Drug Reaction Prediction Based on Genomic Data

BMI 551

Rosemary Ingham
Ognyan Moore

24 March 2017

Setup

First thing we do is import the necessary libraries. We make use of `tidyverse` to manipulate the shape of the data, use the `foreach` library to make writing code for the bagging portion of our analysis a bit simpler, and we use the `randomForest` library to do our random forest model.

```
library(tidyverse)
library(foreach)
library(randomForest)
```

Data Import

Importing the data was fairly straightforward, there were good descriptors provided.

```
subtypes <- read_tsv('subtypes.txt',
                     col_names=TRUE,
                     col_types = list(
                       cellline = col_character(),
                       subtype = col_character())
                     )
subtypes$subtype <- as.factor(subtypes$subtype)

scoring <- read_csv('scoring_and_test_set_id_mappings.csv',
                    col_names = TRUE,
                    col_types = list(
                      cellline = col_character(),
                      drug = col_character(),
                      id = col_integer(),
                      Usage = col_character()))

# discovered drug naming mismatch, making them the same this way
scoring$drug <- gsub("-", ".", scoring$drug)

sample_submission <- read_csv('rand_sub_cont.csv',
                              col_names = TRUE,
                              col_types = list(
                                id = col_integer(),
                                value = col_double()))

expression <- read.table('expression.txt',
                         header=TRUE,
                         sep = "\t",
```

```
                          row.names=1,
                          quote = "",
                          check.names=FALSE)


training_set_results <- read.table('training_set_answers.txt',
                                   header=TRUE,
                                   sep = '\t',
                                   row.names=1,
                                   quote = "",
                                   colClasses="factor")


expression = as_tibble(rownames_to_column(as.data.frame(t(expression)), var='cellline'))
training_set_results = as_tibble(rownames_to_column(training_set_results, var='cellline'))
```

The only oddity is that some of the provided files expected the user to use row names, which tidyverse does not use. We worked around this by importing those files using `read.table` and then using the tibble command `rownames_to_column`.

EDA

First, we look for missing values, as their presence will make a large impact on how we proceed. If we had missing values, we might need to exclude the associated data points.

```
anyNA(expression)
```

```
## [1] FALSE
```

```
anyNA(training_set_results)
```

```
## [1] FALSE
```

```
anyNA(subtypes)
```

```
## [1] FALSE
```

```
anyNA(scoring)
```

```
## [1] FALSE
```

Now that we can see that we have no missing data, we can evaluate data types.

```
sapply(subtypes, class)
```

```
##    cellline     subtype
## "character"    "factor"
```

```
table(sapply(expression, class))
```

```
##
## character   numeric
##         1     18632
```

```
class(expression$cellline)
```

```
## [1] "character"
```

```
table(sapply(training_set_results, class))
```

```
##
## character    factor
##         1        12
```

```
class(training_set_results$cellline)
```

```
## [1] "character"
```

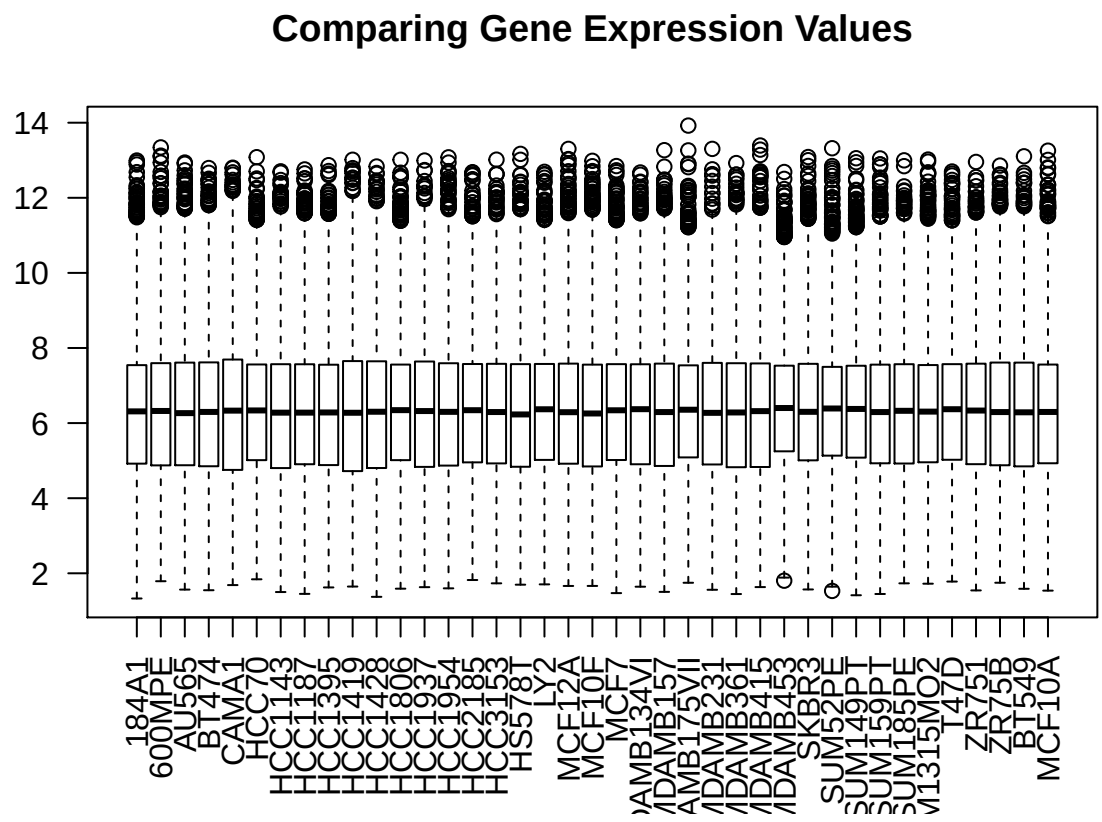
Now all of the data we imported is stored in the correct type for analysis in R, so we can go on to check for normalization in the gene expression data. To do that, we evaluate the quantiles for each cell line, since we're not interested in what the actual values are but in how much they vary.

```
expression_quantiles <- apply(expression[,-1], 1, quantile, probs=c(.25,.5, .75))
apply(expression_quantiles, 1, var)
```

```
##         25%         50%         75%
## 0.011071029 0.001494088 0.001363948
```

If we're not happy with just checking the quantiles alone, we can look at a row-wise boxplot.

```
boxplot(t(subset(expression, select=-c(cellline))),
        names=expression$cellline,
        las=2)
title("Comparing Gene Expression Values")
```



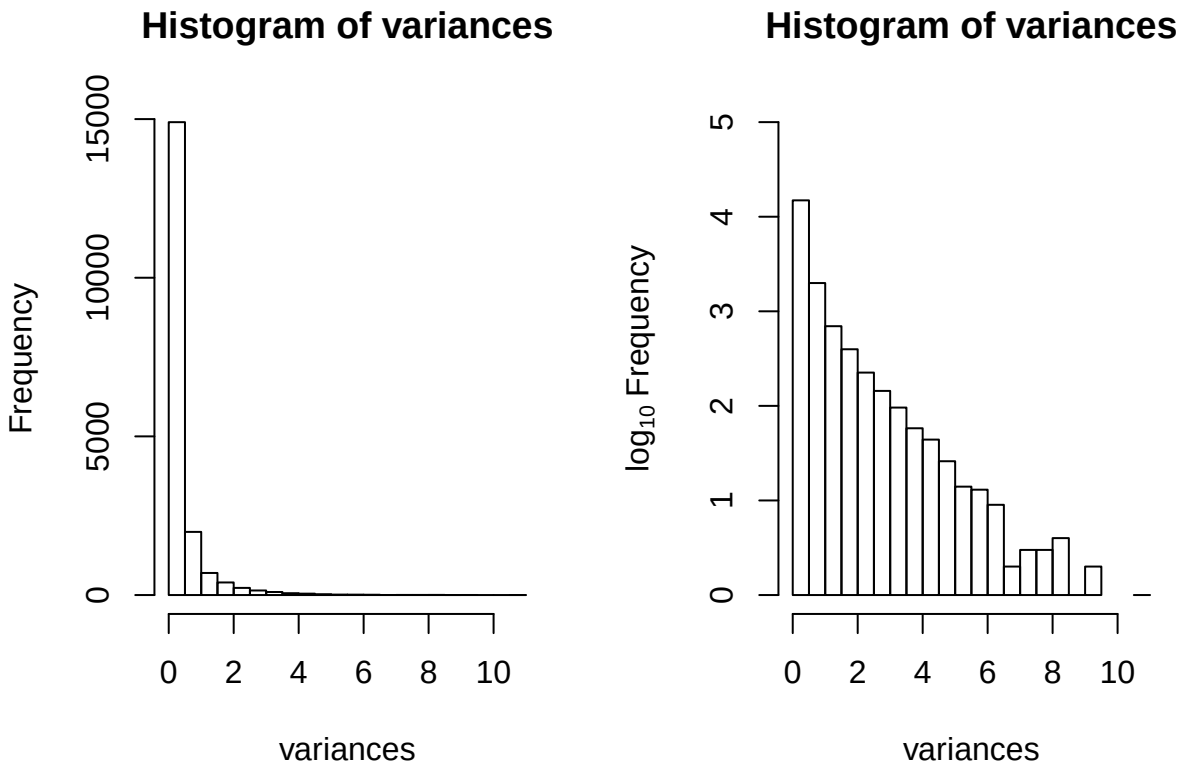
From the results, we can see the quantiles are all similar, and are satisified that the expression data does not require any further normalization.

Classification

The next step in our analysis is to slim down our feature space. We are working with 18633 genes, which is not really feasible to try and make work. The first thing we do is remove from the analysis the genes that have low variance, in other words, genes that seem to not change across different samples. To determine the cutoff, we analyze what the gene variance looks like.

Since we expect an exponential decay of variances, we look at the log-scale of the variances to see if there is an obvious cut-off.

```
variances <- apply(expression[,-1], 2, var)

# Here we first split the test and train expression data
training_expressions <- semi_join(expression, training_set_results, by="cellline")
test_expressions <- anti_join(expression, training_set_results, by="cellline")

par(mfrow=c(1,2))
histogram.data <- hist(variances, breaks=30, plot=TRUE)
histogram.data$counts <- log(histogram.data$counts, 10)
plot(histogram.data, ylim=c(0, 5), ylab=expression("log"[10]*" Frequency"))
```



While we can see that there is a change in the histogram showing the variance as we approach variance values of 5, we will choose a much lower threshold due to our method of dimensionality reduction. The first step is determining the eigen-values and eigen-vectors from the co-variance matrix. Numerical solvers are fairly good, however, calculating the covariance between every gene involves making n^2 co-variance calculations. Since we're starting with 18633 genes, that would result in 18633^2 variance calculations (where each feature has 25 values from the training data), so we need to slim it down somewhat.

