Linear Regression: Resampling

Signal Data Science

In this assignment, we will be covering basic *resampling* methods. First, we'll learn about the usage of n-fold cross-validation in order to detect and combat overfitting. Afterward, we'll discuss the usage of bootstrapping to quantify uncertainty in our estimates of a dataset's properties.

Overfitting

The notion of *overfitting* is defined as:

> 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 perfectly goes through every data point? (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 to split our 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.

Speed dating dataset

We'll be exploring overfitting and resampling in this lesson using aggregated data from a well-known 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. (Get the data from the GitHub repository, **not** from an external source.)

Random number generation in R

In the problems below, we'll be splitting the data into random subsets. This entails using R's `sample()` which uses a R's random number generator. At any given time, R's random number generator has a state. If you want your results to be reproducible, before using a function that depends on random number generation, you can reset the state using `set.seed(n)` (the new state depends on `n`). To see how this works, try using set.seed() in conjunction with `runif(5)`.

A single train/test split

We'll begin by using a *single* train/test split on the data to evaluate the quality of a linear model. It is the simplest method of resampling and consequently the most straightforward one to implement.

- Load the speed dating dataset (using `read.csv()`) and filter the dataset for males only.
- Run a linear regression of attractiveness against the 17 self-ratings of activity participation. Interpret the coefficients.

Next, we will split the task before us into three different functions. As you write each function, test it to ensure that it works correctly.[1]

- Write a function `split_data(df)` that splits the data randomly into a train set and a test set of equal size. (One 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`. We can use the RMSE to evaluate the quality of our predictions on the test set.

Finally, we're ready to see how a linear model performs on the train set versus the test set.

[1] In general, splitting up large tasks into multiple different functions and testing each function individually before combining all of them together is a good strategy for easily catching bugs in your code.

- 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 histograms of the RMSE values for both the train set and the test set on the same graph. Calculate their mean and standard deviation. How does the performance on the train set compare to the performance on the test set?

n-fold cross-validation

Although a single train/test split is *one* way of estimating how well our model would perform on new data, it is not the only or the best way. Indeed, *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.

You'll be implementing the above process in the following problems and comparing the results of *n*-fold cross-validation (for various values of *n*) with the results of a single train/test split.

- 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 distributions of the RMSE values for 2-fold and 10-fold cross-validation on the same graph. Calculate the means and standard deviations of the distributions and compare the results to those for the distribution of RMSE values for the test set when doing a single train/test split.

Stepwise linear regression

Now that we have a way to evaluate model quality – with the 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 substantially improve model performance on a test set. Stepwise regression is one of many ways to choose which variables to include (a task called feature selection).

In *backward stepwise regression* (one type of normal 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.

Implementing backward stepwise regression

Let's find out how much we can improve on a linear regression which includes every variable by using backward stepwise regression.

- Write your own implementation of backward stepwise regression as a function `backward_step(df)` which follows these criteria:
  - Begin with the full dataset – the attractiveness ratings, which we're trying to predict, as well as the 17 activity scores. As before, *don't* include gender or the other 4 ratings.
  - Initialize three empty vectors: `n_removed`, `rmse_cv`, and `rmse_nocv`.
  - Repeatedly iterate until the data frame has fewer than 2 columns left. On each iteration, it should:
    - Append the number of variables removed so far to `n_removed`.
    - 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. Append this value to `rmse_cv`.
    - Fit a linear model for attractiveness with the entire dataset (minus any variables which have previously been removed).
    - Use the linear model to make predictions on the whole dataset and calculate the associated, *non-cross-validated* RMSE. Append this value to `rmse_nocv`.
    - 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`.)

  - At the end, the function should combine `n_removed`, `rmse_cv`, and `rmse_nocv` together into a data frame with three columns and return that data frame.

- Run `backward_step()`. On the same graph, plot both the cross-validated RMSE estimate and the non-cross-validated RMSE against the number of features removed. Interpret the results. Is there any evidence of overfitting when using all 17 predictors in our linear model?

