Introduction\_Machine\_Learning\_Python\_Andreas\_C05

TER 5

Model Evaluation and Improvement

Having discussed the fundamentals of supervised and unsupervised learning, and having explored a variety of machine learning algorithms, we will now dive more deeply into evaluating models and selecting parameters.

We will focus on the supervised methods, regression and classification, as evaluating and selecting models in unsupervised learning is often a very qualitative process (as we saw in Chapter 3).

To evaluate our supervised models, so far we have split our dataset into a training set and a test set using the train\_test\_split function, built a model on the training set by calling the fit method, and evaluated it on the test set using the score method, which for classification computes the fraction of correctly classified samples. Here’s an example of that process:

In[1]:

from sklearn.datasets import make\_blobs

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

# create a synthetic dataset

X, y = make\_blobs(random\_state=0)

# split data and labels into a training and a test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)

# instantiate a model and fit it to the training set

logreg = LogisticRegression().fit(X\_train, y\_train)

# evaluate the model on the test set

print("Test set score: {:.2f}".format(logreg.score(X\_test, y\_test)))

Out[1]:

Test set score: 0.88

253

Remember, the reason we split our data into training and test sets is that we are interested in measuring how well our model generalizes to new, previously unseen data. We are not interested in how well our model fit the training set, but rather in how well it can make predictions for data that was not observed during training.

In this chapter, we will expand on two aspects of this evaluation. We will first introduce cross-validation, a more robust way to assess generalization performance, and discuss methods to evaluate classification and regression performance that go beyond the default measures of accuracy and R2 provided by the score method.

We will also discuss grid search, an effective method for adjusting the parameters in supervised models for the best generalization performance.

**Cross-Validation**

Cross-validation is a statistical method of evaluating generalization performance that is more stable and thorough than using a split into a training and a test set. In cross-validation, the data is instead split repeatedly and multiple models are trained. The most commonly used version of cross-validation is k-fold cross-validation, where k is a user-specified number, usually 5 or 10. When performing five-fold cross-validation, the data is first partitioned into five parts of (approximately) equal size, called folds. Next, a sequence of models is trained. The first model is trained using the first fold as the test set, and the remaining folds (2–5) are used as the training set. The model is built using the data in folds 2–5, and then the accuracy is evaluated on fold 1. Then another model is built, this time using fold 2 as the test set and the data in folds 1, 3, 4, and 5 as the training set. This process is repeated using folds 3, 4, and 5 as test sets. For each of these five splits of the data into training and test sets, we compute the accuracy. In the end, we have collected five accuracy values. The process is illustrated in Figure 5-1:

In[2]:

mglearn.plots.plot\_cross\_validation()

Figure 5-1. Data splitting in five-fold cross-validation

Usually, the first fifth of the data is the first fold, the second fifth of the data is the second fold, and so on.

**Cross-Validation in scikit-learn**

Cross-validation is implemented in scikit-learn using the cross\_val\_score function from the model\_selection module. The parameters of the cross\_val\_score function are the model we want to evaluate, the training data, and the ground-truth labels. Let’s evaluate LogisticRegression on the iris dataset:

In[3]:

from sklearn.model\_selection import cross\_val\_score

from sklearn.datasets import load\_iris

from sklearn.linear\_model import LogisticRegression

iris = load\_iris()

logreg = LogisticRegression()

scores = cross\_val\_score(logreg, iris.data, iris.target)

print("Cross-validation scores: {}".format(scores))

Out[3]:

Cross-validation scores: [ 0.961

0.922

0.958]

By default, cross\_val\_score performs three-fold cross-validation, returning three accuracy values. We can change the number of folds used by changing the cv parameter:

In[4]:

scores = cross\_val\_score(logreg, iris.data, iris.target, cv=5)

print("Cross-validation scores: {}".format(scores))

Out[4]:

Cross-validation scores: [ 1.

0.967

0.933

0.9

1.

]

A common way to summarize the cross-validation accuracy is to compute the mean:

In[5]:

print("Average cross-validation score: {:.2f}".format(scores.mean()))

Out[5]:

Average cross-validation score: 0.96

Using the mean cross-validation we can conclude that we expect the model to be around 96% accurate on average. Looking at all five scores produced by the five-fold cross-validation, we can also conclude that there is a relatively high variance in the accuracy between folds, ranging from 100% accuracy to 90% accuracy. This could imply that the model is very dependent on the particular folds used for training, but it could also just be a consequence of the small size of the dataset.

**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 test 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:

256

|

Chapter 5: Model Evaluation and ImprovementIn[6]:

from sklearn.datasets import load\_iris

iris = load\_iris()

print("Iris labels:\n{}".format(iris.target))

Out[6]:

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 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]

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[7]:

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

Cross-Validation

|

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[8]:

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[9]:

print("Cross-validation scores:\n{}".format(

cross\_val\_score(logreg, iris.data, iris.target, cv=kfold)))

Out[9]:

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:

258

|

Chapter 5: Model Evaluation and ImprovementIn[10]:

kfold = KFold(n\_splits=3)

print("Cross-validation scores:\n{}".format(

cross\_val\_score(logreg, iris.data, iris.target, cv=kfold)))

Out[10]:

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[11]:

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[11]:

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[12]:

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[12]:

Number of cv iterations:

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\_splits 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[13]:

mglearn.plots.plot\_shuffle\_split()

Figure 5-3. ShuffleSplit with 10 points, train\_size=5, test\_size=2, and n\_splits=4

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

In[14]:

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[14]:

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 StratifiedShuffleS plit, 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[15]:

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[15]:

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\_group\_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:

262

| Chapter 5: Model Evaluation and Improvementgamma=0.001C = 0.001

C = 0.01

… C = 10

SVC(C=0.001, gamma=0.001) SVC(C=0.01, gamma=0.001) … SVC(C=10, gamma=0.001)

gamma=0.01SVC(C=0.001, gamma=0.01)

SVC(C=0.01, gamma=0.01)

… SVC(C=10, gamma=0.01)

……

…

… …

gamma=100SVC(C=0.001, gamma=100)

SVC(C=0.01, gamma=100)

… SVC(C=10, gamma=100)

**Simple Grid Search**

We can implement a simple grid search just as for loops over the two parameters, training and evaluating a classifier for each combination:

In[17]:

# naive grid search implementation

from sklearn.svm import SVC

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

iris.data, iris.target, random\_state=0)

print("Size of training set: {}

size of test set: {}".format(

X\_train.shape[0], X\_test.shape[0]))

best\_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:

for C in [0.001, 0.01, 0.1, 1, 10, 100]:

# for each combination of parameters, train an SVC

svm = SVC(gamma=gamma, C=C)

svm.fit(X\_train, y\_train)

# evaluate the SVC on the test set

score = svm.score(X\_test, y\_test)

# if we got a better score, store the score and parameters

if score > best\_score:

best\_score = score

best\_parameters = {'C': C, 'gamma': gamma}

print("Best score: {:.2f}".format(best\_score))

print("Best parameters: {}".format(best\_parameters))

Out[17]:

Size of training set: 112

size of test set: 38

Best score: 0.97

Best parameters: {'C': 100, 'gamma': 0.001}

**The Danger of Overfitting the Parameters and the Validation Set**

Given this result, we might be tempted to report that we found a model that performs with 97% accuracy on our dataset. However, this claim could be overly optimistic (or just wrong), for the following reason: we tried many different parameters and selected the one with best accuracy on the test set, but this accuracy won’t necessarily carry over to new data. Because we used the test data to adjust the parameters, we can no longer use it to assess how good the model is. This is the same reason we needed to split the data into training and test sets in the first place; we need an independent dataset to evaluate, one that was not used to create the model.

One way to resolve this problem is to split the data again, so we have three sets: the training set to build the model, the validation (or development) set to select the parameters of the model, and the test set to evaluate the performance of the selected parameters. Figure 5-5 shows what this looks like:

In[18]:

mglearn.plots.plot\_threefold\_split()

Figure 5-5. A threefold split of data into training set, validation set, and test set

After selecting the best parameters using the validation set, we can rebuild a model using the parameter settings we found, but now training on both the training data and the validation data. This way, we can use as much data as possible to build our model. This leads to the following implementation:

In[19]:

from sklearn.svm import SVC

# split data into train+validation set and test set

X\_trainval, X\_test, y\_trainval, y\_test = train\_test\_split(

iris.data, iris.target, random\_state=0)

# split train+validation set into training and validation sets

X\_train, X\_valid, y\_train, y\_valid = train\_test\_split(

X\_trainval, y\_trainval, random\_state=1)

print("Size of training set: {}

size of validation set: {}

size of test set:"

" {}\n".format(X\_train.shape[0], X\_valid.shape[0], X\_test.shape[0]))

best\_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:

for C in [0.001, 0.01, 0.1, 1, 10, 100]:

# for each combination of parameters, train an SVC

svm = SVC(gamma=gamma, C=C)

svm.fit(X\_train, y\_train)

# evaluate the SVC on the validation set

score = svm.score(X\_valid, y\_valid)

# if we got a better score, store the score and parameters

if score > best\_score:

best\_score = score

best\_parameters = {'C': C, 'gamma': gamma}

264

| Chapter 5: Model Evaluation and Improvement# rebuild a model on the combined training and validation set,

# and evaluate it on the test set

svm = SVC(\*\*best\_parameters)

svm.fit(X\_trainval, y\_trainval)

test\_score = svm.score(X\_test, y\_test)

print("Best score on validation set: {:.2f}".format(best\_score))

print("Best parameters: ", best\_parameters)

print("Test set score with best parameters: {:.2f}".format(test\_score))

Out[19]:

Size of training set: 84

size of validation set: 28

size of test set: 38

Best score on validation set: 0.96

Best parameters: {'C': 10, 'gamma': 0.001}

Test set score with best parameters: 0.92

The best score on the validation set is 96%: slightly lower than before, probably because we used less data to train the model (X\_train is smaller now because we split our dataset twice). However, the score on the test set—the score that actually tells us how well we generalize—is even lower, at 92%. So we can only claim to classify new data 92% correctly, not 97% correctly as we thought before!

