Cross-Validation and Reliable Performance Estimation

August 3, 2024

Table of contents

```r
library(tidyverse)
library(tidymodels)
library(palmerpenguins)
library(patchwork)
library(kableExtra)

tidymodels_prefer()

penguins <- palmerpenguins::penguins

options(kable_styling_bootstrap_options = c("hover", "striped"))

theme_set(theme_bw(base_size = 14))

set.seed(123)
penguins_split <- initial_split(penguins)
penguins_train <- training(penguins_split)
penguins_test <- testing(penguins_split)
```

Recap

In the previous notebook, we saw that adding flexibility to a model improves its ability to approximate complex relationships. Additional flexibility also increases the model's ability to fit the training data – that is, more flexible models will generally have lower training error than less flexible models. Because training error continues to drop with additional flexibility, we need an additional data source which was unseen by the model during the training process in order to provide an unbiased estimate of future model performance. Ideally, this test error will decrease until the optimal level of model flexibility is reached but then the test error will increase once the model becomes *overfit* to the training data.

At the end of the last notebook, we drew an *elbow plot*, showing how the *training* and *test errors* changed as model flexibility was increased. This visual technique allowed us to identify which models were likely *underfit* and what the threshold for an *overfit* model was. We can use these elbow plots in general, to identify an appropriate level of flexibility for our models.

Motivation

We've placed a lot of faith in our *training* and *test* sets over the last few weeks. We've given the training data total power to determine our model coefficients and the test data sole power to determine our expected performance metrics (R-Squared, RMSE, etc.). Perhaps we should have some concern here – especially given our most recent discussion about model flexibility and its relationship to *variance*. Different training data would result in differently estimated coefficients and different testing data would result in differently estimated performance estimates – do we really want to believe that our random data splitting resulted in perfectly fair and representative training and test data? It is certainly possible that we, by chance, would generate a particularly "easy" training set and a particularly "difficult" test set (or vice-versa). Furthermore, if we are in the business of constructing models often, such an occurrence is guaranteed eventually!

To highlight how much influence our training and test sets can have on our models and the expectations we develop about them, each plot below corresponds to the RMSE for models of degree 1, 2, 3, and 5 for a single data set but using a different training/testing splits. To produce the plots below, I've generated a toy dataset and am constructing/analyzing the performance of a variety of models (linear, quadratic, cubic, fifth-degree) for 10 different training/test set combinations below. Interpret the elbow plots to see how the estimated performance metrics change for each different training and test set.

```r
set.seed(345)
num_points <- 100
x <- runif(num_points, 0, 50)
y <- (x - 20)*(x - 40) + rnorm(num_points, 0,150)
toy_data <- tibble(x = x, y = y)
```

