**One-Stop Tutorial on All Cross-Validation Techniques You Can (and Should) Use**

**All CV procedures you need to know as a data scientist, explained**



Photo by [Anni Roenkae](https://www.pexels.com/photo/close-up-photo-of-pink-fluid-2156884/)

**Why cross-validate?**

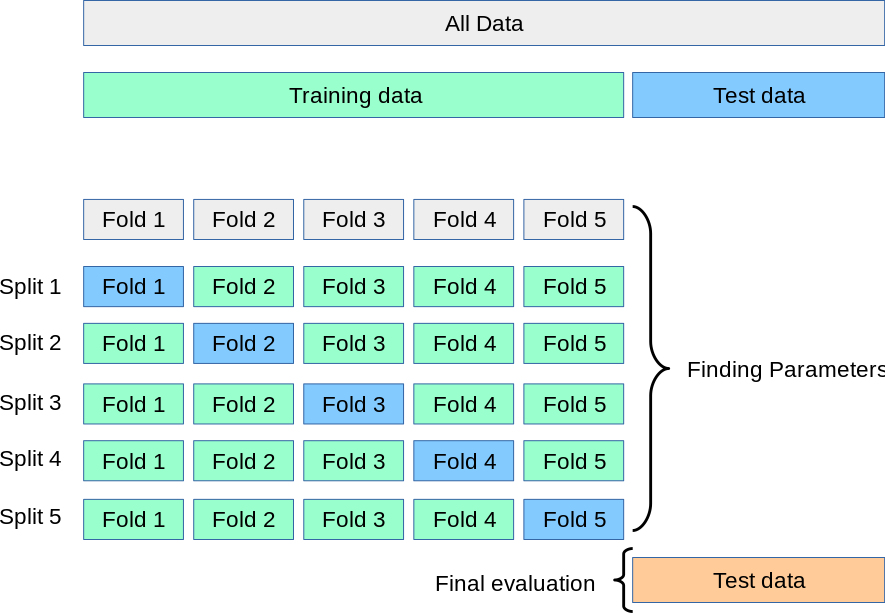
Until I start selling the related merchandise, I gotta advertise the main idea. Here it goes.

Let’s imagine a world where you don’t know what a cross-valıdatıon procedure is. In that crazy world, you obviously split your data into a single train and test sets. The model learns from the training data, and you test its performance by predicting on the so-called unseen data that is your test set. If you are not satisfied with the score, you tune the heck out of your model using the same sets until [GridSearch](https://towardsdatascience.com/11-times-faster-hyperparameter-tuning-with-halvinggridsearch-232ed0160155) (or [Optuna](https://towardsdatascience.com/why-is-everyone-at-kaggle-obsessed-with-optuna-for-hyperparameter-tuning-7608fdca337c)) cries out “enough!”.

Here are *two* of the many ways this process can go *horribly* wrong:

1. **The sets don’t represent the whole population well**. As an extreme example, out of rows with three categories (a, b, c), all **a** and **b** categories may end up in the training set while all **c**s are hanging out in the test set. Or a numeric variable is split so that values to the left and right of some threshold are not distributed well among the train and sets. Or a situation close to that where the new distributions of the variables in both sets are so different than the originals that the model learns from incorrect information.
2. **You leak knowledge about the test set into the model during hyperparameter tuning.** After the search is done, the framework spits out the parameters that work best for that *specific* test set. Since I am using the word *specific*, you should already start thinking about overfitting. Because that’s what happens if you keep testing on the same set repeatedly — the searching framework just gives you the result that makes you happy for that specific test set.

So, if we get back to the world where CV is loved and extensively used by engineers worldwide, all these problems are solved. Here is the magic of CV, as shown in the [Sklearn user guide](https://scikit-learn.org/stable/modules/cross_validation.html) (BSD license):



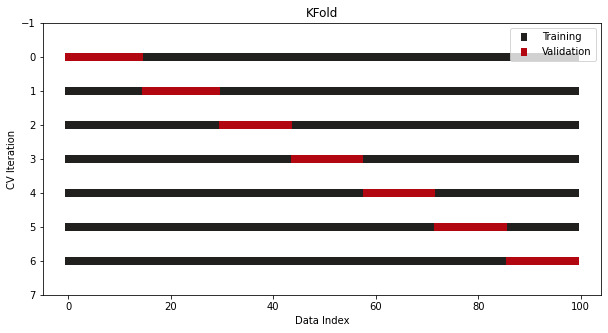
The above is an example of a 5-fold cross-validation process, which takes five iterations to finish. A new model is trained on four folds in each iteration and tested on the last holdout fold. This way, a model is trained and tested on all of the data without wasting any.

