Benefits of Cross-Validation

There are several benefits to using cross-validation instead of a single split into a training and a test set. First, remember that `train_test_split` performs a random split of the data. Imagine that we are "lucky" when randomly splitting the data, and all examples that are hard to classify end up in the training set. In that case, the test set will only contain "easy" examples, and our test set accuracy will be unrealistically high. Conversely, if we are "unlucky," we might have randomly put all the hard-to-classify examples in the test set and consequently obtain an unrealistically low score. However, when using cross-validation, each example will be in the training set exactly once: each example is in one of the folds, and each fold is the test set once. Therefore, the model needs to generalize well to all of the samples in the dataset for all of the cross-validation scores (and their mean) to be high.

Having multiple splits of the data also provides some information about how sensitive our model is to the selection of the training dataset. For the `iris` dataset, we saw accuracies between 90% and 100%. This is quite a range, and it provides us with an idea about how the model might perform in the worst case and best case scenarios when applied to new data.

Another benefit of cross-validation as compared to using a single split of the data is that we use our data more effectively. When using `train_test_split`, we usually use 75% of the data for training and 25% of the data for evaluation. When using five-fold cross-validation, in each iteration we can use four-fifths of the data (80%) to fit the model. When using 10-fold cross-validation, we can use nine-tenths of the data (90%) to fit the model. More data will usually result in more accurate models.

The main disadvantage of cross-validation is increased computational cost. As we are now training k models instead of a single model, cross-validation will be roughly k times slower than doing a single split of the data.

It is important to keep in mind that cross-validation is not a way to build a model that can be applied to new data. Cross-validation does not return a model. When calling `cross_val_score`, multiple models are built internally, but the purpose of cross-validation is only to evaluate how well a given algorithm will generalize when trained on a specific dataset.

Stratified k-Fold Cross-Validation and Other Strategies

Splitting the dataset into k folds by starting with the first one-k-th part of the data, as described in the previous section, might not always be a good idea. For example, let's have a look at the `iris` dataset:

In[7]:

```
from sklearn.datasets import load_iris
iris = load_iris()
print("Iris labels:\n{}".format(iris.target))
```

Out[7]:

```
Iris labels:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

As you can see, the first third of the data is the class 0, the second third is the class 1, and the last third is the class 2. Imagine doing three-fold cross-validation on this dataset. The first fold would be only class 0, so in the first split of the data, the test set would be only class 0, and the training set would be only classes 1 and 2. As the classes in training and test sets would be different for all three splits, the three-fold cross-validation accuracy would be zero on this dataset. That is not very helpful, as we can do much better than 0% accuracy on `iris`.

As the simple *k*-fold strategy fails here, `scikit-learn` does not use it for classification, but rather uses *stratified k-fold cross-validation*. In stratified cross-validation, we split the data such that the proportions between classes are the same in each fold as they are in the whole dataset, as illustrated in Figure 5-2:

In[8]:

```
mglearn.plots.plot_stratified_cross_validation()
```

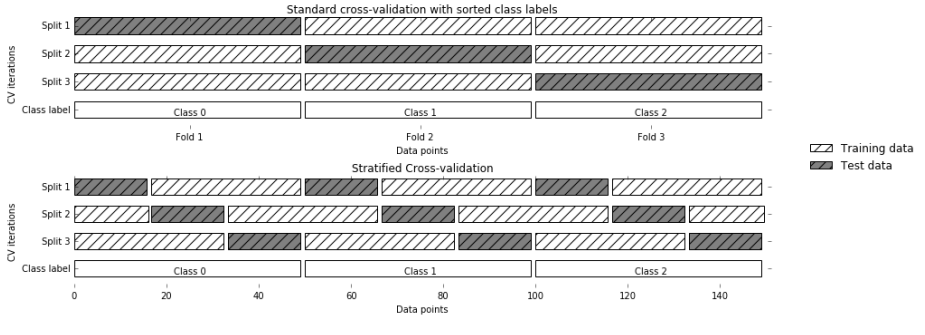

Figure 5-2. Comparison of standard cross-validation and stratified cross-validation when the data is ordered by class label

For example, if 90% of your samples belong to class A and 10% of your samples belong to class B, then stratified cross-validation ensures that in each fold, 90% of samples belong to class A and 10% of samples belong to class B.

It is usually a good idea to use stratified *k*-fold cross-validation instead of *k*-fold cross-validation to evaluate a classifier, because it results in more reliable estimates of generalization performance. In the case of only 10% of samples belonging to class B, using standard *k*-fold cross-validation it might easily happen that one fold only contains samples of class A. Using this fold as a test set would not be very informative about the overall performance of the classifier.

For regression, `scikit-learn` uses the standard *k*-fold cross-validation by default. It would be possible to also try to make each fold representative of the different values the regression target has, but this is not a commonly used strategy and would be surprising to most users.

More control over cross-validation

We saw earlier that we can adjust the number of folds that are used in `cross_val_score` using the `cv` parameter. However, `scikit-learn` allows for much finer control over what happens during the splitting of the data by providing a *cross-validation splitter* as the `cv` parameter. For most use cases, the defaults of *k*-fold cross-validation for regression and stratified *k*-fold for classification work well, but there are some cases where you might want to use a different strategy. Say, for example, we want to use the standard *k*-fold cross-validation on a classification dataset to reproduce someone else's results. To do this, we first have to import the `KFold` splitter class from the `model_selection` module and instantiate it with the number of folds we want to use:

`In[9]:`

