Linear Regresson: Regularization

Signal Data Science

Some helpful notes on the `glmnet` package are in a section at the end of this document. As you work through this assignment, you should **refer to those notes** to understand how `glmnet()` and `cv.glmnet()` work.

Exploring regularization with simulated data

Before using regularized linear regression on real data, we'll build some intuition by using regularization in a simpler context with simulated data.

Define `x` and `y` using:

```
set.seed(1); j = 50; a = 0.25
x = rnorm(j)
error = sqrt(1 - a^2)*rnorm(j)
y = a*x + error
```

If you run `summary(lm(y ~ x - 1))`, corresponding to a linear model with no constant coefficient, you should get an estimated value of 0.2231 for `a`.

- Write a function `cost(x, y, aEst, lambda, p)` which takes
  - two vectors `x` and `y` of equal length,
  - a estimate of the value of `a`, `aEst`,
  - a regularization parameter `lambda`, and
  - a number `p = 1` or `2`, indicating whether L^1 or L^2 regularization is being performed.

  Your function should return the sum of squared errors for the model `y = aEst*x` plus the L^p regularization term. Check that `cost(1, 2, 3, 4, 2)` returns 37.
- Create two vectors, one corresponding to values of λ given by $2^{-2}, 2^{-7}, \ldots, 2^5, 2^6, 2^7$ and another corresponding to values of `aEst` from -0.1 to 0.3 in equally spaced increments of 0.001.
- Use `expand.grid()` on the two vectors of parameter to create a data frame called `grid` with two columns `lambda` and `alpha`, where each row

is a unique pair of `(lambda, aEst)` values.

- Add `costL1` and `costL2` columns to `grid` where we'll store the cost of associated with each pair `(lambda, aEst)` for each of `p = 1` and `p = 2`. Fill in those columns with `cost()`.
- Write a function `get_plot(lambda, p)` which looks at the rows of `grid` with the specified value of `lambda` and returns the `qplot()` object generated by plotting the corresponding values of `aEst` on the abscissa (x-axis) and the corresponding values of either the L^1 or L^2 regularized cost function (depending on `p`) on the ordinate (y-axis).
- Use `lapply()` with `get_plot()` to create two lists, `plotsL1` and `plotsL2`, where the ith plot is generated using the ith value of the vector of `lambda` values.
- Use `multiplot` with `cols=2` and the `plotlist` parameter set to either `plotsL1` or `plotsL2` to visualize the results. Interpret the differences between the sets of plots for the L^1 and L^2 regularized cost function with respect to how L^1 regularization drives coefficient estimates to zero whereas L^2 regularization does not.

Comparing regularization and stepwise regression

We will now begin to use the `glmnet` package, which provides the functions `glmnet()` and `cv.glmnet()` for regularized linear regression.

Continue using the aggregated speed dating dataset (`speed-dating-simple`) from yesterday. For simplicity, we'll restrict to analyzing average *attractiveness ratings* (`attr_o`) for *males*, making predictions for that specific rating in terms of the 17 self-rated activity variables.

Getting acquainted with `glmnet`

`glmnet()` can perform both L^1 and L^2 regularized linear regression as well as a mix of the two (which we'll be exploring later). When calling `glmnet()`, you can set `alpha=1` for L^1 regularization and `alpha=0` for L^2 regularization.

- Create a `activities_scaled` variable which is the result of calling `scale()` on a data frame containing only the 17 activity variables and an `attr_o` variable equal to the (unscaled) `attr_o` column of the original dataset. You'll be passing these variables into `glmnet()`.
- Use `glmnet()` to fit both L^1 and L^2 regularized linear models for `attr_o` in terms of the variables in `activities_scaled`. By default, the function automatically determines a range of different λ values to try. To illustrate

this, access the object returned by `glmnet()` and print out the values of λ used for each of the two regularized models.

- Write a function `get_rmses(fit, df, target)` that takes as input `fit`, the model object generated by a call to `glmnet()`, `features`, the predictors used in the call to `glmnet()`, and `target`, the true values of the target variable being predicted. Iterate over the values of λ which `glmnet()` tried. For each of those values, use `predict()` to generate predictions for the whole dataset and calculate the corresponding (non-cross-validated) RMSE. Return a vector of the calculated RMSE values.
- For both the L^1 and L^2 regularized linear fits, use `get_rmses()` to plot the RMSE corresponding to each value of λ against λ itself.

