# Linear Regresson: Regularization

Signal Data Science

Some helpful notes on the `glmnet` package are at the end of this document. Look there if you have problems.

## Exploring regularization with simulated data

Before using regularized linear regression on real data, we'll explore it in a simpler context with some 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
  - An estimate of the value of `a`, `aEst`
  - A regularization parameter `lambda`
  - A number `p = 1` or `2`, indicating whether $L^1$ or $L^2$ regularization is being performed

  and returns the value of the $L^p$ regularized cost function at `y = aEst*x`.
- Create a dataframe with two columns, one corresponding to the values of $\lambda$ $2^{-8}, 2^{-7}, \ldots, 2^{-1}, 2^0, 2^1$, and for each of these values of $\lambda$, values of `a` from -0.1 to 0.3, in equally spaced increments of 0.001. Use `expand.grid()` to fill in the grid.
- Add `"costL1"` and `"costL2"` columns, where we'll store the cost of associated with each pair `(lambda, a)`, for each of `p = 1` and `p = 2`.

For each of `p = 1` and `p = 2`,

- Use `lapply()` to make a `plots` list with 10 `ggplot()` objects, one for each value of lambda from $2^{-8}$ to $2^{1}$, graphing values of `a` on the abscissa (x-axis) and values of the cost function on the ordinate (y-axis). Then use multiplot with `plotlist = "plots"` to display these graphs in 2 columns of 5. Pay special attention to the values on the y-axis, which vary from plot to plot.

# Comparing regularization and stepwise regression

Install the `glmnet` package.

We'll continue using the simplified speed dating dataset from yesterday. For now, please restrict analyzing *attractiveness ratings* (`"attr_o"`) for *males*.

## Getting acquainted with `glmnet`

The `glmnet()` function 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). Set `alpha=1` for $L^1$ regularization and set `alpha=0` for $L^2$ regularization.

- Use backward stepwise regression to generate attractiveness predictions for the whole dataset. (Don't use cross-validation at this point.)
- Use `glmnet()` to generate similar predictions with both $L^1$ and $L^2$ regularized linear regression.
  - Look at which values of $\lambda$ were used by `glmnet()`.
  - Write a function that (1) takes the model object generated by a call to `glmnet()` and the true values for the target variable, (2) uses `predict()` to generate predictions for every value of $\lambda$ which `glmnet()` tried, and (3) returns the $\lambda$ corresponding to the lowest RMSE and the RMSE itself.
- Compare the minimum RMSE for both regularized fits with the RMSE for backward stepwise regression.
- Compare and interpret the coefficients for $L^1$ and $L^2$ regularized linear regression using various values of $\lambda$.
- For both $L^1$ and $L^2$ regularized linear regression, plot the RMSE as a function of $\lambda$.

The `cv.glmnet()` function uses cross-validation to determine the optimal value of $\lambda$; after getting a model `fit = cv.glmnet(...)`, the best $\lambda$ can be accessed with `fit$lambda.min`. (See the notes on `glmnet` below.)

- Use `cv.glmnet()` and compare its determination of the optimal $\lambda$ values with your own choices of $\lambda$.

## 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, it's easy to understand and, pedagogically, a good stepping stone to regularization, which is why we include it in our curriculum.)

Write a function following these specifications:

- 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 $\lambda$, and use that to make predictions with `predict(fit, test_data, s=fit$lambda.min)`.
- Return 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", "~.")`), call `formula()` on the string to turn it into a *formula*, and then passing the formula into `lm()`.
- You should be running stepwise regression for each training fold separately.

Use your function to 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 residuals 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.[1] We can use cross-validation to find the optimal *pair* of *hyperparameters* $(\alpha, \lambda)$.

Thankfully, we won't have to implement that ourselves (for now)! Instead, we can use the `caret` package to get a cross-validated estimate of the optimal $(\alpha, \lambda)$.

Here's an example of how to use the `caret` package's `train()` function:

```
param_grid = expand.grid(.alpha = 1:10 * 0.1,
                         .lambda = 10^seq(-3, 0, length.out=10))
control = trainControl(method="repeatedcv", number=10,
                       repeats=3, verboseIter=TRUE)
caret_fit = train(x=features, y=target, method="glmnet",
                  tuneGrid=param_grid, trControl=control)
```

In the above example, we perform *10-fold cross-validation* repeated *3 times*.

Whereas `cv.glmnet()` picks an appropriate sequence of $\lambda$ values, the `caret` package does not.

- Run `cv.glmnet()` on the data for any rating to get a rough sense for what the range of $\lambda$ should be, and then use that when doing grid search with `caret`.

Write a function according to the following specifications:

- Use the `caret` package, following the above example, to find the optimal values for $(\alpha, \lambda)$ when predicting attractiveness ratings.

Finally:

- Calculate the corresponding RMSE and compare the different RMSEs for all combinations of (gender, rating).
- Now that you've gotten some experience with `glmnet()` and `cv.glmnet()`, go back and modify your cross-validation function so that you can pass in (1) any of the five rating variables and (2) a gender variable to filter by instead of restricting solely to attractiveness ratings for males.

[1] Read the official documentation to figure out it works.

## 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.[2]

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)`.

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`

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

and `lambda.min.ratio`.[3] 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.

[3] "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."