```
logreg.fit(iris.data, iris.target)
# fit the model again on the blob dataset
logreg.fit(X_train, y_train)
# the outcome is the same as training a "fresh" model:
new_logreg = LogisticRegression()
new_logreg.fit(X_train, y_train)

# predictions made by the two models are the same
pred_new_logreg = new_logreg.predict(X_test)
pred_logreg = logreg.predict(X_test)

pred_logreg == pred_new_logreg
```

```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True], dtype=bool)
```

As you can see, fitting the `logreg` model first on the `iris` dataset has no effect. The `iris` dataset has a different number of features and classes than the `blobs` dataset, but all about the first fit is erased when `fit` is called again.

Next, we will go into several shortcuts that allow you to write less code for common tasks, and speed up some computations. The first way to write more compact code is to make use *method chaining*.

Method chaining

The `fit` method of all scikit-learn models returns `self`. This allows you to write code like this:

```
# instantiate model and fit it in one line
logreg = LogisticRegression().fit(X_train, y_train)
```

Here, we used the return value of `fit` (which is `self`) to assign the trained model to the variable `logreg`. This concatenation of method calls (here `__init__` and then `fit`) is known as *method chaining*. Another common application of method chaining in scikit-learn is to `fit` and `predict` in one line:

```
logreg = LogisticRegression()
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

Finally, you can even do model instantiation, fitting and predicting in one line:

```
y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)
```

This very short variant is not ideal, though. A lot is happening in a single line, which might make the code hard to read. Additionally, the fitted logistic regression model isn't stored in any variable. So we can't inspect it, or use it to predict on any other data.

Shortcuts and efficient alternatives

Often, you want to `fit` a model on some dataset, and then immediately `predict` on the same data, or `transform` it. These are very common tasks, which can often be computed more efficiently than simply calling `fit` and then `predict` or `fit` and then `transform`. For this use-case, all models that have a `predict` method also have a `fit_predict` method, and all model that have a `transform` method also have a `fit_transform` method. Here is an example using `PCA`:

```
from sklearn.decomposition import PCA
pca = PCA()
# calling fit and transform in sequence (using method chaining)
X_pca = pca.fit(X).transform(X)
# same result, but more efficient computation
X_pca_2 = pca.fit_transform(X)
```

While `fit_transform` and `fit_predict` are not more efficient for all algorithms, it is still good practice to use them when trying to predict on, or transform the training set.

For some unsupervised methods that we saw in Chapter 3, like some clustering and manifold learning methods, using `fit_transform` and `fit_predict` are the only options. For example `DBSCAN` does not have a `predict` method, only `fit_predict`, and `t-SNE` does not have a `transform` method, only `fit_transform`. T-SNE and DBSCAN are algorithms that can not be applied to new data, they can only be applied to the training data.

Important Attributes

scikit-learn has some standard attributes that allow you to inspect what a model learned. All these attributes are available after the call to `fit`, and, as we mentioned before, all attributes learned from the data are marked with a trailing underscore.

We already discussed the following common attributes:

- For clustering algorithms, the `labels_` attribute stores the cluster membership for the training data.
- For manifold learning algorithms, the `embedding_` attribute stores the embedding (transformation) of the training data in the lower-dimensional space.
- For linear models, the `coef_` attribute stores the weight or coefficient vector.

- For linear decomposition and dimensionality reduction methods, `components_` stores the array of components (the prototypes in the additive decomposition in Figure decomposition in Chapter 3).

Additionally, for classifiers, `classes_` contains the names of the classes the classifier was trained on, that is the unique entries of the training labels `y_train`:

```
import numpy as np
logreg = LogisticRegression()
# fit model using original data
logreg.fit(iris.data, iris.target)
print("unique entries of iris.target: %s" % np.unique(iris.target))
print("classes using iris.target: %s" % logreg.classes_)

# represent each target by its class name
named_target = iris.target_names[iris.target]
logreg.fit(iris.data, named_target)
print("unique entries of named_target: %s" % np.unique(named_target))
print("classes using named_target: %s" % logreg.classes_)
```

```
unique entries of iris.target: [0 1 2]

classes using iris.target: [0 1 2]

unique entries of named_target: ['setosa' 'versicolor' 'virginica']

classes using named_target: ['setosa' 'versicolor' 'virginica']
```

Summary and outlook

You should now be be intimately familiar with the interfaces of supervised and unsupervised models in scikit-learn, and how to use them. With a good grasp on how to use the different models, we will continue with more complex topics, such as evaluating and selecting models.

CHAPTER 5

Representing Data and Engineering Features

So far, we assumed that our data comes in as a two-dimensional array of floating point numbers, where each column is a *continuous feature* that describes the data points. For many applications, this is not how the data is collected. A particular common type of feature is *categorical features*, also known as *discrete features*, which are usually not numeric. The distinction between categorical feature and continuous feature is analogous to the distinction between classification and regression - only on the input side, not the output side.

Examples for continuous features that we saw are pixel brightnesses and size measurements of plant flowers. Examples for categorical features are the brand of a product, the color of a product, or the department (books, clothing, hardware) it is sold in. These are all properties that can describe a product, but they don't vary in a continuous way. A product belongs either in the clothing department or in the books department. There is no middle ground between books and clothing, and no natural order for the different categories (books is not greater or smaller than clothing, hardware is not between books and clothing, etc).

Regardless of the type of features your data consists of, how you represent them can have an enormous effect on the performance of machine learning models. We saw in Chapter 2 and Chapter 3 that scaling of the data is important. In other words, if you don't rescale your data (say, to unit variance), then it makes a difference whether you represent a measurement in centimeters or inches. We also saw in Chapter 2 that it can be helpful to *augment* your data with additional features, like adding interactions (products) of features or more general polynomials.

The question of how to represent your data best for a particular application is known as *feature engineering*, and it is one of the main tasks of data scientists and machine

learning practitioners trying to solve real-world problems. Representing your data in the right way can have a bigger influence on the performance of a supervised model than the exact parameters you choose.

We will first go over the important and very common case of categorical features, and then give some examples of helpful transformations for specific combinations of features and models.

Categorical Variables

As an example, we will use the dataset of adult incomes in the United States, derived from the 1994 census database.

The task of the `adult` dataset is to predict whether a worker has an income of over 50.000$ or under 50.000$.

The features in this dataset include the workers age, how they are employed (self employed, private industry employee, goverment employee, ...), their education, their gender, their working hours per week, occupation and more.

Below is a table showing the first few entries in the data set:

	age	workclass	education	gender	hours-per-week	occupation	income
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K
5	37	Private	Masters	Female	40	Exec-managerial	<=50K
6	49	Private	9th	Female	16	Other-service	<=50K
7	52	Self-emp-not-inc	HS-grad	Male	45	Exec-managerial	>50K
8	31	Private	Masters	Female	50	Prof-specialty	>50K
9	42	Private	Bachelors	Male	40	Exec-managerial	>50K
10	37	Private	Some-college	Male	80	Exec-managerial	>50K

The task is phrased as a classification task with the two classes being income `<=50k` and `>50k`. It would also be possible to predict the exact income, and make this a regression task. However, that would be much more difficult, and the 50K division is interesting to understand on its own.

In this dataset, `age` and `hours-per-week` are continuous features, which we know how to treat. The `workclass`, `education`, `sex` and `ocupation` features are categorical, however. All of them come from a fixed list of possible values, as opposed to a range, and denote a qualitative property, as opposed to a quantity.

As a starting point, let's say we want to learn a logistic regression classifier on this data. We know from Chapter 2 that a logistic regression makes predictions $\hat{y}$ using the formula

\begin{align*}

&\hat{y} = w[0] *x[0]* + *w[1]* x[1] + \dotsc + w[p] * x[p] + b > 0 &\text{ (1) linear binary classification}

\end{align*}

where `w[i]` and `b` are coefficients learned from the training set and `x[i]` are the input features.

This formula makes sense when `x[i]` are numbers, but not when `x[2]` is `"Masters"` or `"Bachelors"`.

So clearly we need to represent our data in some different way when applying logistic regression.

One-Hot-Encoding (Dummy variables)

By far the most common way to represent categorical variables is using the *one-hot-encoding* or *one-out-of-N* encoding, also known as *dummy variables*.

The idea behind dummy variables is to replace a categorical variable with one or more new features that can have the values 0 and 1. The values 0 and 1 make sense in Formula (1) (and for all other models in scikit-learn), and we can represent any number of categories by introducing one new feature per category as follows.

Let's say for the `workclass` feature we have possible values of `"Government Employee"`, `"Private Employee"`, `"Self Employed"` and `"Self Employed Incorporated"`. To encode this four possible values, we create four new features, called `"Government Employee"`, `"Private Employee"`, `"Self Employed"` and `"Self Employed Incorporated"`. The feature is 1 if `workclass` for this person has the corresponding value, and 0 otherwise.

So exactly one of the four new features will be 1 for each data point. This is why this is called one-hot or one-out-of-N encoding.

The principle is illustrated here. A single feature is encoded using four new features. When using this data in a machine learning algorithm, we would drop the original `workclass` feature and only keep the 0-1 features.

| workclass | Government Employee | Private Employee | Self Employed | Self Employed Incorporated |

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

| Government Employee | 1 | 0 | 0 | 0 |

| Private Employee | 0 | 1 | 0 | 0 |

| Self Employed | 0 | 0 | 1 | 0 |

| Self Employed Incorporated | 0 | 0 | 0 | 1 |

There are two ways to convert your data to a one-hot encoding of categorical variables, either using `pandas` or using `scikit-learn`. At the time of writing, using `pandas` for this setting is slightly easier, so let's go this route. First we load the data using pandas from a comma seperated values (CSV) file:

```
import pandas as pd
# The file has no headers naming the columns, so we pass header=None and provide the column names
data = pd.read_csv("/home/andy/datasets/adult.data", header=None, index_col=False,
                   names=['age', 'workclass', 'fnlwgt', 'education',  'education-num',
                          'marital-status', 'occupation', 'relationship', 'race', 'gender',
                          'capital-gain', 'capital-loss', 'hours-per-week', 'native-country', 'inc
# For illustration purposes, we only select some of the columns:
data = data[['age', 'workclass', 'education', 'gender', 'hours-per-week', 'occupation', 'income']]
# print the first 5 rows
data.head()
```

	age	workclass	education	gender	hours-per-week	occupation	income
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K

Checking string-encoded categorical data

After reading a dataset like this, it is often good to first check if a column actually contains meaningful categorical data. When working with data that was input by humans (say users on a website), there might not be a fixed set of categories, and differences in spelling and capitalization might require preprocessing. For example, it might be that some people specified gender as "male" and some as "man", and we might want to represent these two inputs using the same category.

A good way to check the contents of a column is using the `value_counts` function of a pandas series (the type of a single column in a dataframe), to show us what the unique values are, and how often they appear:

```
data.gender.value_counts()
```

```
 Male      21790

 Female    10771

Name: gender, dtype: int64
```

We can see that there are exactly two values for gender in this datasets, `Male` and `Female`, meaning the data is already in a good format to be represented using one-hot-encoding. In a real application, you should look at all columns, and check their values. We will skip this here for brevity's sake.

There is a very simple way to encode the data in pandas, using the `get_dummies` function:

```
print("Original features:\n", list(data.columns), "\n")
data_dummies = pd.get_dummies(data)
print("Features after get_dummies:\n", list(data_dummies.columns))
```

```
Original features:

 ['age', 'workclass', 'education', 'gender', 'hours-per-week', 'occupation', 'income']



Features after get_dummies:

 ['age', 'hours-per-week', 'workclass_ ?', 'workclass_ Federal-gov', 'workclass_ Local-gov', 'work
```

You can see that the continuous features `age` and `hours-per-week` were not touched, while the categorical features were expanded into one new feature for each possible value:

```
data_dummies.head()
```

	age	hours-per-week	workclass_ ?	workclass_ Federal-gov	workclass_ Local-gov	workclass_ Never-worked	workclass_ Private	workclass_ Self-emp-inc	workclass_ Self-emp-not-inc	workclass_ State-gov	...	occupati Machine-op-inspct
0	39	40	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	...	0.0
1	50	13	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	...	0.0
2	38	40	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	...	0.0
3	53	40	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	...	0.0
4	28	40	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	...	0.0

5 rows × 46 columns

We can use the ``values`` attribute to convert the ``data_dummies`` dataframe into a numpy array, and then train a machine learning model on it. Be careful to separate the

target variable (which is now encoded in two ``income`` columns) from the data before training a model. Including the output variable, or some derived property of the output variable, into the feature representation is a very common mistake in building supervised machine learning models. [Warning box] Careful: column indexing in pandas includes the end of the range, so ```'age':'occupation_ Transport-moving'`` is inclusive of ``occupation_ Transport-moving``. This is in constrast to slicing a numpy array, where the end of a range is not included: ``np.arange(11)[0:10]`` does not include the entry with index 10. [/Warning box] ``` # Get only the columns containing features, that is all columns from 'age' to 'occupation_ Transport-moving' # This range contains all the features but not the target features = data_dummies.ix[:, 'age':'occupation_ Transport-moving'] # extract numpy arrays X = features.values y = data_dummies['income_ >50K'].values print(X.shape, y.shape) ``` ``` (32561, 44) (32561,) ``` Now the data is represented in a way that scikit-learn can work with, and we can proceed as usual: ``` from sklearn.linear_model import LogisticRegression from sklearn.model_selection import train_test_split X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0) logreg = LogisticRegression() logreg.fit(X_train, y_train) print(logreg.score(X_test, y_test)) ``` ``` 0.808745854318 ``` [FIXME Warning callout] Above, we called ``get_dummies`` on a dataframe containing both the training and the test data. This is imporant to ensure categorical values are represented in the same way in the training set and the test set. Imagine we had the training and the test set in two different dataframes. If the "Private Employee" value for the ``workclass`` feature does not appear in the test set, pandas would assume there are only three possible values for this feature, and would create only three new dummy features. Now our training and test set have different numbers of features, and we can't apply the model we learned on the training set to the test set any more. Even worse, imagine the ``workclass`` feature had the values "Government Employee" and "Private Employee" in the training set, and "Self Employed" and "Self Employed Incorporated" in the test set. In both cases, pandas would create two new dummy features, so the encoded dataframes would have the same number of features. However, the two dummy features have entirely different meanings on the training and the test set. The column that means "Goverment Employee" for the training set would encode "Self Employed" for the test set. If we build a machine learning model on this data, it would work very badly, because it assumes the columns mean the same (because they are in the same position), when they mean very different things. To fix this, either call ``get_dummies`` on a dataframe that contains the training and the test data points, or make sure that the column names are the same for training and test set after calling ``get_dummies``, to ensure they have the same semantics. ### Numbers can encode categoricals In the example of the ``adult`` dataset, the categorical variables were encoded as strings. On the one hand, that opens up the possibility of spelling errors, but on the other hand, it clearly marks a variable as categorical. Often, whether for ease of storage or because of the way the data is collected, categorical variables are actually encoded as integers. For example, imagine the census data in the ``adult`` dataset was collected using a questionaire, and the answers for ``workclass`` were recorded as 0 (first box ticked), 1 (second box ticked), 2 (third box ticked) and so on. Now the column contains number from 0 to eight, instead of strings like ``Private``, and looking at the table representing

the dataset, it is not clear from the numbers whether we should treat this variable as continuous or categorical. Knowing that the numbers indicate employment status, it is clear that these are very distinct states, and should not be modelled by a single continuous variable. [Warning box] Categorical features are often encoded using integers. That they are numbers doesn't mean that they should be treated as continuous features, as we mentioned above. It is not always clear whether an integer feature should be treated as continuous or discrete (and [one-hot-encoded). If there is no ordering between the semantics that are encoded (like the ``workclass`` example above), the feature must be treated as discrete. For other cases like 5-star ratings, the better encoding depends on the the particular task and data and which machine learning algorithm is used.[/warning box] The ``get_dummies`` function in pandas treats all numbers as continuous and will not create dummy variables for them. To get around this, you can either use ``scikit-learn``'s ``OneHotEncoder``, for which you can specify which variables are continuous and which are discrete, or convert numeric columns in the dataframe to strings. To illustrate, we create a dataframe object with two columns, one containing strings and one containing integers. ``` # create a dataframe with an integer feature and a categorical string feature demo_df = pd.DataFrame({'Integer Feature': [0, 1, 2, 1], 'Categorical Feature': ['socks', 'fox', 'socks', 'box']}) demo_df ```

	Categorical Feature	Integer Feature
0	socks	0
1	fox	1
2	socks	2
3	box	1

Using get_dummies will only encode the string feature, and not change "Integer Feature": ``` pd.get_dummies(demo_df) ```

	Integer Feature	Categorical Feature_box	Categorical Feature_fox	Categorical Feature_socks
0	0	0.0	0.0	1.0
1	1	0.0	1.0	0.0
2	2	0.0	0.0	1.0
3	1	1.0	0.0	0.0

If you want dummy variables to be created for the "Integer Feature" column, one solution is to convert it to a string. Then, both features will be treated as categorical: ``` demo_df['Integer Feature'] = demo_df['Integer Feature'].astype(str) pd.get_dummies(demo_df) ```

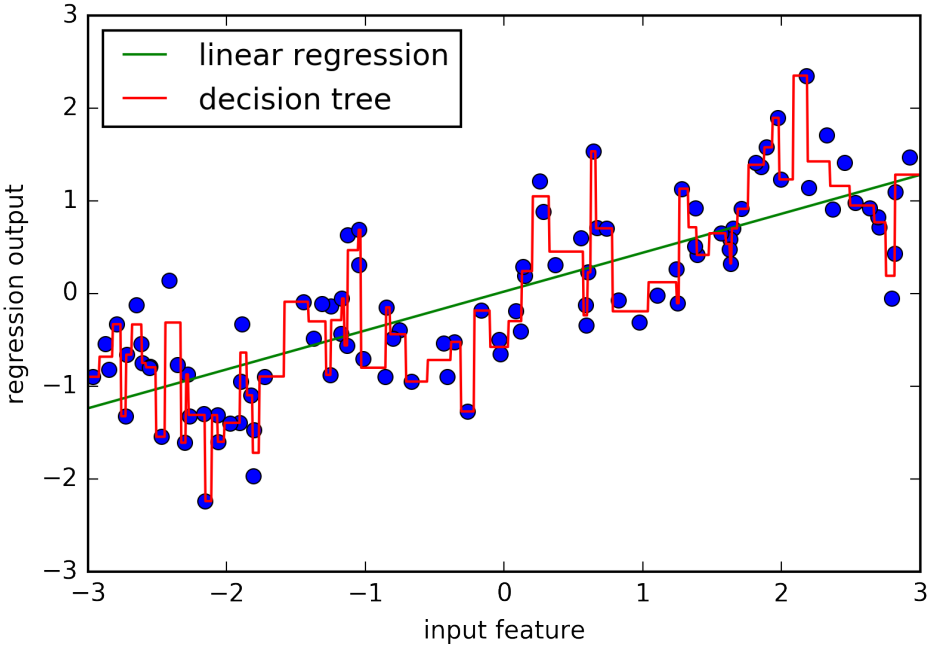
	Categorical Feature_box	Categorical Feature_fox	Categorical Feature_socks	Integer Feature_0	Integer Feature_1	Integer Feature_2
0	0.0	0.0	1.0	1.0	0.0	0.0
1	0.0	1.0	0.0	0.0	1.0	0.0
2	0.0	0.0	1.0	0.0	0.0	1.0
3	1.0	0.0	0.0	0.0	1.0	0.0

Binning, Discretization, Linear Models and Trees

The best way to represent data not only depends on the semantics of the data, but also on the kind of model you are using. Two large and very commonly used families of models are linear models and tree-based models (such as decision trees, gradient boosted trees and random forests). Linear models and tree-based models have very different properties when it comes to how they work with different feature representations.

Let's go back to the `wave` regression dataset that we used in chapter 2. It only has a single input feature. Here is a comparison of a linear regression model and a decision tree regressor on this dataset:

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

X, y = mglearn.datasets.make_wave(n_samples=100)
plt.plot(X[:, 0], y, 'o')
line = np.linspace(-3, 3, 1000)[:-1].reshape(-1, 1)

reg = LinearRegression().fit(X, y)
plt.plot(line, reg.predict(line), label="linear regression")

reg = DecisionTreeRegressor(min_samples_split=3).fit(X, y)
plt.plot(line, reg.predict(line), label="decision tree")
plt.ylabel("regression output")
plt.xlabel("input feature")
plt.legend(loc="best")
```



As you know, linear models can only model linear relationships, which are lines in the case of a single feature. The decision tree an build a much more complex model of the data.

However, this is strongly dependent on our representation of the data. One way to make linear models more powerful on continuous data is to using *binning* (also known as *discretization*) of the feature to split it up into multiple features as follows:

We imagine a partition of the input range of -3 to 3 of this feature into a fixed number of *bins*. Here, we pass bin boundaries from -3 to 3 with 11 equally sized steps. An array of 11 bin boundaries will create 10 bins - they are the space in between two consecutive boundaries.

```
np.set_printoptions(precision=2)
bins = np.linspace(-3, 3, 11)
bins
```

```
array([-3. , -2.4, -1.8, -1.2, -0.6,  0. ,  0.6,  1.2,  1.8,  2.4,  3. ])
```

Here, the first bin contains all data points with feature values -3 to -2.68, the second bin contains all points feature values from -2.68 to -2.37 and so on.

