Evaluating Model Performance by Building Cross-Validation from Scratch

statworx.com/de/blog/evaluating-model-performance-by-building-cross-validation-from-scratch/

2. Oktober 2019

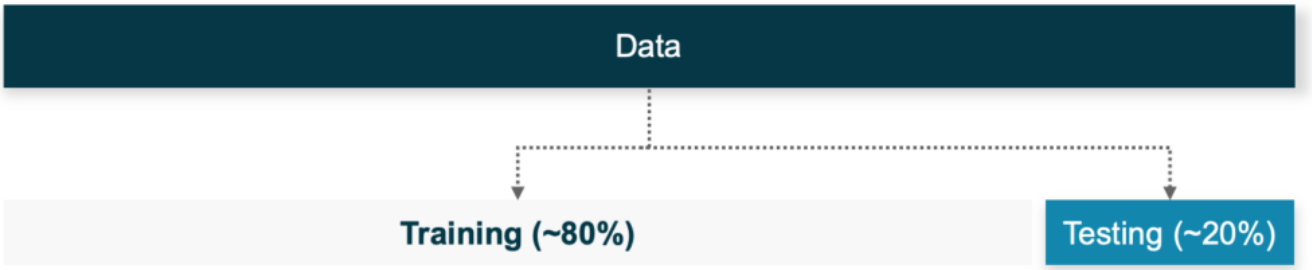
Cross-validation is a widely used technique to assess the generalization performance of a machine learning model. Here at STATWORX, we often discuss performance metrics and how to incorporate them efficiently in our data science workflow. In this blog post, I will introduce the basics of cross-validation, provide guidelines to tweak its parameters, and illustrate how to build it from scratch in an efficient way.

Model evaluation and cross-validation basics

Cross-validation is a model evaluation technique. The central intuition behind model evaluation is to figure out if the trained model is generalizable, that is, whether the predictive power we observe while training is also to be expected on unseen data. We could feed it directly with the data it was developed for, i.e., meant to predict. But then again, there is no way for us to know, or *validate*, whether the predictions are accurate. Naturally, we would want some kind of benchmark of our model's generalization performance before launching it into production. Therefore, the idea is to split the existing training data into an actual training set and a hold-out test partition which is not used for training and serves as the „unseen" data. Since this test partition is, in fact, part of the original training data, we have a full range of „correct" outcomes to validate against. We can then use an appropriate error metric, such as the Root Mean Squared Error (RMSE) or the Mean Absolute Percentage Error (MAPE) to evaluate model performance. However, the applicable evaluation metric has to be chosen with caution as there are pitfalls (as described in this blog post by my colleague Jan). Many machine learning algorithms allow the user to specify hyperparameters, such as the number of neighbors in k-Nearest Neighbors or the number of trees in a Random Forest. Cross-validation can also be leveraged for „tuning" the hyperparameters of a model by comparing the generalization error of different model specifications.