The distinction between the training set, validation set, and test set is fundamentally important to applying machine learning methods in practice. Any choices made based on the test set accuracy “leak” information from the test set into the model. Therefore, it is important to keep a separate test set, which is only used for the final evaluation. It is good practice to do all exploratory analysis and model selection using the combination of a training and a validation set, and reserve the test set for a final evaluation—this is even true for exploratory visualization. Strictly speaking, evaluating more than one model on the test set and choosing the better of the two will result in an overly optimistic estimate of how accurate the model is.

**Grid Search with Cross-Validation**

While the method of splitting the data into a training, a validation, and a test set that we just saw is workable, and relatively commonly used, it is quite sensitive to how exactly the data is split. From the output of the previous code snippet we can see that grid search selects 'C': 10, 'gamma': 0.001 as the best parameters, while the output of the code in the previous section selects 'C': 100, 'gamma': 0.001 as the best parameters. For a better estimate of the generalization performance, instead of using a single split into a training and a validation set, we can use cross-validation to evaluate the performance of each parameter combination. This method can be coded up as follows:

Grid Search

|

265In[20]:

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:

for C in [0.001, 0.01, 0.1, 1, 10, 100]:

# for each combination of parameters,

# train an SVC

svm = SVC(gamma=gamma, C=C)

# perform cross-validation

scores = cross\_val\_score(svm, X\_trainval, y\_trainval, cv=5)

# compute mean cross-validation accuracy

score = np.mean(scores)

# if we got a better score, store the score and parameters

if score > best\_score:

best\_score = score

best\_parameters = {'C': C, 'gamma': gamma}

# rebuild a model on the combined training and validation set

svm = SVC(\*\*best\_parameters)

svm.fit(X\_trainval, y\_trainval)

To evaluate the accuracy of the SVM using a particular setting of C and gamma using five-fold cross-validation, we need to train 36 \* 5 = 180 models. As you can imagine, the main downside of the use of cross-validation is the time it takes to train all these models.

The following visualization (Figure 5-6) illustrates how the best parameter setting is selected in the preceding code:

In[21]:

mglearn.plots.plot\_cross\_val\_selection()

Figure 5-6. Results of grid search with cross-validation

For each parameter setting (only a subset is shown), five accuracy values are computed, one for each split in the cross-validation. Then the mean validation accuracy is computed for each parameter setting. The parameters with the highest mean validation accuracy are chosen, marked by the circle.

As we said earlier, cross-validation is a way to evaluate a given algorithm on a specific dataset. However, it is often used in conjunction with parameter search methods like grid search. For this reason, many people use the term cross-validation colloquially to refer to grid search with cross-validation.

The overall process of splitting the data, running the grid search, and evaluating the final parameters is illustrated in Figure 5-7:

In[22]:

mglearn.plots.plot\_grid\_search\_overview()

Figure 5-7. Overview of the process of parameter selection and model evaluation with

GridSearchCV

Because grid search with cross-validation is such a commonly used method to adjust parameters, scikit-learn provides the GridSearchCV class, which implements it in the form of an estimator. To use the GridSearchCV class, you first need to specify the parameters you want to search over using a dictionary. GridSearchCV will then perform all the necessary model fits. The keys of the dictionary are the names of parameters we want to adjust (as given when constructing the model—in this case, C and gamma), and the values are the parameter settings we want to try out. Trying the values 0.001, 0.01, 0.1, 1, 10, and 100 for C and gamma translates to the following dictionary:

In[23]:

param\_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],

'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}

print("Parameter grid:\n{}".format(param\_grid))

Out[23]:

Parameter grid:

{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}

Grid Search

We can now instantiate the GridSearchCV class with the model (SVC), the parameter grid to search (param\_grid), and the cross-validation strategy we want to use (say, five-fold stratified cross-validation):

In[24]:

from sklearn.model\_selection import GridSearchCV

from sklearn.svm import SVC

grid\_search = GridSearchCV(SVC(), param\_grid, cv=5)

GridSearchCV will use cross-validation in place of the split into a training and validation set that we used before. However, we still need to split the data into a training and a test set, to avoid overfitting the parameters:

In[25]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

iris.data, iris.target, random\_state=0)

The grid\_search object that we created behaves just like a classifier; we can call the standard methods fit, predict, and score on it.1 However, when we call fit, it will run cross-validation for each combination of parameters we specified in param\_grid:

In[26]:

grid\_search.fit(X\_train, y\_train)

Fitting the GridSearchCV object not only searches for the best parameters, but also automatically fits a new model on the whole training dataset with the parameters that yielded the best cross-validation performance. What happens in fit is therefore equivalent to the result of the In[21] code we saw at the beginning of this section. The GridSearchCV class provides a very convenient interface to access the retrained model using the predict and score methods. To evaluate how well the best found parameters generalize, we can call score on the test set:

In[27]:

print("Test set score: {:.2f}".format(grid\_search.score(X\_test, y\_test)))

Out[27]:

Test set score: 0.97

Choosing the parameters using cross-validation, we actually found a model that achieves 97% accuracy on the test set. The important thing here is that we did not use the test set to choose the parameters. The parameters that were found are stored in the

1 A scikit-learn estimator that is created using another estimator is called a meta-estimator. GridSearchCV is the most commonly used meta-estimator, but we will see more later.

best\_params\_ attribute, and the best cross-validation accuracy (the mean accuracy over the different splits for this parameter setting) is stored in best\_score\_:

In[28]:

print("Best parameters: {}".format(grid\_search.best\_params\_))

print("Best cross-validation score: {:.2f}".format(grid\_search.best\_score\_))

Out[28]:

Best parameters: {'C': 100, 'gamma': 0.01}

Best cross-validation score: 0.97

Again, be careful not to confuse best\_score\_ with the generalization performance of the model as computed by the score method on the test set. Using the score method (or evaluating the output of the predict method) employs a model trained on the whole training set. The best\_score\_ attribute stores the mean cross-validation accuracy, with cross-validation performed on the training set.

Sometimes it is helpful to have access to the actual model that was found—for exam‐ ple, to look at coefficients or feature importances. You can access the model with the best parameters trained on the whole training set using the best\_estimator\_attribute:

In[29]:

print("Best estimator:\n{}".format(grid\_search.best\_estimator\_))

Out[29]:

Best estimator:

SVC(C=100, cache\_size=200, class\_weight=None, coef0=0.0,

decision\_function\_shape='ovr', degree=3, gamma=0.01, kernel='rbf',

max\_iter=-1, probability=False, random\_state=None, shrinking=True,

tol=0.001, verbose=False)

Because grid\_search itself has predict and score methods, using best\_estimator\_is not needed to make predictions or evaluate the model.

**Analyzing the result of cross-validation**

It is often helpful to visualize the results of cross-validation, to understand how the model generalization depends on the parameters we are searching. As grid searches are quite computationally expensive to run, often it is a good idea to start with a relatively coarse and small grid. We can then inspect the results of the cross-validated grid search, and possibly expand our search. The results of a grid search can be found in the cv\_results\_ attribute, which is a dictionary storing all aspects of the search. It contains a lot of details, as you can see in the following output, and is best looked at after converting it to a pandas DataFrame:

In[30]:

import pandas as pd

# convert to DataFrame

results = pd.DataFrame(grid\_search.cv\_results\_)

# show the first 5 rows

display(results.head())

Out[30]:

0

1

2

3

4

param\_C

0.001

0.001

0.001

0.001

0.001

param\_gamma

0.001

0.01

0.1

1

10

0

1

2

3

4rank\_test\_score

22

22

22

22

22

0

1

2

3

4split3\_test\_score

0.363

0.363

0.363

0.363

0.363

params

{'C': 0.001, 'gamma': 0.001}

{'C': 0.001, 'gamma': 0.01}

{'C': 0.001, 'gamma': 0.1}

{'C': 0.001, 'gamma': 1}

{'C': 0.001, 'gamma': 10}

split0\_test\_score

0.375

0.375

0.375

0.375

0.375

mean\_test\_score

0.366

0.366

0.366

0.366

0.366

split4\_test\_score

0.380

0.380

0.380

0.380

0.380

split1\_test\_score

0.347

0.347

0.347

0.347

0.347

split2\_test\_score

0.363

0.363

0.363

0.363

0.363

std\_test\_score

0.011

0.011

0.011

0.011

0.011

Each row in results corresponds to one particular parameter setting. For each setting, the results of all cross-validation splits are recorded, as well as the mean and standard deviation over all splits. As we were searching a two-dimensional grid of parameters (C and gamma), this is best visualized as a heat map (Figure 5-8). First we extract the mean validation scores, then we reshape the scores so that the axes correspond to C and gamma:

In[31]:

scores = np.array(results.mean\_test\_score).reshape(6, 6)

# plot the mean cross-validation scores

mglearn.tools.heatmap(scores, xlabel='gamma', xticklabels=param\_grid['gamma'],

ylabel='C', yticklabels=param\_grid['C'], cmap="viridis")

270

|

Chapter 5: Model Evaluation and ImprovementFigure 5-8. Heat map of mean cross-validation score as a function of C and gamma

Each point in the heat map corresponds to one run of cross-validation, with a particular parameter setting. The color encodes the cross-validation accuracy, with light colors meaning high accuracy and dark colors meaning low accuracy. You can see

that SVC is very sensitive to the setting of the parameters. For many of the parameter

settings, the accuracy is around 40%, which is quite bad; for other settings the accu‐

racy is around 96%. We can take away from this plot several things. First, the parame‐

ters we adjusted are very important for obtaining good performance. Both parameters

(C and gamma) matter a lot, as adjusting them can change the accuracy from 40% to

96%. Additionally, the ranges we picked for the parameters are ranges in which we

see significant changes in the outcome. It’s also important to note that the ranges for

the parameters are large enough: the optimum values for each parameter are not on

the edges of the plot.