Next, we record for each data point which bin it falls into. This can be easily computed using the `np.digitize` function:

```
which_bin = np.digitize(X, bins=bins)
print("\nData points:\n", X[:5])
print("\nBin membership for data points:\n", which_bin[:5])
```

```
Data points:

 [[-0.75]

 [ 2.7 ]

 [ 1.39]

 [ 0.59]

 [-2.06]]


Bin membership for data points:

 [[ 4]

 [10]

 [ 8]

 [ 6]

 [ 2]]
```

What we did here is transform the single continuous input feature in the `wave` dataset into a categorical feature which encodes which bin a data point is in. To use a scikit-learn model on this data, we transform this discrete feature to a one-hot encoding using the `OneHotEncoder` from the preprocessing module. The `OneHotEncoder` does the same encoding as `pandas.get_dummies`, though it currently only works on categorical variables that are integers.

```
from sklearn.preprocessing import OneHotEncoder
# transform using the OneHotEncoder.
encoder = OneHotEncoder(sparse=False)
# encoder.fit finds the unique values that appear in which_bin
encoder.fit(which_bin)
# transform creates the one-hot encoding
X_binned = encoder.transform(which_bin)
print(X_binned[:5])
```

```
[[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]

 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
```

```
[ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]

[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]

[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

Because we specified 10 bins, the transformed dataset X_binned now is made up of 10 features:

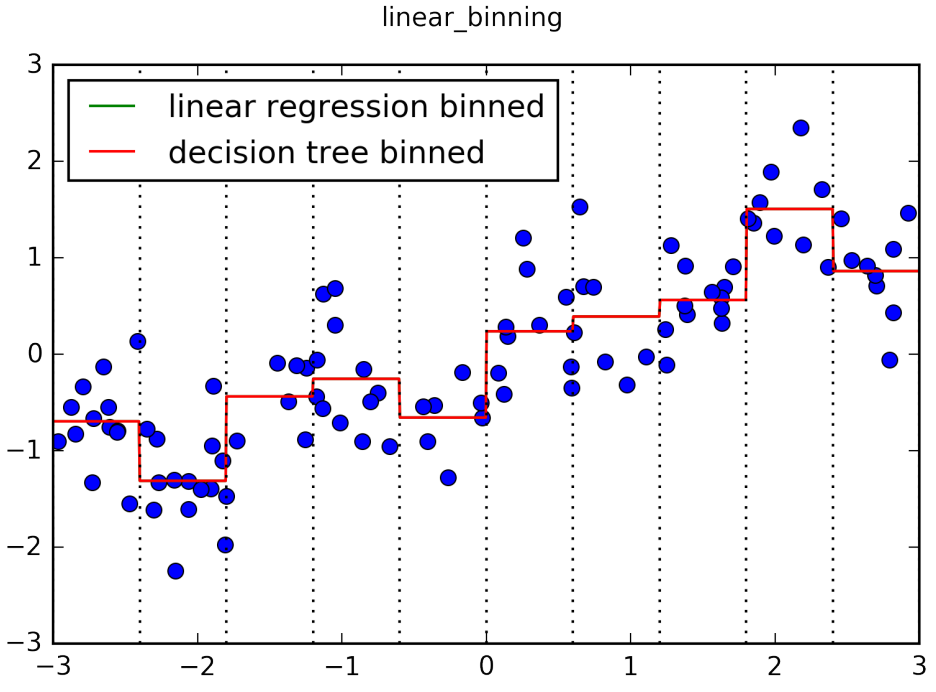
```
X_binned.shape
```

```
(100, 10)
```

Now we build a new linear regression model and a new decision tree model on the one-hot encoded data.

The result is visualized below, together with the bin boundaries, shown as dotted black lines:

```
line_binned = encoder.transform(np.digitize(line, bins=bins))

plt.plot(X[:, 0], y, 'o')
reg = LinearRegression().fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='linear regression binned')

reg = DecisionTreeRegressor(min_samples_split=3).fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='decision tree binned')
for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k')
plt.legend(loc="best")
plt.suptitle("linear_binning")
```



The green line and red line are exactly on top of each other, meaning the linear regression model and the decision tree make exactly the same predictions. For each bin, they predict a constant value. As features are constant within each bin, any model must predict the same value for all points within a bin. Comparing what the models learned before binning the features and after, we see that the linear model became much more flexible, because it now has a different value for each bin, while the decision tree model got much less flexible.

Binning features generally has no beneficial effect for tree-based models, as the model can learn to split up the data anywhere. In a sense, that means the decision trees can learn a binning that is particularly beneficial for predicting on this data. Additionally, decision trees look at multiple features at once, while binning is usually done on a per-feature basis.

However, the linear model benefited greatly in expressiveness from the transformation of the data.

If there are good reasons to use a linear model for a particular data set, say because it is very large and high-dimensional, but some features have non-linear relations with the output, binning can be a great way to increase modelling power.

Interactions and Polynomials

Another way to enrich a feature representation, in particular for linear models, is adding *interaction features* and *polynomial features* of the original data. This kind of feature engineering is often used in statistical modelling, but also common in many practical machine learning applications.

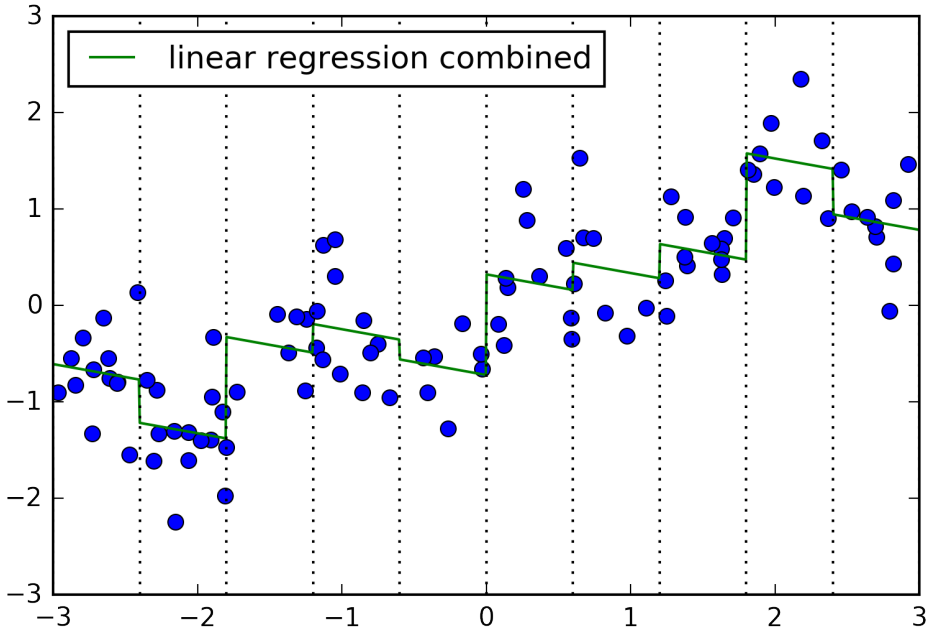
As a first example, look again at Figure linear_binning above. The linear model learned a constant value for each bin on the `wave` dataset. We know, however, that linear models can not only learn offsets, but also slopes. One way to add a slope to the linear model on the binned data, is to add the original feature (the x axis in the plot) back in.

This leads to a 11 dimensional dataset:

```
X_combined = np.hstack([X, X_binned])
print(X_combined.shape)
```

```
(100, 11)
```

```
plt.plot(X[:, 0], y, 'o')

reg = LinearRegression().fit(X_combined, y)

line_combined = np.hstack([line, line_binned])
plt.plot(line, reg.predict(line_combined), label='linear regression combined')

for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k')
plt.legend(loc="best")
```



Now, the model learned an offset for each bin, together with a slope. The learned slope is downward, and shared across all the bins - there is the single x-axis feature which has a single slope. Because the slope is shared across all bins, it doesn't seem to be very helpful. We would rather have a separate slope for each bin! We can achieve this by adding an interaction or product feature that indicates in which bin a data-point is *and* where it lies on the x-axis.

This feature is a product of the bin-indicator and the original feature. Let's create this dataset:

```
X_product = np.hstack([X_binned, X * X_binned])
print(X_product.shape)
```

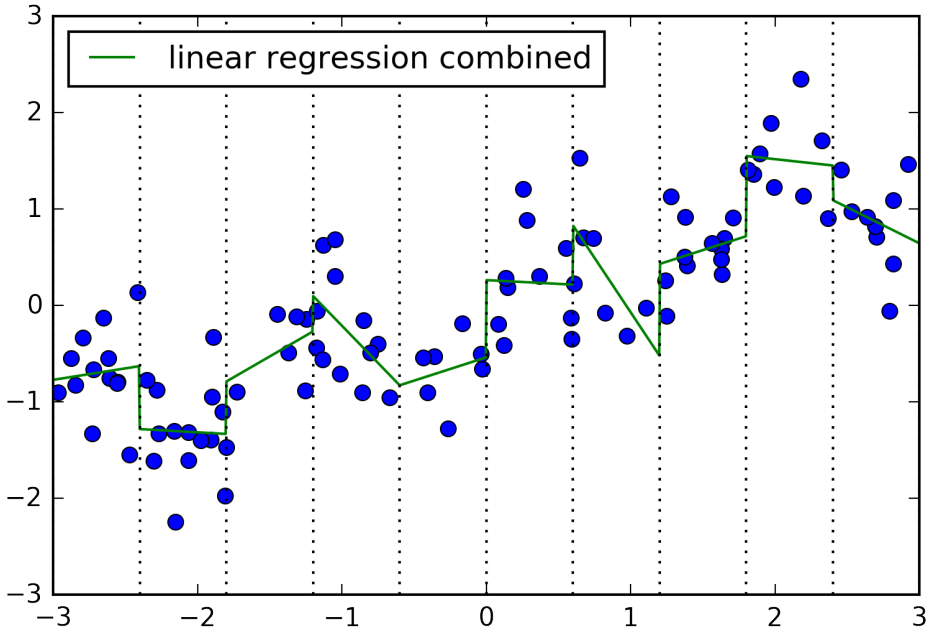
```
(100, 20)
```

This dataset not has 20 features: the indicator for which bin a data point is in, and a product of the original feature and the bin indicator. You can think of the product feature as a separate copy of the x-axis feature for each bin. It is the original feature within the bin, and zero everywhere else.

Here is the linear model on this new representation:

```
plt.plot(X[:, 0], y, 'o')

reg = LinearRegression().fit(X_product, y)
```

```
line_product = np.hstack([line_binned, line * line_binned])
plt.plot(line, reg.predict(line_product), label='linear regression combined')

for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k')
plt.legend(loc="best")
```



As you can see, now each bin has its own offset and slope in this model.

Using binning is one way to expand a continuous feature. Another one is to use *polynomials* of the original features.

For a given feature `x`, we might want to consider `x ** 2`, `x ** 3`, `x ** 4` and so on.

This is implemented in `PolynomialFeatures` in the preprocessing module:

```
from sklearn.preprocessing import PolynomialFeatures

# include polynomials up to x ** 10:
poly = PolynomialFeatures(degree=10)
poly.fit(X)
X_poly = poly.transform(X)
```

Using a degree of 10 yields 11 features, as by default, a constant feature (`x ** 0`) is added, too:

```
X_poly.shape
```

```
(100, 11)
```

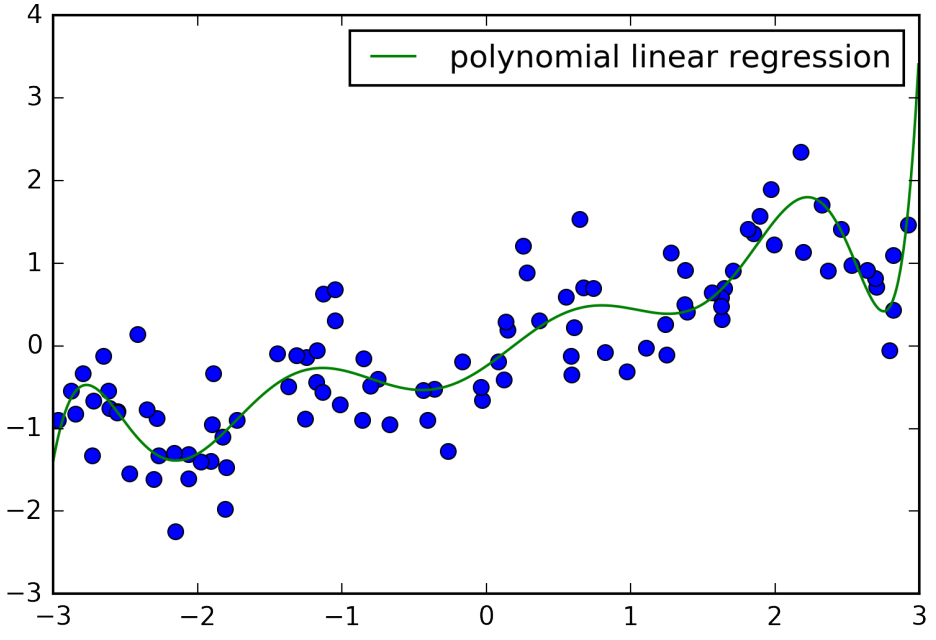
You can obtain the semantics of the features by looking at the `powers_` attribute, which gives the exponents for each feature:

```
poly.powers_
```

```
array([[ 0],
       [ 1],
       [ 2],
       [ 3],
       [ 4],
       [ 5],
       [ 6],
       [ 7],
       [ 8],
       [ 9],
       [10]])
```

Using polynomial features together with a linear regression model yields the classical model of *polynomial regression*:

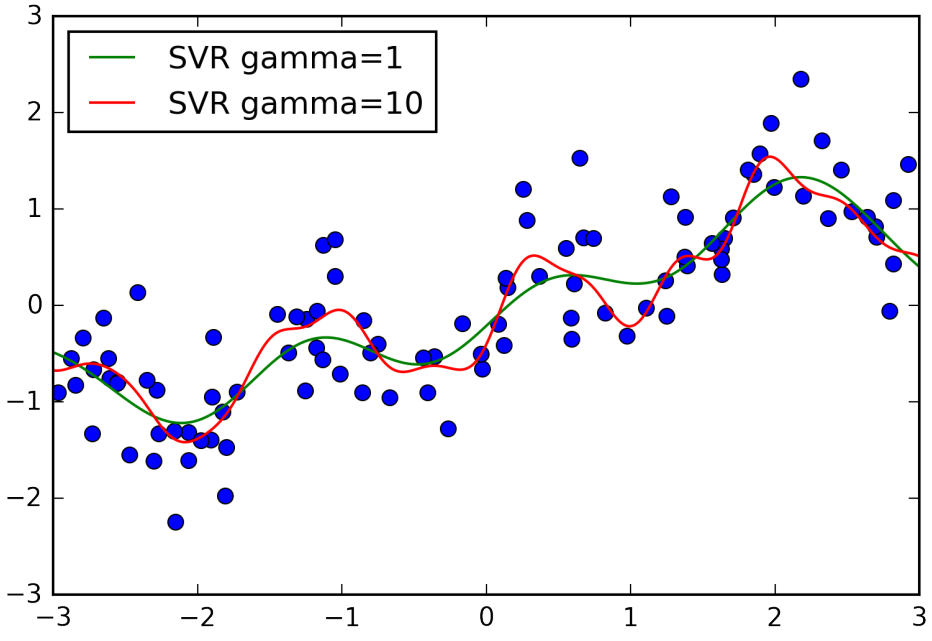
```
plt.plot(X[:, 0], y, 'o')

reg = LinearRegression().fit(X_poly, y)

line_poly = poly.transform(line)
plt.plot(line, reg.predict(line_poly), label='polynomial linear regression')
plt.legend(loc="best")
```



As you can see, polynomial feature yield a very smooth fit on this one-dimensional data. However, polynomials of high degree tend to behave in extreme ways on the boundaries or in regions of little data.

As a comparison, here is a kernel SVM model learned on the original data, without any transformation:

```
from sklearn.svm import SVR
plt.plot(X[:, 0], y, 'o')

for gamma in [1, 10]:
    svr = SVR(gamma=gamma).fit(X, y)
    plt.plot(line, svr.predict(line), label='SVR gamma=%d' % gamma)

plt.legend(loc="best")
```



Using a more complex model, a kernel SVM, we are able to learn a similarly complex prediction to the polynomial regression without using any transformations of the features.

As a more realistic application of interactions and polynomials, let's look again at the Boston Housing data set. We already used polynomial features on this dataset in Chapter 2. Now let us have a look at how these features were constructed, and at how much the polynomial features help. First we load the data, and rescale it to be between 0 and 1 using `MinMaxScaler`:

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target, random_state=0)

# rescale data:
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Now, we extract polynomial features and interactions up to a degree of 2:

```
poly = PolynomialFeatures(degree=2).fit(X_train_scaled)
X_train_poly = poly.transform(X_train_scaled)
```

```
X_test_poly = poly.transform(X_test_scaled)
print(X_train.shape)
print(X_train_poly.shape)
```

```
(379, 13)

(379, 105)
```

The data originally had 13 features, which were expanded into 105 interaction features. These new features represent all possible interactions between two different original features, as well as the square of each original feature. `degree=2` here means that we look at all features that are the product of up to two original features. The exact correspondence between input and output features can be found using the `get_feature_names` method:

```
print(poly.get_feature_names())
```

```
['1', 'x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10', 'x11', 'x12', 'x0^2', 'x0
```

The first new feature is a constant feature, called "1" here. The next 13 features are the original features (called "x0" to "x12"). Then follows the first feature squared ("x0^2") and combinations of the first and the other features.

Let's compare the performance using Ridge on the data with and without interactions:

```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train_scaled, y_train)
print("score without interactions: %f" % ridge.score(X_test_scaled, y_test))
ridge = Ridge().fit(X_train_poly, y_train)
print("score with interactions: %f" % ridge.score(X_test_poly, y_test))
```

```
score without interactions: 0.621370

score with interactions: 0.753423
```

Clearly the interactions and polynomial features gave us a good boost in performance when using Ridge. When using a more complex model like a random forest, the story is a bit different, though:

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=100).fit(X_train_scaled, y_train)
print("score without interactions: %f" % rf.score(X_test_scaled, y_test))
rf = RandomForestRegressor(n_estimators=100).fit(X_train_poly, y_train)
print("score with interactions: %f" % rf.score(X_test_poly, y_test))
```

```
score without interactions: 0.794226

score with interactions: 0.775016
```

You can see that even without additional features, the random forest beats the performance of Ridge. Adding interactions and polynomials actually decreases performance slightly.

Univariate Non-linear transformations

We just saw that adding squared or cubed features can help linear models for regression. There are other transformations that often prove useful for transforming certain features, in particular applying mathematical functions like `log`, `exp` or `sin`. While tree-based models only care about the ordering of the features, linear models and neural networks are very tied to the scale and distribution of each feature, and if there is a non-linear relation between the feature and the target, that becomes hard to model---in particular in regression. The functions `log` and `exp` can help adjusting the relative scales in the data so that they can be captured better by a linear model or neural network.

The `sin` or `cos` functions can come in handy when dealing with data that encodes periodic patterns.

Most models work best when each feature (and in regression also the target) are loosely Gaussian distributed, that is a histogram of each feature should have something resembling the familiar bell-curve shape. Using transformations like `log` and `exp` are a hacky, but simple and efficient way to achieve this. A particular common case when such a transformation can be helpful is when dealing with integer count data. By count data, we mean features like "how often did user A log in". Counts are never negative, and often follow particular statistical patterns. We are using a synthetic dataset of counts here, that has properties similar to those you can find in the wild. The features are all integer, while the response is continuous:

```
rnd = np.random.RandomState(0)
X_org = rnd.normal(size=(1000, 3))
w = rnd.normal(size=3)

X = np.random.poisson(10 * np.exp(X_org))
y = np.dot(X_org, w)
```

Let's look at the first ten entries of the first feature. All are integer and positive, but apart from that it's hard to make out a particular pattern:

If we count the appearance of each value, the distribution of values becomes more clear:

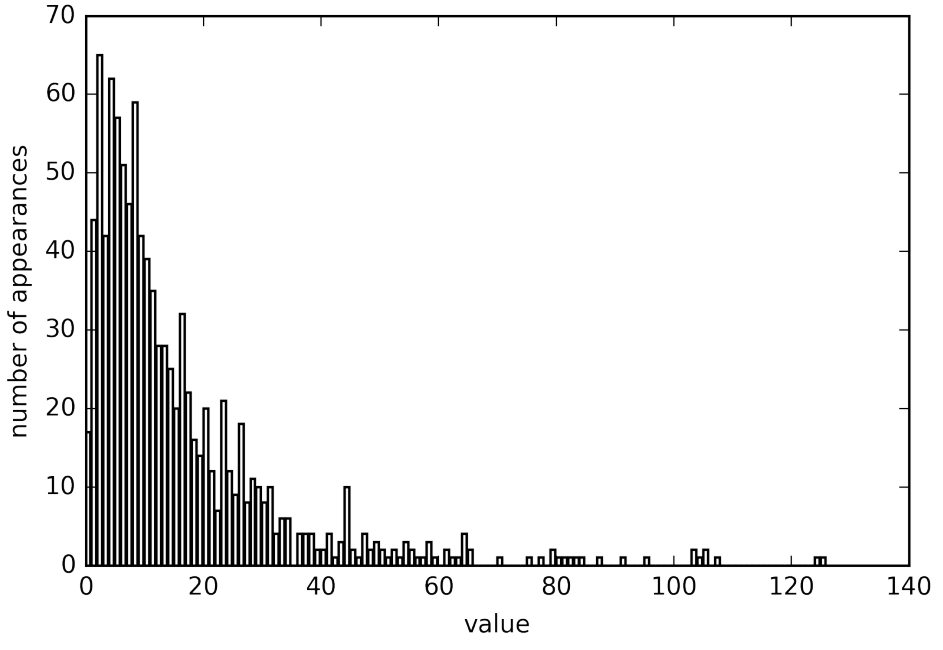
```
np.bincount(X[:, 0])
```

```
array([17, 44, 65, 42, 62, 57, 51, 46, 59, 42, 39, 35, 28, 28, 25, 20, 32,
       22, 16, 14, 20, 12,  7, 21, 12,  9, 18,  8, 11, 10,  8, 10,  4,  6,
        6,  0,  4,  4,  4,  2,  2,  4,  1,  3, 10,  2,  1,  4,  2,  3,  2,
        1,  2,  1,  3,  2,  1,  1,  3,  1,  0,  2,  1,  1,  4,  2,  0,  0,
        0,  0,  1,  0,  0,  0,  0,  1,  0,  1,  0,  2,  1,  1,  1,  1,  1,
```

```
       0,  0,  1,  0,  0,  0,  1,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,
       0,  2,  1,  2,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
       0,  0,  0,  0,  0,  1,  1])
```

The value 2 seems to be the most common with 62 appearances (bincount always starts at 0), and the counts for higher values fall quickly. However, there are some very high values, like 134 appearing twice. We visualized the counts below:

```
bins = np.bincount(X[:, 0])
plt.bar(range(len(bins)), bins, color='w')
plt.ylabel("number of appearances")
plt.xlabel("value")
```



Features `X[:, 1]` and `X[:, 2]` have similar properties. This kind of distribution of values (many small ones, and a few very large ones) is very common in practice [footnote: This is a Poisson distribution, which is quite fundamental to count data]. However, it is something most linear models can't handle very well. Let's try to fit a Ridge regression to this model:

```
from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
Ridge().fit(X_train, y_train).score(X_test, y_test)
```

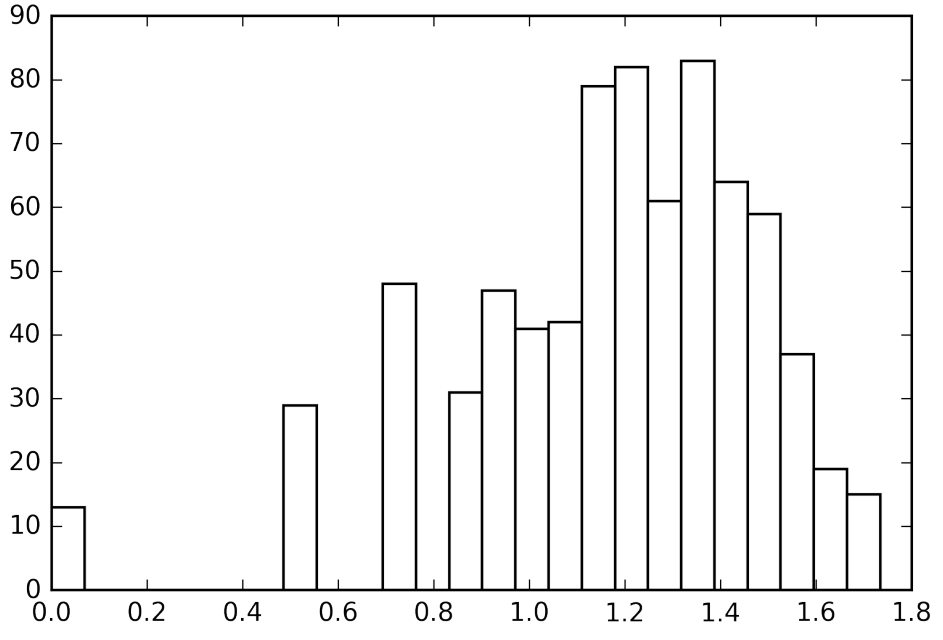
```
0.61099786602482975
```

Applying a logarithmic transformation can help, though. Because the value 0 appears in the data (and the logarithm is not defined at 0), we can not actually just apply `log`, but we have to compute `log(X + 1)`:

```
X_train_log = np.log(X_train + 1)
X_test_log = np.log(X_test + 1)
```

After the transformation, the distribution of the data is less asymmetrical and doesn't have very large outliers any more:

```
plt.hist(np.log(X_train_log[:, 0] + 1), bins=25, color='w');
```



Building a ridge model on the new data provides a much better fit:

```
Ridge().fit(X_train_log, y_train).score(X_test_log, y_test)
```

```
0.85919272057341256
```

Finding the transformation that works best for each combination of dataset and model is somewhat of an art. In this example, all the features had the same properties. This is rarely the case in practice, and usually only a subset of the features should be transformed, or sometimes each feature needs be transformed in a different way. As we mentioned above, these kind of transformations are irrelevant for tree-based models, but might be essential for linear models.

