# 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. It's often useful to look at simple cases when trying to understand complex phenomena, so we'll consider the 1-dimensional case where we only have a single predictor variable.

### Computational exercise

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 $\lambda$ given by $2^{-2}, 2^{-1}, \ldots, 2^5, 2^6, 2^7$ 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 `aEst`, 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 $i$th plot is generated using the $i$th 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.

## Theoretical explanation

We can see that $L^1$ regularization successfully drives coefficient estimates to 0 as $\lambda$ increases while $L^2$ regularization does not. Why does this happen? We can get more insight into what's going on by looking at the underlying mathematics.

*Note:* The reasoning below will consider only the single variable case, where we have a single regression coefficient $\beta$. However, the reasoning applies equally as well to higher-dimensional cases – the notation just get a little bit more cluttered.

Suppose we have a vector of true values $\mathbf{y}$ and a predictor variable $\mathbf{x}$, and consider an $L^p$ regularized linear model for $\mathbf{y}$ in terms of $\mathbf{x}$ with regularization hyperparameter $\lambda$, coefficient estimate $\beta$, and intercept term $I$. That is, our model is given by

$$\mathbf{y} = \beta\mathbf{x} + I.$$

Call the sum of squared errors $\text{SSE} = S(\beta) = \sum_i (y_i - \beta x_i + I)^2$. Then our total cost function for the model is given by

$$C_p(\beta) = \text{SSE} + \lambda|\beta|^p = S(\beta) + \lambda|\beta|^p.$$

First, let's consider the case when we perform $L^2$ regularization. In that situation, $p = 2$ so $|\beta|^p = \beta^2$, and our cost function is

$$C_2(\beta) = S(\beta) + \lambda\beta^2.$$

What value of $\beta$ minimizes $C_2(\beta)$? Since $C_2(\beta)$ is the sum of two quadratic functions of $\beta$, it is smooth (being a quadratic function of $\beta$ itself) and therefore the minimum is achieved when $C_2'(\beta) = 0$, *i.e.*, when

$$S'(\beta) + 2\lambda\beta = 0.$$

Remember that we're interested in the situation where regularization causes the coefficient estimate $\beta$ to be driven to 0. It's then natural to ask: what needs to be true for $C_2(\beta)$ to be minimized at $\beta = 0$? The condition $C_2'(0) = 0$ must hold. Substituting $\beta = 0$ into our expression above, we obtain the condition

$$S'(0) + 2\lambda \cdot 0 = S'(0) = 0.$$

We can conclude that $L^2$ regularization will drive the coefficient estimate $\beta$ to 0 *if and only if* the condition $S'(0) = 0$ holds. Since the sum of squared errors $S(\beta)$ is a smooth quadratic function of $\beta$, the condition $S'(0) = 0$ is equivalent to saying that the sum of squared errors is minimized at $\beta = 0$, *i.e.*, that $\mathbf{y}$ is absolutely uncorrelated with $\mathbf{x}$.

**Therefore:** $L^2$ regularization drives the coefficient estimates to 0 if and only if the target variable is completely uncorrelated with its predictors. This is essentially *never* the case, so $L^2$ regularization will *never* drive coefficient estimates to 0.

We can also think about $L^2$ regularization in the following fashion: The cost function $C_2(\beta)$ is the sum of two convex quadratics, and the minimum of a sum of two convex quadratics has a minimum somewhere in between the minima of the two convex quadratics. *I.e.*, if the two convex quadratics are minimized at $\beta_1$ and $\beta_2$, their sum will be minimized for some $\beta$ *between* but *not equal to* $\beta_1$ and $\beta_2$. If $\mathbf{y}$ and $\mathbf{x}$ have nonzero correlation, then the sum of squared errors is minimized at some value $\beta \neq 0$, whereas the quadratic regularization parameter is minimized at $\beta = 0$. As such, it is *impossible* for their sum to be minimized at $\beta = 0$ precisely; only in the *infinite limit* of $\lambda \to \infty$, where the regularization term *completely dominates* the sum of squared errors, does the minimum of their sum approach $\beta = 0$.

Now, let's consider $L^1$ regularization, where $p = 1$ so

$$C_1(\beta) = S(\beta) + \lambda|\beta|.$$

This is the sum of a quadratic function of $\beta$ and a scaled absolute value function of $\beta$. Each of the two functions has a single local minimum, so the global minimum of $C_1(\beta)$ must be located at *either* (1) at the smooth local minimum of $C_1(\beta)$, where $C_1'(\beta) = 0$, *or* at (2) the minimum of the regularization parameter, where $\beta = 0$.