Now let’s look at some plots (shown in Figure 5-9) where the result is less ideal,

because the search ranges were not chosen properly:

Figure 5-9. Heat map visualizations of misspecified search grids

Grid Search

|

271In[32]:

fig, axes = plt.subplots(1, 3, figsize=(13, 5))

param\_grid\_linear = {'C': np.linspace(1, 2, 6),

'gamma': np.linspace(1, 2, 6)}

param\_grid\_one\_log = {'C': np.linspace(1, 2, 6),

'gamma': np.logspace(-3, 2, 6)}

param\_grid\_range = {'C': np.logspace(-3, 2, 6),

'gamma': np.logspace(-7, -2, 6)}

for param\_grid, ax in zip([param\_grid\_linear, param\_grid\_one\_log,

param\_grid\_range], axes):

grid\_search = GridSearchCV(SVC(), param\_grid, cv=5)

grid\_search.fit(X\_train, y\_train)

scores = grid\_search.cv\_results\_['mean\_test\_score'].reshape(6, 6)

# plot the mean cross-validation scores

scores\_image = mglearn.tools.heatmap(

scores, xlabel='gamma', ylabel='C', xticklabels=param\_grid['gamma'],

yticklabels=param\_grid['C'], cmap="viridis", ax=ax)

plt.colorbar(scores\_image, ax=axes.tolist())

The first panel shows no changes at all, with a constant color over the whole parame‐

ter grid. In this case, this is caused by improper scaling and range of the parameters C

and gamma. However, if no change in accuracy is visible over the different parameter

settings, it could also be that a parameter is just not important at all. It is usually good

to try very extreme values first, to see if there are any changes in the accuracy as a

result of changing a parameter.

The second panel shows a vertical stripe pattern. This indicates that only the setting

of the gamma parameter makes any difference. This could mean that the gamma param‐

eter is searching over interesting values but the C parameter is not—or it could mean

the C parameter is not important.

The third panel shows changes in both C and gamma. However, we can see that in the

entire bottom left of the plot, nothing interesting is happening. We can probably

exclude the very small values from future grid searches. The optimum parameter set‐

ting is at the top right. As the optimum is in the border of the plot, we can expect that

there might be even better values beyond this border, and we might want to change

our search range to include more parameters in this region.

Tuning the parameter grid based on the cross-validation scores is perfectly fine, and a

good way to explore the importance of different parameters. However, you should

not test different parameter ranges on the final test set—as we discussed earlier, eval‐

272

| Chapter 5: Model Evaluation and Improvementuation of the test set should happen only once we know exactly what model we want

to use.

Search over spaces that are not grids

In some cases, trying all possible combinations of all parameters as GridSearchCV

usually does, is not a good idea. For example, SVC has a kernel parameter, and

depending on which kernel is chosen, other parameters will be relevant. If ker

nel='linear', the model is linear, and only the C parameter is used. If kernel='rbf',

both the C and gamma parameters are used (but not other parameters like degree). In

this case, searching over all possible combinations of C, gamma, and kernel wouldn’t

make sense: if kernel='linear', gamma is not used, and trying different values for

gamma would be a waste of time. To deal with these kinds of “conditional” parameters,

GridSearchCV allows the param\_grid to be a list of dictionaries. Each dictionary in the

list is expanded into an independent grid. A possible grid search involving kernel and

parameters could look like this:

In[33]:

param\_grid = [{'kernel': ['rbf'],

'C': [0.001, 0.01, 0.1, 1, 10, 100],

'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},

{'kernel': ['linear'],

'C': [0.001, 0.01, 0.1, 1, 10, 100]}]

print("List of grids:\n{}".format(param\_grid))

Out[33]:

List of grids:

[{'kernel': ['rbf'], 'C': [0.001, 0.01, 0.1, 1, 10, 100],

'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},

{'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]}]

In the first grid, the kernel parameter is always set to 'rbf' (note that the entry for

kernel is a list of length one), and both the C and gamma parameters are varied. In the

second grid, the kernel parameter is always set to linear, and only C is varied. Now

let’s apply this more complex parameter search:

In[34]:

grid\_search = GridSearchCV(SVC(), param\_grid, cv=5)

grid\_search.fit(X\_train, y\_train)

print("Best parameters: {}".format(grid\_search.best\_params\_))

print("Best cross-validation score: {:.2f}".format(grid\_search.best\_score\_))

Out[34]:

Best parameters: {'C': 100, 'kernel': 'rbf', 'gamma': 0.01}

Best cross-validation score: 0.97

Grid Search

|

273Let’s look at the cv\_results\_ again. As expected, if kernel is 'linear', then only C is

varied:

In[35]:

results = pd.DataFrame(grid\_search.cv\_results\_)

# we display the transposed table so that it better fits on the page:

display(results.T)

Out[35]:

param\_C0

0.0011

0.0012

0.0013

0.001… 38

… 0.139

140

1041

100

param\_gamma0.0010.010.11… NaNNaNNaNNaN

param\_kernelrbfrbfrbfrbf… linearlinearlinearlinear

params{C: 0.001,

kernel: rbf,

gamma:

0.001}{C: 0.001,

kernel: rbf,

gamma:

0.01}{C: 0.001,

kernel: rbf,

gamma:

0.1}{C: 0.001, … {C: 0.1,

kernel: rbf,

kernel:

gamma: 1}

linear}{C: 1,

kernel:

linear}{C: 10,

kernel:

linear}{C: 100,

kernel:

linear}

mean\_test\_score 0.370.370.370.37… 0.950.970.960.96

rank\_test\_score27272727… 11133

split0\_test\_score 0.380.380.380.38… 0.9610.960.96

split1\_test\_score 0.350.350.350.35… 0.910.9611

split2\_test\_score 0.360.360.360.36… 1111

split3\_test\_score 0.360.360.360.36… 0.910.950.910.91

split4\_test\_score 0.380.380.380.38… 0.950.950.950.95

std\_test\_score0.0110.0110.011… 0.0330.0220.0340.034

0.011

12 rows × 42 columns

Using different cross-validation strategies with grid search

Similarly to cross\_val\_score, GridSearchCV uses stratified k-fold cross-validation

by default for classification, and k-fold cross-validation for regression. However, you

can also pass any cross-validation splitter, as described in “More control over cross-

validation” on page 258, as the cv parameter in GridSearchCV. In particular, to get

only a single split into a training and a validation set, you can use ShuffleSplit or

StratifiedShuffleSplit with n\_splits=1. This might be helpful for very large

datasets, or very slow models.

Nested cross-validation

In the preceding examples, we went from using a single split of the data into training,

validation, and test sets to splitting the data into training and test sets and then per‐

forming cross-validation on the training set. But when using GridSearchCV as

274

|

Chapter 5: Model Evaluation and Improvementdescribed earlier, we still have a single split of the data into training and test sets,

which might make our results unstable and make us depend too much on this single

split of the data. We can go a step further, and instead of splitting the original data

into training and test sets once, use multiple splits of cross-validation. This will result

in what is called nested cross-validation. In nested cross-validation, there is an outer

loop over splits of the data into training and test sets. For each of them, a grid search

is run (which might result in different best parameters for each split in the outer

loop). Then, for each outer split, the test set score using the best settings is reported.

The result of this procedure is a list of scores—not a model, and not a parameter set‐

ting. The scores tell us how well a model generalizes, given the best parameters found

by the grid. As it doesn’t provide a model that can be used on new data, nested cross-

validation is rarely used when looking for a predictive model to apply to future data.

However, it can be useful for evaluating how well a given model works on a particular

dataset.

Implementing nested cross-validation in scikit-learn is straightforward. We call

cross\_val\_score with an instance of GridSearchCV as the model:

In[36]:

param\_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],

'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}

scores = cross\_val\_score(GridSearchCV(SVC(), param\_grid, cv=5),

iris.data, iris.target, cv=5)

print("Cross-validation scores: ", scores)

print("Mean cross-validation score: ", scores.mean())

Out[36]:

Cross-validation scores: [ 0.967 1.

Mean cross-validation score: 0.98

0.967

0.967

1.

]

The result of our nested cross-validation can be summarized as “SVC can achieve 98%

mean cross-validation accuracy on the iris dataset”—nothing more and nothing

less.

Here, we used stratified five-fold cross-validation in both the inner and the outer

loop. As our param\_grid contains 36 combinations of parameters, this results in a

whopping 36 \* 5 \* 5 = 900 models being built, making nested cross-validation a very

expensive procedure. Here, we used the same cross-validation splitter in the inner

and the outer loop; however, this is not necessary and you can use any combination

of cross-validation strategies in the inner and outer loops. It can be a bit tricky to

understand what is happening in the single line given above, and it can be helpful to

visualize it as for loops, as done in the following simplified implementation:

Grid Search

|

275In[37]:

def nested\_cv(X, y, inner\_cv, outer\_cv, Classifier, parameter\_grid):

outer\_scores = []

# for each split of the data in the outer cross-validation

# (split method returns indices of training and test parts)

for training\_samples, test\_samples in outer\_cv.split(X, y):

# find best parameter using inner cross-validation

best\_parms = {}

best\_score = -np.inf

# iterate over parameters

for parameters in parameter\_grid:

# accumulate score over inner splits

cv\_scores = []

# iterate over inner cross-validation

for inner\_train, inner\_test in inner\_cv.split(

X[training\_samples], y[training\_samples]):

# build classifier given parameters and training data

clf = Classifier(\*\*parameters)

clf.fit(X[inner\_train], y[inner\_train])

# evaluate on inner test set

score = clf.score(X[inner\_test], y[inner\_test])

cv\_scores.append(score)

# compute mean score over inner folds

mean\_score = np.mean(cv\_scores)

if mean\_score > best\_score:

# if better than so far, remember parameters

best\_score = mean\_score

best\_params = parameters

# build classifier on best parameters using outer training set

clf = Classifier(\*\*best\_params)

clf.fit(X[training\_samples], y[training\_samples])

# evaluate

outer\_scores.append(clf.score(X[test\_samples], y[test\_samples]))

return np.array(outer\_scores)

Now, let’s run this function on the iris dataset:

In[38]:

from sklearn.model\_selection import ParameterGrid, StratifiedKFold

scores = nested\_cv(iris.data, iris.target, StratifiedKFold(5),

StratifiedKFold(5), SVC, ParameterGrid(param\_grid))

print("Cross-validation scores: {}".format(scores))

Out[38]:

Cross-validation scores: [ 0.967

1.

0.967

0.967

1.

]

Parallelizing cross-validation and grid search

While running a grid search over many parameters and on large datasets can be com‐

putationally challenging, it is also embarrassingly parallel. This means that building a

276

|

Chapter 5: Model Evaluation and Improvementmodel using a particular parameter setting on a particular cross-validation split can

be done completely independently from the other parameter settings and models.