Sometimes it is also a good idea to transform the target variable `y` in regression. Trying to predict counts (say, number of orders) is a fairly common task, and using the

`log(y + 1)` transformation often helps [footnote: this is a very crude approximation of using Poisson regression, which would be the proper solution from a probabilistic standpoint.]

As you saw in the examples above, binning, polynomials and interactions can have a huge influence on how models perform on a given dataset. This is in particularly true for less complex models like linear models and naive Bayes.

Tree-based models on the other hand are often able to discover important interactions themselves, and don't require transforming the data explicitly most of the time.

Other models like SVMs, nearest neighbors and neural networks might sometimes benefit from using binning, interactions or polynomials, but the implications there are usually much less clear than in the case of linear models.

Automatic Feature Selection

With so many ways to create new features, you might get tempted to increase the dimensionality of the data way beyond the number of original features. However, adding more features makes all models more complex, and so increases the chance of overfitting. When adding new features, or with high-dimensional datasets in general, it can be a good idea to reduce the number of features to only the most useful ones, and discard the rest. This can lead to simpler models that generalize better.

But how can you know how good each feature is?

There are three basic strategies: *Univariate statistics*, *model-based selection* and *iterative selection*. We will discuss all three of them in detail. All three of these methods are supervised methods, meaning they need the target for fitting the model. This means we do need to split the data into training and test set

and fit the feature selection only on the training part of the data.

Univariate statistics

In univariate statistics, we compute whether there is a statistically significant relationship between each feature and the target. Then the features that are related with the highest confidence are selected. In the case of classification, this is also known as analysis of variance (ANOVA).

A key property of these tests are that they are *univariate* meaning that they only consider each feature individually. Consequently a feature will be discarded if it is only informative when combined with another feature. Univariate tests are often very fast to compute, and don't require building a model. On the other hand, they are completely independend of the model that you might want to apply after the feature selection.

To use univariate feature selection in scikit-learn, you need to choose a test, usually either `f_classif` (the default) for classification or `f_regression` for regression, and a method to discard features based on the p-values determined in the test. All methods for discarding parameters use a threshold to discard all features with too high a p-values (which means they are unlikely to be related to the target). The methods differ in how they compute this threshold, with the simplest ones being `SelectKBest` which selects a fixed number `k` of features, and `SelectPercentile`, which selectes a fixed percentage of features.

Let's apply the feature selection for classification on the `cancer` dataset. To make the task a bit harder, we add some non-informative noise features to the data. We expect the feature selection to be able to identify the features that are non-informative and remove them.

```
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectPercentile
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()

# get deterministic random numbers
rng = np.random.RandomState(42)
noise = rng.normal(size=(len(cancer.data), 50))
# add noise features to the data
# the first 30 features are from the dataset, the next 50 are noise
X_w_noise = np.hstack([cancer.data, noise])

X_train, X_test, y_train, y_test = train_test_split(
    X_w_noise, cancer.target, random_state=0, test_size=.5)
# use f_classif (the default) and SelectPercentile to select 10% of features:
select = SelectPercentile(percentile=50)
select.fit(X_train, y_train)
# transform training set:
X_train_selected = select.transform(X_train)

print(X_train.shape)
print(X_train_selected.shape)
```

```
(284, 80)

(284, 40)
```

As you can see, the number of features was reduced from 80 to 40 (50 percent of the original number of features). We can find out which features have been selected using the `get_support` method, which returns a boolean mask of the selected features:

```
mask = select.get_support()
print(mask)
# visualize the mask. black is True, white is False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
```


```
[ True  True  True  True  True  True  True  True  True False  True False
  True  True  True  True  True  True False False  True  True  True  True
  True  True  True  True  True  True False False False  True False  True
 False False  True False False False False  True False False  True False
 False  True False  True False False False False False False  True False
  True False False False False  True False  True False False False False
  True  True False  True False False False False]
```

As you can see from the visualization of the mask above, most of the selected features are the original features, and most of the noise features were removed. However, the recovery of the original features is not perfect.

Let's compare the performance of logistic regression on all features against the performance using only the selected features:

```
from sklearn.linear_model import LogisticRegression

# transform test data:
X_test_selected = select.transform(X_test)

lr = LogisticRegression()
lr.fit(X_train, y_train)
print("Score with all features: %f" % lr.score(X_test, y_test))
lr.fit(X_train_selected, y_train)
print("Score with only selected features: %f" % lr.score(X_test_selected, y_test))
```

```
Score with all features: 0.929825

Score with only selected features: 0.940351
```

In this case, removing the noise features improved performance, even though some of the original features where lost. This was a very simple synthetic example, though, and outcomes on real data is usually mixed. Univariate feature selection can still be very helpful if there is such a large number of features that building a model on them is infeasible, or if you suspect that many features are completely uninformative.

Model-based Feature Selection

Model based feature selection uses a supervised machine learning model to judge the importance of each feature, and keeps only the most important ones. The supervised model that is used for feature selection doesn't need to be the same model that is used for the final supervised modeling.

The model that is used for feature selection needs to provide some measure of importance for each feature, so that they can be ranked by this measure. Decision trees and decision tree based models provide feature importances, which can be used; Linear models have coefficients which can be used by considering the absolute value. As we saw in Chapter 2, linear models with L1 penalty learn sparse coefficients, which only use a small subset of features. This can be viewed as a form of feature selection for the model itself, but can also be used as a preprocessing step to select features for another model.

In contrast to univariate selection, model-based selection considers all features at once, and so can capture interactions (if the model can capture them).

To use model based feature selection, we need to use the `SelectFromModel` transformer:

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(RandomForestClassifier(n_estimators=100, random_state=42), threshold="mec
```

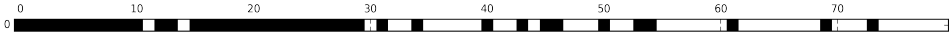
The `SelectFromModel` class selects all features that have an importance measure of the feature (as provided by the supervised model) greater than the provided threshold. To get a comparable result to what we got with univariate feature selection, we used the median as a threshold, so that half of the features will be selected. We use a random forest classifier with 100 trees to compute the feature importances. This is a quite complex model and much more powerful than using univariate tests. Now let's actually fit the model:

```
select.fit(X_train, y_train)
X_train_l1 = select.transform(X_train)
print(X_train.shape)
print(X_train_l1.shape)
```

```
(284, 80)

(284, 40)
```

Again, we can have a look at the features that were selected:

```
mask = select.get_support()
# visualize the mask. black is True, white is False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
```



This time, all but two of the original features were selected. Because we specified to select 40 features, some of the noise features are also selected.

```
X_test_l1 = select.transform(X_test)
LogisticRegression().fit(X_train_l1, y_train).score(X_test_l1, y_test)
```

```
0.9508771929824561
```

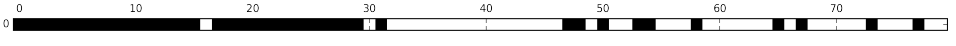
With the better feature selection, we also gained some improvements in performance.

Iterative feature selection

In univariate testing, we build used no model, while in model based selection we used a single model to select features. In iterative feature selection, a series of models is built, with varying numbers of features. There are two basic methods: starting with no features and adding features one by one, until some stopping criterion is reached, or starting with all features and removing features one by one, until some stopping criterion is reached. Because a series of models is built, these methods are much more computationally expensive then the methods we discussed above. One particular method of this kind is *recursive feature elimination* (RFE) which starts with all features, builds a model, and discards the least important feature according to the model. Then, a new model is built, using all but the discarded feature, and so on, until only a pre-specified number of features is left. For this to work, the model used for selection needs to provide some way to determine feature importance, as was the case for the model based selction.

We use the same random forest model that we used above:

```
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42), n_features_to_select=40)
#select = RFE(LogisticRegression(penalty="l1"), n_features_to_select=40)

select.fit(X_train, y_train)
# visualize the selected features:
mask = select.get_support()
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
```



The feature selection got better compared to the univariate and model based selection, but one feature was still missed. Running the above code takes significantly longer than the model based selection, because a random forest model is trained 40 times, once for each feature that is dropped.

```
X_train_rfe= select.transform(X_train)
X_test_rfe= select.transform(X_test)

LogisticRegression().fit(X_train_rfe, y_train).score(X_test_rfe, y_test)
```

```
0.9508771929824561
```

We can also use the model used inside the RFE to make predictions. This uses only the feature set that was selected:

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

```
0.9508771929824561
```

If you are unsure when selecting what to use as input to your machine learning algorithms, automatic feature selection can be quite helpful. It is also great to reduce the amount of features needed, for example to speed up prediction, or allow for more interpretable models. In most real-world cases, applying feature selection is unlikely to provide large gains in performance. However, it is still a valuable tool in the toolbox of the feature engineer.

Utilizing Expert Knowledge

Feature engineering is often an important place to use *expert knowledge* for a particular application. While the purpose of machine learning often is to avoid having to create a set of expert-designed rules, that doesn't mean that prior knowledge of the application or domain should be discarded. Often, domain experts can help in identifying useful features that are much more informative than the initial representation of the data.

Imagine you are a travel agency and want to predict flight prices. Let's say we have a record of prices together with date, airline, start location and destination. A machine learning model might be able to build a decent model from that. Some important factors in flight prices, however can not be learned. For example, flights are usually more expensive during school holidays or around public holidays. While some holidays can potentially be learned from the dates, like Christmas, others might depend on the phases of the moon (like Hannukah and Easter), or be set by authorities like school holidays. These events can not be learned from the data if each flight is only recorded using the (Gregorian) date. It is easy to add a feature that encodes whether a flight was on, preceding, or following a public or school holiday. In this way, prior knowledge about the nature of the task can be encoded in the features to aid a machine learning algorithm. Adding a feature does not force a machine learning algorithm to use it, and even if the holiday information turns out to be non-informative for flight prices, augmenting the data with this information doesn't hurt.

We'll now go through one particular case of using expert knowledge - though in this case is might be more rightfully called "common sense". The task is predicting citibike rentals in front of Andreas' house.

In New York, there is a network of bicycle rental stations, with a subscription system. The stations are all over the city and provide a convenient way to get around. Bike rental data is made public in an anonymized form [Footnote: at https://www.citibikenyc.com/system-data] and has been analyzed in various ways.

The task we want to solve is to predict for a given time and day how many people will rent a bike in front of Andreas' house - so he knows if any bikes will be left for him.

We first load the data for August 2015 of this particular station as a pandas dataframe. We resampled the data into 3 hour intervals to obtain the main trends for each day.

```
from importlib import reload
reload(mglearn.datasets)
citibike = mglearn.datasets.load_citibike()
```

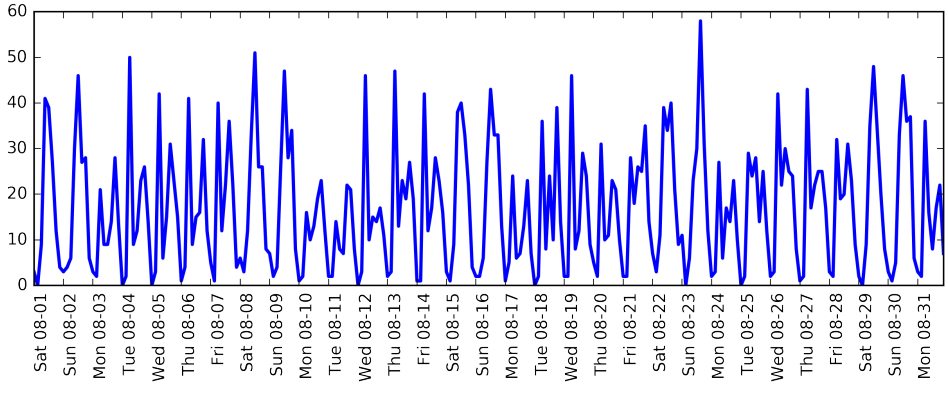
```
/home/andy/checkout/book/notebooks/mglearn/datasets.py:40: FutureWarning: how in .resample() is de

the new syntax is .resample(...).sum()

  data_resampled = data_starttime.resample("3h", how="sum").fillna(0)
```

```
citibike.head()
```

```
starttime

2015-08-01 00:00:00     3.0

2015-08-01 03:00:00     0.0

2015-08-01 06:00:00     9.0

2015-08-01 09:00:00    41.0

2015-08-01 12:00:00    39.0

Freq: 3H, Name: one, dtype: float64
```

Below is a visualization of the rental frequencies for the whole month:

```
plt.figure(figsize=(10, 3))
xticks = pd.date_range(start=citibike.index.min(), end=citibike.index.max(), freq='D')
plt.xticks(xticks, xticks.strftime("%a %m-%d"), rotation=90, ha="left")
plt.plot(citibike, linewidth=2)
```



Looking at the data, we can clearly distinguish day and night for each day. The patterns for week days and weekends also seem to be quite different.

When evaluating a prediction task on a time series like this, we usually want to learn *from the past* and predict *for the future*. This means when doing a split into a training

and a test set, we want to use all the data up to a certain date as training set, and all the data past that date as a test set. This is how we would usually use time series prediction: given everything that we know about rentals in the past, what do we think will happen tomorrow?

We will use the first 184 data points, corresponding to the first 23 days, as our training set, and the remaining 64 data points corresponding to the remaining 8 days as our test set.

The only feature that we are using in our prediction task is the date and time when a particular number of rentals occurred. So the input feature is the time, say `2015-08-01 00:00:00`, and the output is the number of rentals in the following three hours, 3 in this case (according to the dataframe above).

A (surprisingly) common way that dates are stored on computers is using POSIX time, which is the number of seconds since January 1970 00:00:00 (aka the beginning of time). As a first try, we can use this single integer feature as our data representation:

```
# extract the target values (number of rentals)
y = citibike.values
# convert the time to posixtime using "%s"
X = citibike.index.strftime("%s").astype("int").reshape(-1, 1)

from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
```

We first define a function to split the data into training and test set, build the model, and visualize the result:

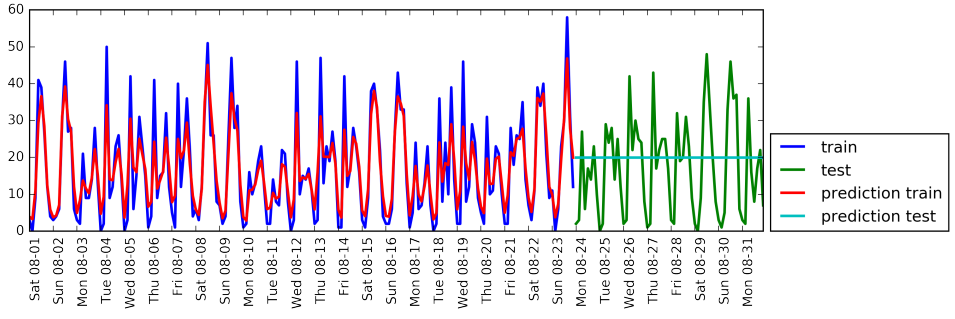
```
# use the first 184 data points for training, the rest for testing
n_train = 184

# function to evaluate and plot a regressor on a given feature set
def eval_on_features(features, target, regressor):
    # split the given features into a training and test set
    X_train, X_test = features[:n_train], features[n_train:]
    # split also the
    y_train, y_test = target[:n_train], target[n_train:]
    regressor.fit(X_train, y_train)
    print("Test-set R^2: ", regressor.score(X_test, y_test))
    y_pred = regressor.predict(X_test)
    y_pred_train = regressor.predict(X_train)
    plt.figure(figsize=(10, 3))

    plt.xticks(range(0, len(X), 8), xticks.strftime("%a %m-%d"), rotation=90, ha="left");

    plt.plot(range(n_train), y_train, label="train", linewidth=2)
    plt.plot(range(n_train, len(y_test) + n_train), y_test, label="test", linewidth=2)
    plt.plot(range(n_train), y_pred_train, label="prediction train", linewidth=2)
```

```
    plt.plot(range(n_train, len(y_test) + n_train), y_pred, label="prediction test", linewidth=2)
    plt.legend(loc=(1.01, 0))
```

We saw above that random forests need very little preprocessing of the data, which makes it seem like a good model to start with.

We use the POSIX time feature `X` and pass a random forest regressor to our `eval_on_features` function:

```
regressor = RandomForestRegressor(n_estimators=100, random_state=0)
plt.figure()
eval_on_features(X, y, regressor);
```




```
Test-set R^2:  -0.035486463626
```

The predictions on the training set are quite good, as is usual for random forests. However, for the test set, a constant line is predicted. The `R^2` is -0.03, which means that we learned nothing. What happened?

The problem lies in the combination of our feature and the random forest. The value of the POSIX time feature for the test set is outside of the range of the feature values on the training set: the points in the test set have time stamps that are later than all the points in the training set.

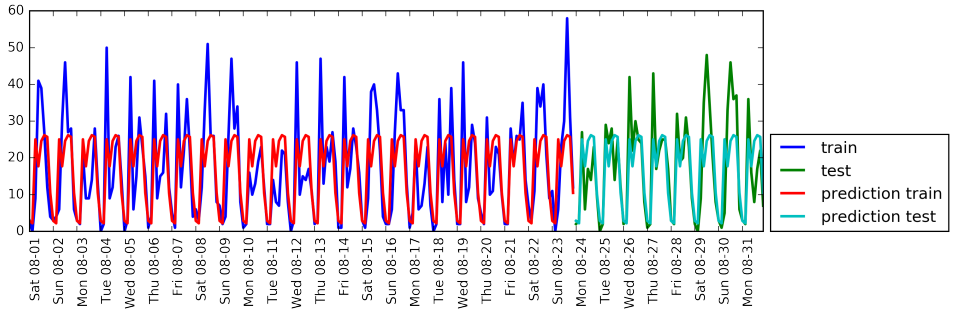
Trees, and therefore random forests, can not *extrapolate* to feature ranges outside the training set.

The result is that the model simply predicts the same as for the closest point in the training set - which is the last time it observed any data.

Clearly we can do better than this. This is where our "expert knowledge" comes in. From looking at the rental figures on the training data, two factors seem to be very important: the time of day, and the day of the week. So let's add these two features. We can't really learn anything from the POSIX time, so we drop that feature.

First, let's use only the hour of the day:

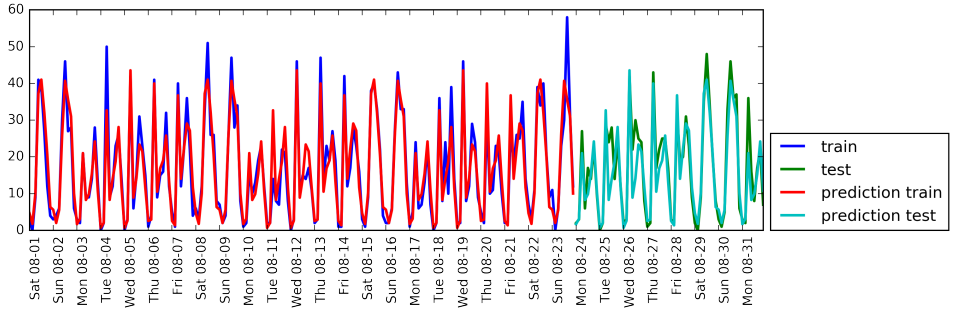
```
X_hour = citibike.index.hour.reshape(-1, 1)
eval_on_features(X_hour, y, regressor)
```




```
Test-set R^2:  0.599577599331
```

Now the predictions have the same pattern for each day of the week. The R^2 is already much better, but the predictions clearly miss the weekly pattern. Now let's also add the day of the week:

```
X_hour_week = np.hstack([citibike.index.dayofweek.reshape(-1, 1), citibike.index.hour.reshape(-1,
eval_on_features(X_hour_week, y, regressor)
```



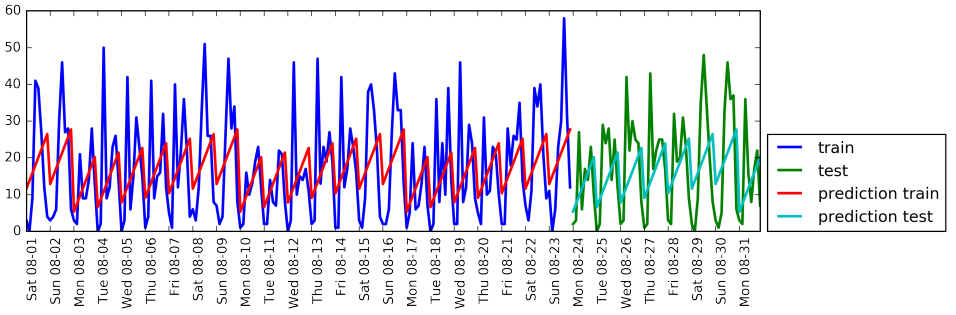

```
Test-set R^2:  0.841948858797
```

Now, we have a model that models the periodic behavior considering day of week and time of day. It has an R^2 of 0.84, and show pretty good predictive performance.

What this model likely is learning is the mean number of rentals for each combination of weekday and time of day from the first 23 days of August. This would actually not require a complex model like a random forest.

So let's try with a simpler model, LinearRegression:

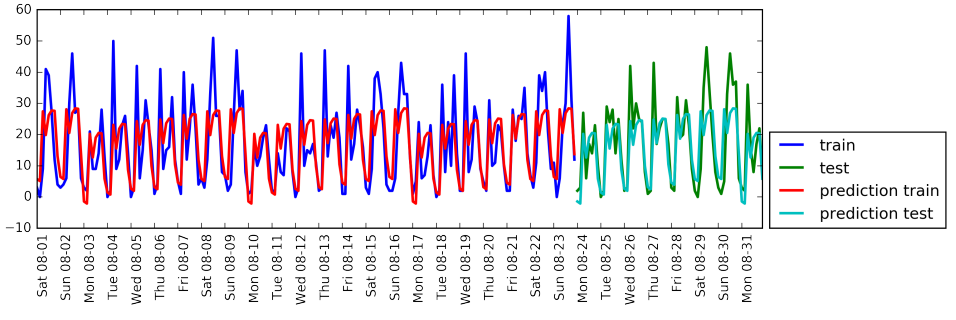
```
eval_on_features(X_hour_week, y, LinearRegression())
```




```
Test-set R^2:  0.132041572622
```

Linear Regression works much worse, and the periodic pattern looks odd. The reason for this is that we encoded day of the week and time of the day using integers, which are interpreted as categorical variables. Therefore, the linear model can only learn a linear function of the time of day - and it learned that later in the day, there are more rentals. However, the patterns are much more complex that that, which we can capture by interpreting the integers as categorical variables, by transforming them using `OneHotEncoder`:

```
enc = OneHotEncoder()
X_hour_week_onehot = enc.fit_transform(X_hour_week).toarray()

eval_on_features(X_hour_week_onehot, y, Ridge())
```



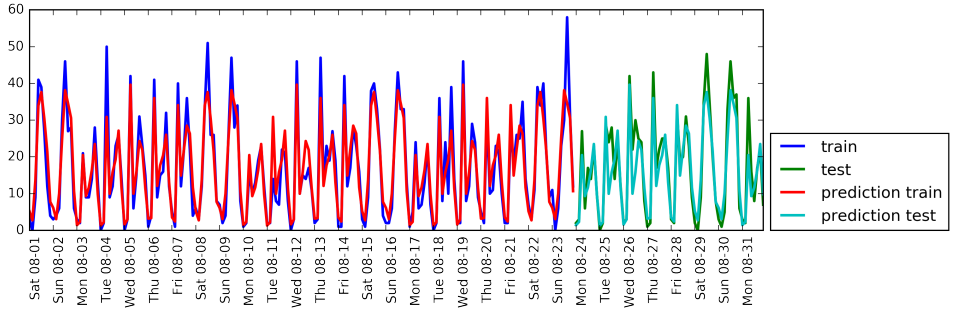