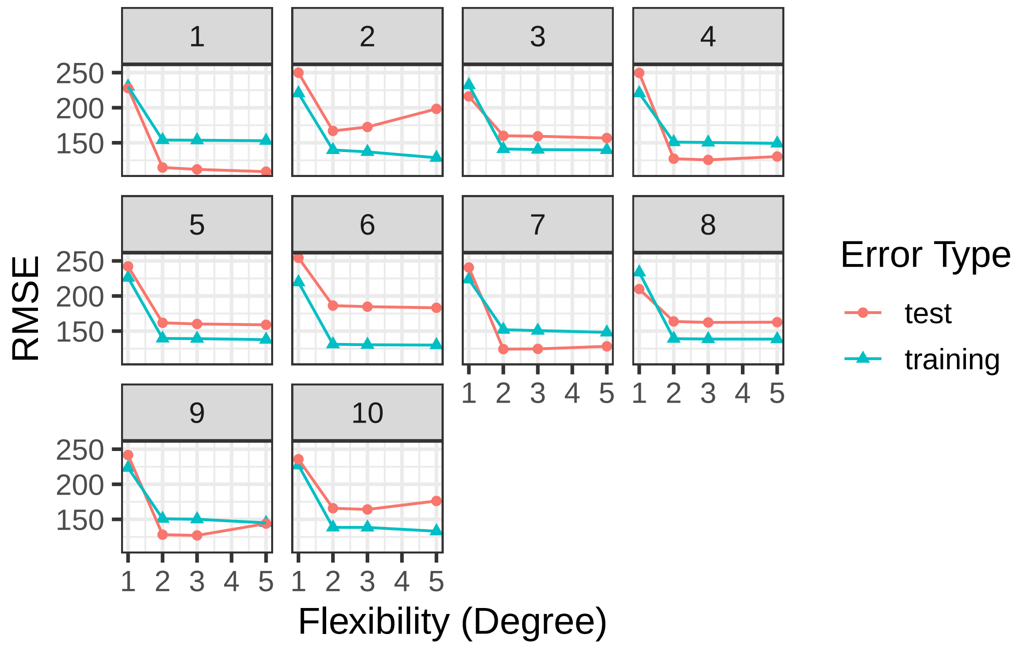
```r
#ggplot(toy_data) +
#  geom_point(aes(x = x, y = y)) +
#  labs(title = "Toy Data")
my_order <- c(1, 2, 3, 5)

collected_metrics_df <- tibble(.metric = NA,
                               .estimator = NA,
                               .estimate = NA,
                               flexibility = NA,
                               trial = NA,
                               type = NA)

for(i in 1:10){
  toy_splits <- initial_split(toy_data)
  toy_train <- training(toy_splits)
  toy_test <- testing(toy_splits)

  for(deg in my_order){
    lr_spec <- linear_reg() %>%
      set_engine("lm")
    lr_rec <- recipe(y ~ x, data = toy_train) %>%
      step_poly(x, degree = deg, options = list(raw = TRUE))
    lr_wf <- workflow() %>%
      add_model(lr_spec) %>%
      add_recipe(lr_rec)
    lr_fit <- lr_wf %>%
      fit(toy_train)
    pred_col_name <- paste0("degree_", deg, "_trial_", i)
    toy_train <- lr_fit %>%
      augment(toy_train) %>%
      rename(!!pred_col_name := .pred)
    toy_test <- lr_fit %>%
      augment(toy_test) %>%
      rename(!!pred_col_name := .pred)
    my_metrics <- metric_set(rsq, rmse)
    collected_metrics_df <- collected_metrics_df %>%
      bind_rows(
        (toy_train %>%
           my_metrics(y, !!pred_col_name) %>%
           mutate(flexibility = deg,
                  trial = i,
                  type = "training")
        )
      ) %>%
      bind_rows(
        (toy_test %>%
           my_metrics(y, !!pred_col_name) %>%
           mutate(flexibility = deg,
                  trial = i,
                  type = "test")
        )
      )
  }
}

collected_metrics_df %>%
  filter(!is.na(.metric)) %>%
  pivot_wider(id_cols = c(flexibility, trial, type),
              names_from = .metric, values_from = .estimate) %>%
  ggplot() +
  geom_point(aes(x = flexibility, y = rmse, shape = type, color = type)) +
  geom_line(aes(x = flexibility, y = rmse, color = type)) +
  labs(x = "Flexibility (Degree)",
       y = "RMSE",
       color = "Error Type",
       shape = "Error Type") +
  facet_wrap(~trial)
```



```
collected_metrics_df %>%
  filter(!is.na(.metric),
         flexibility == 2,
         type == "test") %>%
  pivot_wider(id_cols = c(flexibility, trial, type),
              names_from = .metric, values_from = .estimate) %>%
  kable() %>%
  kable_styling()
```

flexibility	trial	type	rsq	rmse
2	1	test	0.6593570	114.7737
2	2	test	0.7253872	166.9643
2	3	test	0.6374607	160.0023
2	4	test	0.8284092	127.3285
2	5	test	0.4600495	161.7844
2	6	test	0.6501866	186.2976
2	7	test	0.8038373	124.2833
2	8	test	0.6308369	163.7026
2	9	test	0.8273417	128.1685
2	10	test	0.6708070	165.8269

The elbow plots pretty reliably identify the appropriate level of model flexibility. In the toy data set I generated, the true association between x and y was quadratic. However, the *test RMSE*, which we utilize in estimating the "accuracy" of our model's predictions fluctuates quite wildly – between a low of below 115 for the first training/test set combination, and a high of nearly 190 for the sixth training/test combination. There is a big difference between claiming our model's predictions are accurate to within ± 230 units and within ± 380 units. These discrepancies should be worrisome. We are putting lots of trust in our models, so it would be nice to have more certainty about our estimated coefficients, predictions, and error/performance estimates.

Objectives

The discussions above have highlighted that, while we thought we were doing the right things with our training and test set approach all along, we were actually very vulnerable to the random data that – by chance – fell into our training set and our test set. Different splits can lead to different models and certainly different model performance expectations. In this notebook, we'll develop a robust framework which will help us obtain more reliable performance estimates *and* help us estimate how uncertain we are in those estimates.

After working through this notebook you should be able to:

- Describe the pitfalls associated with the *validation set approach* we've been using up to this point in our course.
- Describe how *cross-validation* works as well as how it mitigates the issues associated with a single validation set.
- Implement *cross-validation* using `vfold_cv()` in conjunction with `fit_resamples()` from `{tidymodels}` to fit and assess models using cross-validation.
- Observe and interpret individual performance metrics on each cross-validation *fold* as well as to aggregate those metrics to a more stable performance estimate along with an estimate for our uncertainty in this metric.

Cross-Validation

In cross-validation, rather thank breaking our data set into a single training and test set, we'll break our data into k folds, were k is somewhere between 3 and n (your total number of observations). Each of the folds will take one turn playing the role of the *test* set for a model fit on all of the other folds – that is, with cross-validation, we'll fit k models and get k performance estimates. Once we have those performance estimates, they can be averaged and we can compute the standard error corresponding to the performance metric. This provides us (i) more stable and reliable model performance estimates, and (ii) a way to quantify the uncertainty associated with those metrics.

The most commonly utilized number of folds are 5 and 10. In the special case where you utilize n folds, each observation belongs to its own fold – this is often called *Leave-One-Out*

Cross-Validation – it is very computationally expensive. The greater the number of folds, the more models need to be fit.