This makes grid search and cross-validation ideal candidates for parallelization over

multiple CPU cores or over a cluster. You can make use of multiple cores in Grid

SearchCV and cross\_val\_score by setting the n\_jobs parameter to the number of

CPU cores you want to use. You can set n\_jobs=-1 to use all available cores.

You should be aware that scikit-learn does not allow nesting of parallel operations.

So, if you are using the n\_jobs option on your model (for example, a random forest),

you cannot use it in GridSearchCV to search over this model. If your dataset and

model are very large, it might be that using many cores uses up too much memory,

and you should monitor your memory usage when building large models in parallel.

It is also possible to parallelize grid search and cross-validation over multiple

machines in a cluster, although at the time of writing this is not supported within

scikit-learn. It is, however, possible to use the IPython parallel framework for par‐

allel grid searches, if you don’t mind writing the for loop over parameters as we did

in “Simple Grid Search” on page 263.

For Spark users, there is also the recently developed spark-sklearn package, which

allows running a grid search over an already established Spark cluster.

Evaluation Metrics and Scoring

So far, we have evaluated classification performance using accuracy (the fraction of

correctly classified samples) and regression performance using R2. However, these are

only two of the many possible ways to summarize how well a supervised model per‐

forms on a given dataset. In practice, these evaluation metrics might not be appropri‐

ate for your application, and it is important to choose the right metric when selecting

between models and adjusting parameters.

Keep the End Goal in Mind

When selecting a metric, you should always have the end goal of the machine learn‐

ing application in mind. In practice, we are usually interested not just in making

accurate predictions, but in using these predictions as part of a larger decision-

making process. Before picking a machine learning metric, you should think about

the high-level goal of the application, often called the business metric. The conse‐

quences of choosing a particular algorithm for a machine learning application are

Evaluation Metrics and Scoring

|

277called the business impact.2 Maybe the high-level goal is avoiding traffic accidents, or

decreasing the number of hospital admissions. It could also be getting more users for

your website, or having users spend more money in your shop. When choosing a

model or adjusting parameters, you should pick the model or parameter values that

have the most positive influence on the business metric. Often this is hard, as assess‐

ing the business impact of a particular model might require putting it in production

in a real-life system.

In the early stages of development, and for adjusting parameters, it is often infeasible

to put models into production just for testing purposes, because of the high business

or personal risks that can be involved. Imagine evaluating the pedestrian avoidance

capabilities of a self-driving car by just letting it drive around, without verifying it

first; if your model is bad, pedestrians will be in trouble! Therefore we often need to

find some surrogate evaluation procedure, using an evaluation metric that is easier to

compute. For example, we could test classifying images of pedestrians against non-

pedestrians and measure accuracy. Keep in mind that this is only a surrogate, and it

pays off to find the closest metric to the original business goal that is feasible to evalu‐

ate. This closest metric should be used whenever possible for model evaluation and

selection. The result of this evaluation might not be a single number—the conse‐

quence of your algorithm could be that you have 10% more customers, but each cus‐

tomer will spend 15% less—but it should capture the expected business impact of

choosing one model over another.

In this section, we will first discuss metrics for the important special case of binary

classification, then turn to multiclass classification and finally regression.

Metrics for Binary Classification

Binary classification is arguably the most common and conceptually simple applica‐

tion of machine learning in practice. However, there are still a number of caveats in

evaluating even this simple task. Before we dive into alternative metrics, let’s have a

look at the ways in which measuring accuracy might be misleading. Remember that

for binary classification, we often speak of a positive class and a negative class, with

the understanding that the positive class is the one we are looking for.

Kinds of errors

Often, accuracy is not a good measure of predictive performance, as the number of

mistakes we make does not contain all the information we are interested in. Imagine

an application to screen for the early detection of cancer using an automated test. If

2 We ask scientifically minded readers to excuse the commercial language in this section. Not losing track of the

end goal is equally important in science, though the authors are not aware of a similar phrase to “business

impact” being used in that realm.

278

| Chapter 5: Model Evaluation and Improvementthe test is negative, the patient will be assumed healthy, while if the test is positive, the

patient will undergo additional screening. Here, we would call a positive test (an indi‐

cation of cancer) the positive class, and a negative test the negative class. We can’t

assume that our model will always work perfectly, and it will make mistakes. For any

application, we need to ask ourselves what the consequences of these mistakes might

be in the real world.

One possible mistake is that a healthy patient will be classified as positive, leading to

additional testing. This leads to some costs and an inconvenience for the patient (and

possibly some mental distress). An incorrect positive prediction is called a false posi‐

tive. The other possible mistake is that a sick patient will be classified as negative, and

will not receive further tests and treatment. The undiagnosed cancer might lead to

serious health issues, and could even be fatal. A mistake of this kind—an incorrect

negative prediction—is called a false negative. In statistics, a false positive is also

known as type I error, and a false negative as type II error. We will stick to “false nega‐

tive” and “false positive,” as they are more explicit and easier to remember. In the can‐

cer diagnosis example, it is clear that we want to avoid false negatives as much as

possible, while false positives can be viewed as more of a minor nuisance.

While this is a particularly drastic example, the consequence of false positives and

false negatives are rarely the same. In commercial applications, it might be possible to

assign dollar values to both kinds of mistakes, which would allow measuring the error

of a particular prediction in dollars, instead of accuracy. This might be much more

meaningful for making business decisions on which model to use.

Imbalanced datasets

Types of errors play an important role when one of two classes is much more frequent

than the other one. This is very common in practice; a good example is click-through

prediction, where each data point represents an “impression,” an item that was shown

to a user. This item might be an ad, or a related story, or a related person to follow on

a social media site. The goal is to predict whether, if shown a particular item, a user

will click on it (indicating they are interested). Most things users are shown on the

Internet (in particular, ads) will not result in a click. You might need to show a user

100 ads or articles before they find something interesting enough to click on. This

results in a dataset where for each 99 “no click” data points, there is 1 “clicked” data

point; in other words, 99% of the samples belong to the “no click” class. Datasets in

which one class is much more frequent than the other are often called imbalanced

datasets, or datasets with imbalanced classes. In reality, imbalanced data is the norm,

and it is rare that the events of interest have equal or even similar frequency in the

data.

Now let’s say you build a classifier that is 99% accurate on the click prediction task.

What does that tell you? 99% accuracy sounds impressive, but this doesn’t take the

Evaluation Metrics and Scoring

|

279class imbalance into account. You can achieve 99% accuracy without building a

machine learning model, by always predicting “no click.” On the other hand, even

with imbalanced data, a 99% accurate model could in fact be quite good. However,

accuracy doesn’t allow us to distinguish the constant “no click” model from a poten‐

tially good model.

To illustrate, we’ll create a 9:1 imbalanced dataset from the digits dataset, by classify‐

ing the digit 9 against the nine other classes:

In[39]:

from sklearn.datasets import load\_digits

digits = load\_digits()

y = digits.target == 9

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

digits.data, y, random\_state=0)

We can use the DummyClassifier to always predict the majority class (here

“not nine”) to see how uninformative accuracy can be:

In[40]:

from sklearn.dummy import DummyClassifier

dummy\_majority = DummyClassifier(strategy='most\_frequent').fit(X\_train, y\_train)

pred\_most\_frequent = dummy\_majority.predict(X\_test)

print("Unique predicted labels: {}".format(np.unique(pred\_most\_frequent)))

print("Test score: {:.2f}".format(dummy\_majority.score(X\_test, y\_test)))

Out[40]:

Unique predicted labels: [False]

Test score: 0.90

We obtained close to 90% accuracy without learning anything. This might seem strik‐

ing, but think about it for a minute. Imagine someone telling you their model is 90%

accurate. You might think they did a very good job. But depending on the problem,

that might be possible by just predicting one class! Let’s compare this against using an

actual classifier:

In[41]:

from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier(max\_depth=2).fit(X\_train, y\_train)

pred\_tree = tree.predict(X\_test)

print("Test score: {:.2f}".format(tree.score(X\_test, y\_test)))

Out[41]:

Test score: 0.92

280

|

Chapter 5: Model Evaluation and ImprovementAccording to accuracy, the DecisionTreeClassifier is only slightly better than the

constant predictor. This could indicate either that something is wrong with how we

used DecisionTreeClassifier, or that accuracy is in fact not a good measure here.

For comparison purposes, let’s evaluate two more classifiers, LogisticRegression

and the default DummyClassifier, which makes random predictions but produces

classes with the same proportions as in the training set:

In[42]:

from sklearn.linear\_model import LogisticRegression

dummy = DummyClassifier().fit(X\_train, y\_train)

pred\_dummy = dummy.predict(X\_test)

print("dummy score: {:.2f}".format(dummy.score(X\_test, y\_test)))

logreg = LogisticRegression(C=0.1).fit(X\_train, y\_train)

pred\_logreg = logreg.predict(X\_test)

print("logreg score: {:.2f}".format(logreg.score(X\_test, y\_test)))

Out[42]:

dummy score: 0.80

logreg score: 0.98

The dummy classifier that produces random output is clearly the worst of the lot

(according to accuracy), while LogisticRegression produces very good results.

However, even the random classifier yields over 80% accuracy. This makes it very

hard to judge which of these results is actually helpful. The problem here is that accu‐

racy is an inadequate measure for quantifying predictive performance in this imbal‐

anced setting. For the rest of this chapter, we will explore alternative metrics that

provide better guidance in selecting models. In particular, we would like to have met‐

rics that tell us how much better a model is than making “most frequent” predictions

or random predictions, as they are computed in pred\_most\_frequent and

pred\_dummy. If we use a metric to assess our models, it should definitely be able to

weed out these nonsense predictions.

