CHAPTER 6
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 have seen in Chapter 3).

To evaluate our supervised models, so far we have split our data set in to a training set and a test set using the `train_test_split` function, built a model on the training set 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:

```
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
logreg.score(X_test, y_test)
# we predicted the correct class on 88% of the samples in X_test
```

```
0.88
```

As a reminder, 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, unseen data. We are

not interested in how well our model fit the training set, but rather, 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 a) introduce *cross-validation*, a more robust way to assess generalization performance than a single split of the data into a training and a test set and b) discuss methods to evaluate classification and regression performance that go beyond the default measures of accuracy and R^2 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 to evaluate generalization performance in a more stable and thorough way than using a split into training and test set.

In cross-validation, instead of splitting the data set in to a training set and a test set, the data is 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 five or ten. 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 as the training set. The model is build using the data in the folds 2-5, and then the accuracy is evaluated on fold 1.

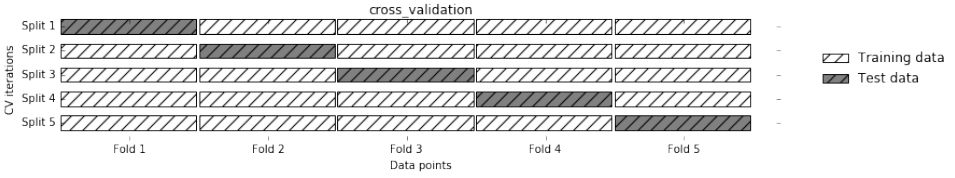
Then another model is build, this time using fold 2 as the test set, and the data in folds 1, 3, 4 and 5 as the training set.

This process is repeated using the folds 3, 4 and 5 as test sets. For each of these five *splits* of the data into training and test set, we computed the accuracy. In the end, we have collected five accuracy values.

The process is illustrated in Figure 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.

```
mglearn.plots.plot_cross_validation()
```



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:

```
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: ", scores)
```

```
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:

```
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
scores
```

```
array([ 1.   ,  0.967,  0.933,  0.9  ,  1.   ])
```

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

```
scores.mean()
```

```
0.96000000000000019
```

Benefits of cross-validation

There are several benefits of using cross-validation instead of a single split into a training and 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 are in the training set. Then, 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 obtain an unrealistically low score.

However, when using cross-validation, each example will be in the training set exactly once: each example is in one of the folds, and each fold is the test set once. Therefore the model needs to generalize well to all of the samples in the data set 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. Looking at the scores for the iris dataset above, we see accuracies between 90% and 100%. This is quite a range, and provides us with an idea about how the model might perform in the worst case and the 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, which is a good rule of thumb. When using five-fold cross-validation, in each iteration we can use 4/5 of the data (or 80% of our data) to fit the model. When using ten-fold cross-validation, we can use 9/10 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.

[info box] 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 build internally, but the purpose of cross-validation is only to evaluate how well a given algorithm will generalize when trained on a specific dataset. [/end infobox]

Stratified K-Fold cross-validation and other strategies

Splitting the dataset into k-folds by starting with the first 1/k-th part of the data as described above might not always be a good idea. Let’s have a look at the iris dataset for example:

```
from sklearn.datasets import load_iris
iris = load_iris()
print(iris.target)
```

```
[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 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 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 above, the first third of the data is the class 0, the second third is the class 1, and the last third is 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 zero, and the training set would be only class 1 and 2.

As the classes in training and test set 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`.

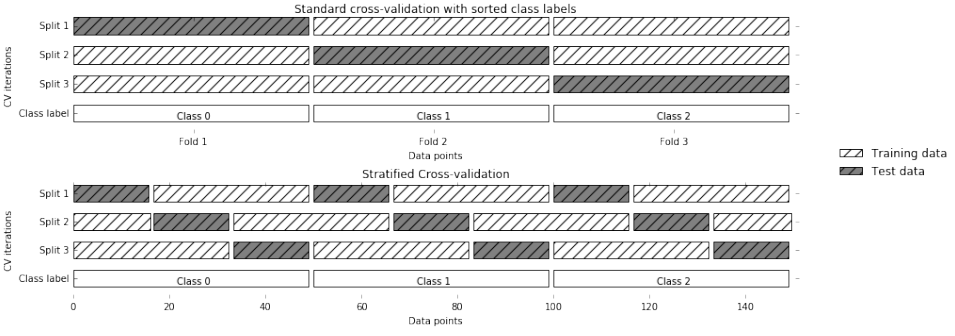
As the simple k-fold strategy fails here, scikit-learn does not use k-fold for classification, but rather *stratified k-fold cross-validation*. In stratified cross-validation, we split the data such that the proportions between classes are the same in each fold as they are in the whole dataset, as illustrated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

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 always 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 of the overall performance of the classifier.

For regression, scikit-learn uses the standard k-fold cross-validation by default. It would be possibly 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.

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



More control over cross-validation

We saw above 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 default of k-fold cross validation for regression and stratified k-fold for classification work well, but there are some cases when 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 you want to use:

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

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

```
cross_val_score(logreg, iris.data, iris.target, cv=kfold)
```

```
array([ 1.   ,  0.933,  0.433,  0.967,  0.433])
```

This way, we can verify that it is indeed a really bad idea to use 3-fold (non-stratified) cross-validation on the iris dataset:

```
kfold = KFold(n_folds=3)
cross_val_score(logreg, iris.data, iris.target, cv=kfold)
```

```
array([ 0.,  0.,  0.])
```

Remember: each fold corresponds to one of the classes, and so nothing can be learned. [specify again that this is on the iris dataset, it's a little unclear]

Another way to resolve this problem instead of stratifying the folds is to shuffle the data, to remove the ordering of the samples by label. We can do that setting the `shuf`

`fle` parameter of `KFold` to `True`. If we shuffle the data, we also need to fix the `ran dom_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).

```
kfold = KFold(n_folds=3, shuffle=True, random_state=0)
cross_val_score(logreg, iris.data, iris.target, cv=kfold)
```

```
array([ 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, in particular for large datasets, but sometimes provides better estimates on small datasets:

```
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: ", scores.mean())
```

```
number of cv iterations:  150

mean accuracy:  0.953333333333
```

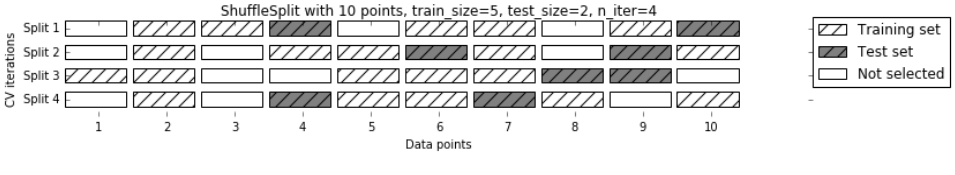
Shuffle-Split cross-validation

Another, very flexible strategy for cross validation is *shuffle-split cross-validation*. In shuffle-split cross-validation, each split samples `train_size` many points for the training set, and `test_size` many (disjoint) point for the test set. This splitting is repeated `n_iter` many times. Figure shuffle_split illustrates running four iterations of splitting a dataset consisting of 10 points, with a training set of 5 points and a test set of 2 points each.

You can use integers for `train_size` and `test_size` to use absolute sizes of these sets, or floating points numbers, to use fractions of the whole dataset.

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

shuffle_split



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

```
from sklearn.model_selection import ShuffleSplit
shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_iter=10)
cross_val_score(logreg, iris.data, iris.target, cv=shuffle_split)
```

```
array([ 1.   ,  0.933,  0.88 ,  0.933,  0.853,  0.973,  0.787,  0.947,

        0.92 ,  0.973])
```

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

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

Cross-validation with groups

Another very common setting for cross-validation is when there are groups in the data that are highly related.

Say you want to build a system to recognize emotions from pictures of faces, and you collect a dataset of pictures of 100 people where each person is captured multiple times, showing various emotions. The goal is to build a classifier that can correctly identify emotions of people not in the dataset.

You could use the default stratified cross-validation to measure the performance of a classifier here. However, it is likely that pictures of the same person will be in 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 set contain images of different people.

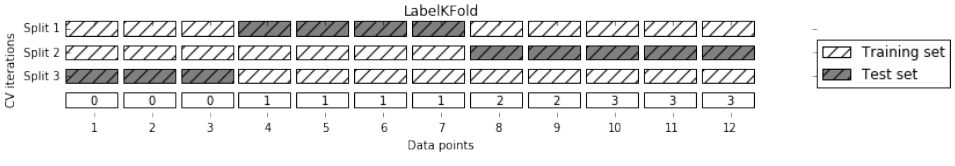
To achieve this, we can use `LabelKFold`, which takes a `label` argument, which we can use to indicate which person is in the image. So `label` here indicates groups in the data that should not be split when creating training and test set, and should not be confused with the class label.

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 you dataset, but are interested in recognizing speech of new speakers.

Below is an example of using a synthetic dataset with a grouping given by the `labels` list. The dataset consists of 12 data points, and for each of the data points, `labels` specifies which group (think patient) the point belongs to. The labels specify 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. 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 label_kfold.

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