Next, the averaged scores are reported with their standard deviations as a confidence interval. Only then can you truly judge your model’s performance with its chosen parameters because the average score you got will represent the model’s true potential to effectively learn from the data and predict accurately on unseen samples.

**1. KFold**

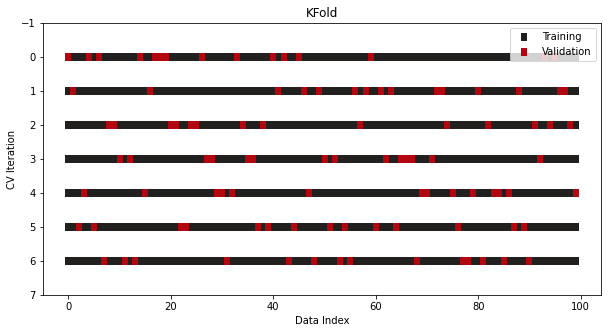
Now, let’s start discussing the many ways you can perform the CV procedure. The simplest one is KFold as seen in the above image. It is implemented with the same name in Sklearn. Here, we will write a quick function that visualizes the split indices of the CV splitter:

Now, let’s pass a KFold splitter with seven splits to this function:



This is what a vanilla KFold looks like.

Another version is shuffling the data before a split is performed. This further minimizes the risk of overfitting by breaking the original order of the samples:



As you can see, the indices of the validation samples are chosen in a shuffled manner. Even so, the overall number of samples is still one-seventh of the whole data because we are doing a 7-fold CV.

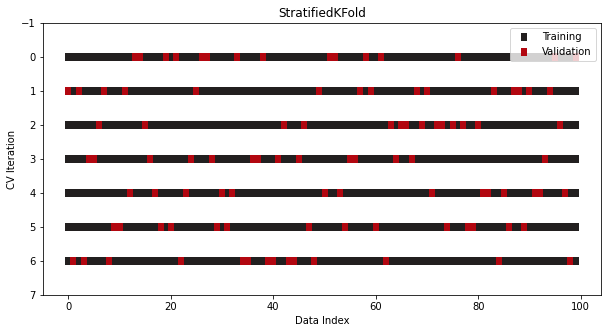
KFold is the most commonly used CV splitter. It is easy to understand and deadly effective. However, depending on the characteristics of your datasets, sometimes you need to be pickier over what CV procedure to use. So, let’s discuss the alternatives.

**2. StratifiedKFold**

Another version of KFold designed explicitly for classification problems is StratifiedKFold.

In classification, the target distribution must be preserved even after the data is split into multiple sets. More specifically, a binary target with 30 to 70 class ratios should still hold the same ratios in both the training and test sets.

The rule is broken in vanilla KFold because it doesn’t care about class ratios or shuffles the data before splitting. As a solution, we use another splitter class in Sklearn — StratifiedKFold:



It looks the same as KFold, but now class ratios are preserved across all folds and iterations.

**3. LeavePOut**

Sometimes, the data you have is so limited that you can’t even afford to divide it into train and test sets. In that case, you can perform a CV where you set aside only a few rows of data in each iteration. This is known as LeavePOut CV, where *p* is the parameter you choose to specify the number of rows in each holdout set.

The most extreme case is the LeaveOneOut splitter where you only use a single row as a test set, and the number of iterations equals the number of rows in the full data. If building 100 models for a small 100-row dataset seems like it is bordering on crazy, I am right there with you.

Even for higher numbers of p, the number of iterations grows exponentially as your dataset size increases. Just imagine how many models will be built when p is five and your data has just 50 rows (hint - use the permutations formula).

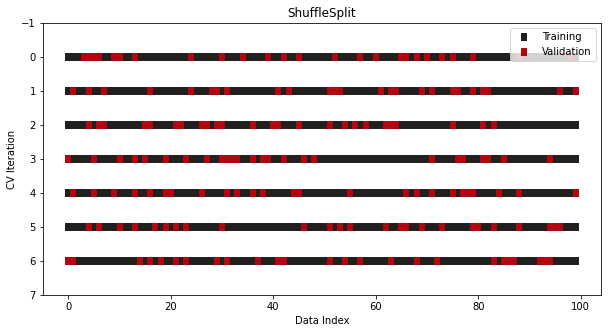
So, you rarely see this one in practice, but it comes up enough times that Sklearn implements these procedures as separate classes:

from sklearn.model\_selection import LeaveOneOut, LeavePOut

**4. ShuffleSplit**

How about we don’t do CV at all and just repeat the train/test split process multiple times? Well, that’s another way you can flirt with the idea of cross-validation and yet still not do it.