Confusion matrices

One of the most comprehensive ways to represent the result of evaluating binary clas‐

sification is using confusion matrices. Let’s inspect the predictions of LogisticRegres

sion from the previous section using the confusion\_matrix function. We already

stored the predictions on the test set in pred\_logreg:

Evaluation Metrics and Scoring

|

281In[43]:

from sklearn.metrics import confusion\_matrix

confusion = confusion\_matrix(y\_test, pred\_logreg)

print("Confusion matrix:\n{}".format(confusion))

Out[43]:

Confusion matrix:

[[401

2]

[ 8 39]]

The output of confusion\_matrix is a two-by-two array, where the rows correspond

to the true classes and the columns correspond to the predicted classes. Each entry

counts how often a sample that belongs to the class corresponding to the row (here,

“not nine” and “nine”) was classified as the class corresponding to the column. The

following plot (Figure 5-10) illustrates this meaning:

In[44]:

mglearn.plots.plot\_confusion\_matrix\_illustration()

Figure 5-10. Confusion matrix of the “nine vs. rest” classification task

282

|

Chapter 5: Model Evaluation and ImprovementEntries on the main diagonal3 of the confusion matrix correspond to correct classifi‐

cations, while other entries tell us how many samples of one class got mistakenly clas‐

sified as another class.

If we declare “a nine” the positive class, we can relate the entries of the confusion

matrix with the terms false positive and false negative that we introduced earlier. To

complete the picture, we call correctly classified samples belonging to the positive

class true positives and correctly classified samples belonging to the negative class true

negatives. These terms are usually abbreviated FP, FN, TP, and TN and lead to the fol‐

lowing interpretation for the confusion matrix (Figure 5-11):

In[45]:

mglearn.plots.plot\_binary\_confusion\_matrix()

Figure 5-11. Confusion matrix for binary classification

Now let’s use the confusion matrix to compare the models we fitted earlier (the two

dummy models, the decision tree, and the logistic regression):

In[46]:

print("Most frequent class:")

print(confusion\_matrix(y\_test, pred\_most\_frequent))

print("\nDummy model:")

print(confusion\_matrix(y\_test, pred\_dummy))

print("\nDecision tree:")

print(confusion\_matrix(y\_test, pred\_tree))

print("\nLogistic Regression")

print(confusion\_matrix(y\_test, pred\_logreg))

3 The main diagonal of a two-dimensional array or matrix A is A[i, i].

Evaluation Metrics and Scoring

|

283Out[46]:

Most frequent class:

[[403

0]

[ 47

0]]

Dummy model:

[[361 42]

[ 43

4]]

Decision tree:

[[390 13]

[ 24 23]]

Logistic Regression

[[401

2]

[ 8 39]]

Looking at the confusion matrix, it is quite clear that something is wrong with

pred\_most\_frequent, because it always predicts the same class. pred\_dummy, on the

other hand, has a very small number of true positives (4), particularly compared to

the number of false negatives and false positives—there are many more false positives

than true positives! The predictions made by the decision tree make much more

sense than the dummy predictions, even though the accuracy was nearly the same.

Finally, we can see that logistic regression does better than pred\_tree in all aspects: it

has more true positives and true negatives while having fewer false positives and false

negatives. From this comparison, it is clear that only the decision tree and the logistic

regression give reasonable results, and that the logistic regression works better than

the tree on all accounts. However, inspecting the full confusion matrix is a bit cum‐

bersome, and while we gained a lot of insight from looking at all aspects of the

matrix, the process was very manual and qualitative. There are several ways to sum‐

marize the information in the confusion matrix, which we will discuss next.

Relation to accuracy. We already saw one way to summarize the result in the confu‐

sion matrix—by computing accuracy, which can be expressed as:

Accuracy =

TP+TN

TP+TN + FP + FN

In other words, accuracy is the number of correct predictions (TP and TN) divided

by the number of all samples (all entries of the confusion matrix summed up).

Precision, recall, and f-score. There are several other ways to summarize the confusion

matrix, with the most common ones being precision and recall. Precision measures

how many of the samples predicted as positive are actually positive:

284

|

Chapter 5: Model Evaluation and ImprovementPrecision =

TP

TP+FP

Precision is used as a performance metric when the goal is to limit the number of

false positives. As an example, imagine a model for predicting whether a new drug

will be effective in treating a disease in clinical trials. Clinical trials are notoriously

expensive, and a pharmaceutical company will only want to run an experiment if it is

very sure that the drug will actually work. Therefore, it is important that the model

does not produce many false positives—in other words, that it has a high precision.

Precision is also known as positive predictive value (PPV).

Recall, on the other hand, measures how many of the positive samples are captured

by the positive predictions:

Recall =

TP

TP+FN

Recall is used as performance metric when we need to identify all positive samples;

that is, when it is important to avoid false negatives. The cancer diagnosis example

from earlier in this chapter is a good example for this: it is important to find all peo‐

ple that are sick, possibly including healthy patients in the prediction. Other names

for recall are sensitivity, hit rate, or true positive rate (TPR).

There is a trade-off between optimizing recall and optimizing precision. You can triv‐

ially obtain a perfect recall if you predict all samples to belong to the positive class—

there will be no false negatives, and no true negatives either. However, predicting all

samples as positive will result in many false positives, and therefore the precision will

be very low. On the other hand, if you find a model that predicts only the single data

point it is most sure about as positive and the rest as negative, then precision will be

perfect (assuming this data point is in fact positive), but recall will be very bad.

Precision and recall are only two of many classification measures

derived from TP, FP, TN, and FN. You can find a great summary of

all the measures on Wikipedia. In the machine learning commu‐

nity, precision and recall are arguably the most commonly used

measures for binary classification, but other communities might

use other related metrics.

So, while precision and recall are very important measures, looking at only one of

them will not provide you with the full picture. One way to summarize them is the

f-score or f-measure, which is with the harmonic mean of precision and recall:

F=2·

precision·recall

precision+recall

Evaluation Metrics and Scoring

|

285This particular variant is also known as the f1-score. As it takes precision and recall

into account, it can be a better measure than accuracy on imbalanced binary classifi‐

cation datasets. Let’s run it on the predictions for the “nine vs. rest” dataset that we

computed earlier. Here, we will assume that the “nine” class is the positive class (it is

labeled as True while the rest is labeled as False), so the positive class is the minority

class:

In[47]:

from sklearn.metrics import f1\_score

print("f1 score most frequent: {:.2f}".format(

f1\_score(y\_test, pred\_most\_frequent)))

print("f1 score dummy: {:.2f}".format(f1\_score(y\_test, pred\_dummy)))

print("f1 score tree: {:.2f}".format(f1\_score(y\_test, pred\_tree)))

print("f1 score logistic regression: {:.2f}".format(

f1\_score(y\_test, pred\_logreg)))

Out[47]:

f1 score most frequent: 0.00

f1 score dummy: 0.10

f1 score tree: 0.55

f1 score logistic regression: 0.89

We can note two things here. First, we get an error message for the most\_frequent

prediction, as there were no predictions of the positive class (which makes the

denominator in the f-score zero). Also, we can see a pretty strong distinction between

the dummy predictions and the tree predictions, which wasn’t clear when looking at

accuracy alone. Using the f-score for evaluation, we summarized the predictive per‐

formance again in one number. However, the f-score seems to capture our intuition

of what makes a good model much better than accuracy did. A disadvantage of the

f-score, however, is that it is harder to interpret and explain than accuracy.

If we want a more comprehensive summary of precision, recall, and f1-score, we can

use the classification\_report convenience function to compute all three at once,

and print them in a nice format:

In[48]:

from sklearn.metrics import classification\_report

print(classification\_report(y\_test, pred\_most\_frequent,

target\_names=["not nine", "nine"]))

286

|

Chapter 5: Model Evaluation and ImprovementOut[48]:

precisionrecallf1-scoresupport

not nine

nine0.90

0.001.00

0.000.94

0.00403

47

avg / total0.800.900.85450

The classification\_report function produces one line per class (here, True and

False) and reports precision, recall, and f-score with this class as the positive class.

Before, we assumed the minority “nine” class was the positive class. If we change the

positive class to “not nine,” we can see from the output of classification\_report

that we obtain an f-score of 0.94 with the most\_frequent model. Furthermore, for the

“not nine” class we have a recall of 1, as we classified all samples as “not nine.” The

last column next to the f-score provides the support of each class, which simply means

the number of samples in this class according to the ground truth.

The last row in the classification report shows a weighted (by the number of samples

in the class) average of the numbers for each class. Here are two more reports, one for

the dummy classifier and one for the logistic regression:

In[49]:

print(classification\_report(y\_test, pred\_dummy,

target\_names=["not nine", "nine"]))

Out[49]:

precisionrecallf1-scoresupport

not nine

nine0.90

0.110.92

0.090.91

0.10403

47

avg / total0.810.830.82450

In[50]:

print(classification\_report(y\_test, pred\_logreg,

target\_names=["not nine", "nine"]))

Out[50]:

precisionrecallf1-scoresupport

not nine

nine0.98

0.951.00

0.830.99

0.89403

47

avg / total0.980.980.98450

Evaluation Metrics and Scoring

|

287As you may notice when looking at the reports, the differences between the dummy

models and a very good model are not as clear any more. Picking which class is

declared the positive class has a big impact on the metrics. While the f-score for the

dummy classification is 0.10 (vs. 0.89 for the logistic regression) on the “nine” class,

for the “not nine” class it is 0.91 vs. 0.99, which both seem like reasonable results.

Looking at all the numbers together paints a pretty accurate picture, though, and we

can clearly see the superiority of the logistic regression model.

Taking uncertainty into account

The confusion matrix and the classification report provide a very detailed analysis of

a particular set of predictions. However, the predictions themselves already threw

away a lot of information that is contained in the model. As we discussed in Chap‐

ter 2, most classifiers provide a decision\_function or a predict\_proba method to

assess degrees of certainty about predictions. Making predictions can be seen as

thresholding the output of decision\_function or predict\_proba at a certain fixed

point—in binary classification we use 0 for the decision function and 0.5 for

predict\_proba.