Using R's `step()`

We'll finish off by using R's built-in stepwise regression function, `step()`, to run backward stepwise regression on all five average ratings. Instead of using a cross-validated RMSE as a metric of model quality, it uses the Akaike information criterion (AIC). You can treat the AIC as a black box – just know that a lower AIC indicates a better model.

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

Here's an example of how to use backward stepwise regression with `step()`:

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

In the following, you will use stepwise regression on *all five* of the average rating variables.

- For *each of the five rating variables* (attractiveness, sincerity, intelligence, fun, and ambition), use `step()` to run backward stepwise regression (with the `direction="backward"` parameter) to predict that rating in terms of the 17 activity ratings. Store the coefficients of the final model for each of the rating variables in a list. Interpret the differences between them.

Note that if you have a column name stored in a *variable*, *e.g.* `var = col_name`, and you would like to regress `col_name` against every other variable in `df`, the call `lm(var ~ ., df)` will *not* work, because `lm()` will look for a column called `var`. Instead, use `paste()` to create a *string* `s = "col_name ~ ."` from `var` and then pass `s` in as the first argument of `lm()`.

Bootstrapping

Aside from n-fold cross-validation, there exists a different resampling technique called *bootstrapping*. In bootstrapping, we take the original dataset and randomly pick rows *with replacement* to form a new dataset with the same total number of rows. It is essentially a way for us to *mimic* the results of obtaining completely new data from the population.

Comparing models

Suppose that we want to use bootstrapping in order to measure model quality. The two simplest ways to do so are to generate a large number of bootstrapped samples and to either (1) train a model on each bootstrapped sample and make predictions on the original dataset or (2) train a model on the original dataset and make predictions on each bootstrapped sample. For each (train, test) pair, we calculate a RMSE; at the end, we take the average of all the calculated RMSE values.

Both of the above methods are fundamentally broken as described. Approach (1) will severely underestimate the RMSE (and consequently the degree of overfitting) because of the high overlap between train and test sets – on average, two-thirds of the rows in the original dataset will show up at least once in each bootstrapped sample. Approach (2) is even worse: if we train a model on the original dataset, it will already have "seen" one copy of every data point, so none of the data in *any* of the test sets will be "new" to the model. This makes approach (2) underestimate the RMSE by an even greater extent *and* be incapable of measuring the degree of overfitting.[2]

It is often illustrative and helpful in building intuition to computationally look at the results of methods which we already know are flawed. To that end:

- Write a function `bootstrap_bad(df, approach)` which takes in a data frame `df` and an integer `approach` equal to 1 or 2. Within the function, generate 100 different bootstrapped samples from `df`. For each of the 100 samples, do the following:
  - Depending on the value of `approach`, set either the train set equal to the bootstrapped sample and the test set equal to `df` or vice versa.
  - Train a linear model for attractiveness in terms of the 17 activities with the train set. Use that model to make predictions on the test set.
  - Calculate the RMSE correspnding to your predictions.

[2]Mathematically speaking, in approach (1) every train set is a subset of the test set and in approach (2) every test set is a subset of the train set.

At the end, calculate and return the average of the 100 calculated RMSE values.

- Copy your code from above for `backward_step()` to make a new function `backward_step_2()`. In addition to the calculations performed in `backward_step()`, call `bootstrap_bad()` twice during each iteration (once for each value of `approach`), store the calculated RMSE values, and return them as well.
- Run `backward_step_2()` and graph the results as before. You should have *four* different series of RMSE values to graph against the number of variables removed. Do the results correspond with the theoretical explanation above?

There is, however, a way to get around these problems. Indeed, we can use approach (1) with a single modification: instead of making predictions on the *entire* original dataset, we only make predictions with the rows which were *not* included in the bootstrapped sample, called the *out-of-bag* data points.

- Write a function `bootstrap_good(df)` which works like `bootstrap_bad()` with `approach=1` *except* with the modification outlined above. Here, you may find `unique()` useful.
- Write a function `backward_step_3()` similar to the other two such functions but using `bootstrap_good()` instead of `bootstrap_bad()`. As before, run it and visualize the results.