```
Test-set R^2:  0.619113916866
```

This gave us a much better match than the continuous feature encoding. Now, the linear model learns one coefficient for each day of the week, and one coefficient for each time of the day. That means that the "time of day" pattern is shared over all days of the week, though.

Using interaction features, we can allow the model to learn one coefficient for each combination of day and time of day:

```
poly_transformer = PolynomialFeatures(degree=2, interaction_only=True, include_bias=False)
X_hour_week_onehot_poly = poly_transformer.fit_transform(X_hour_week_onehot)
```

```
lr = Ridge()
eval_on_features(X_hour_week_onehot_poly, y, lr)
```




```
Test-set R^2:  0.845170635797
```

This transformation finally yields a model that performs similarly well to the random forest.

A big benefit of this model is that it is very clear what is learned: one coefficient for each day and time.

We can simply plot the coefficients learned by the model, something that would not be possible for the random forest.

First, we create feature names for the hour and day features:

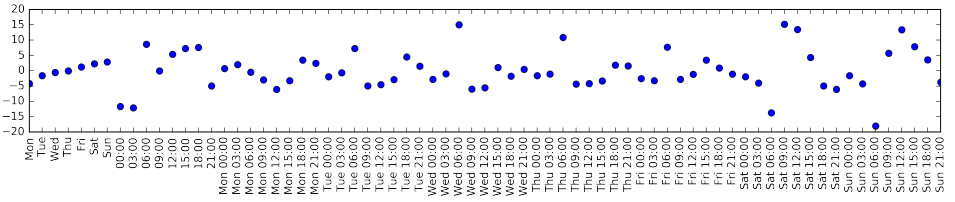
```
hour = ["%02d:00" % i for i in range(0, 24, 3)]
day = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
features =  day + hour
```

Then we name all the interaction feature extracted by the `PolynomialFeatures`, using the `get_feature_names` method, and keep only the feature with non-zero coefficients:

```
features_poly = poly_transformer.get_feature_names(features)
features_nonzero = np.array(features_poly)[lr.coef_ != 0]
coef_nonzero = lr.coef_[lr.coef_ != 0]
```

Now we can visualize the coefficients learned by the linear model:

```
plt.figure(figsize=(15, 2))
plt.plot(coef_nonzero, 'o')
plt.xticks(np.arange(len(coef_nonzero)), features_nonzero, rotation=90);
```



Summary and outlook

In this chapter, we discussed how to deal with different data types, in particular with categorical variables.

We emphasized the importance of representing your data in a way that is suitable for the machine learning algorithm, for example by one-hot-encoding categorical variables.

We also discussed the importance of engineering new features, and the possibility of utilizing expert knowledge in creating derived features from your data.

In particular linear models might benefit greatly from generating new features via binning and adding polynomials and interactions, while more complex, nonlinear models like random forests and SVMs might be able to learn more complex tasks without explicitly expanding the feature space.

In practice, the features that are used, and the match between features and method is often the most important piece in making a machine learning approach work well.

Having our data represented in an appropriate way, and knowing which algorithm to use for which task, the next chapter will focus on evaluating performance of machine learning models and selecting the right parameter settings.

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.

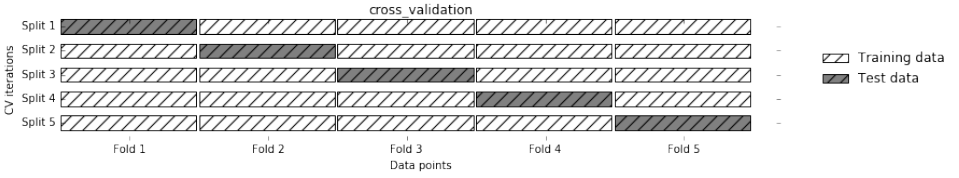
Then another model is build, this time using fold 2 as the test set, and the data in folds 1, 3, 4 and 5 as the training set.

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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

2 2]
```

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`.

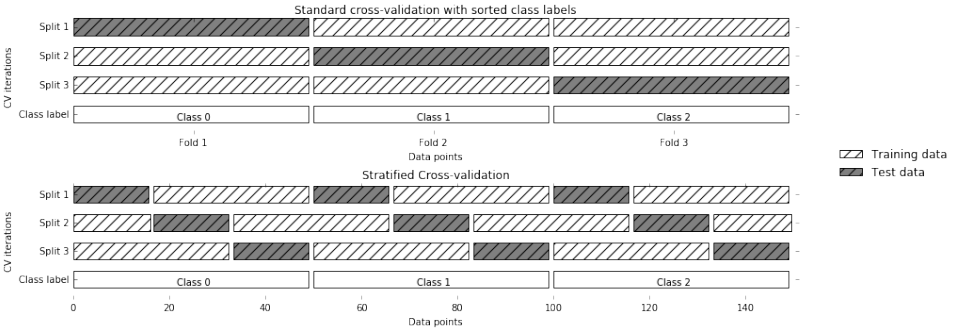
As the simple k-fold strategy fails here, scikit-learn does not use k-fold for classification, but rather *stratified k-fold cross-validation*. In stratified cross-validation, we split the data such that the proportions between classes are the same in each fold as they are in the whole dataset, as illustrated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

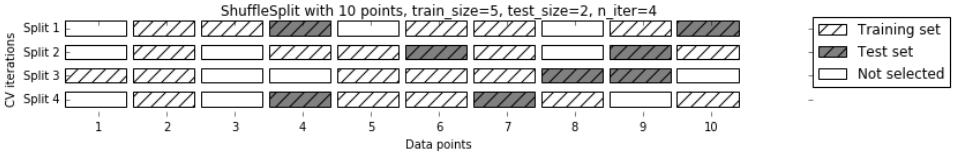
Shuffle-Split cross-validation

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.

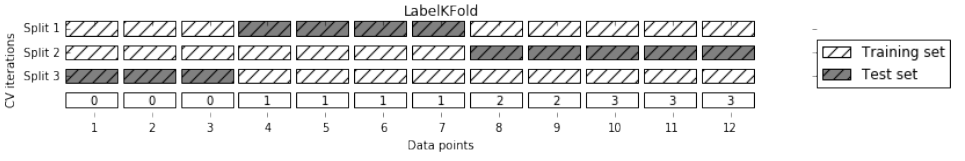
To achieve this, we can use `LabelKFold`, which takes a `label` argument, which we can use to indicate which person is in the image. So `label` here indicates groups in the data that should not be split when creating training and test set, and should not be confused with the class label.

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

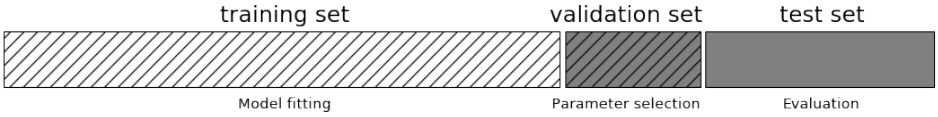
Given this result, we might be tempted to report that we found a model that performs 97.3% accurate on our dataset. However, this claim could be overly optimistic (or just wrong) for the following reason: we tried many different parameters, and selected the one with best accuracy on the test set. However, 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)
```

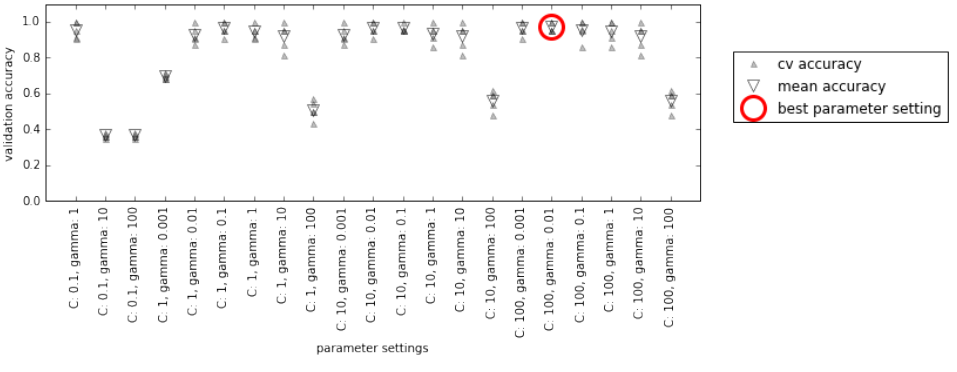
To evaluate the accuracy of the SVM using a particular setting of `C` and `gamma` using five-fold cross-validation we need to train 36 * 5 = 180 models. As you can imagine, the main down-side of the use of cross-validation is the time it takes to train all these models.

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.

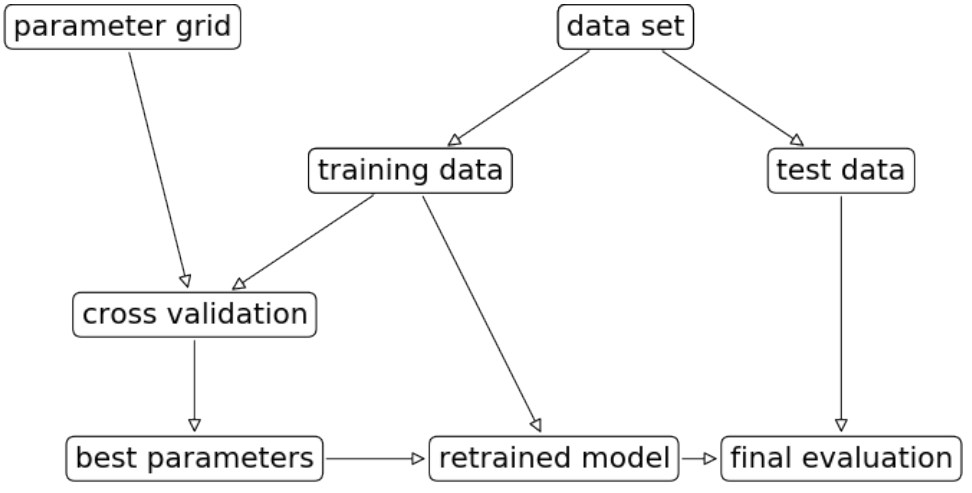
[warning / note box] As we said above, cross-validation is a way to evaluate a given algorithm on a specific dataset.

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:

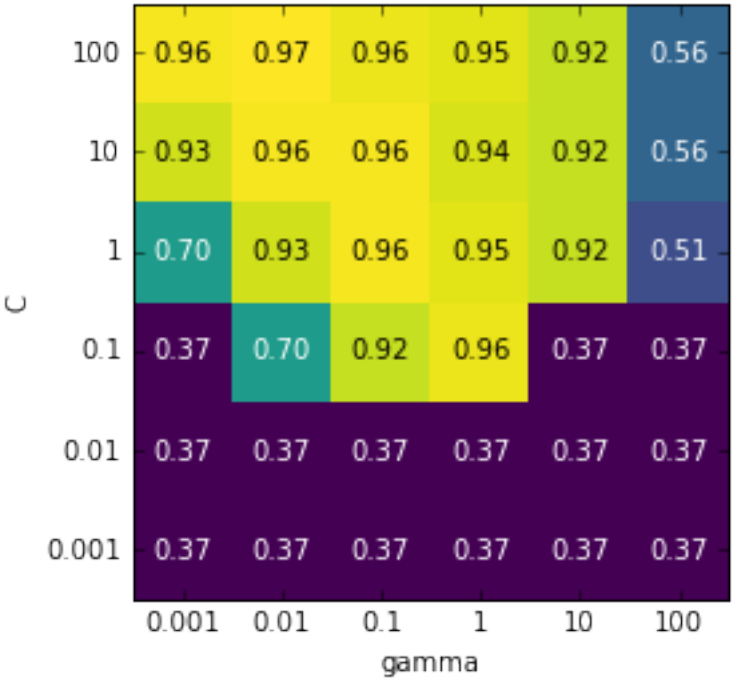
```
grid_search.grid_scores_

[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")
```



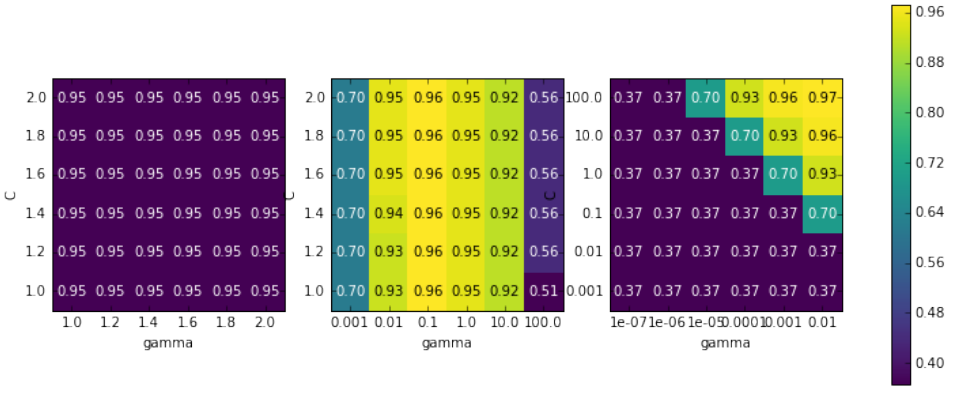
Each point in the heat map corresponds to one run of cross-validation, with a particular parameter setting. The color encodes the cross-validation accuracy, with light colors meaning high accuracy, and dark colors meaning low accuracy.

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`.

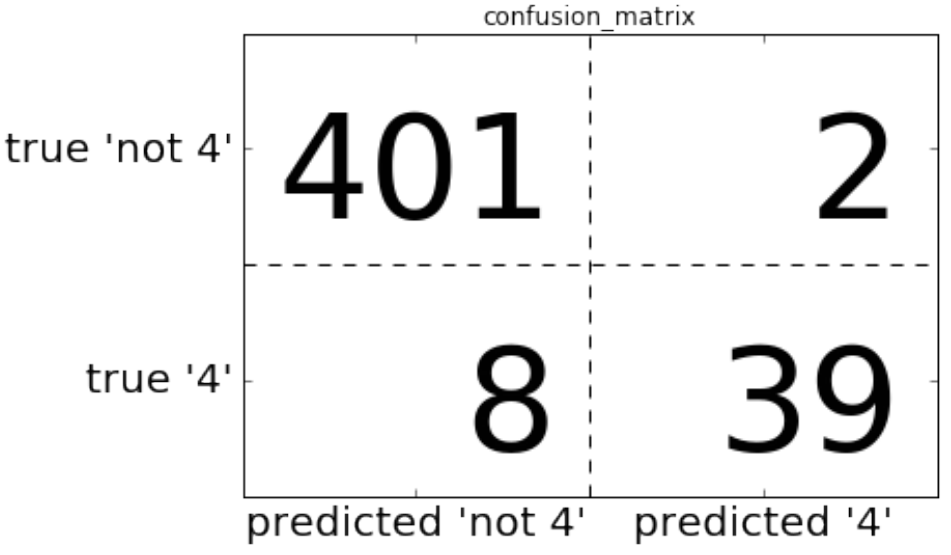
```
from sklearn.metrics import confusion_matrix

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()
```

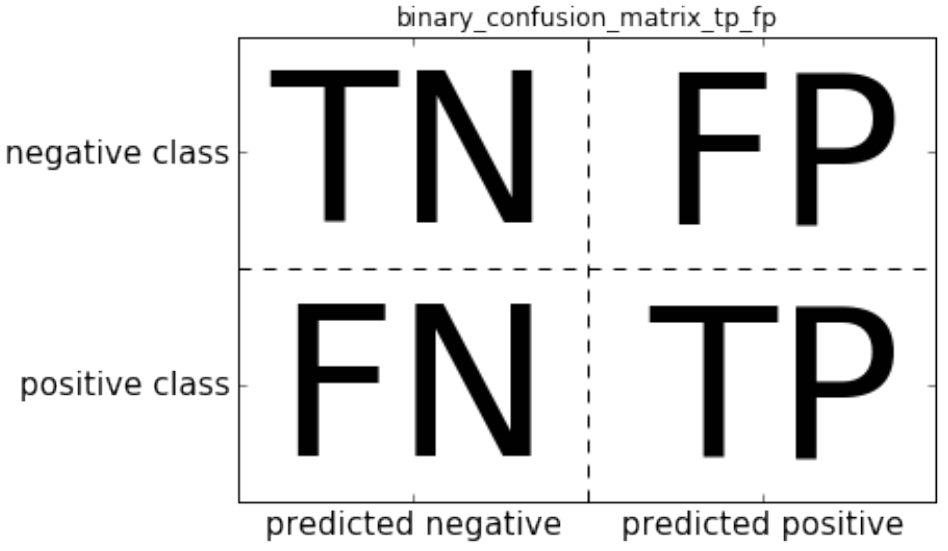


This means of the 403 data points (sum of the first row) that are not a four, 401 got predicted correctly as such, and two of which were incorrectly predicted as a four. Similarly there are 47 data points that are fours, 39 of which got correctly classified, and 8 of which were predicted incorrectly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

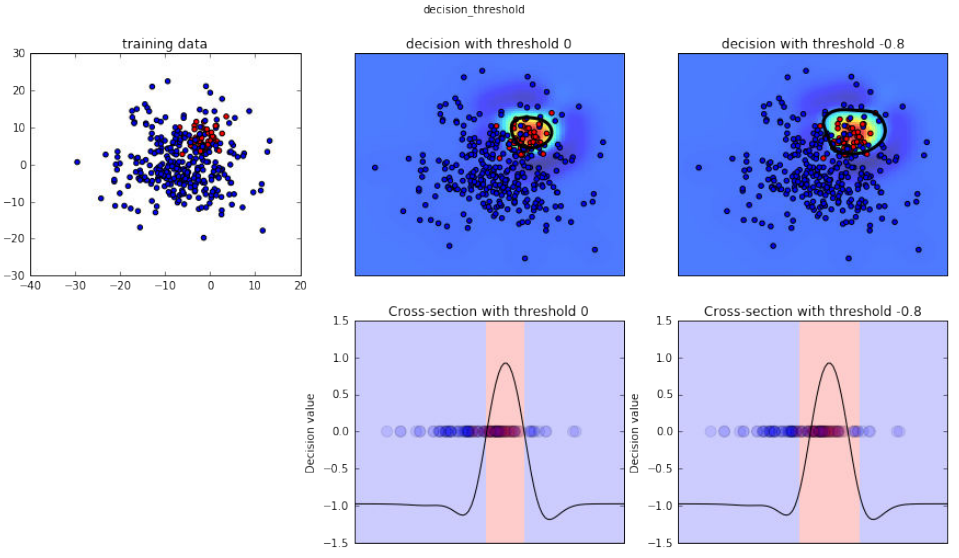
Making predictions can be seen as thresholding the output of `decision_function` or `predict_proba` at a certain fixed point - in binary classification zero for the decision function and 0.5 for `predict_proba`.

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*.

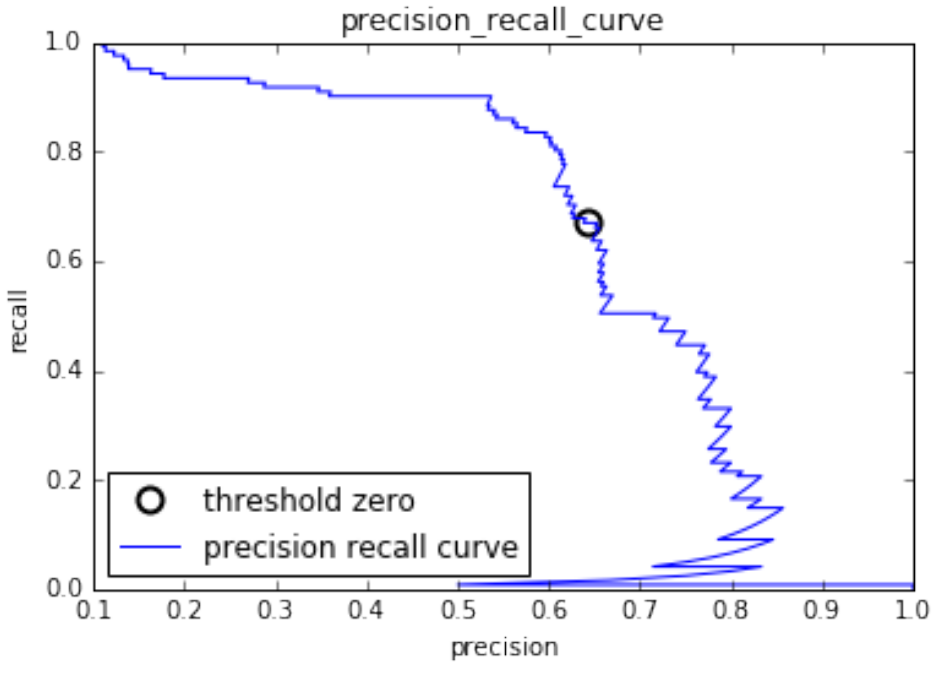
You can find the function to compute the precision-recall curve in the `sklearn.metrics` module. It needs the ground truth labeling and predicted uncertainties, created via `decision_function` or `predict_proba`:

```
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.