The following is an example of an imbalanced binary classification task, with 400

points in the negative class classified against 50 points in the positive class. The train‐

ing data is shown on the left in Figure 5-12. We train a kernel SVM model on this

data, and the plots to the right of the training data illustrate the values of the decision

function as a heat map. You can see a black circle in the plot in the top center, which

denotes the threshold of the decision\_function being exactly zero. Points inside this

circle will be classified as the positive class, and points outside as the negative class:

In[51]:

from mglearn.datasets import make\_blobs

X, y = make\_blobs(n\_samples=(400, 50), centers=2, cluster\_std=[7.0, 2],

random\_state=22)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)

svc = SVC(gamma=.05).fit(X\_train, y\_train)

In[52]:

mglearn.plots.plot\_decision\_threshold()

288

|

Chapter 5: Model Evaluation and ImprovementFigure 5-12. Heatmap of the decision function and the impact of changing the decision

threshold

We can use the classification\_report function to evaluate precision and recall for

both classes:

In[53]:

print(classification\_report(y\_test, svc.predict(X\_test)))

Out[53]:

precisionrecallf1-scoresupport

0

10.97

0.350.89

0.670.93

0.46104

9

avg / total0.920.880.89113

For class 1, we get a fairly small precision, and recall is mixed. Because class 0 is so

much larger, the classifier focuses on getting class 0 right, and not the smaller class 1.

Let’s assume in our application it is more important to have a high recall for class 1, as

in the cancer screening example earlier. This means we are willing to risk more false

positives (false class 1) in exchange for more true positives (which will increase the

recall). The predictions generated by svc.predict really do not fulfill this require‐

ment, but we can adjust the predictions to focus on a higher recall of class 1 by

changing the decision threshold away from 0. By default, points with a deci

sion\_function value greater than 0 will be classified as class 1. We want more points

to be classified as class 1, so we need to decrease the threshold:

Evaluation Metrics and Scoring

|

289In[54]:

y\_pred\_lower\_threshold = svc.decision\_function(X\_test) > -.8

Let’s look at the classification report for this prediction:

In[55]:

print(classification\_report(y\_test, y\_pred\_lower\_threshold))

Out[55]:

precisionrecallf1-scoresupport

0

11.00

0.320.82

1.000.90

0.49104

9

avg / total0.950.830.87113

As expected, the recall of class 1 went up, and the precision went down. We are now

classifying a larger region of space as class 1, as illustrated in the top-right panel of

Figure 5-12. If you value precision over recall or the other way around, or your data is

heavily imbalanced, changing the decision threshold is the easiest way to obtain bet‐

ter results. As the decision\_function can have arbitrary ranges, it is hard to provide

a rule of thumb regarding how to pick a threshold.

For simplicity, we changed the threshold value based on test set

results in the code above. In practice, you need to use a hold-out

validation set, not the test set. As with any other parameter, setting

a decision threshold on the test set is likely to yield overly optimis‐

tic results. Use a validation set or cross-validation instead.

Picking a threshold for models that implement the predict\_proba method can be

easier, as the output of predict\_proba is on a fixed 0 to 1 scale, and models probabil‐

ities. By default, the threshold of 0.5 means that if the model is more than 50% “sure”

that a point is of the positive class, it will be classified as such. Increasing the thres‐

hold means that the model needs to be more confident to make a positive decision

(and less confident to make a negative decision). While working with probabilities

may be more intuitive than working with arbitrary thresholds, not all models provide

realistic models of uncertainty (a DecisionTree that is grown to its full depth is

always 100% sure of its decisions, even though it might often be wrong). This relates

to the concept of calibration: a calibrated model is a model that provides an accurate

measure of its uncertainty. Discussing calibration in detail is beyond the scope of this

book, but you can find more details in the paper “Predicting Good Probabilities with

Supervised Learning” by Alexandru Niculescu-Mizil and Rich Caruana.

290

|

Chapter 5: Model Evaluation and ImprovementPrecision-recall curves and ROC curves

As we just discussed, changing the threshold that is used to make a classification deci‐

sion in a model is a way to adjust the trade-off of precision and recall for a given clas‐

sifier. Maybe you want to miss less than 10% of positive samples, meaning a desired

recall of 90%. This decision depends on the application, and it should be driven by

business goals. Once a particular goal is set—say, a particular recall or precision value

for a class—a threshold can be set appropriately. It is always possible to set a thres‐

hold to fulfill a particular target, like 90% recall. The hard part is to develop a model

that still has reasonable precision with this threshold—if you classify everything as

positive, you will have 100% recall, but your model will be useless.

Setting a requirement on a classifier like 90% recall is often called setting the operat‐

ing point. Fixing an operating point is often helpful in business settings to make per‐

formance guarantees to customers or other groups inside your organization.

Often, when developing a new model, it is not entirely clear what the operating point

will be. For this reason, and to understand a modeling problem better, it is instructive

to look at all possible thresholds, or all possible trade-offs of precision and recalls at

once. This is possible using a tool called the precision-recall curve. You can find the

function to compute the precision-recall curve in the sklearn.metrics module. It

needs the ground truth labeling and predicted uncertainties, created via either

decision\_function or predict\_proba:

In[56]:

from sklearn.metrics import precision\_recall\_curve

precision, recall, thresholds = precision\_recall\_curve(

y\_test, svc.decision\_function(X\_test))

The precision\_recall\_curve function returns a list of precision and recall values

for all possible thresholds (all values that appear in the decision function) in sorted

order, so we can plot a curve, as seen in Figure 5-13:

In[57]:

# Use more data points for a smoother curve

X, y = make\_blobs(n\_samples=(4000, 500), centers=2, cluster\_std=[7.0, 2],

random\_state=22)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)

svc = SVC(gamma=.05).fit(X\_train, y\_train)

precision, recall, thresholds = precision\_recall\_curve(

y\_test, svc.decision\_function(X\_test))

# find threshold closest to zero

close\_zero = np.argmin(np.abs(thresholds))

plt.plot(precision[close\_zero], recall[close\_zero], 'o', markersize=10,

label="threshold zero", fillstyle="none", c='k', mew=2)

plt.plot(precision, recall, label="precision recall curve")

plt.xlabel("Precision")

plt.ylabel("Recall")

plt.legend(loc="best")

Evaluation Metrics and Scoring

|

291Figure 5-13. Precision recall curve for SVC(gamma=0.05)

Each point along the curve in Figure 5-13 corresponds to a possible threshold of the

decision\_function. We can see, for example, that we can achieve a recall of 0.4 at a

precision of about 0.75. The black circle marks the point that corresponds to a thres‐

hold of 0, the default threshold for decision\_function. This point is the trade-off

that is chosen when calling the predict method.

The closer a curve stays to the upper-right corner, the better the classifier. A point at

the upper right means high precision and high recall for the same threshold. The

curve starts at the top-left corner, corresponding to a very low threshold, classifying

everything as the positive class. Raising the threshold moves the curve toward higher

precision, but also lower recall. Raising the threshold more and more, we get to a sit‐

uation where most of the points classified as being positive are true positives, leading

to a very high precision but lower recall. The more the model keeps recall high as

precision goes up, the better.

Looking at this particular curve a bit more, we can see that with this model it is possi‐

ble to get a precision of up to around 0.5 with very high recall. If we want a much

higher precision, we have to sacrifice a lot of recall. In other words, on the left the

curve is relatively flat, meaning that recall does not go down a lot when we require

increased precision. For precision greater than 0.5, each gain in precision costs us a

lot of recall.

Different classifiers can work well in different parts of the curve—that is, at different

operating points. Let’s compare the SVM we trained to a random forest trained on the

same dataset. The RandomForestClassifier doesn’t have a decision\_function, only

predict\_proba. The precision\_recall\_curve function expects as its second argu‐

ment a certainty measure for the positive class (class 1), so we pass the probability of

a sample being class 1—that is, rf.predict\_proba(X\_test)[:, 1]. The default

threshold for predict\_proba in binary classification is 0.5, so this is the point we

marked on the curve (see Figure 5-14):

292

|

Chapter 5: Model Evaluation and ImprovementIn[58]:

from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n\_estimators=100, random\_state=0, max\_features=2)

rf.fit(X\_train, y\_train)

# RandomForestClassifier has predict\_proba, but not decision\_function

precision\_rf, recall\_rf, thresholds\_rf = precision\_recall\_curve(

y\_test, rf.predict\_proba(X\_test)[:, 1])

plt.plot(precision, recall, label="svc")

plt.plot(precision[close\_zero], recall[close\_zero], 'o', markersize=10,

label="threshold zero svc", fillstyle="none", c='k', mew=2)

plt.plot(precision\_rf, recall\_rf, label="rf")

close\_default\_rf = np.argmin(np.abs(thresholds\_rf - 0.5))

plt.plot(precision\_rf[close\_default\_rf], recall\_rf[close\_default\_rf], '^', c='k',

markersize=10, label="threshold 0.5 rf", fillstyle="none", mew=2)

plt.xlabel("Precision")

plt.ylabel("Recall")

plt.legend(loc="best")

Figure 5-14. Comparing precision recall curves of SVM and random forest

From the comparison plot we can see that the random forest performs better at the

extremes, for very high recall or very high precision requirements. Around the mid‐

dle (approximately precision=0.7), the SVM performs better. If we only looked at the

f1-score to compare overall performance, we would have missed these subtleties. The

f1-score only captures one point on the precision-recall curve, the one given by the

default threshold:

Evaluation Metrics and Scoring

|

293In[59]:

print("f1\_score of random forest: {:.3f}".format(

f1\_score(y\_test, rf.predict(X\_test))))

print("f1\_score of svc: {:.3f}".format(f1\_score(y\_test, svc.predict(X\_test))))

Out[59]:

f1\_score of random forest: 0.610

f1\_score of svc: 0.656

Comparing two precision-recall curves provides a lot of detailed insight, but is a fairly

manual process. For automatic model comparison, we might want to summarize the

information contained in the curve, without limiting ourselves to a particular thres‐

hold or operating point. One particular way to summarize the precision-recall curve

is by computing the integral or area under the curve of the precision-recall curve, also

known as the average precision.4 You can use the average\_precision\_score function

to compute the average precision. Because we need to compute the precision-recall

curve and consider multiple thresholds, the result of decision\_function or pre

dict\_proba needs to be passed to average\_precision\_score, not the result of

predict:

In[60]:

from sklearn.metrics import average\_precision\_score

ap\_rf = average\_precision\_score(y\_test, rf.predict\_proba(X\_test)[:, 1])

ap\_svc = average\_precision\_score(y\_test, svc.decision\_function(X\_test))

print("Average precision of random forest: {:.3f}".format(ap\_rf))

print("Average precision of svc: {:.3f}".format(ap\_svc))

Out[60]:

Average precision of random forest: 0.666

Average precision of svc: 0.663

When averaging over all possible thresholds, we see that the random forest and SVC

perform similarly well, with the random forest even slightly ahead. This is quite dif‐

ferent from the result we got from f1\_score earlier. Because average precision is the

area under a curve that goes from 0 to 1, average precision always returns a value

between 0 (worst) and 1 (best). The average precision of a classifier that assigns

decision\_function at random is the fraction of positive samples in the dataset.

