Feature Extraction Methods

Daniel Chen

2023-08-03

Overview

A common obstacle in machine learning is determining the features that optimize (increase the accuracy of our predictions or explain the relationship between variables better) our model. We can throw everything but the kitchen sink at the model but this leads to a handful of problems:

1. Interpretation: How can we make sense of a model that has 100 explanatory variables?
2. Computation issues: Running models with a lot of features can be computationally expensive.
3. Overfitting: The machine learning algorithm may learn the training data very well, but generalize poorly to new samples.

To overcome this problem, practitioners turn to feature selection where a subset of the variables are returned.

The goal if this document is to look at a couple of methods commonly used to select features. It is not intended to be exhaustive nor is it intended to thoroughly analyze different algorithms. We will specifically look at LASSO regression and best subset selection.

Data

For this overview, we'll use COVID-19 data. Our outcome variable is deaths per 100,000 people, and we have a number of explanatory variables from Opportunity Insights and PM COVID. For the purposes of demonstration, we won't use every single variable as it will be too computationally expensive.

```r
set.seed(59000)

data_path <-
  '~/Desktop/GitHub/tutorials/Feature Selection/Data/Covid002.csv'
data <- read.csv(data_path)

# data dictionary explaining what variables mean
data_dictionary_path <- paste0(
  '~/Desktop/GitHub/tutorials/Feature Selection/Data/',
  'Variable Description.xlsx'
)
data_dict <- readxl::read_excel(path = data_dictionary_path)
```

We'll keep the `deathspc` variable, all of the variables that come from `PM COVID`, and a random half of the `Opportunity Insights` variables (again, for computation reasons)

```
final_vars <- c(
  data_dict[data_dict$Source %in% 'PM_COVID', 'Variable', drop = TRUE],
  data_dict |>
    dplyr::filter(Source %in% 'Opportunity Insights') |>
    dplyr::sample_frac(0.5) |>
    dplyr::pull(Variable),
  'deathspc'
)

data <- data[, final_vars]
```

We'll also check to see if there are `NA` values in our data. Although common techniques include imputing the mean or the median, for this exercise we'll simply drop missing data from our data set.

```
purrr::map(.x = data, .f = function(column) { any(is.na(column)) }) |>
  (\(variables) base::Filter(f = isTRUE , x = variables))() |>
  names()
```

```
## [1] "mig_inflow"          "rel_tot"             "mort_30day_hosp_z"
## [4] "diab_hemotest_10"    "mammogram_10"        "frac_middleclass"
## [7] "score_r"             "mig_outflow"         "reimb_penroll_adj10"
```

```
data <- tidyr::drop_na(data)
```

Train Test Split

Next, we'll split our data into a training and testing set, so we can identify patterns in the training set and apply them to the testing set to see how it performs. We'll partition the data such that 80% will be used for training and 20% will be used for testing. Of course, we'll set our seed for reproducibility.

```
sample <- sample(
  x = c(TRUE, FALSE),
  size = nrow(data),
  replace = TRUE,
  prob = c(0.8, 0.2)
)

train <- data[sample, ]
test <- data[!sample, ]

# Return training and testing dimensions
purrr::map(.x = list(train, test), .f = dim) |>
  purrr::set_names(c('train dimensions', 'test dimensions'))
```

```
## $`train dimensions`
## [1] 2386   34
##
## $`test dimensions`
## [1] 571  34
```

```r
# Create targets and features for our training and testing set
xtrain <- data.matrix(frame = train[, !names(train) %in% 'deathspc'])
ytrain <- train$deathspc

xtest <- data.matrix(frame = test[, !names(test) %in% 'deathspc'])
ytest <- test$deathspc
```

LASSO Regression

At a high-level, LASSO regression shrinks coefficients all the way to 0 if they do not add information to our model. By shrinking them to 0, we can remove them from our model entirely, and we'll be left with a subset of our original features.