```
print("label_kfold")
mglearn.plots.plot_label_kfold()
```




```
label_kfold
```

```
from sklearn.model_selection import LabelKFold
# 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.
labels = [0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3]
cross_val_score(logreg, X, y, labels, cv=LabelKFold(n_folds=3))
```

```
array([ 0.75 ,  0.8  ,  0.667])
```

There are more splitting strategies for cross-validation in scikit-learn, which you can find in the scikit-learn user guide, which allow for even more different use-cases. However, the standard `KFold`, `StratifiedKFold` and `LabelKFold` are by far the most commonly used ones.

Grid Search

Now that we know how to evaluate how well a model generalizes, we can do 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 values `0.001, 0.01, 0.1, 1` and `10` 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 below:

| |C = 0.001 | C = 0.01 | C = 0.1 | C = 1 | C = 10 |

|----------|----------|----------|---------|-------|--------|

|gamma=0.001|SVC(C=0.001, gamma=0.001)|SVC(C=0.01, gamma=0.001)|SVC(C=0.1, gamma=0.001)|SVC(C=1, gamma=0.001)|SVC(C=10, gamma=0.001)|

|gamma=0.01|SVC(C=0.001, gamma=0.01)|SVC(C=0.01, gamma=0.01)|SVC(C=0.1, gamma=0.01)|SVC(C=1, gamma=0.001)|SVC(C=10, gamma=0.01)|

|gamma=0.1|SVC(C=0.001, gamma=0.1)|SVC(C=0.01, gamma=0.1)|SVC(C=0.1, gamma=0.1)|SVC(C=1, gamma=0.1)|SVC(C=10, gamma=0.1)|

|gamma=1|SVC(C=0.001, gamma=1)|SVC(C=0.01, gamma=1)|SVC(C=0.1, gamma=1)|SVC(C=1, gamma=1)|SVC(C=10, gamma=1)|

|gamma=10|SVC(C=0.001, gamma=10)|SVC(C=0.01, gamma=10)|SVC(C=0.1, gamma=10)|SVC(C=1, gamma=10)|SVC(C=10, gamma=10)|

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:

```
# 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: %d   size of test set: %d" % (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: ", best_score)
print("best parameters: ", best_parameters)
```

```
Size of training set: 112   size of test set: 38

best score:  0.973684210526

best parameters:  {'C': 100, 'gamma': 0.001}
```

```
best_score
```

```
0.97368421052631582
```

The danger of overfitting the parameters and the validation set

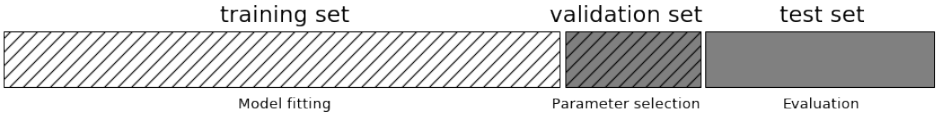
Given this result, we might be tempted to report that we found a model that performs 97.3% accurate on our dataset. However, this claim could be overly optimistic (or just wrong) for the following reason: we tried many different parameters, and selected the one with best accuracy on the test set. However, that doesn't mean that this accuracy carries 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 set in the first place; we need an independent data set 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, as shown in Figure threefold_split below.

After selecting the best parameters using the validation set, we can rebuild a model using the parameters 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.

```
print("threefold_split")
mglearn.plots.plot_threefold_split()
```




```
threefold_split
```

This leads to the following implementation:

```
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 set
X_train, X_valid, y_train, y_valid = train_test_split(X_trainval, y_trainval, random_state=1)

print("Size of training set: %d   size of validation set: %d   size of test set: %d" % (X_train.sh
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_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}

# 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: ", best_score)
print("best parameters: ", best_parameters)
print("test set score with best parameters: ", test_score)
```

```
Size of training set: 84   size of validation set: 28   size of test set: 38

best score on validation set:  0.964285714286

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

test set score with best parameters:  0.921052631579
```

The best score on the validation set is 96.4%: 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 apply 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 above method of splitting the data into a training, a validation and a test set is workable, and relatively commonly used, it is quite sensitive to how exactly the data is split. From the output of the code [FIXME Reference code above] we can see that `GridSearchCV` selects `'C': 10, 'gamma': 0.01` as the best parameters, while the output of the code in [FIXME Reference code two up] selects `'C': 10, '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:

```
# reference: manual_grid_search_cv
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)
```

```
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

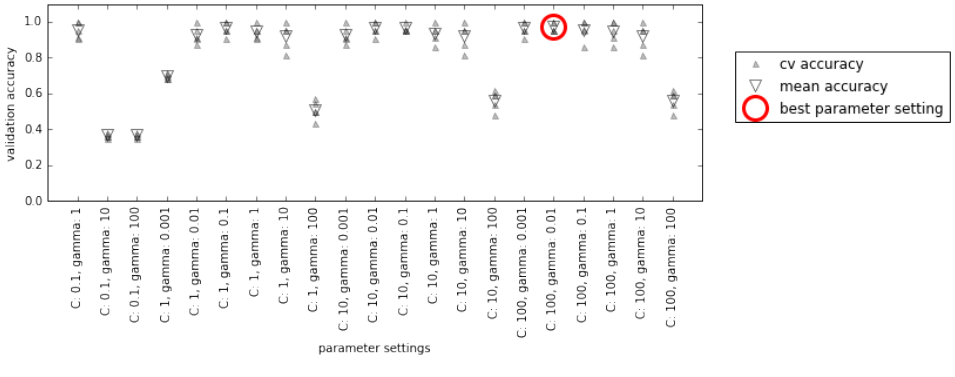
To evaluate the accuracy of the SVM using a particular setting of `C` and `gamma` using five-fold cross-validation we need to train 36 * 5 = 180 models. As you can imagine, the main down-side of the use of cross-validation is the time it takes to train all these models.

Figure cross_val_selection illustrates how the best parameter setting is selected in the code above. 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.

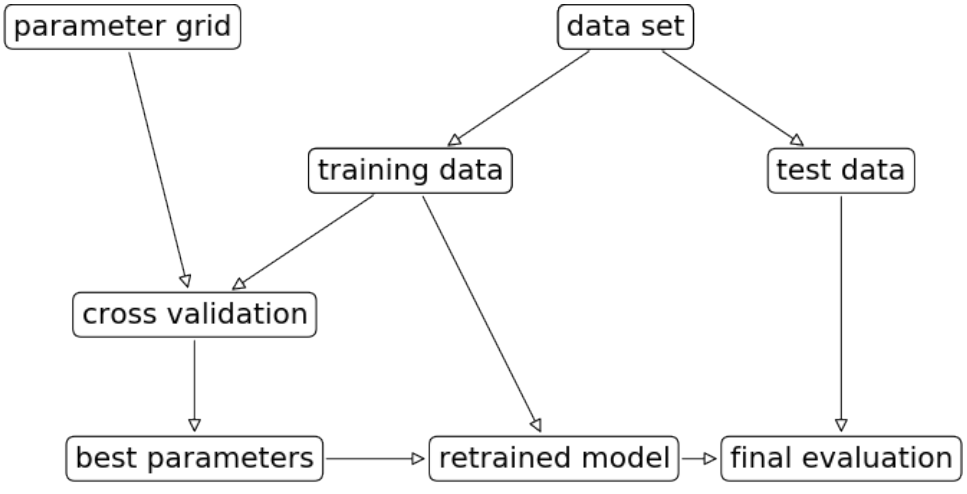
[warning / note box] As we said above, cross-validation is a way to evaluate a given algorithm on a specific dataset.