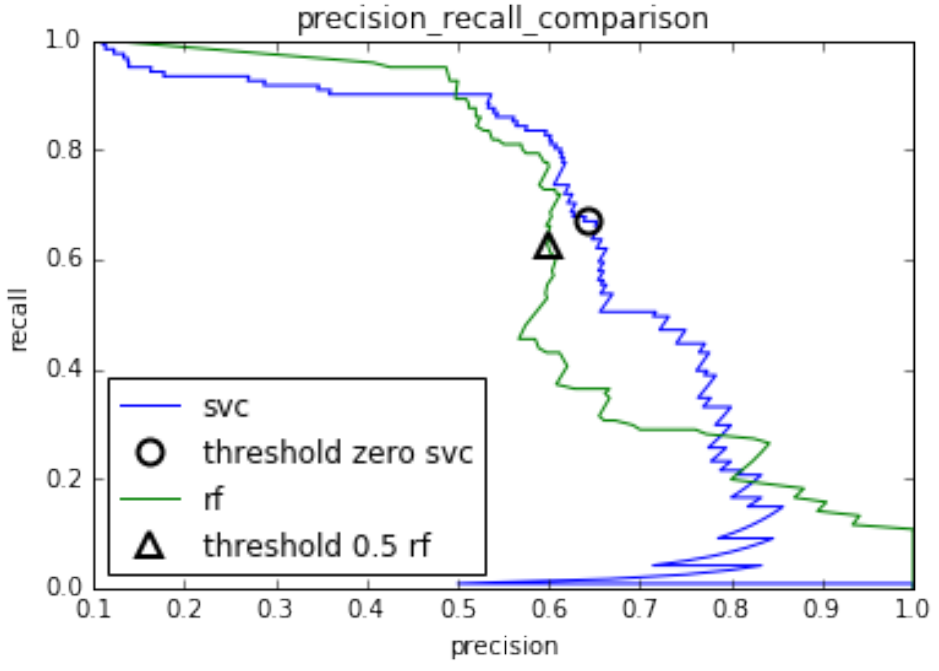
Looking at this particular curve a bit more, we can see that with this model it is possible to get a precision up to around 0.5 with very high recall. If we want a much higher precision, we have to sacrifice a lot of precision. In other words: on the left, the curve is relatively flat, meaning that recall 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}

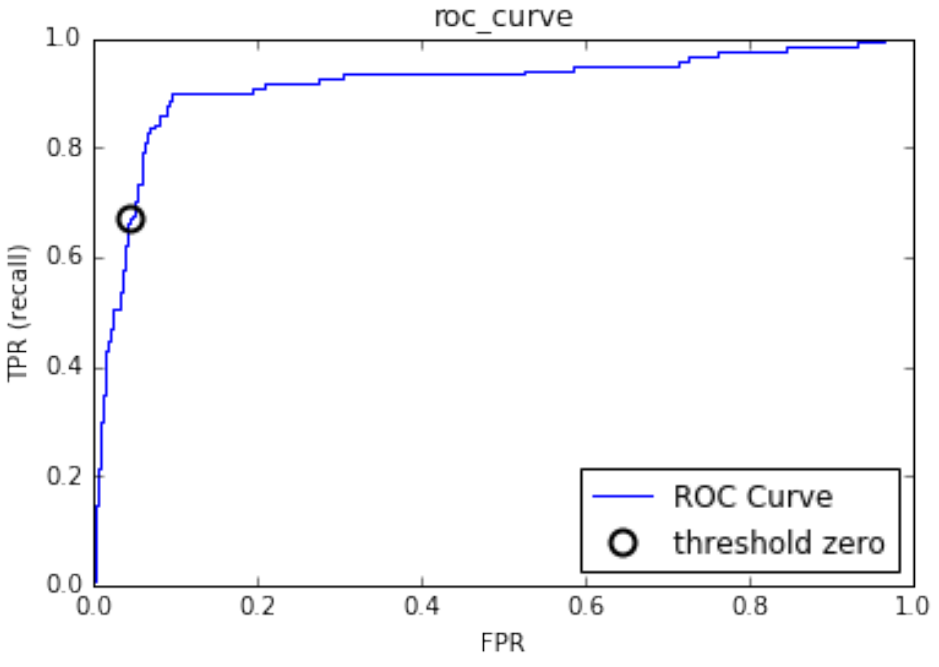
\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}

\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)
```

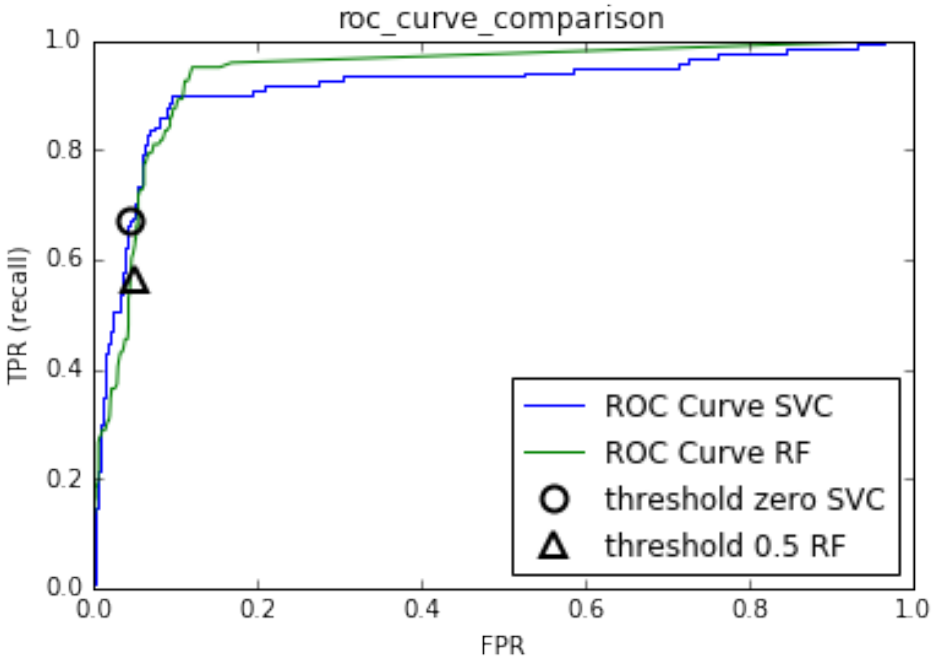


For the ROC curve, the ideal curve is close to the top left: you want a classifier that produces a *high recall* while keeping a *low false positive rate*. Compared to the default threshold of zero, the curve shows that we could achieve a significant higher recall (around 0.9) while only increasing the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

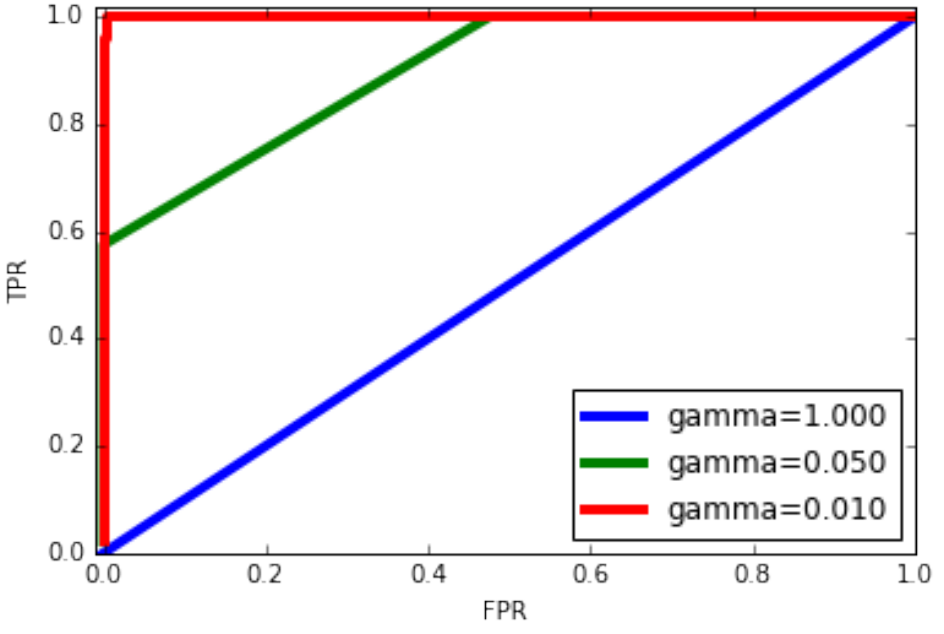
Comparing random forest and SVC using the AUC score, we find that Random Forest performs quite a bit better than SVC.

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.

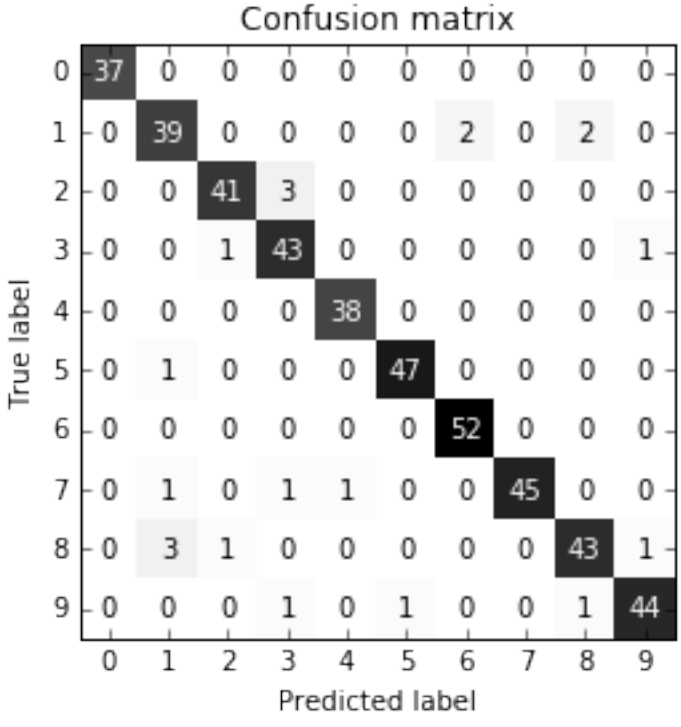
Let's apply these two detailed evaluation methods on the task of classifying the 10 different hand-written digits in the `digits` dataset:

```
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.

CHAPTER 7

Algorithm Chains and Pipelines

For many machine learning algorithms, the particular representation of the data that you provide is very important, as we discussed in Chapter 5. This starts with scaling the data and combining features by hand and goes all the way to learning features using unsupervised machine learning as we saw in Chapter 3.

Consequently, most machine learning applications require not only the application of a single algorithm, but the chaining together of many different processing steps and machine learning models.

For example we noticed that we can greatly improve the performance of a kernel SVM on the `cancer` dataset by using the `MinMaxScaler` for preprocessing. The code for splitting the data, computing minimum and maximum, scaling the data, and training the SVM is shown below:

```
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# load and split the data
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

# compute minimum and maximum on the training data
scaler = MinMaxScaler().fit(X_train)
# rescale training data
X_train_scaled = scaler.transform(X_train)

svm = SVC()
# learn an SVM on the scaled training data
svm.fit(X_train_scaled, y_train)
```

```
# scale test data and score the scaled data
X_test_scaled = scaler.transform(X_test)
svm.score(X_test_scaled, y_test)
```

```
0.95104895104895104
```

Parameter Selection with Preprocessing

Now let's say we want to find better parameters for `SVC` using `GridSearchCV`, as discussed in Chapter 6.

How should we go about doing this? A naive approach might look like this:

```
from sklearn.model_selection import GridSearchCV
# illustration purposes only, don't use this code
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=5)
grid.fit(X_train_scaled, y_train)
print("best cross-validation accuracy:", grid.best_score_)
print("test set score: ", grid.score(X_test_scaled, y_test))
print("best parameters: ", grid.best_params_)
```

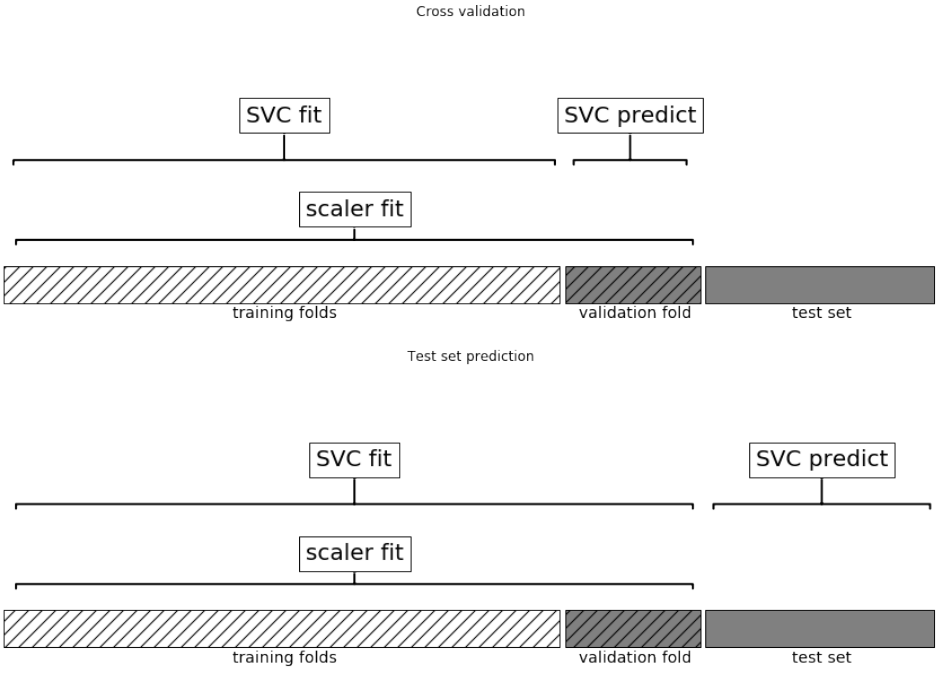
```
best cross-validation accuracy: 0.981220657277

test set score:  0.972027972028

best parameters:  {'gamma': 1, 'C': 1}
```

Here, we ran the grid-search over the parameters of the `SVC` using the scaled data. However, there is a subtle catch in what we just did. When scaling the data, we used *all data in the training set* to find out how to train it.

We then use the *scaled training data* to run our grid-search using cross-validation. For each split in the cross-validation, some part of the original training set will be declared the training part of this split, and some the test part of the split. The test part is used to measure how new data will look like to a model trained on the training part. However, we already used the information contained in the test part of the split, when scaling the data. Remember that the test part in each split in the cross-validation is part of the training set, and we used *the information from the whole training data* to find the right scaling of the data. *This is fundamentally different to how new data looks to the model.* If we observe new data (say in form of our test set), this data will not have been used to scale the training data. This data might have a different minimum and maximum than the training data.

The illustration below shows how the data processing during cross-validation and the final evaluation differ:

```
mglearn.plots.plot_improper_processing()
```



So the splits in the cross-validation no longer correctly mirror how new data will look to the modeling process. We already leaked information from these parts of the data into our modeling process. This will lead to overly optimistic results during cross-validation, and possibly the selection of suboptimal parameters.

To get around this problem, the splitting of the data set during cross-validation should be done *before doing any preprocessing*. Any process that extracts knowledge from the dataset should only ever be applied to the training portion of the data set, so any cross-validation should be the "outermost loop" in your processing.

To achieve this in scikit-learn with the `cross_val_score` function and the `GridSearchCV` function, we can use the `Pipeline` class. The `Pipeline` class is a class that allows "gluing" together multiple processing steps into a single scikit-learn estimator. The `Pipeline` class itself has `fit`, `predict` and `score` methods and behaves just like any other model in scikit-learn. The most common use-case of the pipeline class is in chaining preprocessing steps (like scaling of the data) together with a supervised model like a classifier.

Building Pipelines

Let's look at how we can use the `Pipeline` to express the work-flow for training an SVM after scaling the data `MinMaxScaler` (for now without the grid-search). First, we build a pipeline object, by providing it with a list of steps. Each step is a tuple contain-

ing a name (any string of your choosing [Footnote: With one exception: the name may not contain a double underscore "__".]) and an instance of an estimator:

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC())])
```

Here, we created two steps, the first called `"scaler"` is a `MinMaxScaler`, the second, called `"svm"` is an `SVC`. Now, we can fit the pipeline, like any other scikit-learn estimator:

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

```
Pipeline(steps=[('scaler', MinMaxScaler(copy=True, feature_range=(0, 1))), ('svm', SVC(C=1.0, cach

  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))])
```

Here, `pipe.fit` first calls fit on the first step, the scaler, then transforms the training data using the scaler, and finally fits the SVM with the scaled data. To evaluate on the test data, we simply call `pipe.score`:

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

```
0.95104895104895104
```

Calling the `score` method on the pipeline first transforms the test data using the scaler, and then calls the `score` method on the SVM using the scaled test data. As you can see, the result is identical to the one we got from the code above, doing the transformations "by hand".

Using the pipeline, we reduced the code needed for our "preprocessing + classification" process.

The main benefit of using the pipeline, however, is that we can now use this single estimator in `cross_val_score` or `GridSearchCV`.

Using Pipelines in Grid-searches

Using a pipeline in a grid-search works the same way as using any other estimator. We define a parameter grid to search over, and construct a `GridSearchCV` from the pipeline and the parameter grid. When specifying the parameter grid, there is a slight change, though. We need to specify for each parameter which step of the pipeline it belongs to.

Both parameters that we want to adjust, `C` and `gamma` are parameters of `SVC`, the second step. We gave this step the name `"svm"`. The syntax to define the a parameter grid for a pipeline is to specify for each parameter the step name, followed by `"__"` (dou-

ble underscore), followed by the parameter name. To search over the `C` parameter of the `SVC` we therefore have to use `"svm__C"` as the key in the parameter grid dictionary, and similarly for `gamma`:

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

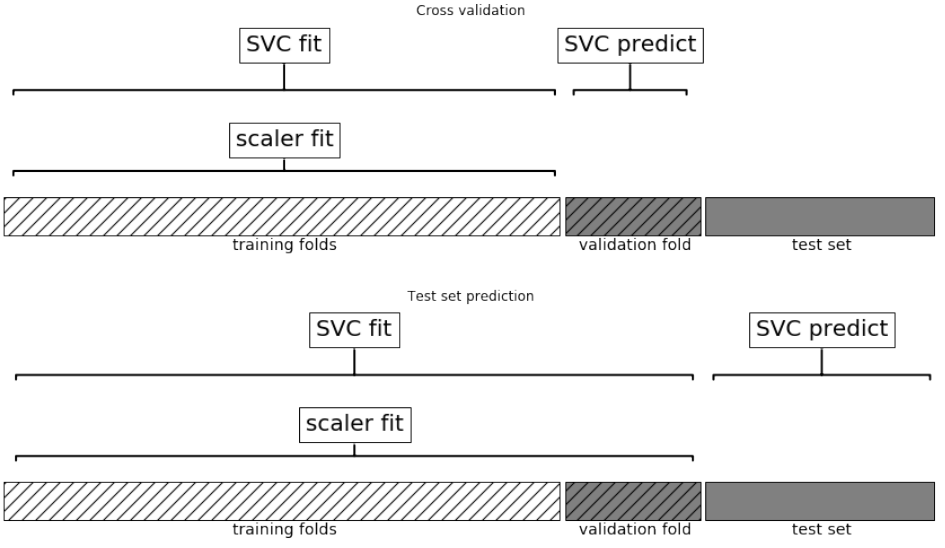
Using this parameter grid we can use `GridSearchCV` as usual:

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)
grid.fit(X_train, y_train)
print("best cross-validation accuracy:", grid.best_score_)
print("test set score: ", grid.score(X_test, y_test))
print("best parameters: ", grid.best_params_)
```

```
best cross-validation accuracy: 0.981220657277

test set score:  0.972027972028

best parameters:  {'svm__C': 1, 'svm__gamma': 1}
```

In contrast to the grid-search we did before, now for each split in the cross-validation, the `MinMaxScaler` is refit with only the training splits, not leaking any information of the test split into the parameter search, as illustrated below. Compare this with Figure improper_preprocessing above.

```
mglearn.plots.plot_proper_processing()
```



The impact of leaking information in the cross-validation varies depending on the nature of the preprocessing step. Estimating the scale of the data using the test fold

usually doesn't have a terrible impact, while using the test fold in feature extraction and feature selection can lead to substantial differences in outcomes.

[FIXME info box] Illustrating information leakage

A great example of leaking information in cross-validation is given in Hastie et al. (FIXME insert cite) and we reproduce an adapted version here.

Let us consider a synthetic regression task with 100 samples and 1000 features that are sampled independently from a Gaussian distribution. We also sample the response from a Gaussian distribution:

```
rnd = np.random.RandomState(seed=0)
X = rnd.normal(size=(100, 10000))
y = rnd.normal(size=(100,))
```

Given the way we created the dataset, there is no relation between the data X and the target y (they are independent), so it should not be possible to learn anything from this data set.

We will now do the following: First select the most informative of the ten features using `SelectPercentile` feature selection, and then evaluate a `Ridge` regressor using cross-validation:

```
from sklearn.feature_selection import SelectPercentile, f_regression

select = SelectPercentile(score_func=f_regression, percentile=5).fit(X, y)
X_selected = select.transform(X)
print(X_selected.shape)
```

```
(100, 500)
```

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import Ridge
np.mean(cross_val_score(Ridge(), X_selected, y, cv=5))
```

```
0.90579530652398221
```

The mean R^2 computed by cross-validation is 0.9, indicating a very good model. This can clearly not be right, as our data is entirely random. What happened here is that our feature selection picked out some features among the 10000 random features that are (by chance) very well correlated with the target. Because we fit the feature selection *outside* of the cross-validation, it could find features that are correlated both on the training and the test folds. The information we leaked from the test-folds was very informative, leading to highly unrealistic results.

Let's compare this to a proper cross-validation using a pipeline:

```
pipe = Pipeline([("select", SelectPercentile(score_func=f_regression, percentile=5)), ("ridge", Ri
np.mean(cross_val_score(pipe, X, y, cv=5))
```

```
-0.24655422384952805
```

This time, we get a *negative* R^2 score, indicating a very poor model.

Using the pipeline, the feature selection is now *inside* the cross-validation loop. This means features can only be selected using the training folds of the data, not the test fold. The feature selection finds features that are correlated with the target on the training set. But because the data is entirely random, these features are not correlated with the target on the test set.

In this example, rectifying the data leakage issue in the feature selection makes the difference between concluding that a model works very well and concluding that a model works not at all.

[end infobox]

The General Pipeline Interface

The `Pipeline` class is not restricted to preprocessing and classification, but can in fact join any number of estimators together.

For example, you could build a pipeline containing feature extraction, feature selection, scaling and classification, for a total of four steps. Similarly the last step could be regression or clustering instead of classification.

The only requirement for estimators in a pipeline is that all but the last step need to have a `transform` method, so they can produce a new representation of the data that can be used in the next step.

Internally, during the call to `Pipeline.fit`, the pipeline calls first `fit` and then `transform` on each step in turn [Footnote: or just `fit_transform`], with the input given by the output of the transform method of the previous step. For the last step in the pipeline, just `fit` is called. Brushing over some finer details, this is implemented as follows. Remember that `pipeline.steps` is a list of tuples, so `pipeline.steps[0][1]` is the first estimator, `pipeline.steps[1][1]` is the second estimator, and so on.

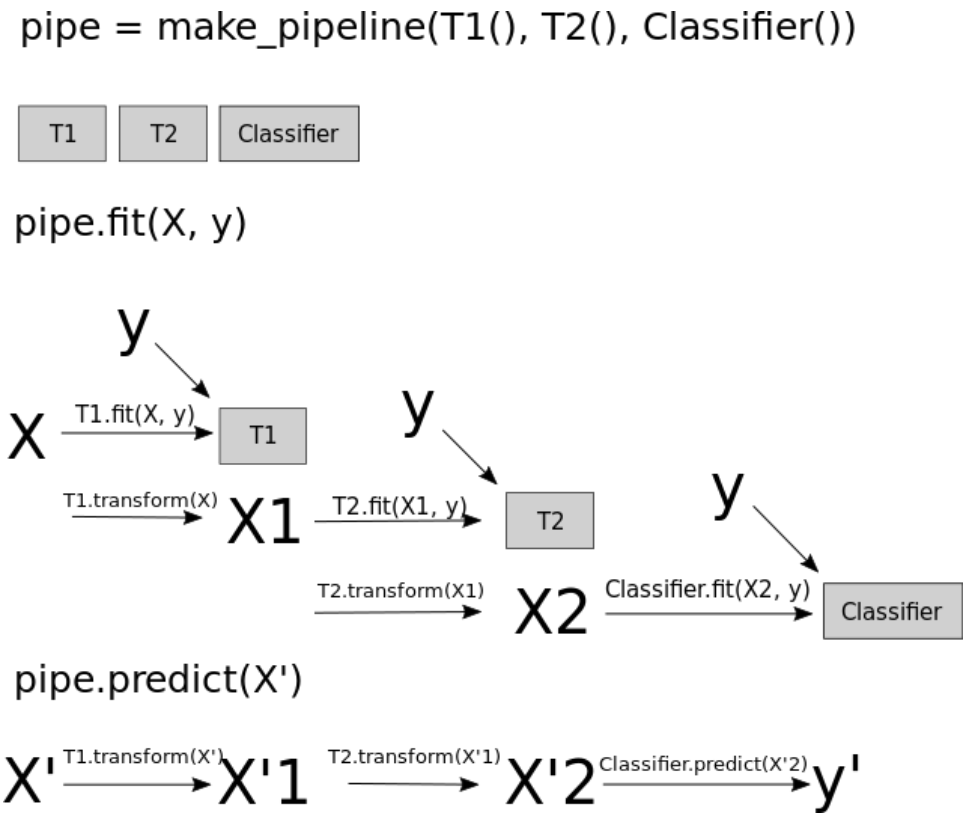
```
def fit(self, X, y):
    X_transformed = X
    for step in self.steps[:-1]:
        # iterate over all but the final step
        # fit and transform the data
        X_transformed = step[1].fit_transform(X_transformed, y)
    # fit the last step
    self.steps[-1][1].fit(X_transformed, y)
    return self
```

When predicting using `Pipeline`, similarly we `transform` the data using all but the last step, and then call `predict` on the last step:

```
def predict(self, X):
    X_transformed = X
```

```
    for step in self.steps[:-1]:
        # iterate over all but the final step
        # transform the data
        X_transformed = step[1].transform(X_transformed)
    # fit the last step
    return self.steps[-1][1].predict(X_transformed)
```

The process is illustrated below for two transformers T1 and T2 and a classifier Clf:



The pipeline is actually even more general than this. There is no requirement for the last step in a pipeline to have a `predict` function, and we could create a pipeline just containing, for example, a scaler and `PCA`. Then, because the last step `PCA` has a `transform` method, we could call `transform` on the pipeline to get the output of `PCA.transform` applied to the data that was processed by the previous step. The last step of a pipeline is only required to have a `fit` method.

Convenient Pipeline creation with `make_pipeline`

Creating a Pipeline using the syntax described above is sometimes a bit cumbersome, and we often don't need user-specified names for each step. There is a convenience

function `make_pipeline` that will create a pipeline for us and automatically name each step based on its class. The syntax for `make_pipeline` is as follows:

```
from sklearn.pipeline import make_pipeline
# standard syntax
pipe_long = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC(C=100))])
# abbreviated syntax
pipe_short = make_pipeline(MinMaxScaler(), SVC(C=100))
```

The pipeline objects `pipe_long` and `pipe_short` do exactly the same, only that `pipe_short` has steps that were automatically named. We can see the name of the steps by looking at the `steps` attribute:

```
pipe_short.steps
```

```
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))),

 ('svc', SVC(C=100, 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))]
```

The steps are named `minmaxscaler` and `svc`. In general the step names are just lowercase version of the class names. If multiple steps have the same class, a number is appended:

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

pipe = make_pipeline(StandardScaler(), PCA(n_components=2), StandardScaler())
pipe.steps
```

```
[('standardscaler-1',

  StandardScaler(copy=True, with_mean=True, with_std=True)),

 ('pca', PCA(copy=True, iterated_power=4, n_components=2, random_state=None,

    svd_solver='auto', tol=0.0, whiten=False)),

 ('standardscaler-2',

  StandardScaler(copy=True, with_mean=True, with_std=True))]
```

As you can see, the first `StandardScaler` was named `"standardscaler-1"` and the second `"standardscaler-2"`. However, in such settings it might be better to use the `Pipeline` construction with explicit names, to give more semantic names to each step.