```
from sklearn.model_selection import KFold
kfold = KFold(n_splits=5)
```

Then, we can pass the `kfold` splitter object as the `cv` parameter to `cross_val_score`:

`In[10]:`

```
print("Cross-validation scores:\n{}".format(
      cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

`Out[10]:`

```
Cross-validation scores:
[ 1.     0.933  0.433  0.967  0.433]
```

This way, we can verify that it is indeed a really bad idea to use three-fold (nonstratified) cross-validation on the `iris` dataset:

In[11]:

```
kfold = KFold(n_splits=3)
print("Cross-validation scores:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

Out[11]:

```
Cross-validation scores:
[ 0.  0.  0.]
```

Remember: each fold corresponds to one of the classes in the `iris` dataset, and so nothing can be learned. Another way to resolve this problem is to shuffle the data instead of stratifying the folds, to remove the ordering of the samples by label. We can do that by setting the `shuffle` parameter of `KFold` to `True`. If we shuffle the data, we also need to fix the `random_state` to get a reproducible shuffling. Otherwise, each run of `cross_val_score` would yield a different result, as each time a different split would be used (this might not be a problem, but can be surprising). Shuffling the data before splitting it yields a much better result:

In[12]:

```
kfold = KFold(n_splits=3, shuffle=True, random_state=0)
print("Cross-validation scores:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

Out[12]:

```
Cross-validation scores:
[ 0.9   0.96  0.96]
```

Leave-one-out cross-validation

Another frequently used cross-validation method is *leave-one-out*. You can think of leave-one-out cross-validation as *k*-fold cross-validation where each fold is a single sample. For each split, you pick a single data point to be the test set. This can be very time consuming, particularly for large datasets, but sometimes provides better estimates on small datasets:

In[13]:

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
print("Number of cv iterations: ", len(scores))
print("Mean accuracy: {:.2f}".format(scores.mean()))
```

Out[13]:

```
Number of cv iterations:  150
Mean accuracy: 0.95
```

Shuffle-split cross-validation

Another, very flexible strategy for cross-validation is *shuffle-split cross-validation*. In shuffle-split cross-validation, each split samples `train_size` many points for the training set and `test_size` many (disjoint) point for the test set. This splitting is repeated `n_iter` times. Figure 5-3 illustrates running four iterations of splitting a dataset consisting of 10 points, with a training set of 5 points and test sets of 2 points each (you can use integers for `train_size` and `test_size` to use absolute sizes for these sets, or floating-point numbers to use fractions of the whole dataset):

`In[14]:`

```
mglearn.plots.plot_shuffle_split()
```

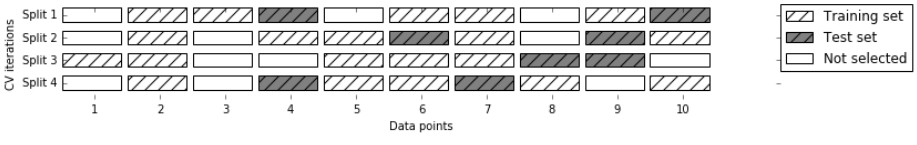

Figure 5-3. ShuffleSplit with 10 points, train_size=5, test_size=2, and n_iter=4

The following code splits the dataset into 50% training set and 50% test set for 10 iterations:

`In[15]:`

```
from sklearn.model_selection import ShuffleSplit
shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_splits=10)
scores = cross_val_score(logreg, iris.data, iris.target, cv=shuffle_split)
print("Cross-validation scores:\n{}".format(scores))
```

`Out[15]:`