However, it is often used in conjunction with parameter search methods like grid search. For this reason, many people colloquially use the term `cross-validation` to refer to grid-search with cross-validation. [/end warning box]

```
mglearn.plots.plot_cross_val_selection()
```




```
mglearn.plots.plot_grid_search_overview()
```



Because grid-search with cross-validation is such a commonly used method to adjust parameters scikit-learn provides the `GridSearchCV` class that 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:

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

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

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 5 fold (stratified) cross-validation:

```
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:

```
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 [footnote: A scikit-learn estimator that is created using another estimator is called a a meta-estimator in scikit-learn. GridSearchCV is the most commonly used meta-estimator, but we will see more later.]. However, when we call `fit`, it will run cross-validation for each combination of parameters we specified in `param_grid`.

```
grid_search.fit(X_train, y_train)
```

```
GridSearchCV(cv=5, error_score='raise',

       estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,

  decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',

  max_iter=-1, probability=False, random_state=None, shrinking=True,

  tol=0.001, verbose=False),

       fit_params={}, iid=True, n_jobs=1,

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

       pre_dispatch='2*n_jobs', refit=True, scoring=None, verbose=0)
```

Fitting the `GridSearchCV` object not only searches for the best parameters, it 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 with FIXME reference manual_grid_search_cv. 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:

```
grid_search.score(X_test, y_test)
```

```
0.97368421052631582
```

Choosing the parameters using cross-validation, we actually found a model that achieves 97.3% accuracy on the test set. The important part here is that we *did not use the test set* to choose the parameters.

The parameters that were found are scored in the `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_`:

```
print(grid_search.best_params_)
print(grid_search.best_score_)
```

```
{'C': 100, 'gamma': 0.01}

0.973214285714
```

[warning box]

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 validation cross-validation accuracy, with *cross-validation performed on the training set*.

[/end warning box]

Sometimes it is helpful to have access to the actual model that was found, for example 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:

```
grid_search.best_estimator_

SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,

  decision_function_shape=None, 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 `grid_scores_` attribute:

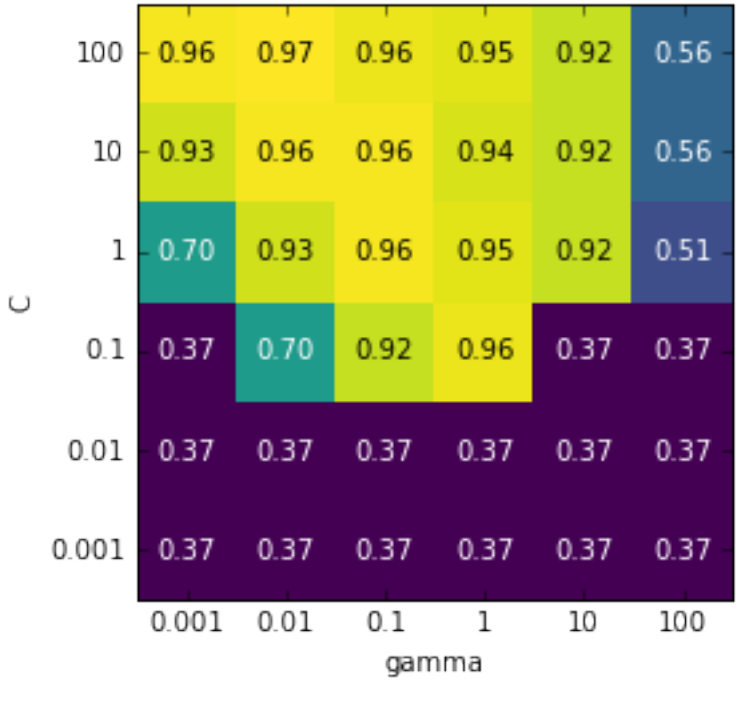
```
grid_search.grid_scores_

[mean: 0.36607, std: 0.01137, params: {'C': 0.001, 'gamma': 0.001},

 mean: 0.36607, std: 0.01137, params: {'C': 0.001, 'gamma': 0.01},

 mean: 0.36607, std: 0.01137, params: {'C': 0.001, 'gamma': 0.1},

 mean: 0.36607, std: 0.01137, params: {'C': 0.001, 'gamma': 1},
```

```
mean: 0.36607, std: 0.01137, params: {'C': 0.001, 'gamma': 10},

mean: 0.36607, std: 0.01137, params: {'C': 0.001, 'gamma': 100},

mean: 0.36607, std: 0.01137, params: {'C': 0.01, 'gamma': 0.001},

mean: 0.36607, std: 0.01137, params: {'C': 0.01, 'gamma': 0.01},

mean: 0.36607, std: 0.01137, params: {'C': 0.01, 'gamma': 0.1},

mean: 0.36607, std: 0.01137, params: {'C': 0.01, 'gamma': 1},

mean: 0.36607, std: 0.01137, params: {'C': 0.01, 'gamma': 10},

mean: 0.36607, std: 0.01137, params: {'C': 0.01, 'gamma': 100},

mean: 0.36607, std: 0.01137, params: {'C': 0.1, 'gamma': 0.001},

mean: 0.69643, std: 0.01333, params: {'C': 0.1, 'gamma': 0.01},

mean: 0.91964, std: 0.04442, params: {'C': 0.1, 'gamma': 0.1},

mean: 0.95536, std: 0.03981, params: {'C': 0.1, 'gamma': 1},

mean: 0.36607, std: 0.01137, params: {'C': 0.1, 'gamma': 10},

mean: 0.36607, std: 0.01137, params: {'C': 0.1, 'gamma': 100},

mean: 0.69643, std: 0.01333, params: {'C': 1, 'gamma': 0.001},

mean: 0.92857, std: 0.04278, params: {'C': 1, 'gamma': 0.01},

mean: 0.96429, std: 0.03405, params: {'C': 1, 'gamma': 0.1},

mean: 0.94643, std: 0.03251, params: {'C': 1, 'gamma': 1},

mean: 0.91964, std: 0.06507, params: {'C': 1, 'gamma': 10},

mean: 0.50893, std: 0.04666, params: {'C': 1, 'gamma': 100},

mean: 0.92857, std: 0.04278, params: {'C': 10, 'gamma': 0.001},

mean: 0.96429, std: 0.03405, params: {'C': 10, 'gamma': 0.01},

mean: 0.96429, std: 0.01793, params: {'C': 10, 'gamma': 0.1},

mean: 0.93750, std: 0.04556, params: {'C': 10, 'gamma': 1},

mean: 0.91964, std: 0.06507, params: {'C': 10, 'gamma': 10},

mean: 0.56250, std: 0.04966, params: {'C': 10, 'gamma': 100},
```

```
 mean: 0.96429, std: 0.03405, params: {'C': 100, 'gamma': 0.001},

 mean: 0.97321, std: 0.02234, params: {'C': 100, 'gamma': 0.01},

 mean: 0.95536, std: 0.04983, params: {'C': 100, 'gamma': 0.1},

 mean: 0.94643, std: 0.05199, params: {'C': 100, 'gamma': 1},

 mean: 0.91964, std: 0.06507, params: {'C': 100, 'gamma': 10},

 mean: 0.56250, std: 0.04966, params: {'C': 100, 'gamma': 100}]
```

This attribute contains the mean cross-validation accuracy (and it's standard deviation) for all parameter settings that we tried. As we were searching a two-dimensional grid of parameters ('C' and 'gamma'), this is best visualized as a heat map. First, we extract the mean validation scores, then we reshape the scores so that the axes correspond to C and gamma:

```
scores = [score.mean_validation_score for score in grid_search.grid_scores_]
scores = np.array(scores).reshape(6, 6)

# plot the mean cross-validation scores
mglearn.tools.heatmap(scores, xlabel='gamma', ylabel='C', xticklabels=param_grid['gamma'],
                      yticklabels=param_grid['C'], cmap="viridis")
```



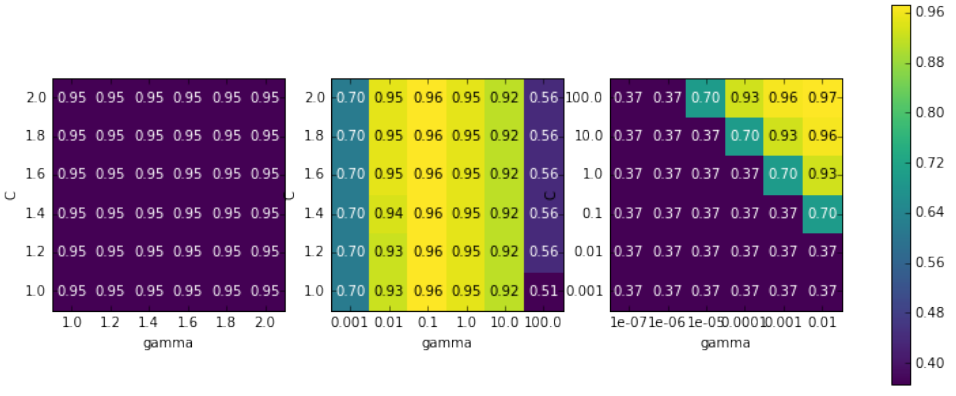
Each point in the heat map corresponds to one run of cross-validation, with a particular parameter setting. The color encodes the cross-validation accuracy, with light colors meaning high accuracy, and dark colors meaning low accuracy.

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 accuracy is around 96%.

We can take away from this plot several things. First, the parameters 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%. Also, 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 is not on the edge of the plot.

Figure gridsearch_failures shows some plots where the result is less ideal, because the search ranges were not chosen properly.