We can see that the non-cross-validated RMSE is minimized at $lambda = 0$! This is not surprising, because *on the whole dataset* the linear fit which minimizes the RMSE is precisely (and almost tautologically) the one obtained by minimizing the sum of squared errors, without adding on any regularization term. However, the same cannot be said for the *cross-validated* RMSE estimates. Typically, adding in some regularization will reduce the amount of overfitting sufficiently well that the optimal value of λ is greater than 0.

We can automatically generate cross-validated error estimates for a range of different λ values with `cv.glmnet()`. For each value of λ tested, `cv.glmnet()` uses n-fold cross-validation ($n = 10$ by default) to calculate an error estimate.[1] After running `fit = cv.glmnet(...)`, the value of λ (out of those tested) corresponding to the lowest error estimate can be accessed with `fit$lambda.min`. Similarly, the entire range of λ values tested can be accessed with `fit$lambda` and the cross-validated error estimates can be accessed with `fit$cvm`. (See the notes on `glmnet` below for additional clarification.)

- Use `cv.glmnet()` to fit L^1 and L^2 regularized linear models for attractiveness ratings in terms of the 17 activity variables. For each one, plot the cross-validated error estimates against the values of λ tested. Interpret the results.

Making cross-validated RMSE predictions

As you saw in the assignment on resampling, we want to use *cross-validation* to get more accurate estimates of model quality. In particular, stepwise regression tends to *overfit*, because of problems with multiple hypothesis testing, so non-cross-validated estimates of a stepwise regression model's quality are often overly optimistic. (However, stepwise regression is easy to understand and,

[1] Technically, by default `cv.glmnet()` calculates the *mean squared error* for linear regression models. However, since the square root function is monotonically increasing, it doesn't matter if one compares models using MSE or RMSE.

pedagogically, a good stepping stone to regularization, which is why we include it in our curriculum.)

- Use 10-fold cross validation to generate predictions for attractiveness with (1) stepwise regression, (2) L^1 regularized linear regression, and (3) L^2 regularized linear regression.
- For regularized linear regression, use `cv.glmnet()` to get cross-validated estimates of the optimal value of λ, and use that to make predictions with `predict(fit, test_data, s=fit$lambda.min)`.
- Calculate and view the RMSE associated with each of the three sets of predictions.

Here are some points to keep in mind:

- Within each cross-validation fold, you'll want to `scale()` the features which you pass into `cv.glmnet()`. When generating predictions on the *held-out* data, you want to scale the features in the same way (*i.e.*, by applying the same linear transformation). The output of `scale()` will contain *attributes* which can be accessed and passed into successive calls of `scale()` to perform the same transformation.
- If you have a string, say, `"attr_o"`, and you want to pass that into `lm()` as part of the regression formula, you can paste together the formula's components (*e.g.*, `paste("attr_o", "~.")`) and then pass that into the first argument of `lm()`.
- You should be running stepwise regression for each training fold separately.

Explore the difference in model quality between backward stepwise regression, L^1 regularized regression, and L^2 regularized regression when predicting attractiveness ratings.

Elastic net regression

Instead of penalizing the sum of squared errors by the L^1 or L^2 norm of the regression coefficients, we can penalize with a *combination* of the two, corresponding to setting the `alpha` parameter in `glmnet()` to a value *between* 0 and 1.[2] This is known as elastic net regularization and usually performs better than pure L^1 or L^2 regularization alone. However, instead of simply finding a *single* hyperparameter λ, we now must find the optimal *pair* of hyperparameters (α, λ) (by testing a large number of such pairs and calculating the cross-validated RMSE corresponding to each one). This is a more difficult task, because we must

[2] Read the official documentation for `glmnet` to figure out how the α parameter works..

search over a 2-dimensional space of hyperparameter combinations instead of a 1-dimensional space for the value of a single hyperparameter.