Receiver operating characteristics (ROC) and AUC

There is another tool that is commonly used to analyze the behavior of classifiers at

different thresholds: the receiver operating characteristics curve, or ROC curve for

4 There are some minor technical differences between the area under the precision-recall curve and average

precision. However, this explanation conveys the general idea.

294

|

Chapter 5: Model Evaluation and Improvementshort. Similar to the precision-recall curve, the ROC curve considers all possible

thresholds for a given classifier, but instead of reporting precision and recall, it shows

the false positive rate (FPR) against the true positive rate (TPR). Recall that the true

positive rate is simply another name for recall, while the false positive rate is the frac‐

tion of false positives out of all negative samples:

FPR =

FP

FP+TN

The ROC curve can be computed using the roc\_curve function (see Figure 5-15):

In[61]:

from sklearn.metrics import roc\_curve

fpr, tpr, thresholds = roc\_curve(y\_test, svc.decision\_function(X\_test))

plt.plot(fpr, tpr, label="ROC Curve")

plt.xlabel("FPR")

plt.ylabel("TPR (recall)")

# find threshold closest to zero

close\_zero = np.argmin(np.abs(thresholds))

plt.plot(fpr[close\_zero], tpr[close\_zero], 'o', markersize=10,

label="threshold zero", fillstyle="none", c='k', mew=2)

plt.legend(loc=4)

Figure 5-15. ROC curve for SVM

For the ROC curve, the ideal curve is close to the top left: you want a classifier that

produces a high recall while keeping a low false positive rate. Compared to the default

threshold of 0, the curve shows that we can achieve a significantly higher recall

(around 0.9) while only increasing the FPR slightly. The point closest to the top left

might be a better operating point than the one chosen by default. Again, be aware that

choosing a threshold should not be done on the test set, but on a separate validation

set.

Evaluation Metrics and Scoring

|

295You can find a comparison of the random forest and the SVM using ROC curves in

Figure 5-16:

In[62]:

fpr\_rf, tpr\_rf, thresholds\_rf = roc\_curve(y\_test, rf.predict\_proba(X\_test)[:, 1])

plt.plot(fpr, tpr, label="ROC Curve SVC")

plt.plot(fpr\_rf, tpr\_rf, label="ROC Curve RF")

plt.xlabel("FPR")

plt.ylabel("TPR (recall)")

plt.plot(fpr[close\_zero], tpr[close\_zero], 'o', markersize=10,

label="threshold zero SVC", fillstyle="none", c='k', mew=2)

close\_default\_rf = np.argmin(np.abs(thresholds\_rf - 0.5))

plt.plot(fpr\_rf[close\_default\_rf], tpr[close\_default\_rf], '^', markersize=10,

label="threshold 0.5 RF", fillstyle="none", c='k', mew=2)

plt.legend(loc=4)

Figure 5-16. Comparing ROC curves for SVM and random forest

As for the precision-recall curve, we often want to summarize the ROC curve using a

single number, the area under the curve (this is commonly just referred to as the

AUC, and it is understood that the curve in question is the ROC curve). We can com‐

pute the area under the ROC curve using the roc\_auc\_score function:

296

|

Chapter 5: Model Evaluation and ImprovementIn[63]:

from sklearn.metrics import roc\_auc\_score

rf\_auc = roc\_auc\_score(y\_test, rf.predict\_proba(X\_test)[:, 1])

svc\_auc = roc\_auc\_score(y\_test, svc.decision\_function(X\_test))

print("AUC for Random Forest: {:.3f}".format(rf\_auc))

print("AUC for SVC: {:.3f}".format(svc\_auc))

Out[63]:

AUC for Random Forest: 0.937

AUC for SVC: 0.916

Comparing the random forest and SVM using the AUC score, we find that the ran‐

dom forest performs quite a bit better than the SVM. Recall that because AUC is the

area under a curve that goes from 0 to 1, AUC always returns a value between 0

(worst) and 1 (best). Predicting randomly always produces an AUC of 0.5, no matter

how imbalanced the classes in a dataset are. This makes AUC a much better metric

for imbalanced classification problems than accuracy. The AUC can be interpreted as

evaluating the ranking of positive samples. It’s equivalent to the probability that a ran‐

domly picked point of the positive class will have a higher score according to the clas‐

sifier than a randomly picked point from the negative class. So, a perfect AUC of 1

means that all positive points have a higher score than all negative points. For classifi‐

cation problems with imbalanced classes, using AUC for model selection is often

much more meaningful than using accuracy.

Let’s go back to the problem we studied earlier of classifying all nines in the digits

dataset versus all other digits. We will classify the dataset with an SVM with three dif‐

ferent settings of the kernel bandwidth, gamma (see Figure 5-17):

In[64]:

y = digits.target == 9

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

digits.data, y, random\_state=0)

plt.figure()

for gamma in [1, 0.05, 0.01]:

svc = SVC(gamma=gamma).fit(X\_train, y\_train)

accuracy = svc.score(X\_test, y\_test)

auc = roc\_auc\_score(y\_test, svc.decision\_function(X\_test))

fpr, tpr, \_ = roc\_curve(y\_test , svc.decision\_function(X\_test))

print("gamma = {:.2f} accuracy = {:.2f} AUC = {:.2f}".format(

gamma, accuracy, auc))

plt.plot(fpr, tpr, label="gamma={:.3f}".format(gamma))

plt.xlabel("FPR")

plt.ylabel("TPR")

plt.xlim(-0.01, 1)

plt.ylim(0, 1.02)

plt.legend(loc="best")

Evaluation Metrics and Scoring

|

297Out[64]:

gamma = 1.00 accuracy = 0.90 AUC = 0.50

gamma = 0.05 accuracy = 0.90 AUC = 0.90

gamma = 0.01 accuracy = 0.90 AUC = 1.00

Figure 5-17. Comparing ROC curves of SVMs with different settings of gamma

The accuracy of all three settings of gamma is the same, 90%. This might be the same

as chance performance, or it might not. Looking at the AUC and the corresponding

curve, however, we see a clear distinction between the three models. With gamma=1.0,

the AUC is actually at chance level, meaning that the output of the decision\_func

tion is as good as random. With gamma=0.05, performance drastically improves to an

AUC of 0.9. Finally, with gamma=0.01, we get a perfect AUC of 1.0. That means that

all positive points are ranked higher than all negative points according to the decision

function. In other words, with the right threshold, this model can classify the data

perfectly!5 Knowing this, we can adjust the threshold on this model and obtain great

predictions. If we had only used accuracy, we would never have discovered this.

For this reason, we highly recommend using AUC when evaluating models on imbal‐

anced data. Keep in mind that AUC does not make use of the default threshold,

though, so adjusting the decision threshold might be necessary to obtain useful classi‐

fication results from a model with a high AUC.

Metrics for Multiclass Classification

Now that we have discussed evaluation of binary classification tasks in depth, let’s

move on to metrics to evaluate multiclass classification. Basically, all metrics for

multiclass classification are derived from binary classification metrics, but averaged

5 Looking at the curve for gamma=0.01 in detail, you can see a small kink close to the top left. That means that at

least one point was not ranked correctly. The AUC of 1.0 is a consequence of rounding to the second decimal

point.

298

| Chapter 5: Model Evaluation and Improvementover all classes. Accuracy for multiclass classification is again defined as the fraction

of correctly classified examples. And again, when classes are imbalanced, accuracy is

not a great evaluation measure. Imagine a three-class classification problem with 85%

of points belonging to class A, 10% belonging to class B, and 5% belonging to class C.

What does being 85% accurate mean on this dataset? In general, multiclass classifica‐

tion results are harder to understand than binary classification results. Apart from

accuracy, common tools are the confusion matrix and the classification report we saw

in the binary case in the previous section. Let’s apply these two detailed evaluation

methods on the task of classifying the 10 different handwritten digits in the digits

dataset:

In[65]:

from sklearn.metrics import accuracy\_score

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

digits.data, digits.target, random\_state=0)

lr = LogisticRegression().fit(X\_train, y\_train)

pred = lr.predict(X\_test)

print("Accuracy: {:.3f}".format(accuracy\_score(y\_test, pred)))

print("Confusion matrix:\n{}".format(confusion\_matrix(y\_test, pred)))

Out[65]:

Accuracy: 0.953

Confusion matrix:

[[37 0 0 0 0 0 0 0 0 0]

[ 0 39 0 0 0 0 2 0 2 0]

[ 0 0 41 3 0 0 0 0 0 0]

[ 0 0 1 43 0 0 0 0 0 1]

[ 0 0 0 0 38 0 0 0 0 0]

[ 0 1 0 0 0 47 0 0 0 0]

[ 0 0 0 0 0 0 52 0 0 0]

[ 0 1 0 1 1 0 0 45 0 0]

[ 0 3 1 0 0 0 0 0 43 1]

[ 0 0 0 1 0 1 0 0 1 44]]

The model has an accuracy of 95.3%, which already tells us that we are doing pretty

well. The confusion matrix provides us with some more detail. As for the binary case,

each row corresponds to a true label, and each column corresponds to a predicted

label. You can find a visually more appealing plot in Figure 5-18:

In[66]:

scores\_image = mglearn.tools.heatmap(

confusion\_matrix(y\_test, pred), xlabel='Predicted label',

ylabel='True label', xticklabels=digits.target\_names,

yticklabels=digits.target\_names, cmap=plt.cm.gray\_r, fmt="%d")

plt.title("Confusion matrix")

plt.gca().invert\_yaxis()

Evaluation Metrics and Scoring

|

299Figure 5-18. Confusion matrix for the 10-digit classification task

For the first class, the digit 0, there are 37 samples in the class, and all of these sam‐

ples were classified as class 0 (there are no false negatives for class 0). We can see that

because all other entries in the first row of the confusion matrix are 0. We can also see

that no other digits were mistakenly classified as 0, because all other entries in the

first column of the confusion matrix are 0 (there are no false positives for class 0).