Accessing step attributes

Often you might want to inspect attributes of one of the steps of the pipeline, say the coefficients of a linear model or the components extracted by `PCA`. The easiest way to access the step in a pipeline is the `named_steps` attribute, which is a dictionary from step names to the estimators:

```
# fit the pipeline defined above to the cancer dataset
pipe.fit(cancer.data)
# extract the first two principal components from the "pca" step
components = pipe.named_steps["pca"].components_
print(components.shape)
```

```
/home/andy/checkout/scikit-learn/sklearn/utils/extmath.py:368: UserWarning: The number of power it

  warnings.warn("The number of power iterations is increased to "
```

Accessing attributes in grid-searched pipeline.

As we discussed above, one of the main reasons to use pipelines is for doing grid-searches. A common task then is to access some of the steps of a pipeline inside a grid-search.

Let's grid-search a `LogisticRegression` classifier on the `cancer` dataset, using `Pipeline` and `StandardScaler` to scale the data before passing it to the `LogisticRegression` classifier.

First we create a pipeline using the `make_pipeline` function:

```
from sklearn.linear_model import LogisticRegression

pipe = make_pipeline(StandardScaler(), LogisticRegression())
```

Next, we create a parameter grid. The regularization parameter to tune for `LogisticRegression` is the parameter `C` as explained in Chapter 2. We use a logarithmic grid for this parameter, searching between 0.01 and 100. Because we used the `make_pipeline` function, the name of the `LogisticRegression` step in the pipeline is the lower-cased class-name `"logisticregression"`. To tune the parameter C, we therefore have to specify a parameter grid for `"logisticregression__C"`:

```
param_grid = {'logisticregression__C': [0.01, 0.1, 1, 10, 100]}
```

We split the cancer dataset into training and test set, and fit a grid-search as usual:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=4)
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
```

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

       estimator=Pipeline(steps=[('standardscaler', StandardScaler(copy=True, with_mean=True, with
```

```
            intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
            penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
            verbose=0, warm_start=False))]),
        fit_params={}, iid=True, n_jobs=1,
        param_grid={'logisticregression__C': [0.01, 0.1, 1, 10, 100]},
        pre_dispatch='2*n_jobs', refit=True, scoring=None, verbose=0)
```

So how do we access the coefficients of the best `LogisticRegression` model that was found by `GridSearchCV`? From Chapter 6 we know that the best model found by `GridSearchCV`, trained on all the training data, is stored in `grid.best_estimator_`:

```
print(grid.best_estimator_)
Pipeline(steps=[('standardscaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('lo
            intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
            penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
            verbose=0, warm_start=False))])
```

This `best_estimator_` in our case is a pipeline with two steps, `"standardscaler"` and `"logisticregression"`. To access the `logisticregression` step, we can use the `named_steps` attribute of the pipeline that we explained above:

```
print(grid.best_estimator_.named_steps["logisticregression"])
LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
            intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
            penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
            verbose=0, warm_start=False)
```

Now that we have the trained `LogisticRegression` instance, we can access the coefficients (weights) associated with each input feature:

```
print(grid.best_estimator_.named_steps["logisticregression"].coef_)
[[-0.389 -0.375 -0.376 -0.396 -0.115  0.017 -0.355 -0.39  -0.058  0.209
  -0.495 -0.004 -0.371 -0.383 -0.045  0.198  0.004 -0.049  0.21   0.224
  -0.547 -0.525 -0.499 -0.515 -0.393 -0.123 -0.388 -0.417 -0.325 -0.139]]
```

This might be a somewhat lengthy expression, but often comes in handy in understanding your models.

Grid-searching preprocessing steps and model parameters

Using pipelines, we can encapsulate all processing steps in our machine learning work flow in a single scikit-learn estimator. Another benefit of doing this is that we can now *adjust the parameters of the preprocessing* using the outcome of a supervised task like regression or classification.

In previous chapters, we used polynomial features on the `boston` dataset before applying the ridge regressor. Let's model that using a pipeline. The pipeline contains three steps: scaling the data, computing polynomial features, and ridge regression:

```
from sklearn.datasets import load_boston
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target, random_state=0)

from sklearn.preprocessing import PolynomialFeatures
pipe = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(),
    Ridge())
```

But how do we know which degree of polynomials to choose, or whether to choose any polynomials or interactions at all? Ideally we want to select the `degree` parameter based on the outcome of the classification.

Using our pipeline, we can search over the `degree` parameter together with the parameter `alpha` of `Ridge`. To do this, we define a `param_grid` that contains both, appropriately prefixed by the step names:

```
param_grid = {'polynomialfeatures__degree': [1, 2, 3],
              'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
```

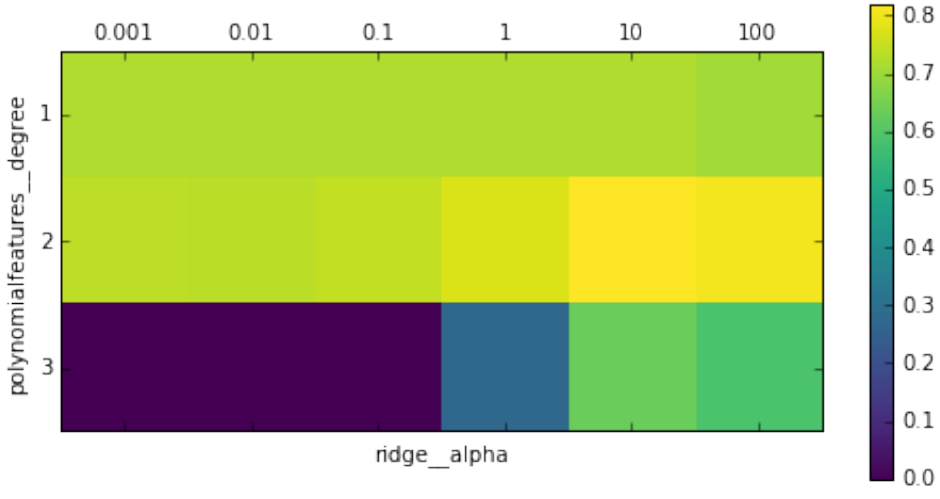
Now we can run our grid-search again:

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5, n_jobs=-1)
grid.fit(X_train, y_train)
```

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

       estimator=Pipeline(steps=[('standardscaler', StandardScaler(copy=True, with_mean=True, with

   normalize=False, random_state=None, solver='auto', tol=0.001))]),

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

       param_grid={'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100], 'polynomialfeatures__degree': [

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

We can visualize the outcome of the cross-validation using a heatmap, as we did in Chapter 6:

```
plt.matshow(np.array([s.mean_validation_score for s in grid.grid_scores_]).reshape(3, -1),
            vmin=0, cmap="viridis")
plt.xlabel("ridge__alpha")
plt.ylabel("polynomialfeatures__degree")
plt.xticks(range(len(param_grid['ridge__alpha'])), param_grid['ridge__alpha'])
plt.yticks(range(len(param_grid['polynomialfeatures__degree'])), param_grid['polynomialfeatures__d

plt.colorbar()
```



Looking at the results produced by the cross-validation, we can see that using polynomials of degree two helps, but that degree three polynomials are much worse than either degree one or two.

This is reflected in the best parameters that were found:

```
print(grid.best_params_)
```

```
{'ridge__alpha': 10, 'polynomialfeatures__degree': 2}
```

Which lead to the following score:

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

```
0.76735803503061784
```

Let's run a grid-search without polynomial features for comparison:

```
param_grid = {'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
pipe = make_pipeline(StandardScaler(), Ridge())
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
grid.score(X_test, y_test)
```

```
0.62717803817745799
```

As we had expected from the grid-search results visualized above, using no polynomial features leads to decidedly worse results. Searching over preprocessing parameters together with model parameters is a very powerful strategy. However, keep in mind that `GridSearchCV` tries *all possible combinations* of the specified parameters. Adding more parameters to your grid therefore increases the number of models that need to be built exponentially.

Summary and Outlook

In this chapter we introduced the `Pipeline` class a general purpose tool to chain together multiple processing steps in a machine learning work flow. Real-world applications of machine learning are rarely an isolated use of a model, and instead a sequence of processing steps. Using pipelines allows us to encapsulate multiple steps into a single python object that adheres to the familiar scikit-learn interface of `fit`, `predict` and `transform`.

In particular when doing model evaluation using cross-validation and parameter selection using grid-search, using the `Pipeline` class to capture all processing steps is essential for proper evaluation.

The `Pipeline` class also allows writing more succinct code, and reduces the likelihood of mistakes that can happen when building processing chains without the pipeline class (like forgetting to apply all transformers on the test set, or maybe not applying them in the right order).

Choosing the right combination of feature extraction, preprocessing and models is somewhat of an art, that often requires some trial-and-error. However, using pipelines, this "trying out" of many different processing steps is quite simple. When experimenting, be careful not to over-complicate your processes, and make sure to evaluate whether every component your are including in your model is necessary.

With this chapter, we complete our survey of general purpose tools and algorithms provided by scikit-learn. You now possess all the required skills and know the necessary mechanisms to apply machine learning in practice. In the next chapter, we will dive in more detail into one particular type of data that is commonly seen in practice, and that requires some special expertise to handle correctly: text data.

CHAPTER 8

Working with Text Data

In Chapter 5, we talked about two kinds of features that can represent properties of the data: continuous features that describe a quantity, and categorical features that are items from a fixed list. There is a third kind of feature that can be found in many application, which is text.

For example, if we want to classify in email into whether it is a legitimate email or spam, the content of the email will certainly contain important information for this classification task. Or maybe we want to learn about the opinion of a politician on the topic of immigration. Here, their speeches or tweets might provide useful information.

In customer services, we often want to find out if a message is a complaint or an inquiry. And depending on the kind of complaint, we might be able to provide automatic advice or forward it to a specific department. These decisions can all be supported by the content of the message that was sent to customer service.

Text data is usually represented as strings, made up of characters. In any of the examples above, the length of the text of each text will be different.

This feature is clearly very different from the numeric features that we discussed so far, and we need to process the text data before we can apply our machine learning algorithms to the text data.

Types of data represented as strings

Before we dive into the processing steps that go into representing text data for machine learning, we want to briefly discuss different kinds of text data that you might encounter. Text is usually just a string in your dataset, but not each string feature should be treated as text. A string feature can sometimes represent categorical

variables, as we discussed in Chapter 5. There is no way to know how to treat a string feature before looking at the data.

There are four kinds of string data you might see:

- Categorical data
- Free strings that can be semantically mapped to categories
- Structured string data
- Text data

Categorical data is data that comes from a fixed list. Say you collect data via a survey where you ask people their favorite color, with a drop-down menu that allows them to select from "red", "green", "blue", "yellow", "black", "white", "purple" and "pink". This will result in a dataset with exactly 8 different possible values, which clearly encode a categorical variable. You can check whether this is the case for your data by eyeballing it (if you see very many different strings it is unlikely that this is a categorical variable), and confirming it by computing the unique values over the dataset, and possibly a histogram over how often each appears. You also might want to check whether each variable actually corresponds to a category that makes sense for your application. Maybe half-way through the existence of your survey, someone found that "black" was misspelled as "blak" and subsequently fixed the survey. As a result your dataset contains both "blak" and "black", which correspond to the same semantic meaning, and should be consolidated.

Now imagine instead of providing a drop-down menu, you provide a text field for the user to provide their own favorite color. Many people might respond with a color name like "black" or "blue". Others might have typographic errors, or use aliases, or different spellings like "gray" and "grey", use more evocative names like "midnight blue", and there will certainly be answers that can not reasonably be related to any color, say "calmly checkered hissing" or "asdfasdfasdf".

The responses you can obtain from a text field belong to the second category, *free strings that correspond to a set of categories*. It will probably be best to encode this data as a categorical variable, where you can select the categories either using the most common entries, or by defining categories that will capture responses in a way that makes sense for your application.

You might then have some categories for standard colors, maybe a category "multicolored" for people that gave answers like "green and red stripes" and an "other" category, for things that can not be encoded otherwise. This kind of preprocessing of strings can take a lot of manual effort, and is not easily automated.

If you are in a position where you can influence data collection, we highly recommend avoiding manually entered values for concepts that are better captured using categorical variables.

Often, manually entered values do not correspond to fixed categories, but still have some *underlying structure*, like addresses, names of places or people, dates, telephone numbers or other identifiers. These kind of strings are often very hard to parse, and their treatment is highly dependent on context and domain. A systematic treatment of these cases is beyond the scope of this book.

The final category of string data is *free form text that consists of phrases or sentences*. Examples of these include tweets, chat logs, hotel reviews, but also the collected works of Shakespeare, the content of Wikipedia or the project Gutenberg collection of 50.000 e-books. All of these collections contain information mostly as sentences of words[footnote: arguably the content of websites linked to in tweets contain more information than the text of the tweet]. For simplicity's sake, let's assume all our documents are in one language, English [footnote: most of what we will talk about in the rest of the chapter also applies to other languages that use the Roman alphabet, and partially also to other alphabets with word boundary delimiters. Chinese for example does not delimit word boundaries, and has other challenges that make applying the techniques of this chapter difficult]. In the context of text analysis, the dataset if often called the *corpus*, and each data point, represented as a single text, is called a *document*.

These terms come from the *information retrieval* (IR) and *natural language processing* (NLP) community, which both deal mostly in text data.

Example application: Sentiment analysis of movie reviews

As a running example in this chapter, we will use a data set of movie reviews collected from the IMDb (Internet Movie Database) website collected by Standford Researcher Andrew Maas [footnote: The dataset is available at http://ai.stanford.edu/~amaas/data/sentiment/]. This dataset contains the text of the reviews, together with a label that indicates "positive" and "negative" reviews. The IMDb website itself contains ratings from one to ten. To simplify the modeling, this annotation is summarized as a two-class classification dataset where reviews with a score of 6 or higher are labeled as positive, and the rest as negative. We will leave the question of whether this is a good representation of the data open, and simply use the data as provided by Andrew Maas.

After unpacking the data, the data set is provided as text files in two separate folders, one for the training data, and one for the test data. Each of these in turn has two subfolders, one called "positive" and one called "negative":

```
!tree -L 2 data/aclImdb
```

```
data/aclImdb
├── test
│   ├── neg
│   └── pos
└── train
    ├── neg
    └── pos


6 directories, 0 files
```

The "positive" folder contains all the positive documents, each as a separate text file, and similarly for the "negative" folder. There is a helper function in scikit-learn to load files stored in such a folder-structure, where each subfolder corresponds to a label, called `load_files`. We apply the `load_files` function first to the training data:

```
from sklearn.datasets import load_files

reviews_train = load_files("data/aclImdb/train/")
# load_files returns a bunch, containing training texts and training labels
text_train, y_train = reviews_train.data, reviews_train.target
print("type of text_train: ", type(text_train))
print("length of text_train: ", len(text_train))
print("text_train[1]:")
# print review number 1
print(text_train[1])
```

```
type of text_train:  <class 'list'>

length of text_train:  25000

text_train[1]:

b'Words can\'t describe how bad this movie is. I can\'t explain it by writing only. You have too s
```

We can see that `text_train` is a list of length 25.000, where each entry is a string containing a review. We printed the review with index one. You can see that the review contains some HTML line breaks (`"<br />"`). While these are unlikely to have a large impact on our machine learning models, it is better to clean the data from this formating before we proceed:

```
text_train = [doc.replace(b"<br />", b" ") for doc in text_train]
```

The type of the entries of `text_train` depends on your Python version. In Python3, they will be of type "bytes" which represents a binary encoding of the string data. In

Python2, `text_train` contains strings. We won't go into the details of the different string types in Python here, but recommend that you read the documentation regarding strings and unicode in Python [Footnote: https://docs.python.org/3/howto/unicode.html for Python 3 and https://docs.python.org/2/howto/unicode.html for Python 2].

The dataset was collected such that the positive class and the negative class balanced, so that there are as many positive as negative strings:

```
print(np.bincount(y_train))
```

```
[12500 12500]
```

We load the test dataset in the same manner:

```
reviews_test = load_files("data/aclImdb/test/")
text_test, y_test = reviews_test.data, reviews_test.target
print("Number of documents in test data: %d" % len(text_test))
print(np.bincount(y_test))
text_test = [doc.replace(b"<br />", b" ") for doc in text_test]
```

```
Number of documents in test data: 25000

[12500 12500]
```

The task we want to solve is given a review, we want to assign the labels "positive" and "negative" based on the text content of the review. This is a standard binary classification task. However, the text data is not in a format that a machine learning model can handle. We need to convert the string representation of the text into a numeric representation that we can apply our machine learning algorithms to.

Representing text data as Bag of Words

One of the most simple, but effective and commonly used ways to represent text for machine learning is using the *bag-of-words* representation. When using bag-of-words, we discard most of the structure of the input text, like chapters, paragraphs, sentences and formatting, and only count *how often each word appears in each text*. Discarding all this structure and counting only occurrence leads to the mental image of representing text as a "bag".

Computing the bag-of-word representation for a corpus of documents consists of the following three steps:

1) Tokenization: Split each document into the words that appear in it (called *tokens*), for example by splitting them by whitespace and punctuation.

2) Vocabulary building: Collect a vocabulary of all words that appear in any of the documents, and number them (say in alphabetical order).

3) Encoding: For each document, count how often each of the words in the vocabulary appear in this document.

bag_of_words

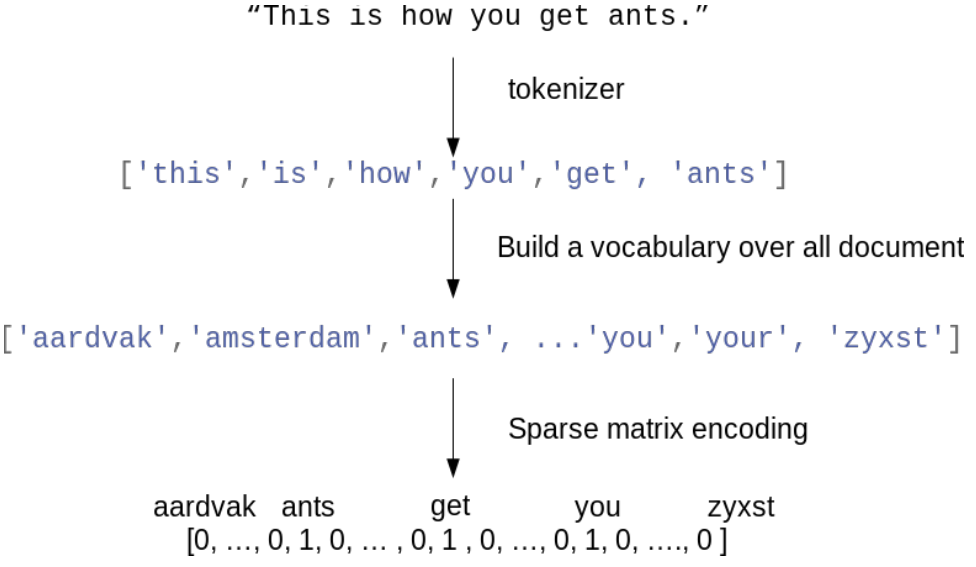


Figure bag_of_words illustrates the process on the string "This is how you get ants". The output of the process is one vector of word-counts for each document. For each word in the vocabulary, we have a count of how often it appears in each document. That means our numeric representation has one feature for each unique word in the whole dataset. Note how the order of the words in the original string is completely irrelevant to the bag of words feature representation. There are some subtleties involved in step 1 and step 2 above, which we will discuss in more detail later in this chapter.

For now, let's look at how we can apply the bag-of-word processing using scikit-learn.

Applying bag-of-words to a toy dataset

The bag-of-word representation is implemented in the `CountVectorizer`, which is a transformer. Let's first apply it to a toy dataset, consisting of two samples, to see it working:

```
bards_words =["The fool doth think he is wise,",
              "but the wise man knows himself to be a fool"]
```

We import and instantiate the `CountVecorizer` and `fit` it to our toy data:

```
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
vect.fit(bards_words)
```

```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

Fitting the `CountVectorizer` consists of the tokenization of the training data and building of the vocabulary, which we can access as the `vocabulary_` attribute:

```
print(len(vect.vocabulary_))
print("vocabulary content:")
vect.vocabulary_
```

```
{'be': 0,
 'but': 1,
 'doth': 2,
 'fool': 3,
 'he': 4,
 'himself': 5,
 'is': 6,
 'knows': 7,
 'man': 8,
 'the': 9,
 'think': 10,
 'to': 11,
 'wise': 12}
13

vocabulary content:
```

The vocabulary consists of 13 words, from "be" to "wise".

To create the bag-of-words representation for the training data, we call the `transform` method:

```
bag_of_words = vect.transform(bards_words)
bag_of_words
```

```
<2x13 sparse matrix of type '<class 'numpy.int64'>'

    with 16 stored elements in Compressed Sparse Row format>
```

The bag of word representation is stored in a SciPy sparse matrix that only stores the entries that are non-zero (see Chapter 1). The matrix is of shape 2 x 13, one row for each of the two data points, and one feature for each of the words in the vocabulary. A sparse matrix is used as most documents only contain a small subset of the words in the vocabulary, meaning most entries in the feature array are zero. Think about how many different words might appear in a movie review compared to all the words in the English language (which is what the vocabulary models). Storing all these zeros would be prohibitive, and a waste of memory.