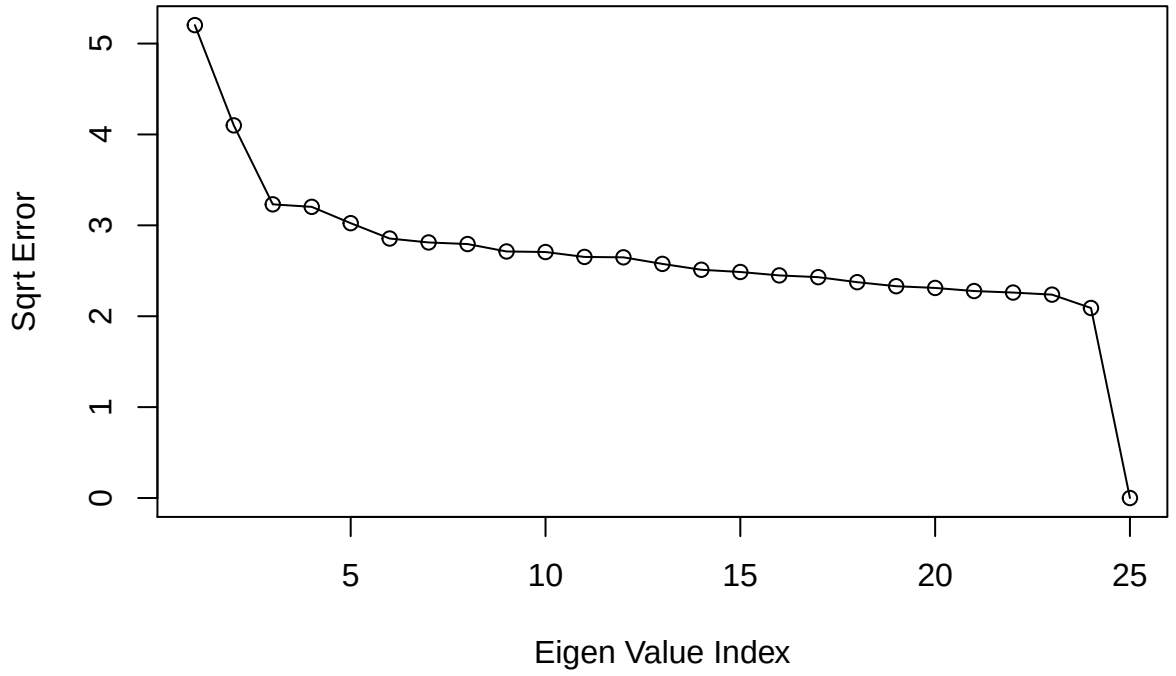
```
cutoff = 2

features_of_interest <- colnames(training_expressions[,-1])[which(variances > cutoff)]
training_expressions_pruned <- training_expressions[,features_of_interest]
test_expressions_pruned <- test_expressions[,features_of_interest]
```

At this point we have 645 genes we are interested in, and given that we have 25 observations in our training set, this feature space is still far too large to do analysis on its own. So we now perform a PCA where we look at doing dimensionality reduction. When performing a PCA, we take the eigen-values of the covariance matrix. Each eigen-value corresponds to the error that would be introduced should we exclude the corresponding eigen-vector and lessen the dimension by 1. While we can do the calculation manually by multiplying the data-frame training_expressions_pruned by our eigen vectors to get the resulting eigen-values, we elected to use the `prcomp` library.

```
pca <- prcomp(training_expressions_pruned)
```

To determine how much of a dimensionality reduction we can do without introducing too much error, we plot the error, which is the square root of the eigen-values.

```
plot(sqrt(pca$sdev), type="o", xlab="Eigen Value Index", ylab="Sqrt Error")
```



The first plot shows what we would expect: the eigen-values which are the square of the error are decreasing, but we effectively need at least 24 of the eigen-vectors to accurately account for the variance in our 25 samples. This result is not unexpected given our very small sample size ($n = $25). This indicates that ideally, for a PCA, we would have more samples to better account for all the variance. So we will make use of all 25 of our eigen-vectors.

```
expressions_pca <- pca$x
test_expressions_pca <- cbind(select(test_expressions, cellline),
                              predict(pca, test_expressions_pruned))
```

We ended up going down to 25 dimensions, and given our sample size, that's far more managable. The next step is to create our training and test data structures that we can test over. In order to minimize repeating ourselves in our code, we use some tivyverse joining functions, and restructure the data using the gather function.

```
training_data <- inner_join(training_set_results,
                            bind_cols(as_tibble(expressions_pca),
                                      semi_join(subtypes,
                                                training_expressions,
                                                by="cellline")),
                            by = "cellline")
training_data <- gather(training_data, "drug", "response", 2:13)
training_data$response <- as.factor(training_data$response)

test_data <- inner_join(tibble('cellline' = setdiff(expression$cellline,
                                                    training_set_results$cellline)),
                        inner_join(test_expressions_pca,
                                   subtypes,
                                   by = "cellline"),
                        by = "cellline")
```

Now that we have the test and training data in place, as well as our feature reduction, we can move onto our modeling methods.

