Regularization Basics

Jim Harner

10/4/2020

6.2 Regularization Basics

The analysis below suggests how difficult it is to choose a model. It is best to partition the data into 3 sets:

- Training set: to explore the type of model (algorithm) using performance measures.
- Validation set: to map models over a parameter grid, e.g., λ and α in the elastic net. The 3-D surface can then be explored to find regions where the performance measures are near optimal.
- Test set: to select an optimal model based on one or a few best model types and their optimal tuning parameters as determined from the validation data set.

If only the training and test data sets are available, tuning parameters can be assessed using cross-validation for various model types. The best of breed for each model type can then be assessed by the test data.

The strategy is to filter the models by the training and validation data sets so only a few are assessed in the test data set.

6.2.1 Spark Algorithms

The Spark algorithms for machine learning have been evolving quickly.

General reference:

MLlib

The RDD-based algorithms can be seen here:

RDD Optimizaton

It is now in maintenance mode.

The new API is Spark DataFrame based:

DataFrame Optimization

6.2.2 Regression

The loss function, i.e., RSS for regression, can be generalized, e.g., by regularization. We now consider fitting all p predictors by constraining, (regularizing) the coefficients, i.e., shrinking the coefficients towards 0. This can often greatly reduce the coefficient variances without appreciably increasing the bias.

The idea is to minimize:

$$RSS(\beta_0, \beta_1, \ldots, \beta_p, \lambda, \alpha) = \sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_{i1} - \cdots - \beta_p x_{ip})^2 + \lambda[(1-\alpha)\sum_{j=1}^{p}\beta_j^2/2 + \alpha\sum_{j=1}^{p}|\beta_j|]$$

with respect to the β's, λ, and α. The parameter λ, a tuning parameter controlling the strength of the penalty, is generally determined by cross validation. The elastic-net penalty α ranges from $\alpha = 0$ (ridge regression) to $\alpha = 1$ (lasso regression). It's value is chosen by the modeler although various choices can be made also using validation.

Ridge regression shrinks correlated predictors (in groups) toward each other, whereas the lasso tends to pick one (in each group) and removes the others. The elastic-net, e.g., at $\alpha = 0.5$, tends to select groups in and out.

Ridge regression and the lasso will be considered initially and then we will explore the more general elastic net. Regularized regression, along with other types of models, can be done with the `glmnet` R package.

Ridge Regression The ridge regression coefficient estimates $\hat{\beta}_j^r$ minimizes:

$$RSS(\beta_0, \beta_1, \ldots, \beta_p, \lambda) = \sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_{i1} - \cdots - \beta_p x_{ip})^2 + \lambda \sum_{j=1}^{p} \beta_j^2/2,$$

where $\lambda \geq 0$ is the tuning parameter. The first term on the right is the RSS, whereas the second term is the shrinkage penalty, which is small when the coefficients are near 0. The product $\hat{\beta}_{j,\lambda}^r x_{ij}$ is not scale invariant, as is the case for least squares, and thus the predictors should be standardized.

As λ increases the variances of the ridge regression coefficient estimates decrease, but the biases increase somewhat. The optimal value of λ is found by minimizing the test MSE generally using cross validation.

Ridge regression can be computed in `glmnet` by setting α to 0. The first argument to `glmnet` is the matrix of predictors, which is standardized by default. The coefficients for the least square model are based on standardized predictors by default. The ridge regression is obtained by using a range or λ's, e.g., `nlambda = 20`. The function `cv.glmnet` can be used to perform cross validation to determine the optimal λ. The optimal ridge regression does differ from the least squares regression, which can be seen by comparing the corresponding regression coefficients and predicted values (below).‘

The lasso The lasso regression coefficient estimates $\hat{\beta}_j^l$ that minimize:

$$RSS(\beta_0, \beta_1, \ldots, \beta_p, \lambda) = \sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_{i1} - \cdots - \beta_p x_{ip})^2 + \lambda \sum_{j=1}^{p} |\beta_j|,$$

where $\lambda \geq 0$ is the tuning parameter. The interpretation is similar to that of ridge regression except that the lasso forces some of the regression coefficients to 0 as the tuning parameter increases. This is a form of variable selection which helps with model interpretation.

Elestic-net The minimization problem at the beginning of this section is for the elastic net. In the case the elastic net penalty α must be specified by the modeler.

6.2.3 Slump Example (glmnet)

Load `slump.csv` into Spark with `spark_read_csv` from the local filesystem.

```
slump_sdf <- spark_read_csv(sc, "slump_sdf",
                  path = "file:///home/rstudio/rspark-tutorial/data/slump.csv")
head(slump_sdf)
```

```
## # Source: spark<?> [?? x 10]
##   cement  slag fly_ash water    sp coarse_aggr fine_aggr slump  flow
##    <dbl> <dbl>   <dbl> <dbl> <dbl>       <dbl>     <dbl> <dbl> <dbl>
## 1    273    82     105   210     9         904       680    23  62
## 2    163   149     191   180    12         843       746     0  20
## 3    162   148     191   179    16         840       743     1  20
## 4    162   148     190   179    19         838       741     3  21.5
## 5    154   112     144   220    10         923       658    20  64
## 6    147    89     115   202     9         860       829    23  55
## # ... with 1 more variable: compressive_strength <dbl>
```

```
# partition into a training and test Spark DataFrames
slump_partition <- tbl(sc, "slump_sdf") %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 2)
slump_train_sdf <- slump_partition$training
slump_test_sdf <- slump_partition$test
```

We do this in order to have the same data partitions as in Section 6.3.