The `caret` package allows us to easily obtain a cross-validated estimate of the optimal (α, λ) values. Here's an example of how to use its `train()` function:

```
# Set grid of parameter values to search over
param_grid = expand.grid(.alpha = 1:10 * 0.1,
                         .lambda = 10^seq(-4, 0, length.out=10))

# Set 10-fold cross validation repeated 3x
control = trainControl(method="repeatedcv", number=10,
                       repeats=3, verboseIter=TRUE)

# Search over the grid
caret_fit = train(x=features, y=target, method="glmnet",
                  tuneGrid=param_grid, trControl=control)

# View the optimal values of alpha and lambda
caret_fit$bestTune

# View the cross-validated RMSE estimates
caret_fit$results$RMSE
```

In the above example, we perform *10-fold cross-validation* for each pair of hyperparameters (α, λ) to estimate the corresponding RMSE. The 10-fold cross validation is *repeated* three times, each time using a different random set of folds, in order to combat potential bias resulting from any particular choice of random folds. The optimal pair of values (α, λ) is the one corresponding to the lowest cross-validated RMSE.

- Use the `caret` package, following the above example, to find the optimal values for (α, λ) when predicting attractiveness ratings with elastic net regularization. Extract the minimum RMSE value obtained from the resulting `caret_fit` object and compare it to the cross-validated RMSE estimates obtained earlier with backward stepwise regression, L^1 regularized linear regression, and L^2 regularized linear regression.

A note on `glmnet`

Here, I'll cover two important points about the behavior of the `glmnet` package.

Passing in data

For `lm()`, you passed in the entire data frame, including both target variable and predictors. `glmnet(features, target, ...)` and `cv.glmnet(features, target, ...)` expect a *scaled matrix of predictors* for `features` and a numeric vector for `target`. Since `scale()` returns a matrix, you can just call `scale()` on a data frame of predictors and pass that in as `features`.

Picking values of λ

Ordinarily, one might expect that, for every different value of λ we want to try using with regularized linear regression, we would have to recompute the entire model from scratch. However, the `glmnet` package, through which we'll be using regularized linear regression, will automatically compute the regression coefficients for *a wide range of λ values simultaneously.[3]

When you call `glmnet()` – or, later, `cv.glmnet()` – you'll get out an object, which we'll call `fit`. (You should generally not be specifying *which* λ values the algorithm should use at this point – it'll try to determine that on its own.) By printing out `fit` in the console, you can see which values of λ were used by `glmnet`.

When you want to make predictions with this `fit` object, you'll have to specify *which* value of λ to use – instead of calling `predict(fit, new_data)`, you'll want to call `predict(fit, new_data, s=lambda)` for some particular λ = `lambda`. Similarly, when extracting coefficients, you'll want to call `coef(fit, s=lambda)`.

Finally, `cv.glmnet()` will use *cross-validation* to determine `fit$lambda.min` and `fit$lambda.1se`. The former is the value of λ (out of all those the algorithm evaluated) which minimizes the cross-validated mean squared error (MSE), and the latter is the greatest value of λ (again, of those evaluated by `glmnet`) such that the MSE corresponding to `fit$lambda.1se` is within 1 standard error of the MSE corresponding to `fit$lambda.min`.

If it turns out that the optimal value of λ lies at either end of the range of λ values used by `glmnet`, then you'll want to modify the range of λ. However, the documentation advises against passing in just a single value for the `lambda` parameter of `glmnet()` and `cv.glmnet()`, instead suggesting modifying `nlambda` and `lambda.min.ratio`.[4] Nevertheless, there are times when passing in a single

[3] "Due to highly efficient updates and techniques such as warm starts and active-set convergence, our algorithms can compute the solution path very fast."

[4] "Typical usage is to have the program compute its own `lambda` sequence based on `nlambda` and `lambda.min.ratio`. Supplying a value of `lambda` overrides this. WARNING: use with care. Do not supply a single value for `lambda` (for predictions after CV use `predict()` instead). Supply instead a decreasing sequence of `lambda` values. `glmnet` relies on its warms starts for speed, and it's often faster to fit a whole path than compute a single fit."

value makes sense, like when you've previously determined the optimal λ and want to just use that instead of a range of different λ values.