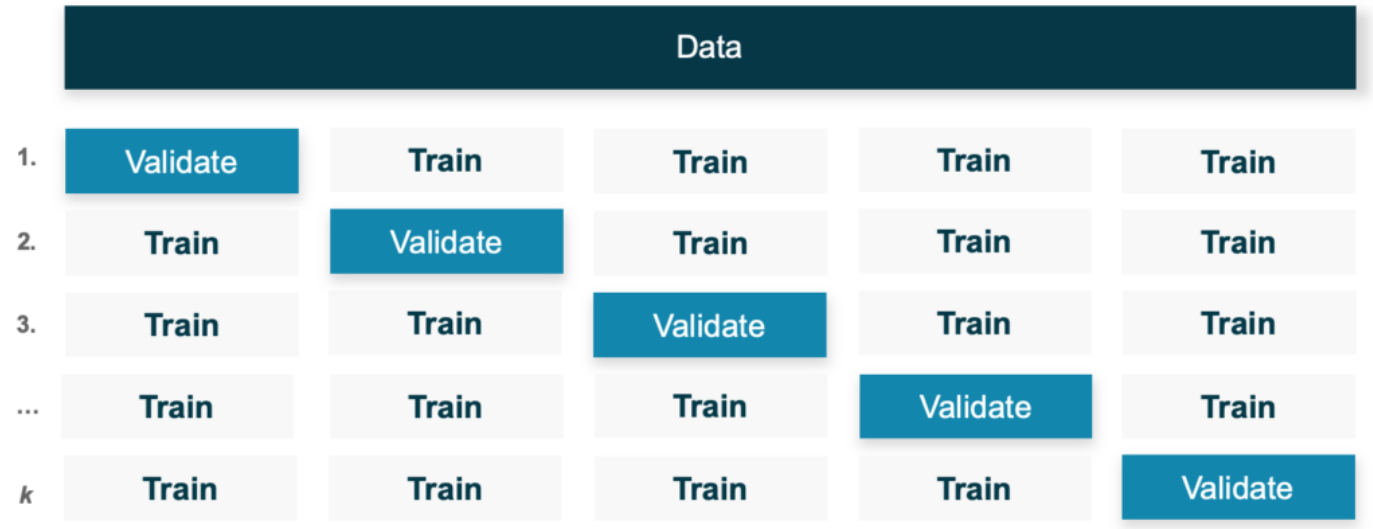
Common approaches to model evaluation

There are dozens of model evaluation techniques that are always trading off between variance, bias, and computation time. It is essential to know these trade-offs when evaluating a model, since choosing the appropriate technique highly depends on the problem and the data we observe. I will cover this topic once I have introduced two of the most common model evaluation techniques: the train-test-split and k-fold cross-validation. In the former, the training data is randomly split into a train and test partition (Figure 1), commonly with a significant part of the data being retained as the training set. Proportions of 70/30 or 80/20 are the most frequently used in the literature, though the exact ratio depends on the size of your data. The drawback of this approach is that this one-time random split can end up partitioning the data into two very imbalanced parts, thus yielding biased generalization error estimates. That is especially critical if you only have limited data, as some features or patterns could end up entirely in the test part. In such a case, the model has no chance to learn them, and you will potentially underestimate its performance.



A more robust alternative is the so-called k-fold cross-validation (Figure 2). Here, the data is shuffled and then randomly partitioned into k folds. The main advantage over the train-test-split approach is that *each* of the k partitions is iteratively used as a test (i.e., validation) set, with the remaining $k-1$ parts serving as the training sets in this iteration. This process is repeated k times, such that every observation is included in both training and test sets. The appropriate error metric is then simply calculated as a mean of all of the k folds, giving the cross-validation error.

This is more of an *extension* of the train-test split rather than a completely new method: That is, the train-test procedure is repeated k times. However, note that even if k is chosen to be as low as $k=2$, i.e., you end up with only two parts. This approach is still superior to the train-test-split in that *both* parts are iteratively chosen for training so that the model has a chance to learn *all* the data rather than just a random subset of it. Therefore, this approach usually results in more robust performance estimates.



Comparing the two figures above, you can see that a train-test split with a ratio of 80/20 is equivalent to *one iteration* of a 5-fold (that is, $k=5$) cross-validation where 4/5 of the data are retained for training, and 1/5 is held out for validation. The crucial difference is that in k-fold the validation set is shifted in each of the k iterations. Note that a k-fold cross-validation is more robust than merely repeating the train-test split k times: In k-fold CV, the partitioning is done *once*, and then you iterate through the folds, whereas in the repeated train-test split, you re-partition the data k times, potentially omitting some data from training.

Repeated CV and LOOCV

There are many flavors of k-fold cross-validation. For instance, you can do „repeated cross-validation" as well. The idea is that, once the data is divided into k folds, this partitioning is fixed for the whole procedure. This way, we're not risking to exclude some portions by chance. In repeated CV, you repeat the process of shuffling and randomly partitioning the data into k folds a certain number of times. You can then average over the resulting cross-validation errors of each run to get a global performance estimate.

Another special case of k-fold cross-validation is „Leave One Out Cross-Validation" (LOOCV), where you set $k=n$. That is, in each iteration, you use a *single* observation from your data as the validation portion and the remaining $n-1$ observations as the training set. While this might sound like a hyper robust version of cross-validation, its usage is generally discouraged for two reasons:

- First, it's usually *very computationally expensive*. For most datasets used in applied machine learning, training your model $n-1$ times is neither desirable nor feasible (although it may be useful for very small datasets).
- Second, even if you had the computational power (and time on your hands) to endure this process, another argument advanced by critics of LOOCV from a statistical point of view is that the resulting cross-validation error can exhibit high variance. The cause of that is that your „validation set" consists of only one observation, and depending on the distribution of your data (and potential outliers), this can vary substantially.