Coefficients are shrunk to 0 with the following equation where SSR stands for the sum of squared residuals:

$$SSR + (\lambda \times |\beta|)$$

Recall that the goal of regression is to find the best fit line that minimizes our error (or sum of squared residuals). If we hold β constant while increasing λ, then the result will always grow larger and larger which is counter to our goal of minimizing the residuals. The only way to minimize the outcome while λ increases is to decrease β until it reaches 0. By shrinking β to 0, we've effectively removed it from the model because we're saying it has no effect on our outcome. Note that we don't want to use a negative value for β because we're ultimately taking the absolute value of it which will ultimately increase our outcome when multiplying by an increasing λ.

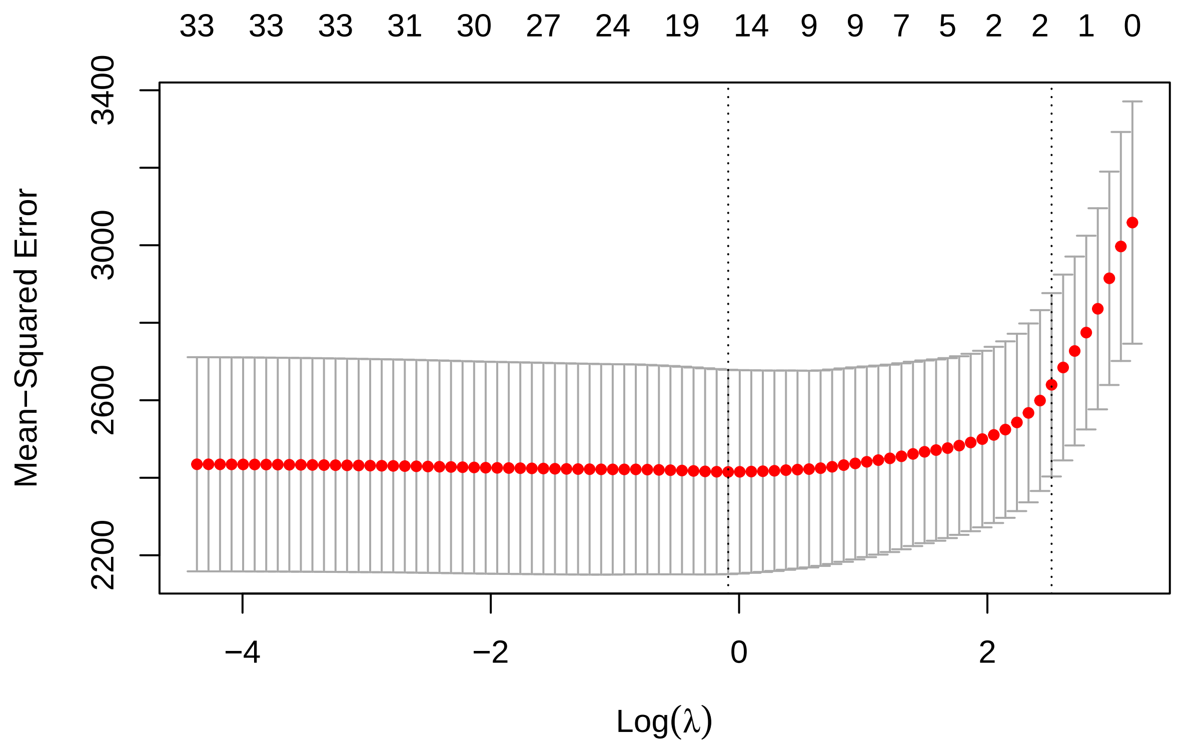
To minimize the error term, β must equal 0 if λ is increasing.