```
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 = [score.mean_validation_score for score in grid_search.grid_scores_]
    scores = np.array(scores).reshape(6, 6)

    # plot the mean cross-validation scores
    scores_image = mglearn.tools.heatmap(scores, xlabel='gamma', ylabel='C', xticklabels=param_gri
                                         yticklabels=param_grid['C'], cmap="viridis", ax=ax)

plt.colorbar(scores_image, ax=axes.tolist())
print("gridsearch_failures")
```



`gridsearch_failures`

The first panel shows no changes at all, with a constant color over the whole parameter grid. 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 parameter is searching over interesting values, but the `C` parameter is not -- or it could mean the `C` parameter is not important at all.

The third panel shows changes in both `C` and `gamma`. However, we can see that in the top right of the plot, nothing interesting is happening. We can probably exclude the very small values from future grid-searches. The optimum parameter setting is on the bottom 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 the final test set---as we discussed above, evaluation of the test set should happen only once we know exactly what model we want to use.

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 section XX as the `cv` parameter in `GridSearchCV`.

In particular, to get only a single split into a training and validation set, you can use `ShuffleSplit` or `StratifiedShuffleSplit` with `n_iter=1`. This might be helpful for very large datasets, or very slow models.

Nested cross-validation

In the above examples, we went from using single split of the data into a training, validation and test set to spitting the data into training and test sets, and then performing cross-validation on the training set. When using `GridSearchCV` as above, we still have a single split of the data in training and test set however, which might still make our results unstable, and may 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 set once, we 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 set. 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 setting. The scores tell us how well a model generalizes, given the best parameters found by grid-search. 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.

It can be good to evaluate how good a given model works on a particular dataset, though.

Implementing nested cross-validation in scikit-learn is straightforward; we call `cross_val_score` with an instance of `GridSearchCV` as the model:

```
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())
```

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

Mean cross-validation score:  0.98
```

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 build, making nested cross-validation a very expensive procedure. Here, we used the same cross-validation splitter in the inner and outer loop; however, this is not necessary and you can use any combination of cross-validation strategies in the inner and outer loop. 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 simplified implementation given below:

```
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)
    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 outer_scores
```

```
from sklearn.model_selection import ParameterGrid, StratifiedKFold
nested_cv(iris.data, iris.target, StratifiedKFold(5), StratifiedKFold(5), SVC, ParameterGrid(param
```

```
[0.96666666666666667, 1.0, 0.96666666666666667, 0.96666666666666667, 1.0]
```

Parallelizing cross-validation and grid-search

While running grid-search over many parameters and on large datasets can be computationally challenging, it is also *embarassingly parallel*. This means that building a

model 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 `GridSearchCV` 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 to 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. However, at the time of writing, this is not supported within scikit-learn. It is, however, possible to use the IPython parallel framework for parallel grid-searches, if you don't mind writing the for-loop over parameters as in [reference "naive implementation"] yourself.

For spark users, there is also the recently developed `spark-sklearn` package [footnote https://github.com/databricks/spark-sklearn] which allows running grid-search over an already established spark cluster.

Evaluation Metrics and scoring

So far, we always evaluated classification performance using accuracy (the fraction of correctly classified samples) and regression performance using R^2. However, these are only two of the many possible ways to summarize how well a supervised model performs on a given dataset.

In practice, these evaluation metrics might not be appropriate for your application, and it is important to choose the right metric when selecting between models and adjusting hyper-parameters.

Keep the end-goal in mind

When selecting a metric, you should always have the end-goal of the machine learning application in mind. In practice, we are usually not interested in just 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 what the high-level goal of the application is, often called *business metric*. The consequences of choosing a particular algorithm for a machine learning application has is called the *business impact*. [Footnote: We ask scientific 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 as "business impact" being used in the sciences.] Maybe this 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 models or adjusting parameters, you should then pick the model that has the most positive influence on the business metric.

Often this is hard, as assessing 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 high business risk 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 evaluate. This closest metric should be used whenever possible for model evaluations and selection. This evaluation might not be a single number---he consequence of your algorithm could be that you have 10% more customers, but each customers will spend 15% less---but it should capture the expected business impact of choosing one model over another.

We will first discuss metrics for the important special case of binary classification, then multi-class classification, and finally regression.

Metrics for binary classification

Binary classification is arguably the most common and conceptually simple application 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 into 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 the 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 (indication 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 consequence of these mistakes are 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 a slight inconvenience for the patient. Making an incorrect positive prediction is called a *false positive*.

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. Making a mistake of this kind, an incorrect negative prediction is called a *false negative*.

In this particular example, it is clear that we want to avoid false negatives as much as possible, while false positives just create 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 - which might be much more meaningful for making business decisions on which model to use.

Imbalanced datasets

Types of errors in particularly play an important role when one of two classes is much more frequent then the other one. This is very common in practice; a good example for this is click-through prediction, where each data point represents an "impression", an item that was shown to a user. This item might be an add, 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 thing 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 dataset*, 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 class 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 and a potentially good model.

To illustrate, we create a 9:1 imbalanced dataset from the digits dataset, by classifying the digit four against the nine other classes:

```
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 four") to see how uninformative accuracy can be:

```
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("predicted labels: %s" % np.unique(pred_most_frequent))
print("score: %f" % dummy_majority.score(X_test, y_test))
```

```
predicted labels: [False]

score: 0.895556
```

We obtained close to 90% accuracy without learning anything. This might seem striking. 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:

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=2).fit(X_train, y_train)
pred_tree = tree.predict(X_test)
tree.score(X_test, y_test)
```

```
0.9177777777777778
```

According to accuracy, the `GaussianNB` model is clearly worse than the constant predictor! This could either indicate that something is wrong with how we use `GaussianNB` or that accuracy is in fact not a good measure here.

For comparison purposes, we evaluate two more classifiers: `SVC` and the default `DummyClassifier` which makes random predictions, but producing classes with the same proportions as in the training set:

```
from sklearn.linear_model import LogisticRegression

dummy = DummyClassifier().fit(X_train, y_train)
pred_dummy = dummy.predict(X_test)
print("dummy score: %f" % 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: %f" % logreg.score(X_test, y_test))
```

```
dummy score: 0.808889

logreg score: 0.977778
```

The dummy that produces random output is still better than the `GaussianNB`, while `LogisticRegression` produces very good results.

Clearly accuracy is an inadequate measure to quantify predictive performance in this imbalanced 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 metrics 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 classification is using confusion matrices. Let's inspect the predictions of `LogisticRegression` above using the `confusion_matrix` function. We already stored the predictions on the test set in `pred_logreg`.

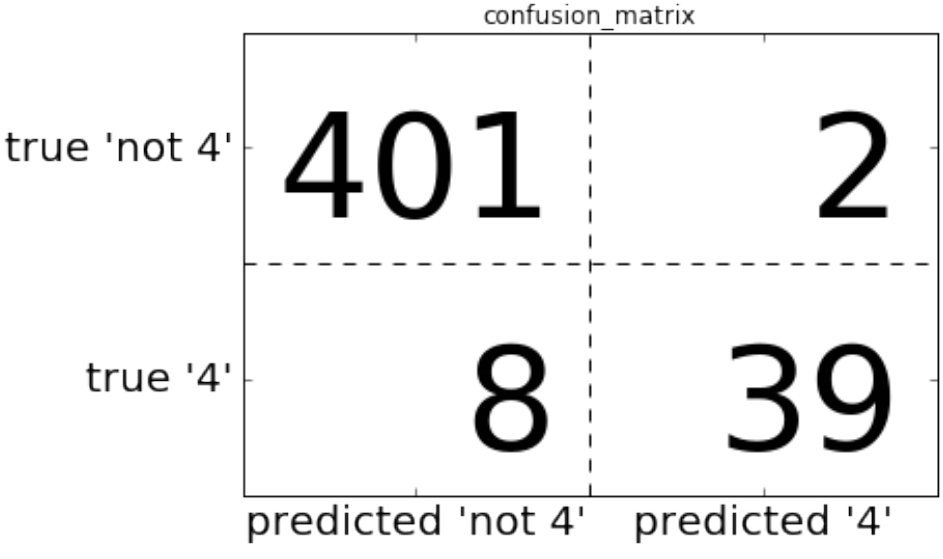
```
from sklearn.metrics import confusion_matrix

confusion = confusion_matrix(y_test, pred_logreg)
print(confusion)
```

```
[[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 corresponds to the predicted classes. Each entry counts for how many data points in the class given by the row the prediction was the class given by the column.

Figure confusion_matrix below illustrates this meaning:

```
mglearn.plots.plot_confusion_matrix_illustration()
```

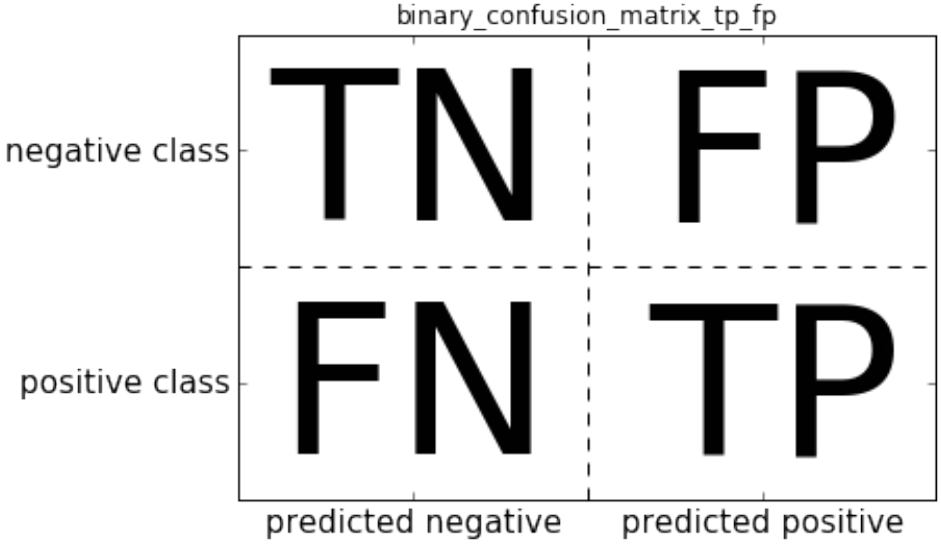


This means of the 403 data points (sum of the first row) that are not a four, 401 got predicted correctly as such, and two of which were incorrectly predicted as a four. Similarly there are 47 data points that are fours, 39 of which got correctly classified, and 8 of which were predicted incorrectly as not a four.

Entries on the main diagonal [footnote: The main diagonal of a two-dimensional array or matrix `A` are `A[i, i]`] of the confusion matrix correspond to correct classifications, while other entries tell us how many samples of one class got mistakenly classified as another class.

If we declare "being a four" the positive class, we can relate the entries of the confusion matrix with the terms *false positives* and *false negatives* that we introduced earlier. To complete the picture, we call correctly classified samples belonging to the positive class *true positives* and correctly classified samples of the negative class *true negatives*. These terms are usually abbreviated FP, FN, TP and TN and lead to the following interpretation for the confusion matrix:

```
mglearn.plots.plot_binary_confusion_matrix()
```



Now let's use the confusion matrix to compare the models we fitted above, the two dummy models, the decision tree and the logistic regression:

```
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))
```

```
Most frequent class:

[[403   0]

 [ 47   0]]



Dummy model:

[[377  26]

 [ 42   5]]



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 (3) in particular compared to the number of false negatives and false positives - there are many more false positives than true positives!

The predictions made by the 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 `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 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 cumbersome, 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 summarize 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 confusion matrix, by computing accuracy, which can be expressed as

\begin{equation}

\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}

\end{equation}

In other words: accuracy is the number of correct prediction (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 summaries the confusion matrix, with the most common one being *precision* and *recall*.

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

\begin{equation}

\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}

\end{equation}

Precision is used as a performance metric when the goal is to limit the number of false positives. As an example, imagine a model to predict 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 predicted drugs will actually work. Therefore, it is important that the model does not produce many false positives, in other words, it has a high precision. Precision is also know as *positive predictive value* (PPV).

Recall on the other hand measures how many of the positive samples are captured by the positive predictions:

\begin{equation}

\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}

\end{equation}

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 the Section "Kinds of Errors" is a good example for this: it is important to find all people that are sick, possible 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 trivially 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, 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.

[info box]Precision and recall are only two of many classification measures derived from TP, FP, TN and FN. You can find a a great summary of all the measures on wikipedia: https://en.wikipedia.org/wiki/Sensitivity_and_specificity In the machine learning community, precision and recall are arguably the most commonly used measures for binary classification, but other communities might use other, related metrics.[/ info box]

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 the harmonic mean of precision and recall:

\begin{equation}

\text{F} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}

\end{equation}

This particular variant is also known as the f_1-score. As it takes precision and recall into account, it can be a better measure than accuracy on imbalanced binary classification datasets. Let's run it on the predictions on the "nine vs rest" dataset that we computed above. 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.

```
from sklearn.metrics import f1_score
print("f1 score most frequent: %.2f" % f1_score(y_test, pred_most_frequent))
print("f1 score dummy: %.2f" % f1_score(y_test, pred_dummy))
print("f1 score tree: %.2f" % f1_score(y_test, pred_tree))
print("f1 score: %.2f" % f1_score(y_test, pred_logreg))
```

```
/home/andy/checkout/scikit-learn/sklearn/metrics/classification.py:1117: UndefinedMetricWarning: F

  'precision', 'predicted', average, warn_for)
```

We can note two things: we get an error message for the `most_frequent` prediction, as there was 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 performance again in one number, which reflect our intuition about how well a model predicts much better than accuracy in the imbalanced class setting. 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:

```
from sklearn.metrics import classification_report
print(classification_report(y_test, pred_most_frequent,
                            target_names=["not nine", "nine"]))
```

```
/home/andy/checkout/scikit-learn/sklearn/metrics/classification.py:1117: UndefinedMetricWarning: F

  'precision', 'predicted', average, warn_for)
```

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 is 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 .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 support) average of the numbers for each class.

Here are two more reports, one for the dummy classifier and one for the logistic regression:

```
print(classification_report(y_test, pred_dummy,
                            target_names=["not nine", "nine"]))
```

```
             precision    recall  f1-score   support


   not nine       0.90      0.94      0.92       403

       nine       0.16      0.11      0.13        47


avg / total       0.82      0.85      0.83       450
```

```
print(classification_report(y_test, pred_logreg,
                            target_names=["not nine", "nine"]))
```

```
             precision    recall  f1-score   support


   not nine       0.98      1.00      0.99       403

       nine       0.95      0.83      0.89        47


avg / total       0.98      0.98      0.98       450
```

As you may notice, when looking at the reports, the differences between the dummy models and a very good model is 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 vs the logistic regression was 0.13 vs 0.89 on the "nine" class, it is 0.90 vs 0.99 on the "not nine" class, which both seem like reasonable results.

Looking at all 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 discusses in Chapter 2, most classifiers provide a `decision_function` or a `predict_proba` method to assess degrees of certainty about predictions.

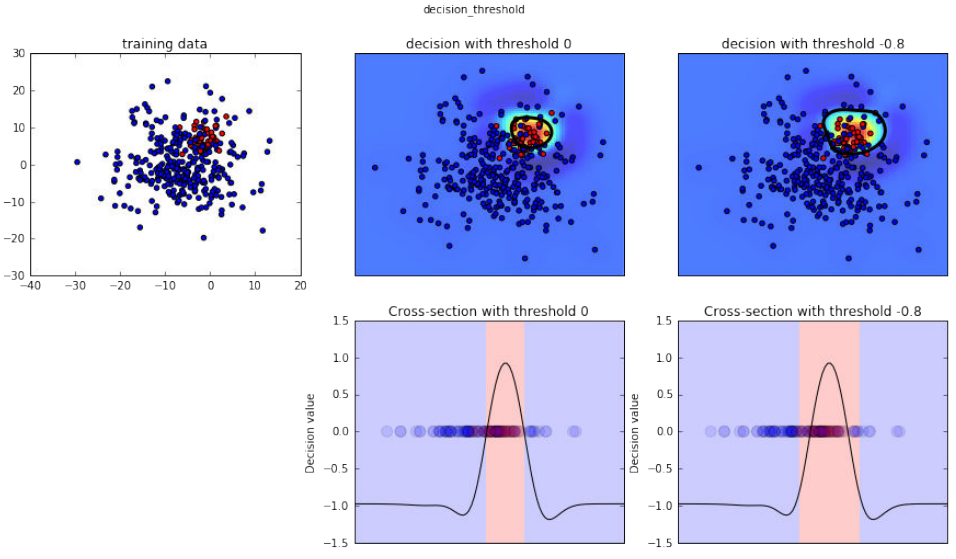
Making predictions can be seen as thresholding the output of `decision_function` or `predict_proba` at a certain fixed point - in binary classification zero for the decision function and 0.5 for `predict_proba`.