In general, note that the performance of LOOCV is a somewhat controversial topic, both in the scientific literature and the broader machine learning community. Therefore, I encourage you to read up on this debate if you consider using LOOCV for estimating the generalization performance of your model (for example, check out this and related posts on StackExchange). As is often the case, the answer might end up being „it depends". In any case, keep in mind the computational overhead of LOOCV, which is hard to deny (unless you have a tiny dataset).

The value of k and the bias-variance trade-off

If $k = n$ is not (necessarily) the best choice, then how to find an appropriate value for k? It turns out that the answer to this question boils down to the notorious *bias-variance trade-off*. Why is that?

The value for k governs how many folds your data is partitioned into and therefore the size of (i.e., number of observations contained in) each fold. We want to choose k in a way that a sufficiently large portion of our data remains in the training set – after all, we don't want to give too many observations away that could be used to train our model. The higher the value of k, the more observations are included in our training set in each iteration.

For instance, suppose we have 1,200 observations in our dataset, then with $k = 3$ our training set would consist of $\frac{k-1}{k} * N = 800$ observations, but with $k = 8$ it would include 1,050 observations. Naturally, with more observations used for training, you approximate your model's actual performance (as if it were trained on the whole dataset), hence reducing the bias of your error estimate compared to a smaller fraction of the data. But with increasing k, the size of your validation partition decreases, and your error estimate in each iteration is more sensitive to these few data points, potentially increasing its overall variance. Basically, it's choosing between the „extremes" of the train-test-split on the one hand and LOOCV on the other. The figure below schematically (!) illustrates the bias-variance performance and computational overhead of different cross-validation methods.

As a rule of thumb, with higher values for k, bias decreases and variance increases. By convention, values like $k = 5$ or $k = 10$ have been deemed to be a good compromise and have thus become the quasi-standard in most applied machine learning settings.

> „These values have been shown empirically to yield test error rate estimates that suffer neither from excessively high bias nor from very high variance.“
>
> *James et al. 2013: 184*

If you are not particularly concerned with the process of cross-validation itself but rather want to seamlessly integrate it into your data science workflow (which I highly recommend!), you should be fine choosing either of these values for k and leave it at that.

Implementing cross-validation in `caret`

Speaking of integrating cross-validation into your daily workflow—which possibilities are there? Luckily, cross-validation is a standard tool in popular machine learning libraries such as the `caret` package in R. Here you can specify the method with the `trainControl` function. Below is a script where we fit a random forest with 10-fold cross-validation to the `iris` dataset.

```
library(caret)

set.seed(12345)
inTrain <- createDataPartition(y = iris$Species, p = .7, list = FALSE)

iris.train <- iris[inTrain, ]
iris.test <- iris[- inTrain, ]

fit.control <- caret::trainControl(method = "cv", number = 10)

rf.fit <- caret::train(Species ~ .,
                       data = iris.train,
                       method = "rf",
                       trControl = fit.control)
```

We define our desired cross-validation method in the `trainControl` function, store the output in the object `fit.control`, and then pass this object to the `trControl` argument of the `train` function. You can specify the other methods introduced in this post in a similar fashion:

```
# Leave-One-Out Cross-validation:
fit.control <- caret::trainControl(method = "LOOCV", number = 10)

# Repeated CV (remember to specify the number of repeats!)
fit.control <- caret::trainControl(method = "repeatedcv", number = 10, repeats = 5)
```

The old-fashioned way: Implementing k-fold cross-validation by hand

However, data science projects can quickly become so complex that the ready-made functions in machine learning packages are not suitable anymore. In such cases, you will have to implement the algorithm—including cross-validation techniques—by hand, tailored to the specific project needs. Let me walk you through a make-shift script for implementing simple k-fold cross-validation in R by hand (we will tackle the script step by step here; you can find the whole code on our GitHub).

