Linear Regression: Resampling

Signal Data Science

We'll be covering **resampling** in this lecture using aggregated data from a famous speed dating dataset. The dataset is described by the authors as such:

> *Subjects*—Our subjects were drawn from students in graduate and professional schools at Columbia University. Participants were recruited through a combination of mass e-mail and fliers posted throughout the campus and handed out by research assistants. [. . .]
>
> *Setting*—The Speed Dating events were conducted in an enclosed room within a popular bar/restaurant near the campus. The table arrangement, lighting, and type and volume of music played were held constant across events. Rows of small square tables were arranged with one chair on either side of each table.
>
> *Procedure*—The events were conducted over weekday evenings during 2002–2004; data from fourteen of these sessions are utilized in this study. In general, two sessions were run in a given evening, with participants randomly distributed between them. Participants were not aware of the number of partners they would be meeting at the Speed Dating event. [. . .] Upon checking in, each participant was given a clipboard, a pen, and a nametag on which only his or her ID number was written. Each clipboard included a scorecard with a cover over it so that participants' responses would remain confidential. The scorecard was divided into columns in which participants indicated the ID number of each person they met. Participants would then circle "yes" or "no" under the ID number to indicate whether they would like to see the other person again. Beneath the Yes/No decision was a listing of the six attributes on which the participant was to rate his or her partner: Attractive, Sincere; Intelligent; Fun; Ambitious; Shared Interests.

In the `speed-dating-simple` dataset, `speed-dating-simple.csv` has the data and `speed-dating-info.txt` has an explanation of the variables; the data has been aggregated on the level of each person being rated, so each row corresponds to a unique participant in the speed dating event and contains information about (1) the average ratings they received from others on 5 different scales and (2) their own self-ratings of interest in 17 different activities.

We will be using the speed dating dataset to explore the problem of overfitting, which is:

> a problem where a functional form or algorithm performs substantially better on the data used to train it than on new data drawn from the same distribution. It occurs when the parameters used to describe the functional form end up fitting the noise or random fluctuations in the training data rather than the attributes that are common between the training data and test data.

To illustrate a straightforward example of overfitting, consider the following problem:

- Given n data points and a model $a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$ where we fit the coefficients a_i to the data, how large does n need to be before we can come up with a model that goes through every data point precisely? (Think about the linear case, where only a_0 and a_1 are nonzero.)

In general, with an excess of parameters, we run the risk of them being fit to non-generalizable aspects of our data, such as random noise and fluctuations, which may fool us into thinking that our model is better than it really is. However, if we simply train a model on all of our data and evaluate the quality of its fit on the *same* data, we won't be able to detect such problems. As such, we want to use *resampling* techniques, most of which involve splitting data into *train* and *test* sets. Instead of training the model on the entire dataset, we'll in general train it on a *subset* of the data and estimate the quality of the model against the data which was *not* fed into the model, in order to approximate how well the model would perform on *new* data.

Random number generation in R

We'll begin with a brief discussion of the random number generator (RNG) in R.

The RNG can be *seeded* with `set.seed(n)` for any integer `n`. The values that the RNG outputs will depend on its seed, and setting the seed "resets" it to the initial state corresponding to that particular seed.

- Try using `set.seed()` in conjunction with `runif(5)` to get a sense of how this works.

This is important because we may want, *e.g.*, to generate the same random partitioning of our data consistently, in which case we would put a `set.seed(1)` call before our shuffling of the indices with `sample()`.[1] Or, alternatively, you may want a sequence of reproducibly different calls of `sample()`, etc.

[1]In some cases, you may want to (1) save the state of the RNG for later, (2) set the seed to something specific and generate a consistent splitting of the data, and (3) change the RNG back to its saved state. This is possible using `.Random.seed` and is described in Cookbook for R – we won't need this for this lesson, but it's important to be aware of (as it will eventually surely come up).

A single train/test split

We'll begin by using a *single* train/test split on the data.

- Load the speed dating dataset (using `read.csv()`) and filter the dataset for the gender of your choice.
- Run a linear regression of attractiveness against the 17 self-ratings of activity participation. Interpret the coefficients.
- Write a function `split_data(df)` that splits the data randomly into a train set and a test set of equal size. (An easy way to do this is to call `sample()` to shuffle the row indices of the data and then to use the `%%` operator, taking the remainder upon division by 2, to assign each row to 0 or 1.) Your function should return a *named list* so that `split_data(df)$train` yields the train set and `split_data(df)$test` yields the test set.
- Write a function `split_predict(train, test)` that trains a linear model on the train set to predict *attractiveness* from the 17 *activities*, and uses `predict(model, data)` to generate predictions for both the train set and the test set. Your function should return a *named list* so that `split_predict(df)$train` yields the predictions on the train set and `split_predict(df)$test` yields the predictions on the test set.
- Write a function `rmse(x, y)` to calculate the root-mean-square error between `x` and `y`.
- Run `split_data()` and `split_predict()` 100 times to generate predictions for 100 different train/test splits. For each prediction, calculate the associated RMSE against the true values. Plot the distribution of RMSEs for both the train set and the test set. Calculate their mean and standard error (the standard deviation divided by $sqrt(n)$ for n samples). In general, how does the performance on the train set compare to the performance on the test set?

n-fold cross validation

n-fold cross validation has two advantages over making a single train/test split:

1. In n-fold cross validation, we fit multiple models to multiple train/test splits and then aggregate the results. Between all of these different models, each observation in the dataset ends up in a training set at least once.
2. With a single train/test split, the outcome can depend too much on our choice of RNG seed, which determines whether a particular observation ends up in the train set or the test set. With n-fold cross validation, this problem is somewhat reduced – the outcome is more consistent and stable.

The process is as follows:

1. We randomly split the data into *n* different, equally-sized subsets.
2. For each of the *n* subsets, we train the model against *all the other subsets* and use that model to make predictions on our held-out subset.
3. We combine all of the predictions and calculate a measure of model quality such as the RMSE.

We'll be continuing with predicting attractiveness from activities. Now, choose one of two different approaches:

- Write a function `nfold_cv(df, n_folds)` that splits `df` into `n_folds` folds, generates predictions in the fashion described above, and calculates the RMSE on the whole dataset with those predictions. It should return the RMSE at the end.

- Run `nfold_cv()` 100 times for both 2-fold and 10-fold cross-validation. Plot the distribution of the associated RMSEs. Also, calculate their means and standard errors.

- Compare the results of a single train/test split, 2-fold cross validation, and 10-fold cross validation with regard to getting an accurate measure of model quality.

Stepwise regression

Now that we have a way to evaluate model quality – with cross-validated RMSE – we can use this metric of model quality to choose between *different models*. After all, we need not include *every* available variable into our linear model; well-chosen omissions can improve model performance on a test set.

In *backward stepwise regression*, we do the following:

1. We start with every predictor variable added to the model.
2. Then, we iterate, at each step removing the variable which adds the least to the model.
3. We eventually reach a stopping point based on some statistical criteria.

Let's find out how much improvement we can get in our model by using this method.

- Write your own implementation of backward stepwise regression as a function `backward_step(df)` which follows these criteria:

  – It begins with the full dataset – the attractiveness ratings, which we're trying to predict, as well as the 17 activity scores.

  – The function should repeatedly iterate. On each iteration, it should:

    * Use *10-fold cross validation* via `nfold_cv()` to calculate a cross-validated RMSE for the current model, measuring the quality of how well the currently selected variables can predict attractiveness.
    * Fit a linear model for attractiveness with the entire dataset (minus any variables which have previously been removed).
    * Remove the variable associated with the coefficient which has the *highest* p-value. (If a linear model is stored in `fit`, you can access data about the coefficients by looking at `summary(fit)$coefficients`.)
  - The function should store the cross-validated estimates of RMSE in association with the number of features moved at each step. At the end, it should return both.
- Run `backward_step()` and plot the number of variables removed against the cross-validated RMSE. Interpret the results.

Using R's `step()`

We'll finish off by using R's built-in stepwise regression function, `step()`. You can treat its functionality as a black box, but:

- Spend a minute or two skimming the official documentation for `step()`, especially the *Details* section.

By default, `step()` uses the Akaike information criterion as a measure of model quality – both for determining which variable to add or remove and for determining when to stop – which you can also treat as a black box. (Just know that *a lower AIC is better*.)

Here's an example of backward stepwise regression:

```
model_init = lm(col ~ ., df)
model = formula(lm(col ~ ., df))
step_reg = step(model_init, model, direction="backward")
```

Now, your turn:

- For *each of the five rating variables* (attractiveness, sincerity, intelligence, fun, ambition), use `step()` to run backward stepwise regression (with the `direction="backward"` parameter).
- Store the coefficients of the final linear model into a data frame. Interpret the results.
- Repeat the above for the other gender (the opposite of the one you selected at the beginning). Interpret the differences.