Taking the deriative of $C_1'(\beta)$, we obtain

$$C_1'(\beta) = S'(\beta) + \lambda \frac{|\beta|}{\beta}.$$

Since $S(\beta)$ is a quadratic function of $\beta$, $S'(\beta)$ is a linear function of $\beta$. Without any regularization (at $\lambda = 0$), $S'(\beta)$ is guaranteed to be 0 for *some* value of $\beta$ (any straight line on the $x$–$y$ axis will pass through $y = 0$ eventually).

Now, note that $|\beta|/\beta$ is equal to 1 for $\beta > 0$, equal to $-1$ for $\beta < 0$, and is undefined at $\beta = 0$. As such, adding on the regularization term $\lambda|\beta|/\beta$ to $S'(\beta)$ is equivalent to *shifting* the $\beta < 0$ side of the graph of $S'(\beta)$ down $\lambda$ units, shifting the $\beta > 0$ side of the graph up $\lambda$ units, and making the $\beta = 0$ point undefined. Intuitively, it must be the case that after a sufficiently large shift—after $\lambda$ exceeds some finite threshold—the two halves of the graph are driven completely above and below the $\beta = 0$ line, and neither one attains the value of 0 anywhere. As such, the only remaining candidate for the minimum of $C_1(\beta)$ is at the nondifferentiable corner $\beta = 0$.

Formally, let $S'(\beta) = a\beta + b$ without loss of generality, where $a > 0$ is guaranteed because $S(\beta)$ is convex. Suppose also that $\mathbf{y}$ has a nonzero correlation with $\mathbf{x}$, so $b \neq 0$ (*i.e.*, $\beta = 0$ is no tthe solution to $S'(\beta) = 0$). Then

$$C_1'(\beta) = a\beta + b + \lambda \frac{|\beta|}{\beta}.$$

We aim to show that for sufficiently large $\lambda$, $C_1'(\beta) = 0$ has no solution. Setting everything equal to 0, we obtain

$$a\beta + b + \lambda|\beta|/\beta = 0$$

Suppose that we have a solution where $\beta > 0$, meaning that $a\beta + b + \lambda = 0$. Rearranging, we obtain $\beta = -b/a - \lambda/a$. Since $a$ is guaranteed to be positive, increasing $\lambda$ will decrease the value of the entire expression; indeed, for $\lambda > -b$ we obtain $\beta < 0$, a contradiction.

Similarly, suppose that we have a solution where $\beta < 0$, meaning that $a\beta + b - \lambda = 0$. Rearranging, we obtain $\beta = \lambda/a - b/a$, and for $L > b$ we obtain $\beta > 0$, a contradiction.

As such, for $\lambda > |b|$, where the right hand side is purely a function of $\mathbf{y}$ and $\mathbf{x}$, the only possible global minimum of $C_1(\beta)$ is at the nondifferentiable cusp $\beta = 0$.

**Therefore:** For sufficiently large $\lambda$, $L^1$ regularization is *guaranteed* to drive coefficient estimates to 0, unless the target variable is completely uncorrelated with its predictors.

Here's an alternative explanation for why $L^1$ regularization drives coefficient estimates to 0. Consider the simpler model where our variables have been appropriately rescaled (to mean 0) and reflected such that the model is just $\mathbf{y} = \beta\mathbf{x}$ for $\beta \geq 0$. Then the $L^1$ regularized cost function is

$$C_1(\beta) = \sum_i (y_i - \beta x_i)^2 + \lambda\beta.$$

Expanding out the sum, we obtain

$$C_1(\beta) = \beta^2 \mathrm{Var}(\mathbf{x}) - 2\beta \mathrm{Cov}(\mathbf{x}, \mathbf{y}) + \mathrm{Var}(\mathbf{y}) + \lambda\beta.$$

Doing a bit of factoring, we arrive at

$$C_1(\beta) = \beta^2 \mathrm{Var}(\mathbf{x}) + \mathrm{Var}(\mathbf{y}) + \beta\left(\lambda - 2\mathrm{Cov}(\mathbf{x}, \mathbf{y})\right).$$

Notice that $C_1(0) = \mathrm{Var}(\mathbf{y})$, a fixed value independent of $\lambda$. Assume that $C_1(\beta)$ is minimized at some value $\beta > 0$ for all $\lambda$. However, this is a contradiction, because if $\beta > 0$ we can increase the value of $C_1(\beta)$ to arbitrarily large values by increasing $\lambda$ and thereby increasing $\beta\left(\lambda - 2\mathrm{Cov}(\mathbf{x}, \mathbf{y})\right)$. It must therefore be the case that for sufficiently large $\lambda$, the only possible minimum of $C_1(\beta)$ is at $\beta = 0$.