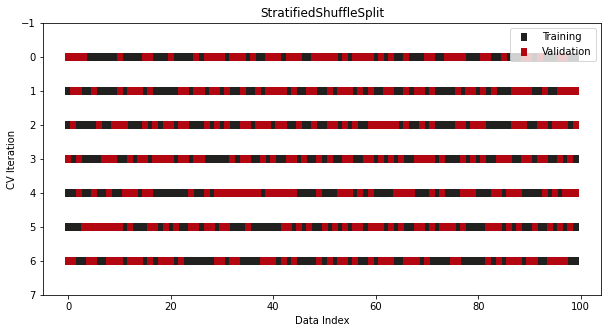
By logic, generating multiple train/test sets using different random seeds should resemble a robust CV process if done for enough iterations. That’s why there is a splitter that performs this process in Sklearn:



The advantage of ShuffleSplit is that you have complete control over the sizes of the train and sets in each fold. The size of the sets doesn’t have to be inversely proportionate to the number of splits.

However, contrary to other splitters, there is no guarantee that random splits will generate different folds in each iteration. So, use this class with caution.

By the way, there is also a stratified version of ShuffleSplit for classification:

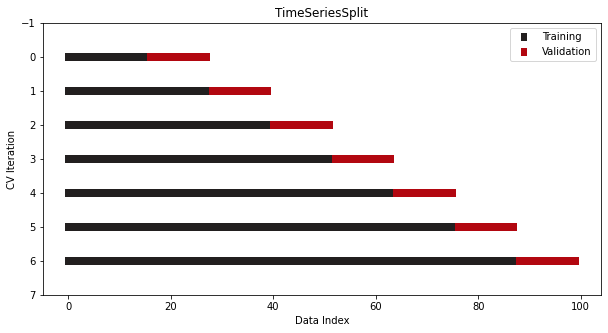


**5. TimeSeriesSplit**

Finally, we have the special case of time series data where the ordering of samples matters.

We can’t use any of the traditional CV classes because they would lead to a disaster. There is a high chance you would be training on the future samples and predicting the past ones.

To solve this, Sklearn offers yet another splitter — TimeSeriesSplit where it ensures that the above does not happen:



Nice and neat!

**Other CV splitters for non-IID data**

So far, we have been dealing with IID (independent and identically distributed) data. In other words, the process that generated the data does not have a memory of the past samples.

However, there are cases where your data is not IID — that some groups of samples are dependent on each other. For example, in the [Google Brain Ventilator Pressure](https://www.kaggle.com/c/ventilator-pressure-prediction/leaderboard) competition on Kaggle, the participants should work with non-IID data.

The data records thousands of breaths (in, out) that an artificial lung takes and records the air pressure for each breath at some millisecond intervals. As a result, the data contains about 80 rows for each breath taken, making those rows dependent.

Here, traditional CV splitters won’t work as expected because there is a definite chance that a split might occur “right in the middle of a breath.” Here is another example from the [Sklearn user guide](https://scikit-learn.org/stable/modules/cross_validation.html#:~:text=Such%20a%20grouping,paired%20training%20fold.):

Such a grouping of data is domain specific. An example would be when there is medical data collected from multiple patients, with multiple samples taken from each patient. And such data is likely to be dependent on the individual group. In our example, the patient id for each sample will be its group identifier.

It also states the solution right after that:

In this case we would like to know if a model trained on a particular set of groups generalizes well to the unseen groups. To measure this, we need to ensure that all the samples in the validation fold come from groups that are not represented at all in the paired training fold.

Then, Sklearn lists five different classes that can work with grouped data. If you grasped the ideas from the previous sections and understood what non-IID data is, you won’t have trouble working with them:

1. [GroupKFold](https://scikit-learn.org/stable/modules/cross_validation.html#group-k-fold)
2. [StratifiedGroupKFold](https://scikit-learn.org/stable/modules/cross_validation.html#stratifiedgroupkfold)
3. [LeaveOneGroupOut](https://scikit-learn.org/stable/modules/cross_validation.html#leave-one-group-out)
4. [LeavePGroupsOut](https://scikit-learn.org/stable/modules/cross_validation.html#leave-p-groups-out)
5. [GroupShuffleSplit](https://scikit-learn.org/stable/modules/cross_validation.html#group-shuffle-split)

Each of these splitters has a groups argument where you should pass the column the group ids are stored. This tells the classes how to differentiate between each group.

**Summary**

Finally, the dust settles, and we are here.

One question I probably left unanswered is, “Should you always use cross-validation?” The answer is a tentative yes. When your dataset is sufficiently large, any random split will probably resemble the original data well in both sets. In that case, a CV is not a strict requirement.

However, statisticians and folks much more experienced than me on StackExchange say that you should perform at least two or 3-fold cross-validation no matter the data size. You just can never be too cautious.

Thank you for readıng!