Some digits were confused with others, though—for example, the digit 2 (third row),

three of which were classified as the digit 3 (fourth column). There was also one digit

3 that was classified as 2 (third column, fourth row) and one digit 8 that was classified

as 2 (third column, ninth row).

With the classification\_report function, we can compute the precision, recall,

and f-score for each class:

In[67]:

print(classification\_report(y\_test, pred))

Out[67]:

300

precisionrecallf1-scoresupport

0

1

2

3

4

5

6

7

8

91.00

0.89

0.95

0.90

0.97

0.98

0.96

1.00

0.93

0.961.00

0.91

0.93

0.96

1.00

0.98

1.00

0.94

0.90

0.941.00

0.90

0.94

0.92

0.99

0.98

0.98

0.97

0.91

0.9537

43

44

45

38

48

52

48

48

47

avg / total0.950.950.95450

| Chapter 5: Model Evaluation and ImprovementUnsurprisingly, precision and recall are a perfect 1 for class 0, as there are no confu‐

sions with this class. For class 7, on the other hand, precision is 1 because no other

class was mistakenly classified as 7, while for class 6, there are no false negatives, so

the recall is 1. We can also see that the model has particular difficulties with classes 8

and 3.

The most commonly used metric for imbalanced datasets in the multiclass setting is

the multiclass version of the f-score. The idea behind the multiclass f-score is to com‐

pute one binary f-score per class, with that class being the positive class and the other

classes making up the negative classes. Then, these per-class f-scores are averaged

using one of the following strategies:

• "macro" averaging computes the unweighted per-class f-scores. This gives equal

weight to all classes, no matter what their size is.

• "weighted" averaging computes the mean of the per-class f-scores, weighted by

their support. This is what is reported in the classification report.

• "micro" averaging computes the total number of false positives, false negatives,

and true positives over all classes, and then computes precision, recall, and f-

score using these counts.

If you care about each sample equally much, it is recommended to use the "micro"

average f1-score; if you care about each class equally much, it is recommended to use

the "macro" average f1-score:

In[68]:

print("Micro average f1 score: {:.3f}".format(

f1\_score(y\_test, pred, average="micro")))

print("Macro average f1 score: {:.3f}".format(

f1\_score(y\_test, pred, average="macro")))

Out[68]:

Micro average f1 score: 0.953

Macro average f1 score: 0.954

Regression Metrics

Evaluation for regression can be done in similar detail as we did for classification—

for example, by analyzing overpredicting the target versus underpredicting the target.

However, in most applications we’ve seen, using the default R2 used in the score

method of all regressors is enough. Sometimes business decisions are made on the

basis of mean squared error or mean absolute error, which might give incentive to

tune models using these metrics. In general, though, we have found R2 to be a more

intuitive metric to evaluate regression models.

Evaluation Metrics and Scoring

|

301Using Evaluation Metrics in Model Selection

We have discussed many evaluation methods in detail, and how to apply them given

the ground truth and a model. However, we often want to use metrics like AUC in

model selection using GridSearchCV or cross\_val\_score. Luckily scikit-learn

provides a very simple way to achieve this, via the scoring argument that can be used

in both GridSearchCV and cross\_val\_score. You can simply provide a string

describing the evaluation metric you want to use. Say, for example, we want to evalu‐

ate the SVM classifier on the “nine vs. rest” task on the digits dataset, using the AUC

score. Changing the score from the default (accuracy) to AUC can be done by provid‐

ing "roc\_auc" as the scoring parameter:

In[69]:

# default scoring for classification is accuracy

print("Default scoring: {}".format(

cross\_val\_score(SVC(), digits.data, digits.target == 9)))

# providing scoring="accuracy" doesn't change the results

explicit\_accuracy = cross\_val\_score(SVC(), digits.data, digits.target == 9,

scoring="accuracy")

print("Explicit accuracy scoring: {}".format(explicit\_accuracy))

roc\_auc = cross\_val\_score(SVC(), digits.data, digits.target == 9,

scoring="roc\_auc")

print("AUC scoring: {}".format(roc\_auc))

Out[69]:

Default scoring: [ 0.9 0.9 0.9]

Explicit accuracy scoring: [ 0.9 0.9

AUC scoring: [ 0.994 0.99

0.996]

0.9]

Similarly, we can change the metric used to pick the best parameters in Grid

SearchCV:

In[70]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

digits.data, digits.target == 9, random\_state=0)

# we provide a somewhat bad grid to illustrate the point:

param\_grid = {'gamma': [0.0001, 0.01, 0.1, 1, 10]}

# using the default scoring of accuracy:

grid = GridSearchCV(SVC(), param\_grid=param\_grid)

grid.fit(X\_train, y\_train)

print("Grid-Search with accuracy")

print("Best parameters:", grid.best\_params\_)

print("Best cross-validation score (accuracy)): {:.3f}".format(grid.best\_score\_))

print("Test set AUC: {:.3f}".format(

roc\_auc\_score(y\_test, grid.decision\_function(X\_test))))

print("Test set accuracy: {:.3f}".format(grid.score(X\_test, y\_test)))

302

|

Chapter 5: Model Evaluation and ImprovementOut[70]:

Grid-Search with accuracy

Best parameters: {'gamma': 0.0001}

Best cross-validation score (accuracy)): 0.970

Test set AUC: 0.992

Test set accuracy: 0.973

In[71]:

# using AUC scoring instead:

grid = GridSearchCV(SVC(), param\_grid=param\_grid, scoring="roc\_auc")

grid.fit(X\_train, y\_train)

print("\nGrid-Search with AUC")

print("Best parameters:", grid.best\_params\_)

print("Best cross-validation score (AUC): {:.3f}".format(grid.best\_score\_))

print("Test set AUC: {:.3f}".format(

roc\_auc\_score(y\_test, grid.decision\_function(X\_test))))

print("Test set accuracy: {:.3f}".format(grid.score(X\_test, y\_test)))

Out[71]:

Grid-Search with AUC

Best parameters: {'gamma': 0.01}

Best cross-validation score (AUC): 0.997

Test set AUC: 1.000

Test set accuracy: 1.000

When using accuracy, the parameter gamma=0.0001 is selected, while gamma=0.01 is

selected when using AUC. The cross-validation accuracy is consistent with the test set

accuracy in both cases. However, using AUC found a better parameter setting in

terms of AUC and even in terms of accuracy.6

The most important values for the scoring parameter for classification are accuracy

(the default); roc\_auc for the area under the ROC curve; average\_precision for the

area under the precision-recall curve; f1, f1\_macro, f1\_micro, and f1\_weighted for

the binary f1-score and the different weighted variants. For regression, the most com‐

monly used values are r2 for the R2 score, mean\_squared\_error for mean squared

error, and mean\_absolute\_error for mean absolute error. You can find a full list of

supported arguments in the documentation or by looking at the SCORER dictionary

defined in the metrics.scorer module:

6 Finding a higher-accuracy solution using AUC is likely a consequence of accuracy being a bad measure of

model performance on imbalanced data.

Evaluation Metrics and Scoring

|

303In[72]:

from sklearn.metrics.scorer import SCORERS

print("Available scorers:\n{}".format(sorted(SCORERS.keys())))

Out[72]:

Available scorers:

['accuracy', 'adjusted\_rand\_score', 'average\_precision', 'f1', 'f1\_macro',

'f1\_micro', 'f1\_samples', 'f1\_weighted', 'log\_loss', 'mean\_absolute\_error',

'mean\_squared\_error', 'median\_absolute\_error', 'precision', 'precision\_macro',

'precision\_micro', 'precision\_samples', 'precision\_weighted', 'r2', 'recall',

'recall\_macro', 'recall\_micro', 'recall\_samples', 'recall\_weighted', 'roc\_auc']

Summary and Outlook

In this chapter we discussed cross-validation, grid search, and evaluation metrics, the

cornerstones of evaluating and improving machine learning algorithms. The tools

described in this chapter, together with the algorithms described in Chapters 2 and 3,

are the bread and butter of every machine learning practitioner.

There are two particular points that we made in this chapter that warrant repeating,

because they are often overlooked by new practitioners. The first has to do with

cross-validation. Cross-validation or the use of a test set allow us to evaluate a

machine learning model as it will perform in the future. However, if we use the test

set or cross-validation to select a model or select model parameters, we “use up” the

test data, and using the same data to evaluate how well our model will do in the future

will lead to overly optimistic estimates. We therefore need to resort to a split into

training data for model building, validation data for model and parameter selection,

and test data for model evaluation. Instead of a simple split, we can replace each of

these splits with cross-validation. The most commonly used form (as described ear‐

lier) is a training/test split for evaluation, and using cross-validation on the training

set for model and parameter selection.

The second point has to do with the importance of the evaluation metric or scoring

function used for model selection and model evaluation. The theory of how to make

business decisions from the predictions of a machine learning model is somewhat

beyond the scope of this book.7 However, it is rarely the case that the end goal of a

machine learning task is building a model with a high accuracy. Make sure that the

metric you choose to evaluate and select a model for is a good stand-in for what the

model will actually be used for. In reality, classification problems rarely have balanced

classes, and often false positives and false negatives have very different consequences.

7 We highly recommend Foster Provost and Tom Fawcett’s book Data Science for Business (O’Reilly) for more

information on this topic.

304

|

Chapter 5: Model Evaluation and ImprovementMake sure you understand what these consequences are, and pick an evaluation met‐

ric accordingly.

The model evaluation and selection techniques we have described so far are the most

important tools in a data scientist’s toolbox. Grid search and cross-validation as we’ve

described them in this chapter can only be applied to a single supervised model. We

have seen before, however, that many models require preprocessing, and that in some

applications, like the face recognition example in Chapter 3, extracting a different

representation of the data can be useful. In the next chapter, we will introduce the

Pipeline class, which allows us to use grid search and cross-validation on these com‐

plex chains of algorithms.

Summary and Outlook

|

305CHAPTER 6

Algorithm Chains and Pipelines

For many machine learning algorithms, the particular representation of the data that

you provide is very important, as we discussed in Chapter 4. This starts with scaling

the data and combining features by hand and goes all the way to learning features

using unsupervised machine learning, as we saw in Chapter 3