We now `collect` the training and test Spark DataFrames into R as regular data frames.

```
slump_train_df <- collect(slump_partition$training)
slump_test_df <- collect(slump_partition$test)
```

We can now use `glmnet` to model compressive strength. This is a regular regression since both `alpha` and `lambda` are 0.

```
slump.x <- model.matrix(compressive_strength ~ cement + slag + fly_ash + water + sp + coarse_aggr + fin
slump.y <- slump_train_df$compressive_strength
slump_glmfit <- glmnet(x = slump.x, y = slump.y, alpha=0, lambda=0,
                       standardize = TRUE, intercept = TRUE)
coef(slump_glmfit)
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                        s0
## (Intercept) 105.66906735
## cement        0.07446521
## slag         -0.01319228
## fly_ash       0.06060996
## water        -0.20827231
## sp           -0.03243234
## coarse_aggr  -0.04174206
## fine_aggr    -0.02240826
```

The regression coefficient estimates from `glmnet` are similar to, but slightly unequal to the coefficient estimates from `sparklyr`'s `ml_linear_regression`. This is due to differences in the package algorithms underlying linear regression.

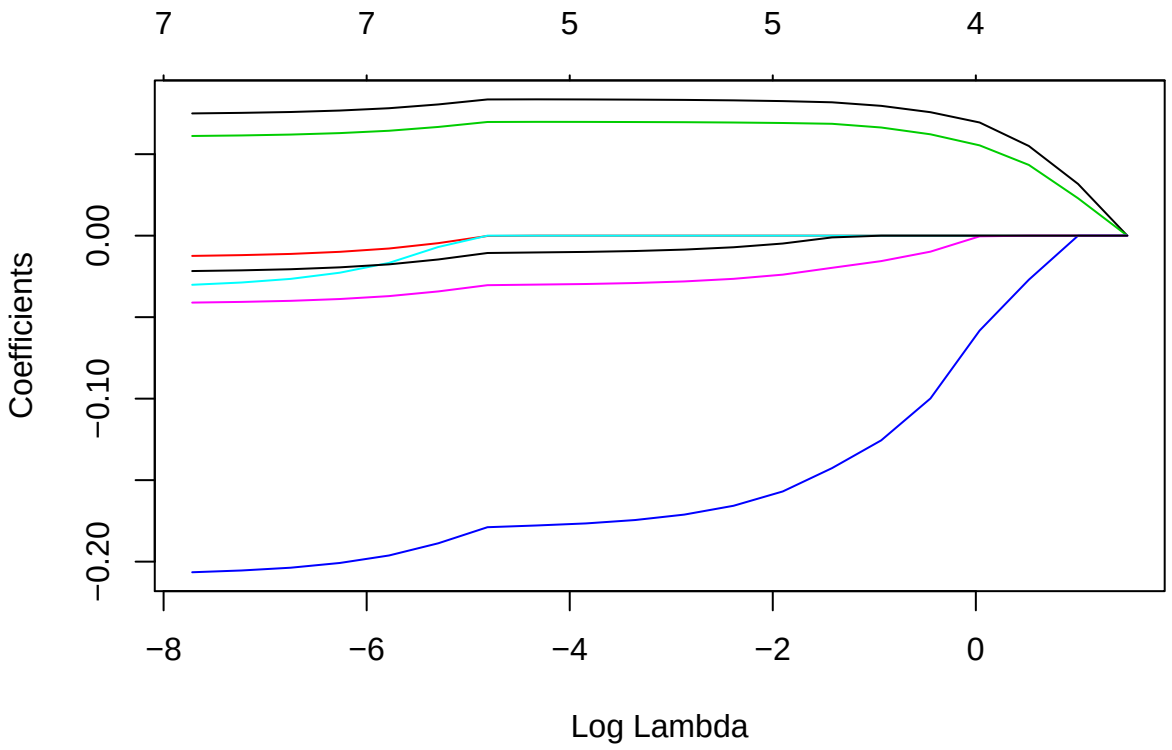
Note that the predictors (features) are standardized. This is essential for classifiers, but good practice in regression. Convergence is faster for most algorithms.

```
predict(slump_glmfit, slump.x[1:5,])
```

```
##         s0
## 1 32.85766
## 2 30.19840
## 3 32.11886
## 4 23.81682
## 5 28.27475
```

We now build the model with a lasso penalty.

```
slump_glm_lasso <- glmnet(x = slump.x, y = slump.y, alpha = 1, nlambda = 20,
                          standardize = TRUE, intercept = TRUE)
plot(slump_glm_lasso, xvar = "lambda", labels = TRUE)
```




```
slump_glm_lasso_cv <- cv.glmnet(x = slump.x, y = slump.y, type.measure = "mse",
                                alpha = 1, nfolds = 10)
slump_glm_lasso_cv$lambda.min
```

```
## [1] 0.09801078
```

Our choice of λ based on 10-fold cross-validation is less than what we choose in Section 6.3.