## 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 started 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()`. (Since regularization penalizes all of the coefficients in a 'uniform' manner, not weighting some more than others in the calculation of the regularization penalty term, the variables should be scaled so that they are expressed in roughly equivalent units.[1])
- 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 $\lambda$ values to try. To illustrate this, access the object returned by `glmnet()` and print out the values of $\lambda$ 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 $\lambda$ 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 $\lambda$ against $\lambda$ itself.

## Making cross-validated RMSE estimates

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 $\lambda$ is greater than 0.

[1] For example, consider a regression of height against (1) age and (2) testosterone level. The coefficient of age will be much higher if age is expressed in seconds rather than in decades, and correspondingly a regularized model would overly penalize the coefficient on age expressed in seconds relative to what it would do for age expressed in decades.

We can automatically generate cross-validated error estimates for a range of different $\lambda$ values with `cv.glmnet()`. For each value of $\lambda$ tested, `cv.glmnet()` uses $n$-fold cross-validation ($n = 10$ by default) to calculate an error estimate.[2] After running `fit = cv.glmnet(...)`, the value of $\lambda$ (out of those tested) corresponding to the lowest error estimate can be accessed with `fit$lambda.min`. Similarly, the entire range of $\lambda$ 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 $\lambda$ tested. Interpret the results.

## Stepwise regression vs. regularization

Stepwise regression tends to *overfit* because of problems with multiple hypothesis testing. It's easy to understand as an introduction to model comparison and serve as a good stepping stone to regularization, which is why we include it in our curriculum. However, *in practice*, regularizing a linear model is typically strictly superior to selecting a subset of the predictors with a stepwise method.

**Read the entirety of this section before writing any code.**

- Implement **20-fold** cross-validation. Within each fold, you should fit a model predicting attractiveness in terms of the 17 activities (1) with `step()` for backward stepwise regression and (2) with `cv.glmnet()` for both $L^1$ and $L^2$ regularized linear regression. Use the fitted model to make predictions on the held-out fold for all three methods. At the end of the 10 iterations, calculate and return the RMSE corresponding to each method. Interpret the results.

Keep the following in mind while you work:

- After fitting a model with `cv.glmnet()`, be sure to pass `s=fit$lambda.min` to `predict()`, because by default `predict()` does *not* use the value of $\lambda$ which minimizes the cross-validated error.
- Within each cross-validation fold, you'll want to `scale()` the features which you pass into `cv.glmnet()`. However, when making predictions on the *held-out* data, you don't want to simply call `scale()` on that data directly; instead, you want to apply the *same transformations* which were applied to the training set. To do so, (1) extract the scaling parameters from the scaled training data (stored as attributes) and (2) pass them in as additional parameters to `scale()` when calling it on the test set.

[2]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.

## 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.[3] 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 $\lambda$, we now must find the optimal *pair* of hyperparameters $(\alpha, \lambda)$ (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 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 $(\alpha, \lambda)$ 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,
                  preProc=c("center", "scale"))

# 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 $(\alpha, \lambda)$ 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 $(\alpha, \lambda)$ is the one corresponding to the lowest cross-validated RMSE.

- Use the `caret` package, following the above example, to find the optimal values for $(\alpha, \lambda)$ when predicting attractiveness ratings with elastic net regularization. Extract the minimum RMSE value obtained from the resulting

[3]Read the official documentation for `glmnet` to figure out how the $\alpha$ parameter works..

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.

As a closing note, keep in mind that the technique of regularization is not *specific* to linear regression. Indeed, the penalization of the cost function with a regularization term can be applied to a very wide range of machine learning techniques (some of which will be covered in the future assignments), resulting in methods such as sparse PCA and penalized SVMs.

# 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 $\lambda$

Ordinarily, one might expect that, for every different value of $\lambda$ 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 $\lambda$ values simultaneously.[4]

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* $\lambda$ 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 $\lambda$ were used by `glmnet`.

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

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

Finally, `cv.glmnet()` will use *cross-validation* to determine `fit$lambda.min` and `fit$lambda.1se`. The former is the value of $\lambda$ (out of all those the algorithm evaluated) which minimizes the cross-validated mean squared error (MSE), and the latter is the greatest value of $\lambda$ (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 $\lambda$ lies at either end of the range of $\lambda$ values used by `glmnet`, then you'll want to modify the range of $\lambda$. 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`.[5] Nevertheless, there are times when passing in a single value makes sense, like when you've previously determined the optimal $\lambda$ and want to just use that instead of a range of different $\lambda$ values.

[5] "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."