Implementing Cross-Validation in `{tidymodels}`

It's been a bit since we worked with the `palmerpenguins` data, but let's bring that data set back. If you remember, we built several linear regression models using that data set. Our most successful model was a model which included [DESCRIBE MODEL HERE] and which had an estimated test RMSE of [INSERT TEST RMSE HERE]. For several of the models we built with the penguins data, our test RMSE was actually lower than our training RMSE. In this case, random splitting had obtained an *easy* test set for us. Knowing what we do now, we should be skeptical of using that test RMSE number to estimate future model performance. Cross-validation is a way to protect against just getting an "easy" or "difficult" test set by chance.

We'll update our workflow as follows:

1. Split data into training and validation sets using `initial_split()`, `training()`, and `testing()` as usual, but with a larger proportion of data belonging to the "training" set. We'll use `prop = 0.9` for the penguins data – this will leave about 34 penguins in our holdout set.
   - We'll reserve this smaller *test* set as a final sanity check for our model before we "push it to production".
2. We'll use `vfold_cv()` to split the *training* data into *cross-validation folds.*
3. We'll build our model(s) as usual (create a model specification, build a recipe, package the model and recipe into a workflow) and then use `fit_resamples()` rather than `fit()` to fit and assess our model on our cross-validation folds.
4. Once we've fit our models on the folds (the resamples), we'll collect the performance metrics on them using `collect_metrics()`.

We'll take care of the first three steps below.

```r
penguins_split <- initial_split(penguins, prop = 0.9)
penguins_train <- training(penguins_split)
penguins_test <- testing(penguins_split)

penguins_folds <- vfold_cv(penguins_train)

lr_spec <- linear_reg() %>%
  set_engine("lm")
```

```r
lr_rec <- recipe(body_mass_g ~ ., data = penguins_train) %>%
  step_dummy(species) %>%
  step_dummy(island) %>%
  step_interact(~ starts_with("species"):contains("length")) %>%
  step_interact(~ bill_length_mm:bill_depth_mm)

lr_wf <- workflow() %>%
  add_model(lr_spec) %>%
  add_recipe(lr_rec)

lr_results <- lr_wf %>%
  fit_resamples(penguins_folds)
```

Now that we've fit our model, we can extract the results from Cross-Validation. There are two types of result we can get back – individual results from each fold:

```r
lr_results %>%
  collect_metrics(summarize = FALSE) %>%
  pivot_wider(id_cols = id, names_from = .metric, values_from = .estimate) %>%
  kable() %>%
  kable_styling()
```

id	rmse	rsq
Fold01	362.4834	0.7540292
Fold02	274.2758	0.8794673
Fold03	302.1292	0.7981302
Fold04	334.4922	0.7574754
Fold05	242.2274	0.9157606
Fold06	261.8536	0.9053921
Fold07	312.0557	0.9015039
Fold08	296.0174	0.8761237
Fold09	241.9070	0.9119503
Fold10	248.9867	0.9231618

or a single set of metrics summarized across all of the folds:

```r
lr_results %>%
  collect_metrics() %>%
  kable() %>%
  kable_styling()
```

.metric	.estimator	mean	n	std_err	.config
rmse	standard	287.6428273	10	12.980473	Preprocessor1_Model1
rsq	standard	0.8622995	10	0.021012	Preprocessor1_Model1

Seeing the results on the individual folds gives us an idea about how our performance metrics fluctuate from one set to the next. Are the performance metrics relatively stable or do they vary wildly from one fold to the next? Seeing the aggregated results gives us a more trustworthy estimate for each of our performance metrics and also reports a standard error for each metric. That standard error measures the average fluctuation in the performance estimate from one fold to the next.

In summary, we can be confident that the R-Squared value for this model is around 86.2%$\pm$4%. Similarly, we can be confident that the predictions our model makes will be accurate to within about $\pm 2\,(287.64 \pm 2 \cdot 12.98)$ grams. Check with the other students in the class – we should all have similar estimates for the Cross-Validation RMSE, even when we don't set the same seed. This was not the case when we used the *validation-set approach*.

Summary

In this notebook we saw that different test sets result in different model performance estimates. In fact, those model performance estimates can vary quite wildly from one choice of test set to another. For this reason, we've developed the notion of *cross-validation*, a powerful tool for producing stable and more reliable performance estimates than what we get from the *validation-set approach* alone.

In *cross-validation*, our training data is split into several folds. Each fold takes one turn being left out of model training and serves as the validation set for a model fit on the remaining folds. In doing this, we obtain several model performance estimates which can be averaged to obtain a more stable performance estimate along with an estimate for the standard error of that performance estimate.

- To split our training data into cross-validation folds, we use `vfold_cv()` on the training set.
- To assess our model using cross-validation we use `fit_resamples()` in place of the `fit()` function we have been using up to this point.
- We obtain and aggregate the cross-validation performance metrics by calling `collect_metrics()` on our "fitted" model.

Cross-validation is a very powerful tool and we'll continue to use it in a variety of ways throughout the remainder of our course.