Resampling

We'll be covering **resampling** in this lecture using aggregated data from Andrew Gelman's speed dating dataset.

Check `speedDatingSimple.csv` for the data and `speed-dating-info.txt` for an explanation of the variables; the data has been aggregated on the level of each person being rated, so each row corresponds to a particular person's average ratings by their partners (on five different scales) as well as their self-ratings of interest in 17 different activities.

In machine learning in general, we have the problem of overfitting, where:

> 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.

Exercise. Given n data points and a model $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ 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.)

We run the risk of the parameters of our model 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. We also want a better measure of model quality in general, because estimates of model quality tend to be overly optimistic compared to performance on new data. 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.

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.

A single train/test split

- Load the speed dating dataset (using `read.csv()`) and restrict to a gender of your choice.
- Run a linear regression of attractiveness against the 17 activities. Interpret the coefficients.
- Write a function that splits the data randomly into a train set and a test set (of equal sizes), trains a linear model on the train set to predict *attractiveness* from the 17 *activities*, and uses `predict(model, newdata)` to generate predictions for the test set.
  - 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.
  - Run the function 100 times to generate predictions for many different train/test splits.
  - For each prediction, calculate the associated R^2 (just square the `cor()`relation).[2]
  - Plot the distribution of R^2 values and calculate their 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?

[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).

[2]Technically, the square of the correlation isn't the same as R^2, the percentage of variance explained by the predictors – r^2, the correlation squared, is an *imperfect estimate* of R^2. In general, it's better to use the root mean squared error, but it's fine to just use R^2 for this.

n-fold cross validation

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

- 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.
- 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 goes as follows:

1. We randomly split the data into n different 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 some measure of model quality, like R^2 or root-mean-square error.

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

- Implement a function that uses n-fold cross validation to generate a set of predictions for any n, or
- Implement 2-fold and 10-fold cross validation separately in two different functions.

Either way,

- Make 100 such predictions, with different folds each time, using both 2-fold and 10-fold cross validation.
- As before, plot the distribution of the associated R^2 values and calculate their 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

We can use our methods of evaluating model quality to choose between different models.

In *backward stepwise regression*:

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

To that end, you'll be writing your own implementation of backward stepwise regression.

Write a function that does the following:

- 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* to calculate an R^2 value, measuring the quality of how well the currently selected variables can predict attractiveness.
  - Fit a linear model for attractiveness against the full set of data.
  - 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 R^2 in association with the number of features moved at each step.

Use this function to explore how cross-validated R^2 changes as you remove more and more variables from the model. Plot the results.

Using R's `step()`

We'll finish off by using R's built-in stepwise regression function, `step()`. You can more-or-less treat its functionality as a black box, but spend a minute or two skimming the official documentation. It 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="direction")
```

- 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 between the results.