Simulating data, defining the error metric, and setting k

```
# devtools::install_github("andrebleier/Xy")
library(tidyverse)
library(Xy)

sim <- Xy(n = 1000,
          numvars = c(2,2),
          catvars = 0,
          cor = c(-0.5, 0.9),
          noisevars = 0)

sim_data <- sim$data

RMSE <- function(f, o){
  sqrt(mean((f - o)^2))
}

k <- 5
```

We start by loading the required packages and simulating some simulation data with 1,000 observations with the `Xy()` package developed by my colleague André (check out his blog post on simulating regression data with Xy). Because we need some kind of error metric to evaluate model performance, we define our RMSE function which is pretty straightforward: The RMSE is the root of the mean of the squared error, where error is the difference between our fitted (`f`) und observed (`o`) values—you can pretty much read the function from left to right. Lastly, we specify our k, which is set to the value of 5 in the example and is stored as a simple integer.

Partitioning the data

```
set.seed(12345)
sim_data <- mutate(sim_data,
                   my.folds = sample(1:k,
                                     size = nrow(sim_data),
                                     replace = TRUE))
```

Next up, we partition our data into k folds. For this purpose, we add a new column, `my.folds`, to the data: We sample (with replacement) from 1 to the value of k, so 1 to 5 in our case, and randomly add one of these five numbers to each row (observation) in the data. With 1,000 observations, each number should be assigned about 200 times.

Training and validating the model

```
cv.fun <- function(this.fold, data){

  train <- filter(data, my.folds != this.fold)
  validate <- filter(data, my.folds == this.fold)

  model <- lm(y ~ NLIN_1 + NLIN_2 + LIN_1 + LIN_2,
              data = train)

  pred <- predict(model, newdata = validate) %>% as.vector()

  this.rmse <- RMSE(f = pred, o = validate$y)

  return(this.rmse)
}
```

Next, we define `cv.fun`, which is the heart of our cross-validation procedure. This function takes two arguments: `this.fold` and `data`. I will come back to the meaning of `this.fold` in a minute, let's just set it to 1 for now. Inside the function, we divide the data into a training and validation partition by subsetting according to the values of `my.folds` and `this.fold`: Every observation with a randomly assigned `my.folds` value **other than 1** (so approximately 4/5 of the data) goes into training. Every observation with a `my.folds` value **equal to 1** (the remaining 1/5) forms the validation set. For illustration purposes, we then fit a simple linear model with the simulated outcome and four predictors. Note that we only fit this model on the `train` data! We then use this model to `predict()` our validation data, and since we have true observed outcomes for this *subset of the original overall training data* (this is the whole point!), we can compute our RMSE and return it.

Iterating through the folds and computing the CV error

```
cv.error <- sapply(seq_len(k),
                   FUN = cv.fun,
                   data = sim_data) %>%
  mean()

cv.error
```

Lastly, we wrap the function call to `cv.fun` into a `sapply()` loop—this is where all the magic happens: Here we iterate over the range of k, so `seq_len(k)` leaves us with the vector `[1] 1 2 3 4 5` in this case. We apply each element of this vector to `cv.fun`. In `apply()` statements, the iteration vector is always passed as the first argument of the function which is called, so in our case, each element of this vector at a time is passed to `this.fold`. We also pass our simulated `sim_data` as the `data` argument.

Let us quickly recap what this means: In the first iteration, `this.fold` equals 1. This means that our train set consists of all the observations where `my.folds` is not 1, and observations with a value of 1 form the validation set (just as in the example above). In the next iteration of the loop, `this.fold` equals 2. Consequently, observations with 1, 3, 4, and 5 form the training set, and observations with a value of 2 go to validation, and so on. Iterating over all values of k, this schematically provides us with the diagonal pattern seen in Figure 2 above, where each data partition at a time is used as a validation set.

To wrap it all up, we calculate the mean: This is the mean of our k individual RMSE values and leaves us with our cross-validation error. And there you go: We just defined our custom cross-validation function! This is merely a template: You can insert any model and any error metric. If you've been following along so far, feel free to try implementing repeated CV yourself or play around with different values for k.

Conclusion

As you can see, implementing cross-validation yourself isn't all that hard. It gives you great flexibility to account for project-specific needs, such as custom error metrics. If you don't need that much flexibility, enabling cross-validation in popular machine learning packages is a breeze. I hope that I could provide you with a sufficient overview of cross-validation and how to implement it both in pre-defined functions as well as by hand. If you have questions, comments, or ideas, feel free to drop me an e-mail.

References

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. An Introduction to Statistical Learning. New York: Springer.

Über den Autor

Lukas Feick

I am a data scientist at STATWORX. I have always enjoyed using data-driven approaches to tackle complex real-world problems, and to help people gain better insights.