Below is an example of an imbalanced binary classification task, with 400 blue points classified against 50 red points.

The training data is shown on the right of Figure decision_threshold. We train a kernel SVM model on this data, and the left of Figure decision_threshold illustrates the values of the decision function as a heat-map. A red background means points there will be classified as red, blue background means that points there will be classified as blue.

You can see a black circle, which denotes the threshold of the `decision_function` being exactly zero. Points inside this circle will be classified as red, and outside as blue.

```
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)

mglearn.plots.plot_decision_threshold()
```



We can use the `classification_report` to evaluate precision and recall for both classes:

```
print(classification_report(y_test, svc.predict(X_test)))
```

```
             precision    recall  f1-score   support

          0       0.97      0.89      0.93       104
          1       0.35      0.67      0.46         9

avg / total       0.92      0.88      0.89       113
```

In Figure decision_threshold, blue is the negative class and red is the positive class.

For class 1, we get a fairly small recall, and precision 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 above. This means we are willing to risk more false positives (false class 1) in exchange for true positives (which will increase the recall).

The predictions generated by `svc.predict` above really do not fulfill this requirement. 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 `decision_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:

```
y_pred_lower_threshold = svc.decision_function(X_test) > -.8
```

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

```
print(classification_report(y_test, y_pred_lower_threshold))
```

```
             precision    recall  f1-score   support

          0       1.00      0.82      0.90       104
          1       0.32      1.00      0.49         9

avg / total       0.95      0.83      0.87       113
```

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 right panel of Figure decision_threshold.

If you value precision over recall or the other way around, or you data is heavily imbalanced, changing the decision threshold is the easiest way to obtain better results. As the `decision_function` can have arbitrary ranges, how to pick the right threshold is hard to say. If you do set a threshold, you need to be careful not to do this on the test set. As with any other parameter, setting a decision threshold on the test set is likely to give you too optimistic 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 zero to one scale, and models probabilities. By default, the threshold of 0.5 means that if the model is more than 50% "sure" that a point if of the positive class, it will be classified as such. Increasing the threshold 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 it's full depth is always 100% sure on its decisions - even though it might be often wrong). This relates to the concept of *calibration*: A calibrated model is a model that provides an accurate measure of it's uncertainty. Discussing calibration in detail is beyond the scope of this book, unfortunately.

Precision-Recall curves and ROC curves

As we just discussed, changing the threshold that is used to make a classification decision in a model is a way to adjust the trade-off of precision and recall for a given classifier. Maybe you want miss less than 10% of positive samples - meaning a desired recall of 90%. This decision is a decision that depends on the application, and is (or 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 threshold 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 is useless.

Setting a requirement on a classifier like 90% recall is often called the *operating point*. Fixing an operating point is often helpful in business settings to make performance guarantees to customer 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 recall *at once*. This is possible using a tool called the *precision-recall curve*.

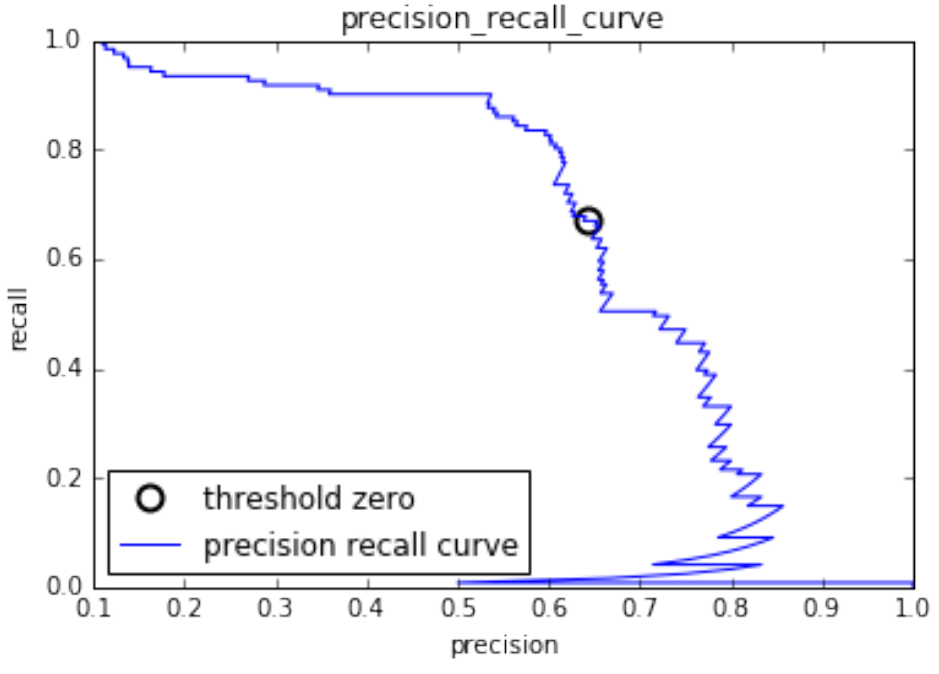
You can find the function to compute the precision-recall curve in the `sklearn.metrics` module. It needs the ground truth labeling and predicted uncertainties, created via `decision_function` or `predict_proba`:

```
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:

```
# create a similar dataset as before, but with more samples to get 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.title("precision_recall_curve");
plt.legend(loc="best")
```



Each point along the curve in "precision_recall_curve" corresponds to a possible threshold on 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 threshold of zero, 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 towards higher precision, but also lower recall. Raising the threshold more and more, we get to a situation where most of the points classified as being positive are true positives, leading to a very high precision at lower recall. The more the model keeps recall high as precision goes up, the better.

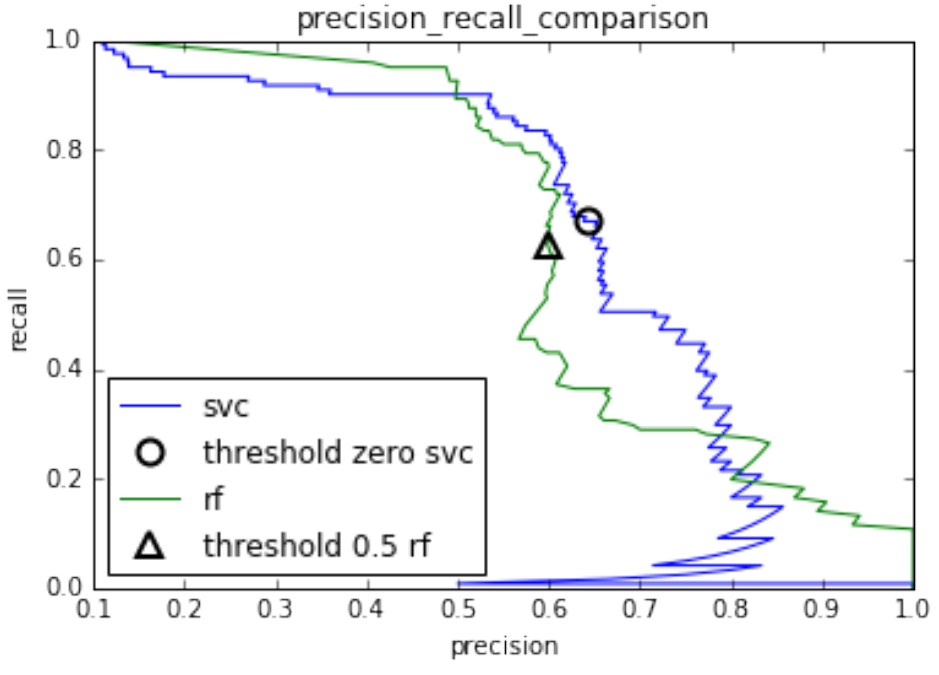
Looking at this particular curve a bit more, we can see that with this model it is possible to get a precision up to around 0.5 with very high recall. If we want a much higher precision, we have to sacrifice a lot of precision. In other words: on the left, the curve is relatively flat, meaning that recall does not go down a lot when we require

increased precision. For precision greater than 0.5, reach 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. Below we compare the SVC 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 second argument a certainty measure for the positive class (class 1), so we pass the probability of a sample being class one, 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.