What is the optimal value of λ then? We turn to cross validation to identify it on our training data. We'll use the `cv.glmnet` function from the `glmnet` package, and set the `alpha` argument equal to 1. This is important because it specifies LASSO regression and not ridge regression.

```r
cv_lasso <- glmnet::cv.glmnet(
  x = xtrain,
  y = ytrain,
  alpha = 1,
  nfolds = 10,
  keep = TRUE
)
```

The nice thing about the output of `cv.glmnet` is that it allows us to create a plot of the log values of λ and their corresponding mean squared error (MSE). The optimal value of λ minimizes the MSE, and we can see that it corresponds to the dotted vertical line that is closer to the y-axis on the plot below. This value of λ also corresponds to about 12 predictors which is much fewer than the 35 in our original model.

```r
plot(cv_lasso)
```



We can also easily identify the minimum value of λ and its corresponding MSE with the following

```
cv_lasso$lambda.min
```

```
## [1] 0.9160583
```

```
min(cv_lasso$cvm)
```

```
## [1] 2415.04
```

And we can always verify that the MSE is correct on our end by manually calculating it ourselves:

```
# Helper function to calculate MSE
get_mse <- function(x, y) { mean((x - y) ^ 2) }

# This will return the same MSE as the training fold from cross-validation
get_mse(
  x = cv_lasso$fit.preval[, which(cv_lasso$lambda == cv_lasso$lambda.min)],
  y = ytrain
)
```

```
## [1] 2415.04
```

Now that we have the optimal value of λ, we can fit the model to our training data and identify which features have been dropped from the model. These zeroed-out coefficeints are denoted by `.` in the `s0` column of our output.

```r
optimal_lasso <- glmnet::glmnet(
  x = xtrain,
  y = ytrain,
  alpha = 1,
  lambda = cv_lasso$lambda.min
)

optimal_lasso$beta
```

```
## 33 x 1 sparse Matrix of class "dgCMatrix"
##                                    s0
## pm25                       1.64392549
## pm25_mia                   .
## summer_tmmx               -0.10514591
## summer_rmax                .
## winter_tmmx                .
## winter_rmax                .
## bmcruderate                .
## exercise_any_q1            .
## brfss_mia                  .
## mig_inflow               -84.64955239
## bmi_obese_q2               .
## cs00_seg_inc               .
## cs_race_theil_2000        29.74704347
## cs00_seg_inc_aff75         .
## rel_tot                    .
## exercise_any_q3            .
## cur_smoke_q2              -0.46480445
## mort_30day_hosp_z          0.28379229
## pop_density                0.01107039
## diab_hemotest_10          -0.56290200
## cs_born_foreign            0.60117143
## mammogram_10               0.03107146
## frac_middleclass        -126.74844296
## adjmortmeas_chfall30day    .
## puninsured2010            -1.08730299
## bmi_obese_q3              -1.31455105
## score_r                    .
## poor_share               -18.48570823
## mig_outflow                .
## cur_smoke_q3               .
## unemp_rate               -62.92406706
## reimb_penroll_adj10        .
## taxrate                  156.73039099
```

We can see that there are only 16 of the initial 33 features in our model that return estimates.

We can of course also make predictions from our training data once our model has been fit:

```r
# Make predictions and calculate MSE
train_predictions <-
  stats::predict(optimal_lasso, newx = xtrain, s = optimal_lasso$lambda)
train_MSE <- get_mse(x = train_predictions, y = ytrain)
```

```
train_MSE
```

```
## [1] 2242.012
```

And we can also do the same for the test data:

```
optimal_lasso <- glmnet::glmnet(
  x = xtrain,
  y = ytrain,
  alpha = 1,
  lambda = cv_lasso$lambda.min
)

# Make predictions and calculate MSE
test_predictions <-
  stats::predict(optimal_lasso, newx = xtest, s = optimal_lasso$lambda)
test_MSE <- get_mse(x = test_predictions, y = ytest)

test_MSE
```

```
## [1] 2048.943
```

Note that our test MSE is much much lower than our training MSE. Generally, good machine learning models will return similar MSEs for both the training and the testing sets. Since our test MSE is so much lower than our training set, LASSO is likely not a good fit for this specific data set. In general, LASSO performs better when there are fewer features to begin with. However, the purpose of this exercise is to demonstrate how LASSO works in returning a subset of features.

Best Subset

```
best_AIC <- bestglm::bestglm(Xy = train, IC = 'AIC', method = 'exhaustive')
```