Linear Regression

Since we need to make two submissions, the first submission will make use of linear regression. Given that we have as many parameters as we have samples, we do not expect linear regression to perform well, but are treating this as an opportunity to learn to implement bagging from the ground-up.

First we create our bagging function, which given n observations will make n samples with replacement as many times as we want (defaulting to 500 iterations), calculate the prediction parameters, and then average them out.

```
bagging <- function(training, test_vector, drug, iterations=500)
{
  training_data_slim <- subset(filter(training,
                                      drug == drug),
                               select=-c(drug, cellline))
  predictions <- foreach(m=1:iterations, .combine=cbind)  %do% {
    training_bag_set <- sample_n(training_data_slim,
                                 size=nrow(training_data_slim),
                                 replace=TRUE)
    suppressWarnings(fit <- lm(as.numeric(levels(response))[response] ~ .,
                               data=training_bag_set))
    suppressWarnings(predict(fit, newdata=test_data))
  }
  rowMeans(predictions)
}
```

```
outcome <- tibble('cellline' = scoring$cellline,
                  'drug' = scoring$drug,
                  'id' = scoring$id,
                  'value' = NA)

for(drug in unique(training_data$drug)){
  predictions <- bagging(training_data, test_data, drug)
  for(j in seq_along(test_data$cellline)){
    cellline <- test_data$cellline[j]
    row <- which(outcome$cellline == cellline & outcome$drug == drug)
    outcome$value[row] <- predictions[j]
  }
}

write_csv(outcome[,3:4], 'submission_linear_regression.csv')
```

We make individual models per drug, then run the appropriate model for each drug in the test set.

Amazingly enough, the linear regression model didn't do too poorly. At the time of submission, it was not the worst submssion made (though it was the second worst), with a score of 0.51. The linear regression model did not take into account the subtype of the tumor (due to the limintation of input parameters), and since we were taking samples with replacement, the `lm` was throwing up warnings of having dependent samples, which is why we used the `suppressWarnings` option.

In short, linear regression is not a great modeling tool when you have a highly dimensional feature space but very few samples (n << p).

Random Forests

The next model type we used was random forests. Random forests lends itself really nicely to this data set, as it has no problem handling high-dimensional data with very small sample sizes. Even though we reduced the expression set data down to 38 dimensions through the PCA, that's still more than the number of training samples we have to work with.

```
ntrees = 200000
models <- training_data %>%
            select(-cellline) %>%
            group_by(drug) %>%
            nest() %>%
            mutate(model = map(data, ~ randomForest(response ~ .,
                                                    data=.,
                                                    importance=TRUE,
                                                    ntree=ntrees)))


outcome <- tibble('cellline' = scoring$cellline,
                  'drug' = scoring$drug,
                  'id' = scoring$id,
                  'value' = NA)

for(i in seq_along(models$drug)){
  drug <- models$drug[i]
  predictions <- as.vector(predict(models$model[[i]], test_data[,-1]))
  for(j in seq_along(test_data$cellline)){
```

```
    cellline <- test_data$cellline[j]
    row <- which(outcome$cellline == cellline & outcome$drug == drug)
    outcome$value[row] <- predictions[j]
  }
}

write_csv(outcome[,3:4], 'submission_random_forests.csv')
```

We used a similar modeling structure here as with linear regression, making a separate forest for each drug.

We settled on the number of trees by trying several different values; tree counts well below 200000 give worse results, while tree counts above it take longer to compute for no gain in performance. Most of the other defaults for random forest were in line with best practices, such as defaults for node size and number of candidates per split. We tested a version where the factor variables were identified as such in the R code, but found it actually yielded worse performance, perhaps due to random forests' tendency to over-weight factor variables with more factors. We also tested conditional random forests but excluded them from the final analysis for performing significantly worse.

The random forests model generates significantly better results than the linear regression model, giving us a score of 0.64. Given the simplicity of the algorithm, for a safe and fast analysis, random forests seem to be an excellent way to go.