To look at the actual content of the sparse matrix, we can convert it to a "dense" NumPy array (that also stores all the zero entries) using the `toarray` method. This is possible because we are using a small toy dataset that contains only 13 words. For any real dataset, this would result in a `MemoryError`.

```
print(bag_of_words.toarray())
```

```
[[0 0 1 1 1 0 1 0 0 1 1 0 1]

 [1 1 0 1 0 1 0 1 1 1 0 1 1]]
```

We can see that the word counts for each word are either zero or one, none of the two strings in `bards_words` contain a word twice. You can read these feature vectors as follows: The first string `"The fool doth think he is wise,"` is represented as the first row in, and it contains the first word in the vocabulary, `"be"`, zero times. It also contains the second word in the vocabulary, `"but"`, zero times. It does contain the third word, `"doth"`, once, and so on. Looking at both rows, we can see that the fourth word, `"fool"`, the tenth word `"the"` and the thirteenth word `"wise"` appear in both strings.

Bag-of-word for movie reviews

Now that we went through the bag-of-word process in detail, let's apply it to our task of sentiment analysis for movie reviews. Above, we already loaded our training and test data from the IMDb reviews into lists of strings (`text_train` and `text_test`), which we will now process:

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print(repr(X_train))
```

```
<25000x74849 sparse matrix of type '<class 'numpy.int64'>'

    with 3431196 stored elements in Compressed Sparse Row format>
```

The shape of `X_train`, the bag-of-words representation of the training data, is 25.000 x 74.849, indicating that the vocabulary contains 74.849 entries. Again, the data is stored as a SciPy sparse matrix. Let's look in a bit more detail at the vocabulary. Another way to access the vocabulary is using the `get_feature_name` method of the vectorizer, which returns a convenient list where each entry corresponds to one feature:

```
feature_names = vect.get_feature_names()
print(len(feature_names))
# print first fifty features
print(feature_names[:20])
# print feature 20010 to 20030
print(feature_names[20010:20030])
# get every 2000th word to get an overview
print(feature_names[::2000])
```

```
74849

['00', '000', '0000000000001', '00001', '00015', '000s', '001', '003830', '006', '007', '0079', '0

['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'drawback', 'drawbacks', 'dra

['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bête', 'chicanery', 'condensing', 'cunning',
```

As you can see, possibly a bit surprisingly, is that the first ten entries in the vocabulary are all numbers. All these numbers appear somewhere in the reviews, and are therefore extracted as words. Most of these numbers don't have any immediate semantic meaning---apart from "007", which, in particular in the context of movies, is likely to refer to the James Bond character [footnote: A quick analysis of the data confirms that this is indeed the case. Try confirming it yourself.]. Weeding out the meaningful from the non-meaningful "words" is sometimes tricky. Looking at some words further along in the vocabulary, we find a collection of English words starting with "dra". You might notice that for "draught", "drawback" and "drawer" both the singular and plural form are contained in the vocabulary as distinct words. These words have very closely related semantic meanings, and counting them as different words, corresponding to different features, might not be ideal.

Before we try to improve our feature extraction, let us obtain a quantitative measure of performance by actually building a classifier. We have the training labels stored in `y_train` and the bag-of-word representation of the training data in `X_train`, so we can train a classifier on this data. For high-dimensional, sparse data like this, linear models, like `LogisticRegression` often work best. Let's start by evaluating `LogisticRegresssion` using cross-validation[footnote: The attentive reader might notice that we violate our lesson from Chapter 7 on cross-validation with preprocessing. Using

the default settings of `CountVectorizer`, it actually does not collect any statistics, so our results are valid. Using `Pipeline` from the start would be a better choice for applications, but we defer it for ease of exposure.]

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression

scores = cross_val_score(LogisticRegression(), X_train, y_train, cv=5)
np.mean(scores)
```

```
0.88131999999999999
```

We obtain a mean cross-validation score of 88.2%, which indicates reasonable performance for a balanced binary classification task. We know that `LogisticRegression` has a regularization parameter `C` which we can tune via cross-validation:

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: ", grid.best_score_)
print("Best parameters: ", grid.best_params_)
```

```
Best cross-validation score:  0.88816


Best parameters:  {'C': 0.1}
```

We obtain a cross-validation score of 88.8% using `C=0.1`. We can now assess the generalization-performance of this parameter setting on the test set:

```
X_test = vect.transform(text_test)
grid.score(X_test, y_test)
```

```
0.87895999999999996
```

Now, let's see if we can improve the extraction of words. The way the `CountVectorizer` extracts tokens is using a regular expression. By default, the regular expression that is used is `"\b\w\w+\b"`. If you are not familiar with regular expressions, this means it finds all sequences of characters that consist of at least two letters or numbers (`"\w"`) and that are separated by word boundaries (`"\b"`), in particular it does not find single-letter words, and it splits up contractions like "doesn't" or "bit.ly", but matches "h8ter" as a single word. The `CountVectorizer` then converts all words to lower-case characters, so that "soon", "Soon" and "sOon" all correspond to the same token (and therefore feature).

This simple mechanism works quite well in practice, but as we saw above, we get many uninformative features like the numbers. One way to cut back on these is to only use tokens that appear in at least 2 documents (or at least 5 documents etc). A token that appears only in a single document is unlikely to appear in the test set and is therefore not helpful.

We can set the minimum number of documents a token needs to appear in with the `min_df` parameter:

```
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print(repr(X_train))
```

```
<25000x27271 sparse matrix of type '<class 'numpy.int64'>'

    with 3354014 stored elements in Compressed Sparse Row format>
```

By requiring at least five appearances of each token, we can bring down the number of features to 27.272 (see the output above), only about a third of the original features. Let's look at some tokens again:

```
feature_names = vect.get_feature_names()

# print first fifty features
print(feature_names[:50])
# print feature 20010 to 20020
print(feature_names[20010:20030])
#
print(feature_names[::700])
```

```
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '100', '10

['repentance', 'repercussions', 'repertoire', 'repetition', 'repetitions', 'repetitious', 'repetit

['00', 'affections', 'appropriately', 'barbra', 'blurbs', 'butchered', 'cheese', 'commitment', 'co
```

There are clearly much fewer numbers, and some of the more obscure words or misspellings seem to have vanished. Let's see how well our model performs by doing a grid-search again:

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: ", grid.best_score_)
```

```
Best cross-validation score:  0.88812
```

The best validation accuracy of the grid-search is still 88.8%, unchanged from before. We didn't improve our model, but having less features to deal with speeds up processing and throwing away useless features might make the model more interpretable.

Stop-words

Another way that we can get rid of uninformative words is by discarding words that are too frequent to be informative. There are two main approaches: using a language-specific list of stop words, or discarding words that appear too frequently. Scikit-learn had a built-in list of English stop-words in the `feature_extraction.text` module:

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
# print number of stop words
```

```
print(len(ENGLISH_STOP_WORDS))
# print some of the stop words
print(list(ENGLISH_STOP_WORDS)[::10])
```

```
318

['show', 'however', 'something', 'is', 'of', 'are', 'about', 'least', 'eight', 'thereupon', 'alway
```

Clearly, removing the stop-words in the list can only decrease the number of features by the lenght of the list, here 318, but it might lead to an improvement in performance. Let's give it a try:

```
# specifying "english" uses the build-in list. We could also augment it and pass our own.
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print(repr(X_train))
```

```
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'

    with 2149958 stored elements in Compressed Sparse Row format>
```

There are now 305 (=27272 - 26967) less features in the dataset, which means that most, but not all of the stop-words appeared. Let's run the grid-search again:

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: ", grid.best_score_)
```

```
Best cross-validation score:  0.88296
```

The grid-search performance decreased slightly using the stop words. The change is very slight, but given that excluding 305 features is unlikely to change performance or interpretability a lot, it doesn't seem worth using this list. Fixed lists are mostly helpful for small datasets, that might not contain enough information for the model to determine which words are stop words from the data itself. As an exercise, you can try out the other approach, discarding frequently appearing words, by setting the `max_df` option of `CountVectorizer` and see how it influences the number of features and the performance.

Rescaling the data with TFIDF

Instead of dropping features that are deemed unimportant, another approach is to rescale features by how informative we expect them to be. One of the most common ways to do this is using the term frequency–inverse document frequency (tf-idf) method. The intuition of this method is to give high weight to a term that appears often in a particular document, but not in many documents in the corpus. If a word appears often in a particular document, but not in very many documents, it is likely to be very descriptive of the content of that document.

Scikit-learn implements the tf-idf method in two classes, the `TfidfTransformer`, which takes in the sparse matrix output produced by `CountVectorizer` and transforms it, or `TfidfVectorizer`, which takes in the text data and does both the bag-of-words feature extraction and the tf-idf transformation.

There are several variants of the tf-idf rescaling scheme, which you can find on the wikipedia page [footnote: https://en.wikipedia.org/wiki/Tf-idf]. The tf-idf score for word w in document d as implemented in both the `TfidfTransformer` and `TfidfVectorizer` is given by:

\begin{equation*}

\text{tfidf}(w, d) = \text{tf} \log\big(\frac{N + 1}{N_w + 1}\big) + 1

\end{equation*}

where N is the number of documents in the training set, N_w is the number of documents in the training set that the word d appears in, and tf, the term frequency, is the number of times that the word w appears in the query document (the document you want to transform or encode). Both classes also apply l2 normalization after computing the tf-idf representation, in other words they rescale the representation of each document to have euclidean norm 1. Rescaling in this way means that the length of a document (the number of words) does not change the vectorized representation. We provide this formula here mostly for completeness, and you don't need to remember it to use the tf-idf encoding.

Because tf-idf actually makes use of the statistical properties of the training data, we will use a pipeline, as described in Chapter 7, to ensure the results of our grid-search are valid. This leads to the following code:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5, norm=None), LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: ", grid.best_score_)
```

```
Best cross-validation score:  0.89392
```

As you can see, there is some improvement of using tf-idf instead of using just word counts. We can also inspect which words tf-idf found most important. Keep in mind that the tf-idf scaling is meant to find words that distinguish documents, but it is a purely unsupervised technique. So "important" here does not necessarily related to the "positive review" and "negative review" labels we are interested in. First we extract the `TfidfVectorizer` from the pipeline:

```
vectorizer = grid.best_estimator_.named_steps["tfidfvectorizer"]
# transform the training dataset:
X_train = vectorizer.transform(text_train)
# find maximum value for each of the features over dataset:
max_value = X_train.max(axis=0).toarray().ravel()
sorted_by_tfidf = max_value.argsort()
# get feature names
feature_names = np.array(vectorizer.get_feature_names())

print("features with lowest tfidf")
print(feature_names[sorted_by_tfidf[:20]])

print("features with highest tfidf")
print(feature_names[sorted_by_tfidf[-20:]])
```

```
features with lowest tfidf

['poignant' 'disagree' 'instantly' 'importantly' 'lacked' 'occurred'

 'currently' 'altogether' 'nearby' 'undoubtedly' 'directs' 'fond' 'stinker'

 'avoided' 'emphasis' 'commented' 'disappoint' 'realizing' 'downhill'

 'inane']

features with highest tfidf

['coop' 'homer' 'dillinger' 'hackenstein' 'gadget' 'taker' 'macarthur'

 'vargas' 'jesse' 'basket' 'dominick' 'the' 'victor' 'bridget' 'victoria'

 'khouri' 'zizek' 'rob' 'timon' 'titanic']
```

Features with low tf-idf are those that are either very commonly used across documents, or are only used sparingly, and only in very long documents. Interestingly, many of the high tf-idf features actually identify certain shows or movies. These terms only appear in reviews for this particular show or franchise, but tend to appear very often in these particular reviews. This is very clear for example for "pokemon", "smallville" and "doodlebops", but "scanners" here actually also refers to a movie title. These words are unlikely to help us in our sentiment classification task (unless maybe some franchises are universally reviewed positively or negatively) but certainly contain a lot of specific information about the review.

We can also find the words that have low inverse document frequency, that is those that appear frequently and are therefore deemed less important. The inverse document frequency values found on the training set are stored in the `idf_` attribute:

```
sorted_by_idf = np.argsort(vectorizer.idf_)
print("features with lowest idf")
print(feature_names[sorted_by_idf[:100]])
```

```
features with lowest idf

['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with'

 'was' 'as' 'on' 'movie' 'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all'

 'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they' 'there' 'if' 'his' 'out'

 'just' 'about' 'he' 'or' 'has' 'what' 'some' 'good' 'can' 'more' 'when'

 'time' 'up' 'very' 'even' 'only' 'no' 'would' 'my' 'see' 'really' 'story'

 'which' 'well' 'had' 'me' 'than' 'much' 'their' 'get' 'were' 'other'

 'been' 'do' 'most' 'don' 'her' 'also' 'into' 'first' 'made' 'how' 'great'

 'because' 'will' 'people' 'make' 'way' 'could' 'we' 'bad' 'after' 'any'

 'too' 'then' 'them' 'she' 'watch' 'think' 'acting' 'movies' 'seen' 'its'

 'him']
```

As expected, these are mostly English stop words like "the" and "no". But some are clearly domain specific to the movie reviews, like "movie", "film", "time", "story" and so on. Interestingly, "good", "great" and "bad" are also among the most frequent, and therefore "least relevant" words, even though we might expect these to be very important for our sentiment analysis task.

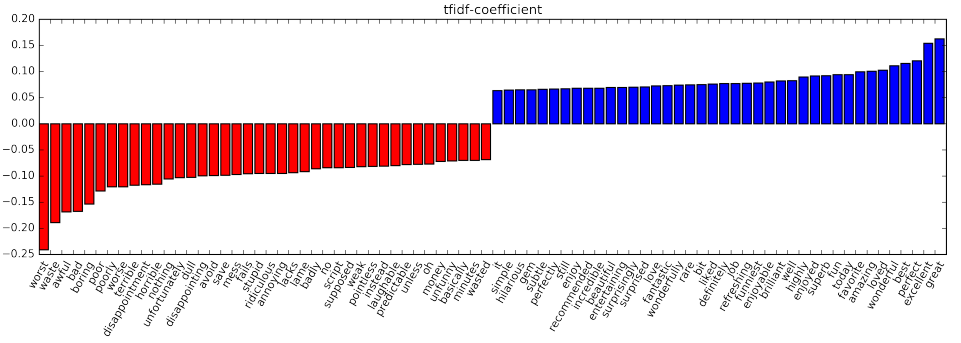
Investigating model coefficients

Finally, let us look into a bit more detail into what our logistic regression model actually learned from the data.

Because there are so many features, 27.272 after removing the infrequent ones, we can clearly not look at all of the coefficients at the same time. However, we can look at the largest coefficients, and see which words these correspond to.

We will use the last model that we trained, based on the tf-idf features.

```
mglearn.tools.visualize_coefficients(grid.best_estimator_.named_steps["logisticregression"].coef_,
                                     feature_names, n_top_features=40)
plt.title("tfidf-coefficient")
```



The bar-chart in Figure tfidf-coefficient shows the 25 largest and 25 smallest coefficients of the logistic regression model, with the bar showing the size of each coefficient. The negative coefficients on the left belong to words that according to the model are indicative of negative reviews, while the positive coefficients on the right belong to word that according to the model indicate positive reviews. Most of the terms are quite intuitive, like "worst", "waste", "disappointment" and "laughable" indicating bad movie reviews, while "excellent", "wonderful", "enjoyable" and "refreshing" indicate positive movie reviews. Some words are slightly less clear, like "bit", "job" and "today", but these might be part of phrases like "good job" or "best today".

Bag of words with more than one word (n-grams)

One of the main disadvantages of using a bag-of-word representation is that word order is completely discarded. Therefore the two strings "it's bad, not good at all" and "it's good, not bad at all" have exactly the same representation, even though the meanings are inverted. Putting "not" in front of a word is only one (if extreme) example of how context matters. There is a way of capturing context when using a bag-of-word representation, by not only considering the counts of single tokens, but also the counts of pairs or triples of tokens that appear next to each other.

Pairs of tokens are known as *bigrams*, triplets of tokens are known as *trigrams* and more generally sequences of tokens are known as *n-grams*. We can change the range of tokens that are considered as features by changing the `ngram_range` parameter of the `CountVecorizer` or `TfidfVectorizer`. The `ngram_range` parameter is a tuple, consisting of the minimum length and the maximum length of the sequences of tokens that are considered. Here is an example on the toy data from above:

```
print(bards_words)
```

```
['The fool doth think he is wise,', 'but the wise man knows himself to be a fool']
```

The default is to create one feature per sequence of tokens that are at least one token long, and at most one token long, in other words exactly one token long (single tokens are also called *unigrams*):

```
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
print(len(cv.vocabulary_))
print(cv.get_feature_names())
```

```
13

['be', 'but', 'doth', 'fool', 'he', 'himself', 'is', 'knows', 'man', 'the', 'think', 'to', 'wise']
```

To look only at bigrams, that is only at sequences of two tokens following each other, we can set `ngram_range` to `(2, 2)`:

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print(len(cv.vocabulary_))
print(cv.get_feature_names())
```

```
14

['be fool', 'but the', 'doth think', 'fool doth', 'he is', 'himself to', 'is wise', 'knows himself
```

Using longer sequences of tokens usually results in many more features, and in more specific features. There is no common bigram between the two phrases in `bard_words`:

```
cv.transform(bards_words).toarray()
```

```
array([[0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0],

       [1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1]])
```

For most applications, the minimum number of tokens should be one, as single words often capture a lot of meaning. Adding bigrams helps in most cases, and adding longer sequences, up to 5-grams, might help, but will lead to an explosion of the number of features, and might lead to overfitting, as there are many very specific features.

Here is what using unigrams, bigrams and trigrams on `bards_words` looks like:

```
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print(len(cv.vocabulary_))
print(cv.get_feature_names())
```

```
39

['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think', 'doth think he', 'fool',
```

Let's use the `TfidfVectorizer` on the IMDb movie review data and find the best setting of n-gram range using grid-search:

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
              "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (1, 3)]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
```

```
print("Best cross-validation score: ", grid.best_score_)
grid.best_params_
```

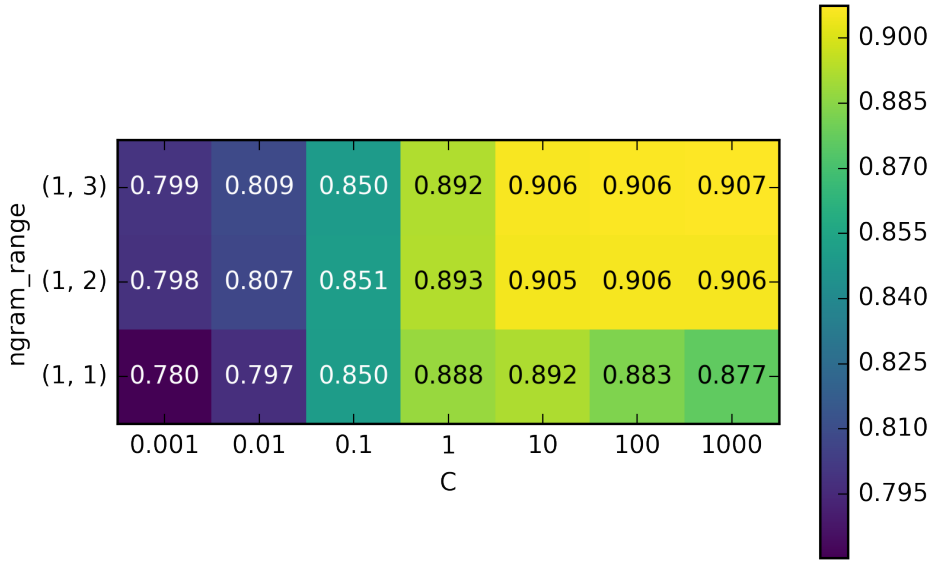
```
{'logisticregression__C': 1000, 'tfidfvectorizer__ngram_range': (1, 3)}
```

```
Best cross-validation score:  0.9074
```

As you can see from the results, we improved performance a bit more than a percent by adding bigram and trigram features.

We can visualize the cross-validation accuracy as a function of the `ngram_range` and `C` parameter as a heat map, as we did in Chapter 6:

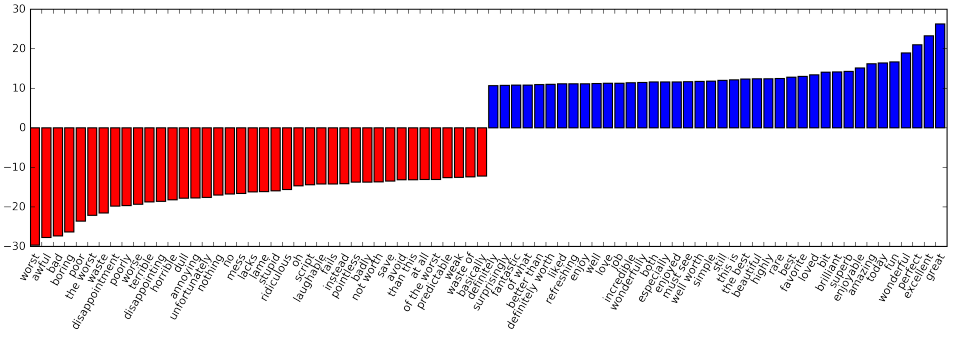
```
# extract scores from grid_search
scores = [s.mean_validation_score for s in grid.grid_scores_]
scores = np.array(scores).reshape(-1, 3).T
# visualize heatmap
heatmap = mglearn.tools.heatmap(scores, xlabel="C", ylabel="ngram_range",
                                xticklabels=param_grid['logisticregression__C'],
                                yticklabels=param_grid['tfidfvectorizer__ngram_range'],
                                cmap="viridis", fmt="%.3f")
plt.colorbar(heatmap);
```



From the heat map we can see that using bigrams increases performance quite a bit, while adding three-grams only provides a very small benefit in terms of accuracy. To understand better how the model improved, we visualize the important coefficient for the best model (which includes unigrams, bigrams and trigrams):

```
# extract feature names and coefficients
feature_names = np.array(grid.best_estimator_.named_steps['tfidfvectorizer'].get_feature_names())
coef = grid.best_estimator_.named_steps['logisticregression'].coef_
```

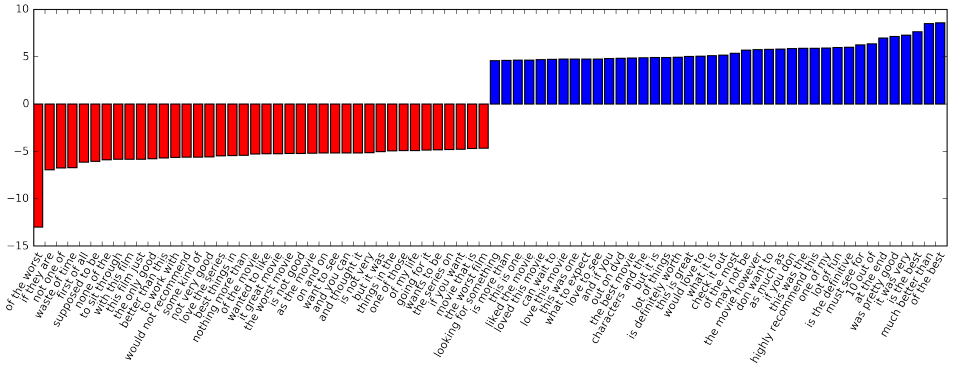
```
mglearn.tools.visualize_coefficients(coef, feature_names, n_top_features=40)
plt.title("ngram-coefficient")
```



There are particularly interesting features containing the word "worth" that were not present in the unigram model: "not worth" is indicative of a negative review, while "definitely wroth" and "well worth" are indicative of a positive review. This is a prime example of context influencing the meaning of the word "worth".

Below, we visualize only bigrams and trigrams, to provide further insight into why these features are helpful. Many of the useful bigrams and trigrams consist of common words that would not be informative on their own, as in the phrases "none of the", "the only good", "on and on", "this was one of", "of the most" and so on. However, the impact of these features is quite limited compared to the importance of the unigram features.

