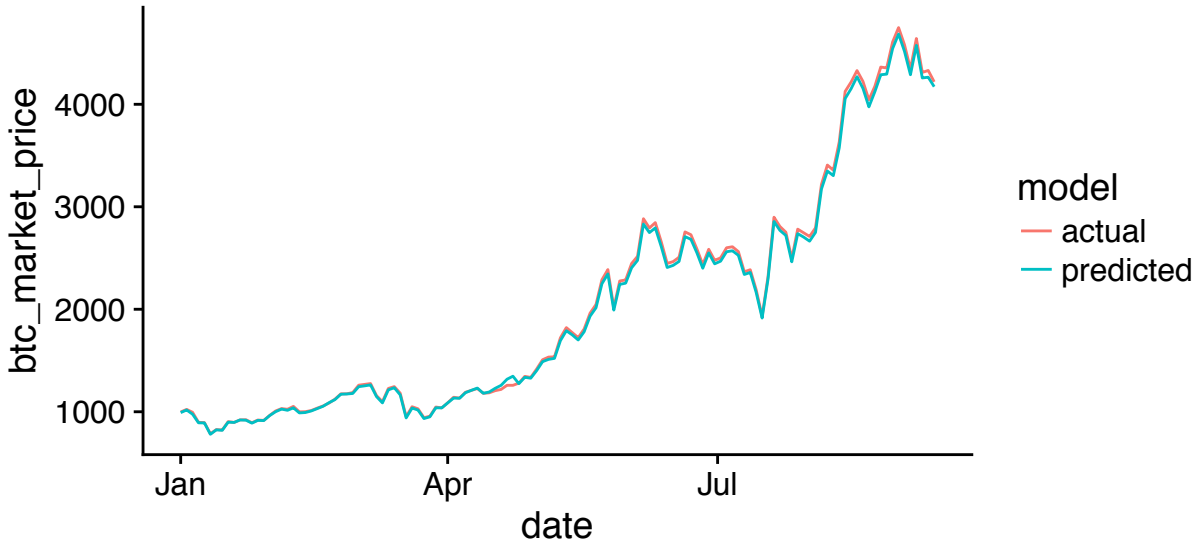
```
## # A tibble: 9 x 2
##                       variable_name         value
##                               <chr>         <dbl>
## 1                   btc_market_cap  5.822657e-08
## 2               btc_avg_block_size  3.538485e+00
## 3     btc_n_transactions_per_block  7.525819e-04
## 4     btc_median_confirmation_time  3.100144e-01
## 5               btc_miners_revenue  6.348366e-06
## 6           btc_n_unique_addresses  1.584942e-07
## 7 btc_estimated_transaction_volume -2.558547e-05
## 8                     day_of_week  7.146605e-02
```

```
## 9                      day_of_month  7.910710e-02
```

We see that we've select 9 out of the 24 variables.

Finally, we can plot the performance curve of the selected lasso model.

```
data <- bitcoin_test %>%
  select(date, btc_market_price) %>%
  rename(actual = btc_market_price) %>%
  mutate(date = unix_to_date(date)) %>%
  mutate(predicted = preds[, best.index]) %>%
  gather(predicted, actual, key = model, value = btc_market_price)

ggplot(data = data, aes(x = date, y = btc_market_price, color = model)) +
  geom_line()
```




We see that the model matches up well with the test data.