The coefficient estimates, based on 10-fold cross-validation, are:

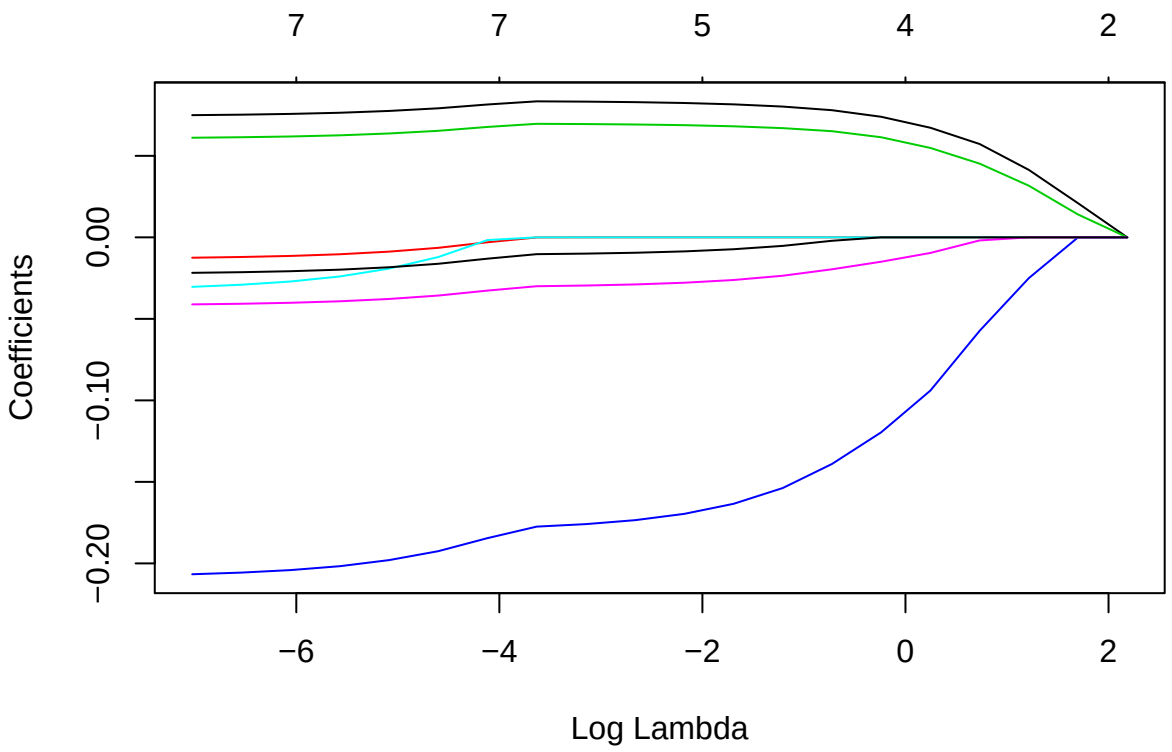
```
coef(slump_glm_lasso_cv, s = "lambda.min")
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                         1
## (Intercept) 67.294181648
## cement       0.082977762
## slag         .
## fly_ash      0.069400892
## water       -0.164799076
## sp           .
## coarse_aggr -0.026216369
## fine_aggr   -0.006875596
```

Notice that `cement`, `fly_ash`, and `water` (and possibly coarse_aggr‘) converge more slowly to 0, which agrees with our results in Sections 6.1 and 6.3.

We now look at the elastic net with `alpha = 0.5`.

```
slump_glm_enet <- glmnet(x = slump.x, y = slump.y, alpha = 0.5, nlambda = 20)
plot(slump_glm_enet, xvar = "lambda", labels = TRUE)
```




```
slump_glm_enet_cv <- cv.glmnet(x = slump.x, y = slump.y, type.measure = "mse",
                               alpha = 0.5, nfolds = 10)
slump_glm_enet_cv$lambda.min
```

```
## [1] 0.08485306
```

```
coef(slump_glm_enet_cv, s = "lambda.min")
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                         1
## (Intercept) 72.549824085
## cement       0.082757731
## slag         .
## fly_ash      0.069094944
## water       -0.172104902
## sp           .
## coarse_aggr -0.028511365
## fine_aggr   -0.009149299
```

The results are similar.

It would be possible to build a parameter grid extending the idea in Section 6.1. We could assess performance metrics over this grid and then test. For now you can experiment with different values of α and λ or minimize λ for various α directly with `glmnet`.

```
spark_disconnect(sc)
```