```
# find 3-gram features
mask = np.array([len(feature.split(" ")) for feature in feature_names]) == 3
# visualize only 3-gram features:
mglearn.tools.visualize_coefficients(coef.ravel()[mask],
                                     feature_names[mask], n_top_features=40)
```



Advanced tokenization, stemming and lemmatization

We mentioned above that the feature extraction in the `CountVectorizer` and `TfidfVectorizer` is relatively simple, and much more elaborate methods are possible. One particular step that is often improved in more sophisticated text processing applications is the first step in the bag-of-word model, the tokenization, the step defines what constitutes a word for the purpose of feature extraction.

We saw above that the vocabulary often contains singular and plural version of words as in `'drawback', 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings'`. For the purpose of a bag-of-words model, the semantics of "drawback" and "drawbacks" are so close that distinguishing them will only increase overfitting, and not allow the model to fully exploit the training data. Similarly, we found the vocabulary includes words like `'replace', 'replaced', 'replacement', 'replaces', 'replacing'`, which are different verb forms and a nouns relating to the verb "to replace".

Similarly to having singular and plural of a noun, treating different verb-forms and related words as distinct tokens is disadvantageous for building a model that generalizes well. This problem can be overcome by representing each word using its *word stem*, identifying (or *conflating*) all the words that have the same word stem. If this is done by using a rule-based heuristic, like dropping common suffixes, this is usually referred to as *stemming*. If instead a dictionary of known word forms is used (that is using an explicit and human-verified system), and the role of the word in the sentence taken into account, the process is referred to as *lemmatization* and the standardized form of the word is referred to as *lemma*. Both processing methods, lemmatization and stemming, are forms of *normalization* that try to extract some normal form of a word. Another interesting case of normalization is spell correction, which can be helpful in practice, but is outside of the scope of this book.

To get a better feeling for normalization, let's compare a method for stemming, the Porter stemmer, a widely used collection of heuristics (here imported from the `nltk` package) to lemmatization as implemented in the SpaCy package. For details of the interface, consult the `nltk` and `SpaCy` documentations. We are more interested in the general principles here.

```
import spacy
import nltk

# load spacy's English language models
en_nlp = spacy.load('en')
# instantiate NLTK's Porter stemmer
stemmer = nltk.stem.PorterStemmer()

# define function to compare lemmatization in spacy with stemming in NLKT
def compare_normalization(doc):
```

```
    # tokenize document in spacy:
    doc_spacy = en_nlp(doc)
    # print lemmas found by spacy
    print("Lemmatization:")
    print([token.lemma_ for token in doc_spacy])
    # print tokens found by Porter stemmer
    print("Stemming:")
    print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])
```

We will compare lemmatization and the Porter stemmer on a sentence designed to show some of the differences:

```
compare_normalization(u"Our meeting today was worse than yesterday, I'm scared of meeting the clie

Lemmatization:

['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be', 'scared', 'of', 'mee

Stemming:

['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', "'m", 'scare', 'of', 'meet',
```

Stemming is always restricted to trimming the word to a stem, so "was" becomes "wa", while lemmatization can retrieve the correct base verb form, "be". Similarly, lemmatization can normalize "worse" to "bad", while stemming produces "wors". Another major difference is that stemming reduces both occurrences of "meeting" to "meet". Using lemmatization, the first occurrence of "meeting" is recognized as a noun, and left as-is, while the second occurrence is recognized as verb, and reduced to "meet". In general, lemmatization is a much more involved process than stemming, but usually produces better results when used for normalizing tokens for machine learning.

While scikit-learn implements neither form of normalization, `CountVectorizer` allows specifying your own tokenizer to convert each document into a list of tokens using the `tokenizer` parameter. We can use the lemmatization from SpaCy to create a callable that will take a string and produce a list of lemmas:

```
# Technicallity: we want to use the regexp based tokenizer that is used by CountVectorizer
# and only use the lemmatization from SpaCy. To this end, we replace en_nlp.tokenizer (the SpaCy t
# with the regexp based tokenization
import re
# regexp used in CountVectorizer:
regexp = re.compile('(?u)\\b\\w\\w+\\b')

# load spacy language model and save old tokenizer
en_nlp = spacy.load('en')
old_tokenizer = en_nlp.tokenizer
# replace the tokenizer with the regexp above
en_nlp.tokenizer = lambda string: old_tokenizer.tokens_from_list(regexp.findall(string))

# create a custom tokenizer using the SpaCy document processing pipeline
# (now using our own tokenizer)
```

```
def custom_tokenizer(document):
    doc_spacy = en_nlp(document, entity=False, parse=False)
    return [token.lemma_ for token in doc_spacy]

# define a count vectorizer with the custom tokenizer
lemma_vect = CountVectorizer(tokenizer=custom_tokenizer, min_df=5)
```

Let's transform the data and inspect the vocabulary size:

```
# transform text_train using CountVectorizer with lemmatization
X_train_lemma = lemma_vect.fit_transform(text_train)
print("X_train_lemma.shape: ", X_train_lemma.shape)

# Standard CountVectorizer for reference
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train.shape: ", X_train.shape)
```

```
X_train_lemma.shape:  (25000, 21596)

X_train.shape:  (25000, 27271)
```

As you can see from the output above, lemmatization reduced the number of features from 27.272 (with the standard `CountVectorizer` processing) to 21.596. Lemmatization can be seen as a kind of regularization, as it conflates certain features. Therefore, we expect lemmatization to improve performance most when the dataset is small. To illustrate how lemmatization can help, we will use `StratifiedShuffleSplit` for cross-validation, using only 1% of the data as training data, and the rest as test data:

```
# build a grid-search using only 1% of the data as training set:
from sklearn.model_selection import StratifiedShuffleSplit

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
cv = StratifiedShuffleSplit(n_iter=5, test_size=0.99, train_size=0.01, random_state=0)
grid = GridSearchCV(LogisticRegression(),
                    param_grid, cv=cv)
# Perform grid-search with standard CountVectorizer
grid.fit(X_train, y_train)
print("Best cross-validation score (standard CountVectorizer): {:.3f}".format(grid.best_score_))
# Perform grid-search with Lemmatization
grid.fit(X_train_lemma, y_train)
print("Best cross-validation score (lemmatization): {:.3f}".format(grid.best_score_))
```

```
Best cross-validation score (standard CountVectorizer): 0.721

Best cross-validation score (lemmatization): 0.731
```

In this case, lemmatization provided a modest improvement in performance. As with many of the different feature extraction techniques, the result varies depending on the dataset. Lemmatization and stemming can sometimes help in building better, or at least more compact models, so we suggest you give these techniques a try when trying to squeeze out the last bit of performance on a particular task.

Topic Modeling and Document Clustering

One particular technique that is often applied to text data is *topic modeling*, which is an umbrella term, describing the task of assigning each document to one or multiple *topics*, usually without supervision. A good example for this is news data, which might be categorized into topics like "politics", "sports", "finance" and so on. If each document is assigned a single topic, this is the task of clustering the documents, as discussed in Chapter 3.

If each document can have more than one topic, the task relates to decomposition methods from Chapter 3. Each of the components we learn then corresponds to one topic, and the coefficient of the components in the representation of a document tells us how much each document is about a particular topic.

Often, when people talk about topic modeling, they refer to one particular decomposition method called Latent Dirichlet Allocation (often LDA for short [footnote: There is another machine learning model called LDA, which is Linear Discriminant Analysis, a linear classification model. This leads to quite some confusion. In this book, LDA refers to Latent Dirichlet Allocation).

Intuitively, the LDA model tries to find groups of words (the topics) that appear together frequently. LDA also requires that each document can be understood as a "mixture" of a subset of the topics. It is important to understand that for the machine learning model a "topic" might not be what we would normally call a topic in everyday speech, but that it resembles more the components extracted by PCA or NMF, which might or might not have a semantic meaning.

Even if there is a semantic meaning for an LDA "topic", it might not be something we'd usually call a topic. Going back to the example of news articles, we might have a collection of articles about sports, politics and finance, written by two specific authors. In a politics article, we might expect words like "govenor", "vote", "party" etc, while in a sports article we might expect words like "team", "score" and "season". Each of these groups will likely appear together, while it's less likely that "team" and "govenor" appear together.

However, these are not the only groups of words we might expect to appear together. The two reporters might prefer different phrases or different choices of words. Maybe one of them likes to use he word "demarcate" and one likes the word "polarize". Another "topic" would then be "words often used by reporter A" and "words often used by reporter B", thought these are not topics in the usual sense of the word.

Let's apply LDA to our movie review dataset to see how it works in practice. For unsupervised text document models, it is often good to remove very common words, as they might otherwise dominate the analysis. We remove words that appear in at

least 20 percent of the documents, and we limit the bag-of-words model to the 10.000 that are most common after removing the top 20 percent:

```
vect = CountVectorizer(max_features=10000, max_df=.15)
X = vect.fit_transform(text_train)
```

We learn a topic model with 10 topics, which is few enough that we can look at all of them.

Similarly to the components in NMF, topics don't have an inherent ordering, and changing the number of topics will change all of the topics. [footnote: In fact, NMF and LDA solve quite related problems, and we could also use NMF to extract "topics".]

We choose the "batch" learning method, which is somewhat slower than the default, but usually provides better results, and increase "max_iter", which can also lead to better models.

```
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_topics=10, learning_method="batch", max_iter=25, random_state=0)
# be build the model and transform the data in one step
# computing transform takes some time, and we can save time by doing both at once.
document_topics = lda.fit_transform(X)
```

As in the decomposition methods we saw in Chapter 3, `LDA` has a `components_` attribute, that stores how important each word is for each topic. The size of `components_` is `(n_topics, n_words)`.

```
lda.components_.shape
```

```
(10, 10000)
```

To understand better what the different topics mean, we will look at the most important word for each of the topics. The `print_topics` function we use below provides a nice formatting for these features.

```
# for each topic (a row in the components_), sort the features (ascending).
# Invert rows with [:, ::-1] to make sorting descending
sorting = np.argsort(lda.components_, axis=1)[:, ::-1]
# get the feature names from the vectorizer:
feature_names = np.array(vect.get_feature_names())
```

```
# print out the 10 topics:
mglearn.tools.print_topics(topics=range(10), feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=5, n_words=10)
```

```
topic 0       topic 1       topic 2       topic 3       topic 4

--------      --------      --------      --------      --------

between       war           funny         show          didn

young         world         worst         series        saw
```

```
family        us            comedy        episode       am
real          our           thing         tv            thought
performance   american      guy           episodes      years
beautiful     documentary   re            shows         book
work          history       stupid        season        watched
each          new           actually      new           now
both          own           nothing       television    dvd
director      point         want          years         got



topic 5       topic 6       topic 7       topic 8       topic 9
--------      --------      --------      --------      --------
horror        kids          cast          performance   house
action        action        role          role          woman
effects       animation     john          john          gets
budget        game          version       actor         killer
nothing       fun           novel         oscar         girl
original      disney        both          cast          wife
director      children      director      plays         horror
minutes       10            played        jack          young
pretty        kid           performance   joe           goes
doesn         old           mr            performances  around
```

Judging from the important words, topic 1 seems to be about historical and war movies, topic 2 might be about bad comedy, topic 3 might be about tv series, topic 4 seems to caputre some very common words, topic 6 seem to capture children's movies, and topic 8 seems to capture award-related reviews. Using only ten topics, each of the topics needs to be very broad, so that they can together cover all the different kinds of reviews in our dataset.

Next, we will learn another model, this time with 100 topics. Using more topics makes the analysis much harder, but makes it more likely that topics can specialize to intesting subsets of the data.

```
lda100 = LatentDirichletAllocation(n_topics=100, learning_method="batch", max_iter=25, random_stat
document_topics100 = lda100.fit_transform(X)
```

Looking at all 100 topics would be a bit overwhelming, so we selected some interesting and representative topics.

```
topics = np.array([7, 16, 24, 25, 28, 36, 37, 45, 51, 53, 54, 63, 89, 97])
```

```
sorting = np.argsort(lda100.components_, axis=1)[:, ::-1]
feature_names = np.array(vect.get_feature_names())
mglearn.tools.print_topics(topics=topics, feature_names=feature_names, sorting=sorting,
                           topics_per_chunk=7, n_words=20)
```

```
topic 7       topic 16      topic 24      topic 25      topic 28      topic 36      topic 37

--------      --------      --------      --------      --------      --------      --------

thriller      worst         german        car           beautiful     performance   excellent

suspense      awful         hitler        gets          young         role          highly

horror        boring        nazi          guy           old           actor         amazing

atmosphere    horrible      midnight      around        romantic      cast          wonderful

mystery       stupid        joe           down          between       play          truly

house         thing         germany       kill          romance       actors        superb

director      terrible      years         goes          wonderful     performances  actors

quite         script        history       killed        heart         played        brilliant

bit           nothing       new           going         feel          supporting    recommend

de            worse         modesty       house         year          director      quite

performances  waste         cowboy        away          each          oscar         performance

dark          pretty        jewish        head          french        roles         performances

twist         minutes       past          take          sweet         actress       perfect

hitchcock     didn          kirk          another       boy           excellent     drama

tension       actors        young         getting       loved         screen        without

interesting   actually      spanish       doesn         girl          plays         beautiful
```

```
mysterious  re          enterprise  now       relationship  award    human
murder      supposed    von         night     saw           work     moving
ending      mean        nazis       right     both          playing  world
creepy      want        spock       woman     simple        gives    recommended
```

```
topic 45    topic 51    topic 53    topic 54    topic 63    topic 89    topic 97
--------    --------    --------    --------    --------    --------    --------
music       earth       scott       money       funny       dead        didn
song        space       gary        budget      comedy      zombie      thought
songs       planet      streisand   actors      laugh       gore        wasn
rock        superman    star        low         jokes       zombies     ending
band        alien       hart        worst       humor       blood       minutes
soundtrack  world       lundgren    waste       hilarious   horror      got
singing     evil        dolph       10          laughs      flesh       felt
voice       humans      career      give        fun         minutes     part
singer      aliens      sabrina     want        re          body        going
sing        human       role        nothing     funniest    living      seemed
musical     creatures   temple      terrible    laughing    eating      bit
roll        miike       phantom     crap        joke        flick       found
fan         monsters    judy        must        few         budget      though
metal       apes        melissa     reviews     moments     head        nothing
concert     clark       zorro       imdb        guy         gory        lot
playing     burton      gets        director    unfunny     evil        saw
hear        tim         barbra      thing       times       shot        long
fans        outer       cast        believe     laughed     low         interesting
```

```
prince        men           short         am            comedies      fulci         few

especially    moon          serial        actually      isn           re            half
```

The topics we extracted this time seem to be more specific, though many are hard to interpret. Topic 7 seems to be about horror movies and thrillers, topics 16 and 54 see, to capture bad reviews, while topic 63 mostly seems to be capturing positive reviews of comedies.

If you want to make further inferences using the topics that were discovered, it is good to confirm the intuition we gained from looking the highest ranking words for each topic, by looking at the documents that are assigned to these topics. For example, topic 45 seems to be about music. Let's check which kind of reviews are assigned this topic:

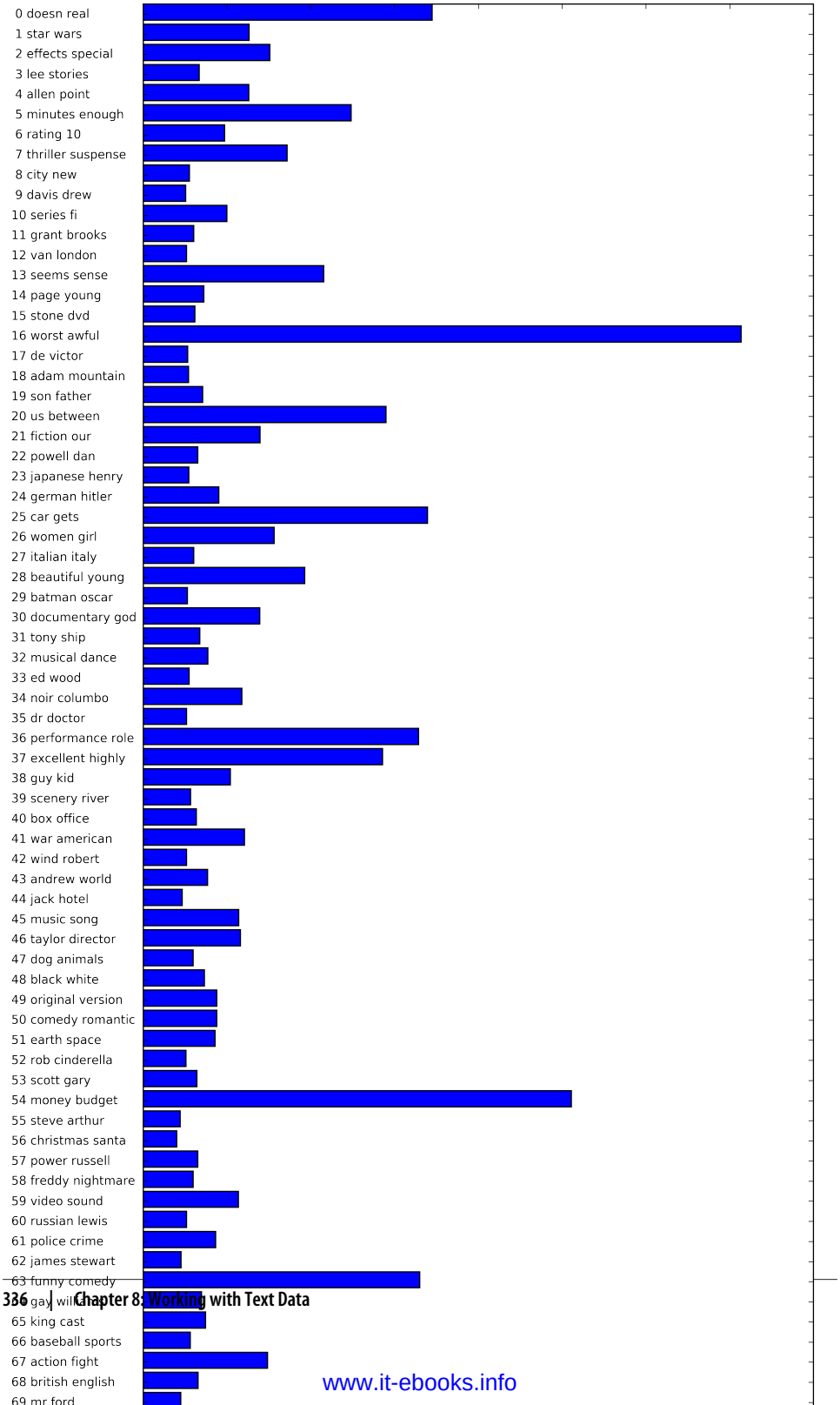
```
# sort by weight of "music" topic 45
music = np.argsort(document_topics100[:, 45])[::-1]
# print the five documents where the topic is most important
for i in music[:10]:
    # pshow first two sentences
    print(b".".join(text_train[i].split(b".")[:2]) + b".\n")
```

```
b'I love this movie and never get tired of watching. The music in it is great.\n'

b"I enjoyed Still Crazy more than any film I have seen in years. A successful band from the 70's c

b'Hollywood Hotel was the last movie musical that Busby Berkeley directed for Warner Bros. His dir

b"What happens to washed up rock-n-roll stars in the late 1990's? They launch a comeback / reunion

b'As a big-time Prince fan of the last three to four years, I really can\'t believe I\'ve only jus

b"This film is worth seeing alone for Jared Harris' outstanding portrayal of John Lennon. It doesn

b"The funky, yet strictly second-tier British glam-rock band Strange Fruit breaks up at the end of

b"I just finished reading a book on Anita Loos' work and the photo in TCM Magazine of MacDonald in

b'I love this movie!!! Purple Rain came out the year I was born and it has had my heart since I ca

b"This movie is sort of a Carrie meets Heavy Metal. It's about a highschool guy who gets picked on
```

As we can see, this topic covers a wide variety of music-centered reviews, from musicals, to bigraphic movies, to some hard-to-specify genre in the last review. Another interesting way to inspect the topics is to see how much weight each topic gets overall, by summing the `document_topics` over all reviews. We name each topic by the two most comen words:

```
plt.figure(figsize=(10, 30))
plt.barh(np.arange(100), np.sum(document_topics100, axis=0))
topic_names = ["{:>2} ".format(i) + " ".join(words) for i, words in enumerate(feature_names[sortin
```

```
plt.yticks(np.arange(100) + .5, topic_names, ha="left");
ax = plt.gca()
ax.invert_yaxis()
yax = ax.get_yaxis()
yax.set_tick_params(pad=110)
```



The most important topics are 97, which seems to consist mostly of stop-words, possibly with a slight negative direction, topic 16, which is clearly about bad reviews, followed by some genre-specific and 36 and 37, both of which seem to contain laudatory words.

It seems like LDA mostly discovered two kind of topics: genre-specific and rating-specific, in addition to several more unspecific topics. This seems like an interesting discovery, as most reviews are made of some movie-specific comments, and some comments that justify or emphasize the rating.

Topic models like LDA are an interesting methods to understand large text corpora in the absence of labels --- or, as here, even if labels are available. The LDA algorithms is randomized, though, and changing the `random_state` parameter can lead to quite different outcomes. While identifying topics can be helpful, any conclusions you draw from an unsupervised model should be taken with a grain of salt, and we recommend verifying your intution by looking at the documents in a specific topic.

Summary and Outlook

In this chapter we talked about the basics of processing text, also known as *natural language processing* (NLP) with an example application classifying movie reviews. The tools discussed here should serve as a great starting point when trying to process text data. In particular for text classification such as spam and fraud detection or sentiment analysis, bag of word representations provide a simple and powerful solution. As so often in machine learning, the representation of the data is key in NLP applications, and inspecting the tokens and n-grams that are extracted can give powerful insights into the modeling process. In text processing applications, it is often possible to introspect models in a meaningful way, as we saw above, both for supervised and unsupervised tasks. You should take full advantage of this ability when using NLP based methods in practice.

NLP and text processing is a large research field, and discussing the details of advanced methods is far beyond the scope of this book. If you want to learn more about text processing and natural language processing, we recommend the O'Reilly book Natural Language Processing with Python by Bird, Klein and Loper, which provides an overview of NLP together with an introduction to the `nltk` python package for NLP. Another great and more conceptual book is the standard reference Introduction to information retrieval by Manning, Raghavan and Schütze, which describes fundamental algorithms in information retrieval, NLP and machine learning. Both books have online versions that can be accessed free of charge.

As we discussed above, the classes `CountVectorizer` and `TfidfVectorizer` only implement relatively simple text processing methods. For more advanced text processing methods, we recommend the Python packages `SpaCy`, a relatively new, but

very efficient and well-designed package, `nltk`, a very well-established and complete, but somewhat dated library, and `gensim`, an NLP package with an emphasis on topic modelling.

There have been several very exciting new developments in text processing in recent years, which are outside of the scope of this book and relate to neural networks. The first is the use of continuous vector representations, also known as word vectors or distributed word representations, as implemented in the `word2vec` library. The original paper "Distributed representations of words and phrases and their compositionality" by Mikolov, Suskever, Chen, Corrado and Dean is a great introduction to the subject. Both `SpaCy` and `gensim` provide functionality for the techniques discussed in this paper and its follow-ups.

Another direction in NLP that has picked up momentum in recent years are *recurrent neural networks* (RNNs) for text processing. RNNs are a particularly powerful type of neural network that can produce output that is again text, in contrast to classification models that can only assign class labels. The ability to produce text as output makes RNNs well-suited for automatic translation and summarization. An introduction to the topic can be found in the relatively technical paper "Sequence to Sequence Learning

with Neural Networks" by Suskever, Vinyals and Le. A more practical tutorial using `tensorflow` framework can found on the tensorflow website [footnote https://www.tensorflow.org/versions/r0.8/tutorials/seq2seq/index.html].