Although the modified version of approach (1) certainly works, we can see that it tends to slightly overestimate the RMSE. However, it also has a minor advantage over n-fold cross validation in that its RMSE estimates tend to have lower variance (when compared against commonly used values of n). Bootstrapped estimates can be corrected for the RMSE overestimation, and we can use repeated cross-validation to reduce the variance of cross-validated RMSE estimates, so in practice the differences are quite minor – it's more important to just pick one and use it consistently.

Moving forward, we'll stick to using n-fold cross-validation for measuring model quality. It's conceptually simpler than bootstrapping and is encountered more often.

Estimating parameter distributions

There is, however, a different task for which bootstrapping is very well suited, namely the task of estimating the *variance of parameter estimates*. We'll illustrate with a computational example.

Suppose that you want to invest a fixed sum of money into two different financial assets X and Y. You can invest a fraction α of your money into X and the remaining $1 - \alpha$ into Y. You'll implement a simple strategy: instead of

looking at the returns at all, simply *minimize the risk* (*i.e.*, the variance of the returns). Somewhat counterintuitively, this strategy actually performs well in practice.[3]

We can make the simplifying assumption that the returns of X and Y are uncorrelated. In that case, it can be mathematically shown that the value of α which minimizes the risk is

$$\alpha = \frac{\sigma_Y^2}{\sigma_X^2 + \sigma_Y^2}$$

where σ_X^2 and σ_Y^2 are respectively the *variances* of the returns of X and Y.

We don't know the values of σ_X^2 or σ_Y^2, but given data on the returns of X and Y you can estimate those values and consequently gain an estimate of α, which we denote $\hat{\alpha}$. However, this naturally leads to the question: how variable is our estimate $\hat{\alpha}$? Equivalently, given only a finite dataset and no knowledge about the *true* values of σ_X^2 and σ_Y^2, is it possible for us to quantify how *certain* we are about our estimate $\hat{\alpha}$? The answer is *yes*.

- Write a function `calc_alpha(X, Y)` which takes as inputs equivalently sized vectors containing data about the returns of X and Y and uses the above formula to calculate an estimate for α.
- Write a function `gen_alphas(sdX, sdY)` which takes as input two positive numbers `sdX` and `sdY`. We will suppose that the true returns of the assets X and Y are normally distributed with mean 10 and standard deviation equal to `sdX` and `sdY` respectively. Generate 100 observations of both distributions and store them as `X` and `Y`. Next, generate 1000 bootstrapped samples of both `X` and `Y`. For each pair of bootstrapped samples, calculate and store the corresponding value of α with `calc_alpha()`. Finally, return the list of 1000 estimates of α.
- For each of the following pairs of `(sdX, sdY)` parameters, run `gen_alphas()` to obtain 1000 bootstrapped estimates of α, plot a histogram of them, and calculate their mean and standard deviation: $(1, 3)$, $(3, 1)$, $(1, 1)$. Do these results correspond to what you would expect?
- Run `gen_alphas()` for each of the following `(sdX, sdY)` pairs and plot histograms of the results on the same graph: $(1, 2)$, $(1, 3)$, $(1, 4)$. Interpret the results.

Bootstrapping is very powerful in cases when the parameters to estimate are complex functions of the data, like in the above example. (However, if the parameter is as simple as the mean of the data, it's typically better to use the *t*-test from classical statistics, which will give more precise results with theoretical

[3] See Why does the minimum variance portfolio provide good returns? and Baker *et al.* (2010), Benchmarks as Limits to Arbitrage: Understanding the Low Volatility Anomaly.

justification.) For example, bootstrapping can be used to calculate estimates of standard error for parameters like regression coefficients or to even estimate the stability of clustering algorithms. It is also useful for revealing non-normality in parameter distributions.

We'll seldom implement bootstrapping ourselves, but we'll see it incorporated into a variety of different algorithms in the future.