```
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], '^', markersize=10,
         label="threshold 0.5 rf", fillstyle="none", c='k', mew=2)
plt.xlabel("precision")
plt.ylabel("recall")
plt.legend(loc="best")
plt.title("precision_recall_comparison");
```



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 middle (around precision=0.7), the SVM performance 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:

```
print("f1_score of random forest: %f" % f1_score(y_test, rf.predict(X_test)))
print("f1_score of svc: %f" % f1_score(y_test, svc.predict(X_test)))
```

```
f1_score of random forest: 0.609756


f1_score of svc: 0.655870
```

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 threshold or operating point.

One particular way to summarize the precision-recall curve by computing the integral or area under the curve of the precision-recall curve, also known as *average precision*.

You can compute the average precision using the `average_precision_score`. Because we need to compute the ROC curve, and consider multiple thresholds, we

need to pass the result of `decision_function` or `predict_proba` to `average_precision_score`, not the result of `predict`:

```
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: %f" % ap_rf)
print("average precision of svc: %f" % ap_svc)
```

```
average precision of random forest: 0.665737

average precision of svc: 0.662636
```

When averaging over all possible thresholds, we see that random forest and SVC perform similarly well, with the random forest even slightly ahead. This is quite different than the result we got from `f1_score` above.

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 commonly used to analyze the behavior of classifiers at different thresholds: the *receiver operating characteristics curve*, or *ROC curve* for short. The ROC curve similarly 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 fraction of false positives out of all negative samples:

\begin{equation}

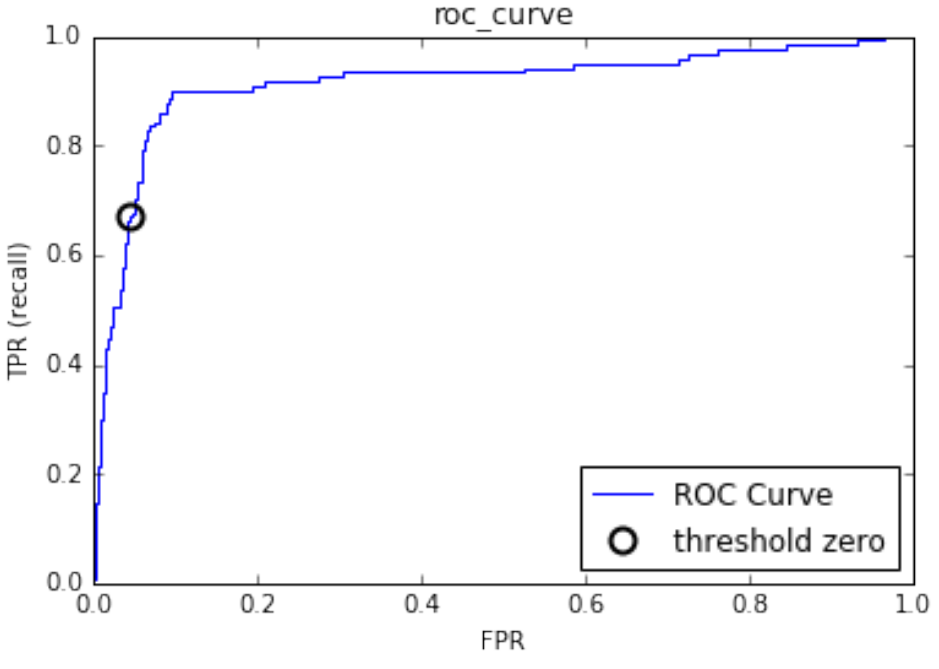
\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}

\end{equation}

The ROC curve can be computed using the `roc_curve` function:

```
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)")
plt.title("roc_curve");
# 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)
```

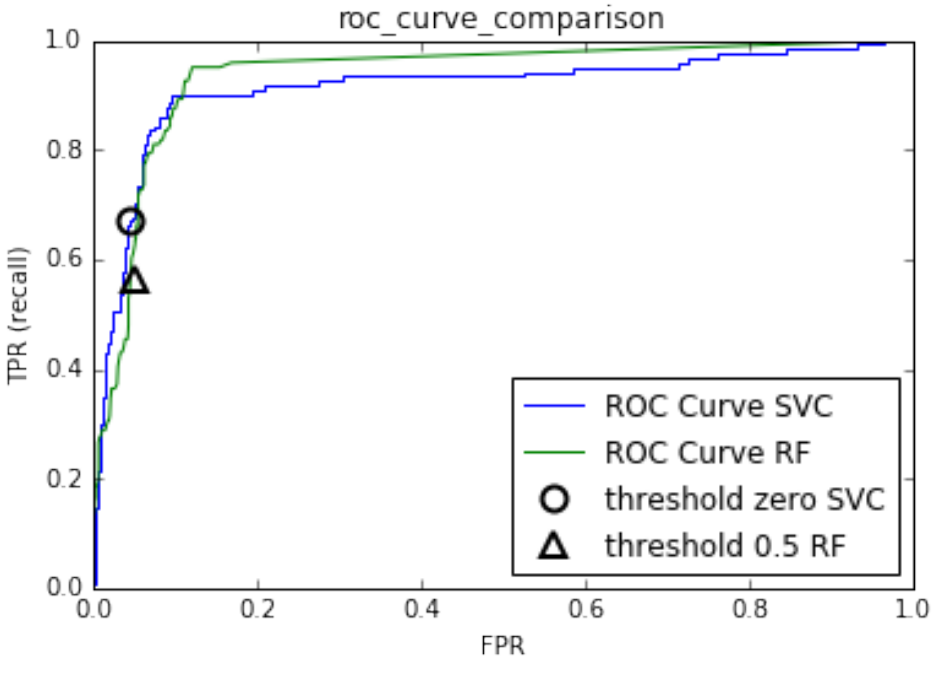


For the ROC curve, the ideal curve is close to the top left: you want a classifier that produces a *high recall* while keeping a *low false positive rate*. Compared to the default threshold of zero, the curve shows that we could achieve a significant higher recall (around 0.9) while only increasing the FPR slightly. The point closes 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.

You can find a comparison of the Random Forest and the SVC using ROC curves in Figure roc_curve_comparison.

```
from sklearn.metrics import roc_curve
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.title("roc_curve_comparison");
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)
```



As for the precision-recall curve, we often want to summarize the ROC curve using a single number, the area under the curve. Often the area under the ROC-curve is just called *AUC* (*area under the curve*) and it is understood that the curve in question is the ROC curve. We can compute the area under the ROC curve using the `roc_auc_score` function:

```
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: %f" % rf_auc)
print("AUC for SVC: %f" % svc_auc)
```

```
AUC for Random Forest: 0.936695


AUC for SVC: 0.916294
```

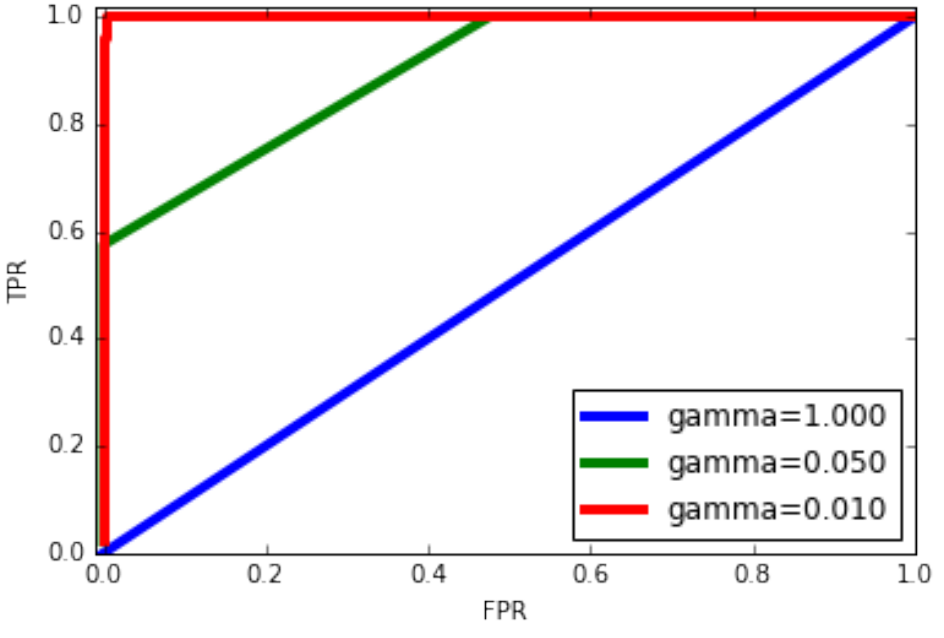
Comparing random forest and SVC using the AUC score, we find that Random Forest performs quite a bit better than SVC.

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). Predicting randomly always produces an AUC of 0.5, not matter how imbalanced the classes in a dataset are. This makes it a much better metric for imbalanced classification problems than accuracy.

The AUC can be interpreted as evaluating the *ranking* of positive samples. The AUC is equivalent to the probability that a randomly picked point of the positive class will have a higher score according to the classifier than a randomly picked point from the negative class. So an perfect AUC of 1 means that all positive points have a higher score than all negative points.

For classification 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 above of classifying all nines in the `digits` dataset versus all other digits. We will classify the dataset with an SVM with three different settings of the kernel bandwidth gamma:

```
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 = %.02f  accuracy = %.02f  AUC = %.02f" % (gamma, accuracy, auc))
    plt.plot(fpr, tpr, label="gamma=%.03f" % gamma, linewidth=4)
plt.xlabel("FPR")
plt.ylabel("TPR")
plt.xlim(-0.01, 1)
plt.ylim(0, 1.02)
plt.legend(loc="best")
```




```
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
```

The accuracy of all three settings of gamma is the same, 90%. This could either be chance performance, or it could be 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_function` is as good as random. With gamma=0.05, performance drastically improves to an AUC of 0.5. 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! [Footnote: 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.] Knowing this, we can adjust the threshold on this model, and obtain great predictions.

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

For this reason, we highly recommend using AUC when evaluating models on imbalanced data. Keep in mind that AUC does not make use of the default threshold, so

adjusting the decision threshold might be necessary to obtain useful classification results from a model with high AUC.

Multi-class classification

Now that we have discussed evaluation of binary classification tasks in-depth, let's move on to metrics to evaluate multi-class classification. Basically all metrics for multi-class classification are derived from binary classification metrics, but averaged over all classes.

Accuracy for multi-class 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, multi-class classification 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 above.

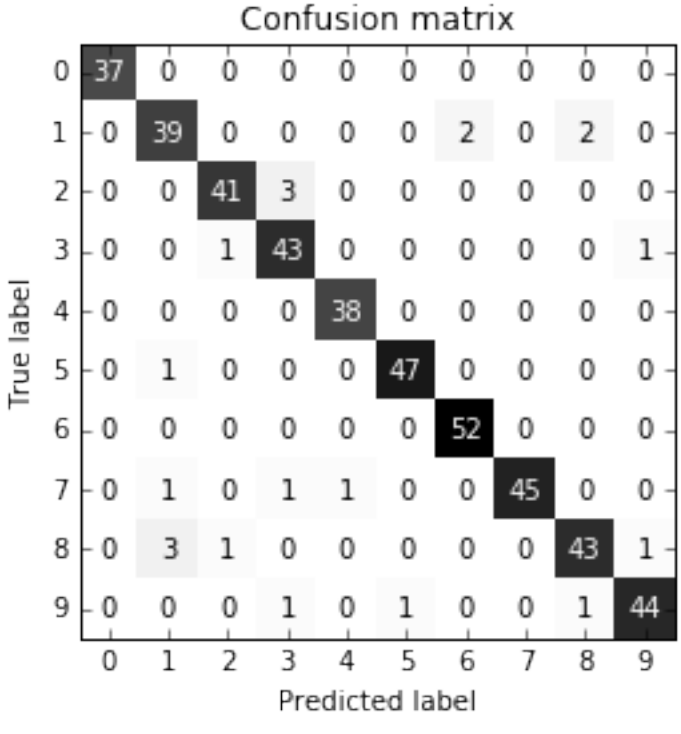
Let's apply these two detailed evaluation methods on the task of classifying the 10 different hand-written digits in the `digits` dataset:

```
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: %0.3f" % accuracy_score(y_test, pred))
print("confusion matrix:")
print(confusion_matrix(y_test, pred))
```

```
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]]
```

```
scores_image = mglearn.tools.heatmap(confusion_matrix(y_test, pred), xlabel='Predicted label', yla
                                     xticklabels=digits.target_names, yticklabels=digits.target_na
                                     cmap=plt.cm.gray_r, fmt="%d")
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```



The model has an accuracy of 95.6%, 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 visualy more appealing plot in Figure multi_class_confusion_matrix. For the first class, the digit 0, there are 37 samples in the class, and all of these samples were classified as class 0 (no false negatives for the zero class). We can see that because all other entries in the first row of the confusion matrix are zero. We can also see that no other digits was mistakenly classified as zero, because all other

entries in the first column of the confusion matrix are zero (no false positives for class zero).

Some digits that were confused with others are the digit two (third row), three of which were classified as the digit three (fourth column). There was also one digit three that was classified as two (third coloumn, fourth row) and one digit eight that was classified as two (thrid column, fourth row).

With the `classification_report` function, we can compute the precision, recall and f-score for each class:

```
print(classification_report(y_test, pred))
```

```
             precision    recall  f1-score   support


          0       1.00      1.00      1.00        37
          1       0.89      0.91      0.90        43
          2       0.95      0.93      0.94        44
          3       0.90      0.96      0.92        45
          4       0.97      1.00      0.99        38
          5       0.98      0.98      0.98        48
          6       0.96      1.00      0.98        52
          7       1.00      0.94      0.97        48
          8       0.93      0.90      0.91        48
          9       0.96      0.94      0.95        47


avg / total       0.95      0.95      0.95       450
```

Unsurprisingly, precision and recall are a perfect 1 for class zero, as there are no confusions with this class. For class seven on the other hand, precision is 1 because no other class was mistakenly classified as seven, while for class six, there are not false negatives, so the recall is 1. We can also see that the model has particular difficulties with classes eight and three.

The most commonly used metric for imbalanced datasets in the multi-class setting is the multi-class version of the f-score. The idea behind multi-class F-score is to compute 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 the 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 total number of false positives, false negatives and true positives over all classes, and then compute precision, recall and f-score using these counts.

If you care about each *sample* equally much, it is recommended to use `"micro"` average f1-score, if you care about each *class* equally much, it is recommended to use the `"macro"` average f1-score:

```
print("micro average f1 score: %f" % f1_score(y_test, pred, average="micro"))
print("macro average f1 score: %f" % f1_score(y_test, pred, average="macro"))
```

```
micro average f1 score: 0.953333

macro average f1 score: 0.954000
```

Regression metrics

Evaluation for regression can be done in similar detail as we did for classification above, for example by analyzing over-predicting the target versus under-predicting the target. However, in most application we've seen, using the default R^2 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 R^2 to be a more intuitive metric to evaluate regression models.

Using evaluation metrics in model selection

We now 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 desired evaluation metric you want to use. Say, for example, we want to evaluate the SVC 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 providing `"roc_auc"` as the `scoring` parameter:

```
# default scoring for classification is accuracy
print("default scoring ", 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 ", explicit_accuracy)
roc_auc =  cross_val_score(SVC(), digits.data, digits.target == 9, scoring="roc_auc")
print("AUC scoring ", roc_auc)
```

```
default scoring  [ 0.9  0.9  0.9]

explicit accuracy scoring  [ 0.9  0.9  0.9]

AUC scoring  [ 0.994  0.99   0.996]
```

Similarly we can change the metric used to pick the best parameters in `GridSearchCV`:

```
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)):", grid.best_score_)
print("Test set AUC: %.3f" % roc_auc_score(y_test, grid.decision_function(X_test)))
print("Test set accuracy %.3f: " % grid.score(X_test, y_test))

# 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):", grid.best_score_)
print("Test set AUC: %.3f" % roc_auc_score(y_test, grid.decision_function(X_test)))
print("Test set accuracy %.3f: " % grid.score(X_test, y_test))
```

```
Grid-Search with accuracy

Best parameters: {'gamma': 0.0001}

Best cross-validation score (accuracy)): 0.970304380104

Test set AUC: 0.992

Test set accuracy 0.973:
```

```
Grid-Search with AUC

Best parameters: {'gamma': 0.01}

Best cross-validation score (AUC): 0.997467845028

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, both in terms of AUC and even in terms of accuracy [Footnote: Finding a higher accuracy solution using AUC is likely a consequence of accuracy being a bad measure of model performance on imbalanced data].

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 commonly used values are `r2` for the R^2 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:

```
from sklearn.metrics.scorer import SCORERS
print(sorted(SCORERS.keys()))
```

```
['accuracy', 'adjusted_rand_score', 'average_precision', 'f1', 'f1_macro', 'f1_micro', 'f1_samples
```

Summary and outlook

In this chapter we discussed cross-validation, grid-search and evaluation metrics, the corner-stones 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: 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 "used 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 above is a train-test split for evaluation, and using cross-validation on the training set for model and parameter selection.

The second important point is 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. 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 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. Make sure you understand what these consequences are, and pick an evaluation metric accordingly.

The techniques model evaluation and selection techniques we described so far are the most important tools in a data scientists toolbox. However, grid search and cross validation as we described it 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 complex chains of algorithms.