```
Cross-validation scores:
[ 0.96   0.907  0.947  0.96   0.96   0.907  0.893  0.907  0.92   0.973]
```

Shuffle-split cross-validation allows for control over the number of iterations independently of the training and test sizes, which can sometimes be helpful. It also allows for using only part of the data in each iteration, by providing `train_size` and `test_size` settings that don't add up to one. Subsampling the data in this way can be useful for experimenting with large datasets.

There is also a stratified variant of `ShuffleSplit`, aptly named `StratifiedShuffleSplit`, which can provide more reliable results for classification tasks.

Cross-validation with groups

Another very common setting for cross-validation is when there are groups in the data that are highly related. Say you want to build a system to recognize emotions from pictures of faces, and you collect a dataset of pictures of 100 people where each person is captured multiple times, showing various emotions. The goal is to build a classifier that can correctly identify emotions of people not in the dataset. You could use the default stratified cross-validation to measure the performance of a classifier here. However, it is likely that pictures of the same person will be in both the training and the test set. It will be much easier for a classifier to detect emotions in a face that is part of the training set, compared to a completely new face. To accurately evaluate the generalization to new faces, we must therefore ensure that the training and test sets contain images of different people.

To achieve this, we can use `GroupKFold`, which takes an array of `groups` as argument that we can use to indicate which person is in the image. The `groups` array here indicates groups in the data that should not be split when creating the training and test sets, and should not be confused with the class label.

This example of groups in the data is common in medical applications, where you might have multiple samples from the same patient, but are interested in generalizing to new patients. Similarly, in speech recognition, you might have multiple recordings of the same speaker in your dataset, but are interested in recognizing speech of new speakers.

The following is an example of using a synthetic dataset with a grouping given by the `groups` array. The dataset consists of 12 data points, and for each of the data points, `groups` specifies which group (think patient) the point belongs to. The groups specify that there are four groups, and the first three samples belong to the first group, the next four samples belong to the second group, and so on:

`In[17]:`

```
from sklearn.model_selection import GroupKFold
# create synthetic dataset
X, y = make_blobs(n_samples=12, random_state=0)
# assume the first three samples belong to the same group,
# then the next four, etc.
groups = [0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3]
scores = cross_val_score(logreg, X, y, groups, cv=GroupKFold(n_splits=3))
print("Cross-validation scores:\n{}".format(scores))
```

`Out[17]:`

```
Cross-validation scores:
[ 0.75   0.8    0.667]
```

The samples don't need to be ordered by group; we just did this for illustration purposes. The splits that are calculated based on these labels are visualized in Figure 5-4.

As you can see, for each split, each group is either entirely in the training set or entirely in the test set:

In[16]:

```
mglearn.plots.plot_label_kfold()
```

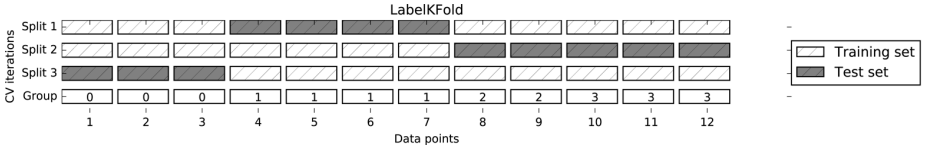

Figure 5-4. Label-dependent splitting with GroupKFold

There are more splitting strategies for cross-validation in `scikit-learn`, which allow for an even greater variety of use cases (you can find these in the `scikit-learn` user guide). However, the standard `KFold`, `StratifiedKFold`, and `GroupKFold` are by far the most commonly used ones.

Grid Search

Now that we know how to evaluate how well a model generalizes, we can take the next step and improve the model's generalization performance by tuning its parameters. We discussed the parameter settings of many of the algorithms in `scikit-learn` in Chapters 2 and 3, and it is important to understand what the parameters mean before trying to adjust them. Finding the values of the important parameters of a model (the ones that provide the best generalization performance) is a tricky task, but necessary for almost all models and datasets. Because it is such a common task, there are standard methods in `scikit-learn` to help you with it. The most commonly used method is *grid search*, which basically means trying all possible combinations of the parameters of interest.

Consider the case of a kernel SVM with an RBF (radial basis function) kernel, as implemented in the `SVC` class. As we discussed in Chapter 2, there are two important parameters: the kernel bandwidth, `gamma`, and the regularization parameter, `C`. Say we want to try the values `0.001`, `0.01`, `0.1`, `1`, `10`, and `100` for the parameter `C`, and the same for `gamma`. Because we have six different settings for `C` and `gamma` that we want to try, we have 36 combinations of parameters in total. Looking at all possible combinations creates a table (or grid) of parameter settings for the SVM, as shown here: