Utilizing Python for Machine Learning Applications

Introduction to Machine Learning

In many ways, machine learning is the primary means by which data science manifests itself to the broader world. Machine learning is where these computational and algorithmic skills of data science meet the statistical thinking of data science.

What Is Machine Learning?

Fundamentally, machine learning involves building mathematical models to help understand data. "Learning" comes into play when we give these models *tunable parameters* that can be adapted to observed data; in this way the program can be considered to be "learning" from the data. Once these models have been fit to previously seen data, they can be used to predict and understand aspects of newly observed data.

Categories of Machine Learning

At the most fundamental level, machine learning can be categorized into two main types: supervised learning and unsupervised learning.

Supervised learning involves somehow modeling the relationship between measured features of data and some label associated with the data; once this model is determined, it can be used to apply labels to new, unknown data. This is further subdivided into *classification* tasks and *regression* tasks: in classification, the labels are discrete categories, while in regression, the labels are continuous quantities. We will see examples of both types of supervised learning in the following section.

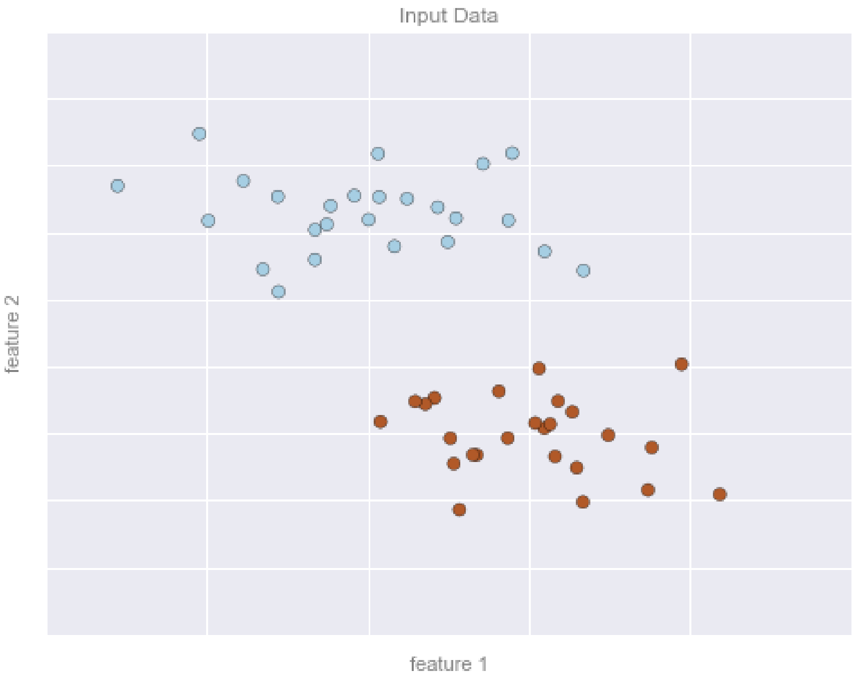
Unsupervised learning involves modeling the features of a dataset without reference to any label, and is often described as "letting the dataset speak for itself." These models include tasks such as *clustering* and *dimensionality reduction.* Clustering algorithms identify distinct groups of data, while dimensionality reduction algorithms search for more succinct representations of the data. We will see examples of both types of unsupervised learning in the following section.

Qualitative Examples of Machine Learning Applications

To make these ideas more concrete, let's take a look at a few very simple examples of a machine learning task. These examples are meant to give an intuitive, non-quantitative overview of the types of machine learning tasks we will be looking at in this lecture. In later sections, we will go into more depth regarding the particular models and how they are used.

Classification: Predicting discrete labels

We will first take a look at a simple *classification* task, in which you are given a set of labeled points and want to use these to classify some unlabeled points.Imagine that we have the data shown in this figure:



Here we have two-dimensional data: that is, we have two *features* for each point, represented by the *(x,y)* positions of the points on the plane. In addition, we have one of two *class labels* for each point, here represented by the colors of the points. From these features and labels, we would like to create a model that will let us decide whether a new point should be labeled "blue" or "red."

There are a number of possible models for such a classification task, but here we will use an extremely simple one. We will make the assumption that the two groups can be separated by drawing a straight line through the plane between them, such that points on each side of the line fall in the same group. Here the *model* is a quantitative version of the statement "a straight line separates the classes", while the *model parameters* are the particular numbers describing the location and orientation of that line for our data. The optimal values for these model parameters are learned from the data (this is the "learning" in machine learning), which is often called *training the model*.

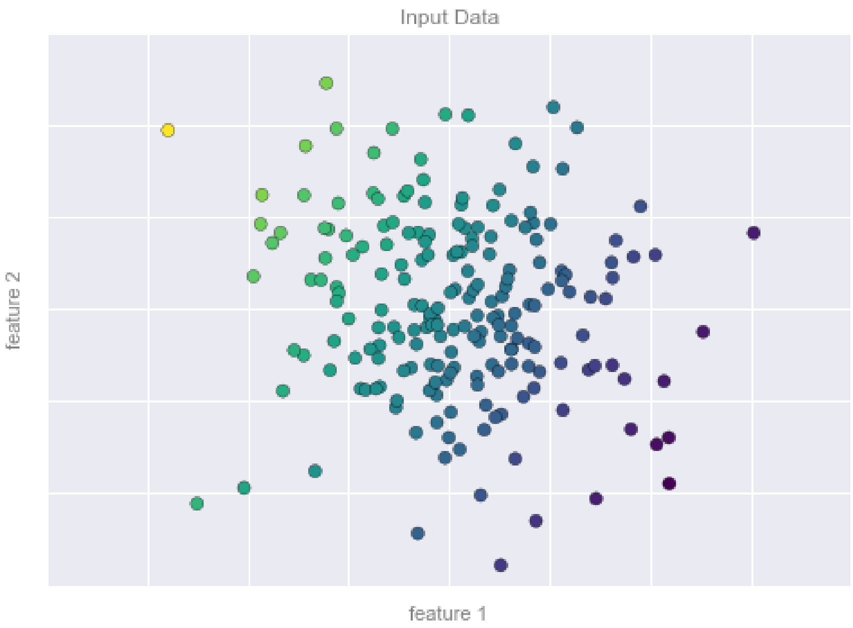
Now that this model has been trained, it can be generalized to new, unlabeled data. In other words, we can take a new set of data, draw this model line through it, and assign labels to the new points based on this model. This stage is usually called *prediction*.

This is the basic idea of a classification task in machine learning, where "classification" indicates that the data has discrete class labels. At first glance this may look fairly trivial: it would be relatively easy to simply look at this data and draw such a discriminatory line to accomplish this classification. A benefit of the machine learning approach, however, is that it can generalize to much larger datasets in many more dimensions.

For the training set, these labels might be determined by individual inspection of a small representative sample of emails; for the remaining emails, the label would be determined using the model. For a suitably trained classification algorithm with enough well-constructed features (typically thousands or millions of words or phrases), this type of approach can be very effective.

Regression: Predicting continuous labels

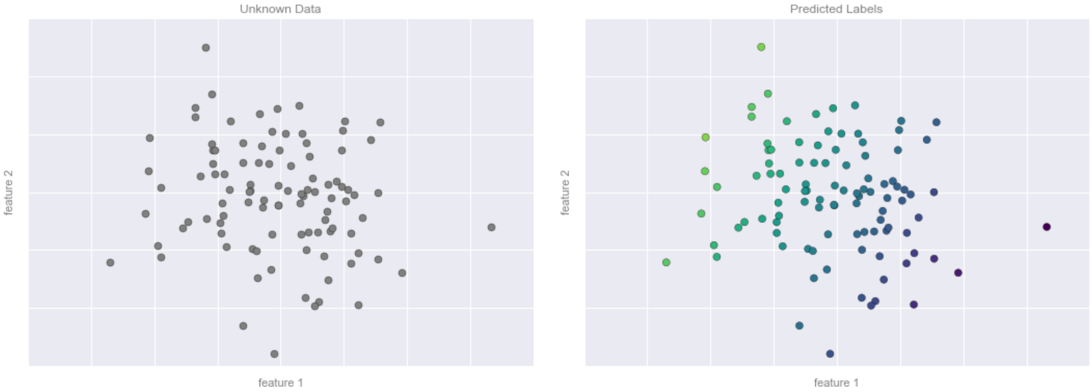
In contrast with the discrete labels of a classification algorithm, we will next look at a simple *regression* task in which the labels are continuous quantities.Consider the data shown in the following figure, which consists of a set of points each with a continuous label:



As with the classification example, we have two-dimensional data: that is, there are two features describing each data point. The color of each point represents the continuous label for that point.

There are a number of possible regression models we might use for this type of data, but here we will use a simple linear regression to predict the points. This simple linear regression model assumes that if we treat the label as a third spatial dimension, we can fit a plane to the data. This is a higher-level generalization of the well-known problem of fitting a line to data with two coordinates.

We need to predict labels for new points.



As with the classification example, this may seem rather trivial in a low number of dimensions. But the power of these methods is that they can be straightforwardly applied and evaluated in the case of data with many, many features.

For example, this is similar to the task of computing the distance to galaxies observed through a telescope—in this case, we might use the following features and labels:

- *feature 1*, *feature 2*, etc. → brightness of each galaxy at one of several wave lengths or colors
- *label* → distance or redshift of the galaxy

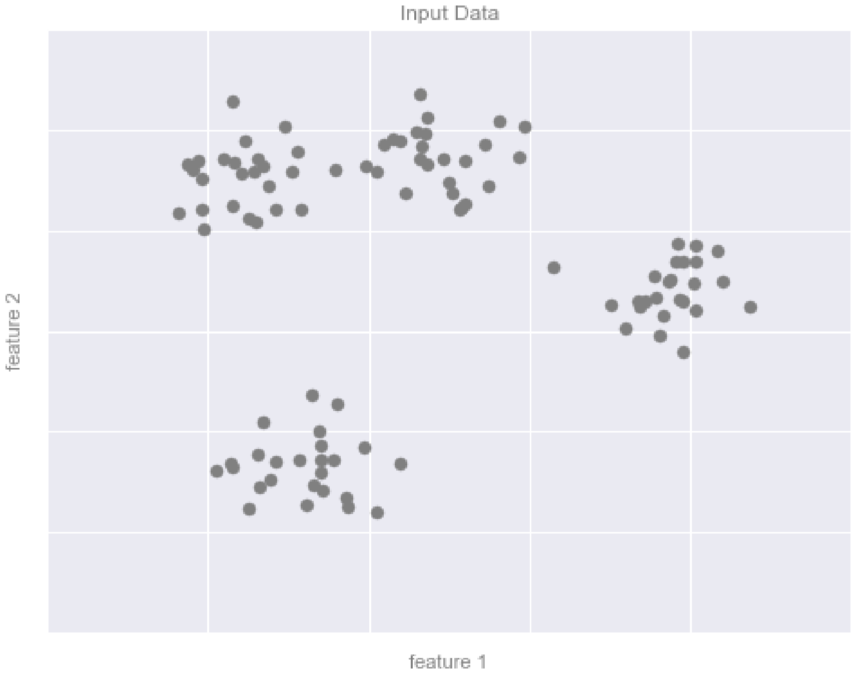
The distances for a small number of these galaxies might be determined through an independent set of (typically more expensive) observations. Distances to remaining galaxies could then be estimated using a suitable regression model, without the need to employ the more expensive observation across the entire set. In astronomy circles, this is known as the "photometric redshift" problem.

Some important regression algorithms that we will discuss are linear regression, support vector machines, and random forest regression.

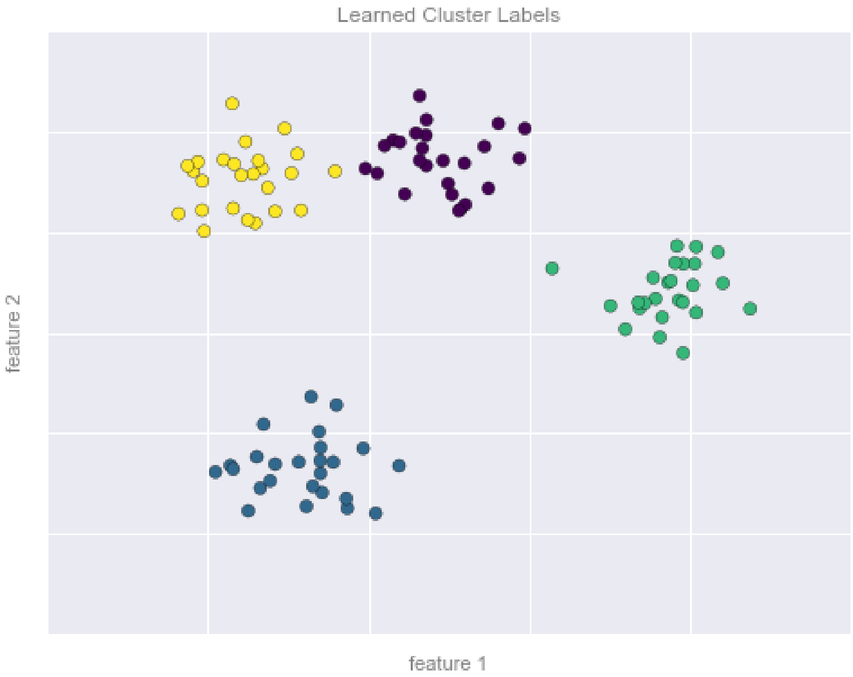
Clustering: Inferring labels on unlabeled data

The classification and regression illustrations we just looked at are examples of supervised learning algorithms, in which we are trying to build a model that will predict labels for new data. Unsupervised learning involves models that describe data without reference to any known labels.

One common case of unsupervised learning is "clustering," in which data is automatically assigned to some number of discrete groups. For example, we might have some two-dimensional data like that shown in the following figure:



By eye, it is clear that each of these points is part of a distinct group. Given this input, a clustering model will use the intrinsic structure of the data to determine which points are related. Using the very fast and intuitive *k*-means algorithm, we find the clusters shown in the following figure:



Introducing Scikit-Learn package

There are several Python libraries which provide solid implementations of a range of machine learning algorithms. One of the best known is Scikit-Learn (http://scikit-learn.org), a package that provides efficient versions of a large number of common algorithms. Scikit-Learn is characterized by a clean, uniform, and streamlined API, as well as by very useful and complete online documentation. A benefit of this uniformity is that once you understand the basic use and syntax of Scikit-Learn for one type of model, switching to a new model or algorithm is very straightforward.

This section provides an overview of the Scikit-Learn API; a solid understanding of these API elements will form the foundation for understanding the deeper practical discussion of machine learning algorithms and approaches.

Data Representation in Scikit-Learn

Machine learning is about creating models from data: for that reason, we'll start by discussing how data can be represented in order to be understood by the computer. The best way to think about data within Scikit-Learn is in terms of tables of data.

Data as table

A basic table is a two-dimensional grid of data, in which the rows represent individual elements of the dataset, and the columns represent quantities related to each of these elements. For example, consider the Iris dataset (https://en.wikipedia.org/wiki/Iris_flower_data_set), famously analyzed by Ronald Fisher in 1936. We can download this dataset in the form of a Pandas `DataFrame` using the seaborn (http://seaborn.pydata.org/) library:

In [3]:

```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

Out[3]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Here each row of the data refers to a single observed flower, and the number of rows is the total number of flowers in the dataset. In general, we will refer to the rows of the matrix as *samples*, and the number of rows as `n_samples` .

Likewise, each column of the data refers to a particular quantitative piece of information that describes each sample. In general, we will refer to the columns of the matrix as *features*, and the number of columns as `n_features` .

Features matrix

This table layout makes clear that the information can be thought of as a two-dimensional numerical array or matrix, which we will call the *features matrix*. By convention, this features matrix is often stored in a variable named `X` . The features matrix is assumed to be two-dimensional, with shape `[n_samples, n_features]` , and is most often contained in a NumPy array or a Pandas `DataFrame` , though some Scikit-Learn models also accept SciPy sparse matrices.

The samples (i.e., rows) always refer to the individual objects described by the dataset. For example, the sample might be a flower, a person, a document, an image, a sound file, a video, an astronomical object, or anything else you can describe with a set of quantitative measurements.

The features (i.e., columns) always refer to the distinct observations that describe each sample in a quantitative manner. Features are generally real-valued, but may be Boolean or discrete-valued in some cases.

Target array

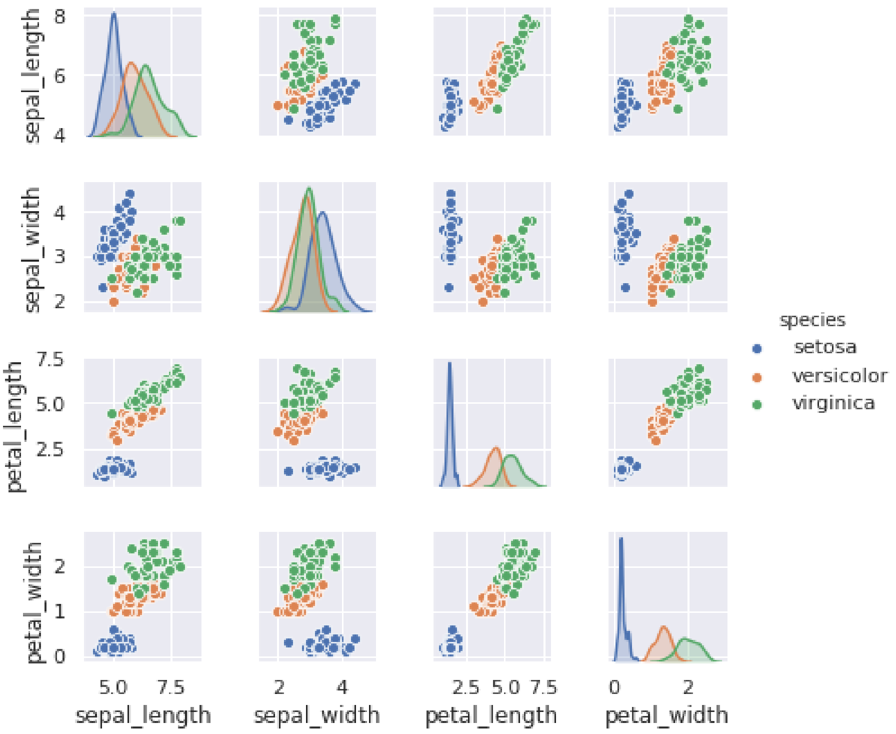
In addition to the feature matrix `X` , we also generally work with a *label* or *target* array, which by convention we will usually call `y` . The target array is usually one dimensional, with length `n_samples` , and is generally contained in a NumPy array or Pandas `Series` . The target array may have continuous numerical values, or discrete classes/labels. While some Scikit-Learn estimators do handle multiple target values in the form of a two-dimensional, `[n_samples, n_targets]` target array, we will primarily be working with the common case of a one-dimensional target array.

Often one point of confusion is how the target array differs from the other features columns. The distinguishing feature of the target array is that it is usually the quantity we want to *predict from the data*: in statistical terms, it is the dependent variable. For example, in the preceding data we may wish to construct a model that can predict the species of flower based on the other measurements; in this case, the `species` column would be considered the target array.

With this target array in mind, we can use Seaborn to conveniently visualize the data:

In [5]:

```
%matplotlib inline
import seaborn as sns; sns.set()
sns.pairplot(iris, hue='species', height=1.5);
```



For use in Scikit-Learn, we will extract the features matrix and target array from the `DataFrame` , which we can do using some of the Pandas `DataFrame` operations discussed before:

In [35]:

```
X_iris = iris.drop('species', axis=1)
X_iris.shape
```

Out[35]:

```
(150, 4)
```

In [36]:

```
y_iris = iris['species']
y_iris.shape
```

Out[36]:

```
(150,)
```

With this data properly formatted, we can move on to consider the *estimator* API of Scikit-Learn:

Scikit-Learn's Estimator API

The Scikit-Learn API is designed with the following guiding principles in mind, as outlined in the Scikit-Learn API paper (http://arxiv.org/abs/1309.0238):

- *Consistency*: All objects share a common interface drawn from a limited set of methods, with consistent documentation.
- *Inspection*: All specified parameter values are exposed as public attributes.
- *Limited object hierarchy*: Only algorithms are represented by Python classes; datasets are represented in standard formats (NumPy arrays, Pandas `DataFrame` s, SciPy sparse matrices) and parameter names use standard Python strings.
- *Composition*: Many machine learning tasks can be expressed as sequences of more fundamental algorithms, and Scikit-Learn makes use of this wherever possible.
- *Sensible defaults*: When models require user-specified parameters, the library defines an appropriate default value.

In practice, these principles make Scikit-Learn very easy to use, once the basic principles are understood. Every machine learning algorithm in Scikit-Learn is implemented via the Estimator API, which provides a consistent interface for a wide range of machine learning applications.

Basics of the API

Most commonly, the steps in using the Scikit-Learn estimator API are as follows (we will step through a handful of detailed examples in the sections that follow).

1. Choose a class of model by importing the appropriate estimator class from Scikit-Learn.
2. Choose model hyperparameters by instantiating this class with desired values.
3. Arrange data into a features matrix and target vector following the discussion above.
4. Fit the model to your data by calling the `fit()` method of the model instance.
5. Apply the Model to new data:
    - For supervised learning, often we predict labels for unknown data using the `predict()` method.
    - For unsupervised learning, we often transform or infer properties of the data using the `transform()` or `predict()` method.

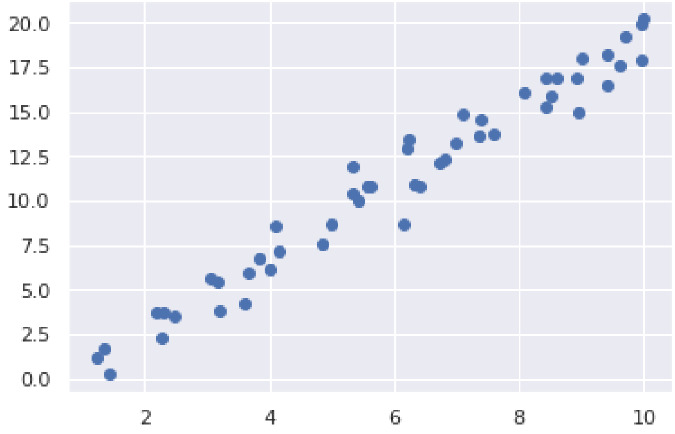
We will now step through several simple examples of applying supervised and unsupervised learning methods.

Supervised learning example: Simple linear regression

As an example of this process, let's consider a simple linear regression—that is, the common case of fitting a line to (x, y) data. We will use the following simple data for our regression example:

In [6]:

```
import matplotlib.pyplot as plt
import numpy as np

rng = np.random.RandomState(24)
x = 10 * rng.rand(50)
y = 2 * x - 1 + rng.randn(50)
plt.scatter(x, y);
```



With this data in place, we can use the recipe outlined earlier. Let's walk through the process:

1. Choose a class of model

In Scikit-Learn, every class of model is represented by a Python class. So, for example, if we would like to compute a simple linear regression model, we can import the linear regression class:

In [7]:

```
from sklearn.linear_model import LinearRegression
```

Note that other more general linear regression models exist as well; you can read more about them in the `sklearn.linear_model` module documentation (http://Scikit-Learn.org/stable/modules/linear_model.html).

2. Choose model hyperparameters

An important point is that *a class of model is not the same as an instance of a model*.

Once we have decided on our model class, there are still some options open to us. Depending on the model class we are working with, we might need to answer one or more questions like the following:

- Would we like to fit for the offset (i.e., *y*-intercept)?
- Would we like the model to be normalized?
- Would we like to preprocess our features to add model flexibility?
- What degree of regularization would we like to use in our model?
- How many model components would we like to use?

These are examples of the important choices that must be made *once the model class is selected*. These choices are often represented as *hyperparameters*, or parameters that must be set before the model is fit to data. In Scikit-Learn, hyperparameters are chosen by passing values at model instantiation. We will explore how you can quantitatively motivate the choice of hyperparameters later.

For our linear regression example, we can instantiate the `LinearRegression` class and specify that we would like to fit the intercept using the `fit_intercept` hyperparameter:

In [18]:
```
model = LinearRegression(fit_intercept=True)
model
```

Out[18]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

Keep in mind that when the model is instantiated, the only action is the storing of these hyperparameter values. In particular, we have not yet applied the model to any data: the Scikit-Learn API makes very clear the distinction between *choice of model* and *application of model to data*.

3. Arrange data into a features matrix and target vector

Previously we detailed the Scikit-Learn data representation, which requires a two-dimensional features matrix and a one-dimensional target array. Here our target variable `y` is already in the correct form (a length- `n_samples` array), but we need to massage the data `x` to make it a matrix of size `[n_samples, n_features]` . In this case, this amounts to a simple reshaping of the one-dimensional array:

In [19]:
```
X = x[:, np.newaxis]
X.shape
```

Out[19]:
```
(50, 1)
```

4. Fit the model to your data

Now it is time to apply our model to data. This can be done with the `fit()` method of the model:

In [20]:
```
model.fit(X, y)
```

Out[20]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

This `fit()` command causes a number of model-dependent internal computations to take place, and the results of these computations are stored in model-specific attributes that the user can explore. In Scikit-Learn, by convention all model parameters that were learned during the `fit()` process have trailing underscores; for example in this linear model, we have the following:

```
In [21]: model.coef_

Out[21]: array([2.08215472])
```

```
In [22]: model.intercept_

Out[22]: -1.5098663849702465
```

These two parameters represent the slope and intercept of the simple linear fit to the data. Comparing to the data definition, we see that they are very close to the input slope of 2 and intercept of -1.

One question that frequently comes up regards the uncertainty in such internal model parameters. In general, Scikit-Learn does not provide tools to draw conclusions from internal model parameters themselves: interpreting model parameters is much more a *statistical modeling* question than a *machine learning* question. Machine learning rather focuses on what the model *predicts*. If you would like to dive into the meaning of fit parameters within the model, other tools are available, including the Statsmodels Python package (http://statsmodels.sourceforge.net/).

5. Predict labels for unknown data

Once the model is trained, the main task of supervised machine learning is to evaluate it based on what it says about new data that was not part of the training set. In Scikit-Learn, this can be done using the `predict()` method. For the sake of this example, our "new data" will be a grid of _x* values, and we will ask what *y_ values the model predicts:

```
In [27]: xfit = np.linspace(-1, 11)
         xfit

Out[27]: array([-1.        , -0.75510204, -0.51020408, -0.26530612, -0.02040816,
                 0.2244898 ,  0.46938776,  0.71428571,  0.95918367,  1.20408163,
                 1.44897959,  1.69387755,  1.93877551,  2.18367347,  2.42857143,
                 2.67346939,  2.91836735,  3.16326531,  3.40816327,  3.65306122,
                 3.89795918,  4.14285714,  4.3877551 ,  4.63265306,  4.87755102,
                 5.12244898,  5.36734694,  5.6122449 ,  5.85714286,  6.10204082,
                 6.34693878,  6.59183673,  6.83673469,  7.08163265,  7.32653061,
                 7.57142857,  7.81632653,  8.06122449,  8.30612245,  8.55102041,
                 8.79591837,  9.04081633,  9.28571429,  9.53061224,  9.7755102 ,
                10.02040816, 10.26530612, 10.51020408, 10.75510204, 11.        ])
```

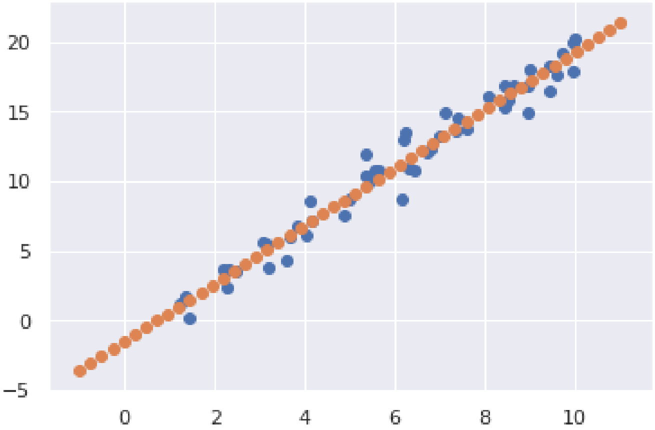
As before, we need to coerce these *x* values into a `[n_samples, n_features]` features matrix, after which we can feed it to the model:

```
In [25]: Xfit = xfit[:, np.newaxis]
         yfit = model.predict(Xfit)
```

Finally, let's visualize the results by plotting first the raw data, and then this model fit:

In [30]:

```
plt.scatter(x, y)
plt.scatter(xfit, yfit);
```



Typically the efficacy of the model is evaluated by comparing its results to some known baseline, as we will see in the next example

Supervised learning example: Iris classification

Let's take a look at another example of this process, using the Iris dataset we discussed earlier. Our question will be this: given a model trained on a portion of the Iris data, how well can we predict the remaining labels?

For this task, we will use an extremely simple generative model known as Gaussian naive Bayes, which proceeds by assuming each class is drawn from an axis-aligned Gaussian distribution. Because it is so fast and has no hyperparameters to choose, Gaussian naive Bayes is often a good model to use as a baseline classification, before exploring whether improvements can be found through more sophisticated models.

We would like to evaluate the model on data it has not seen before, and so we will split the data into a *training set* and a *testing set*. This could be done by hand, but it is more convenient to use the `train_test_split` utility function:

In [33]:

```
import sklearn
sklearn.__version__
```

Out[33]:

```
'0.20.2'
```

In [37]:

```
from sklearn.model_selection  import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,
                                                random_state=1)
```

With the data arranged, we can follow our recipe to predict the labels:

In [38]:

```
from sklearn.naive_bayes import GaussianNB # 1. choose model class
model = GaussianNB()                      # 2. instantiate model
model.fit(Xtrain, ytrain)                 # 3. fit model to data
y_model = model.predict(Xtest)            # 4. predict on new data
```

Finally, we can use the `accuracy_score` utility to see the fraction of predicted labels that match their true value:

In [39]:

```
from sklearn.metrics import accuracy_score
accuracy_score(ytest, y_model)
```

Out[39]:

```
0.9736842105263158
```

With an accuracy topping 97%, we see that even this very naive classification algorithm is effective for this particular dataset!

Unsupervised learning: Iris clustering

Let's next look at applying clustering to the Iris data. A clustering algorithm attempts to find distinct groups of data without reference to any labels. Here we will use a powerful clustering method called a Gaussian mixture model (GMM). A GMM attempts to model the data as a collection of Gaussian blobs.

We can fit the Gaussian mixture model as follows:

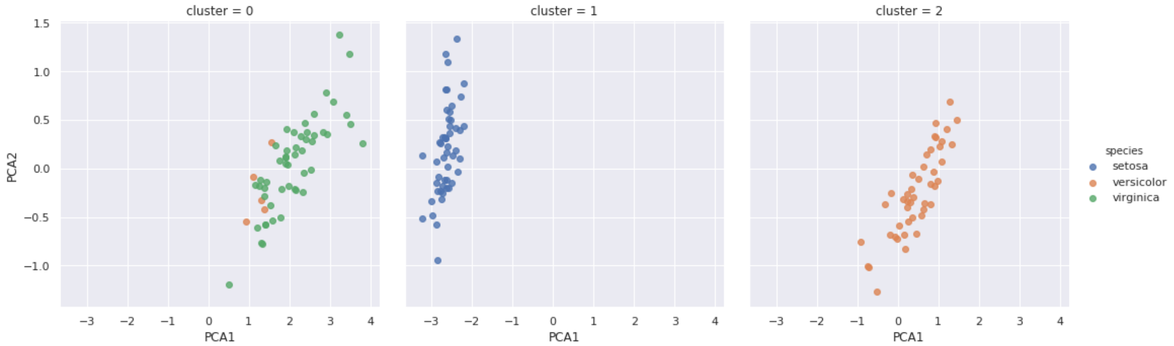
In [43]:

```
from sklearn.mixture import GaussianMixture      # 1. Choose the model class
model = GaussianMixture(n_components=3,
            covariance_type='full')  # 2. Instantiate the model with hyperparameters
model.fit(X_iris)                    # 3. Fit to data. Notice y is not specified!
y_gmm = model.predict(X_iris)        # 4. Determine cluster labels
```

As before, we will add the cluster label to the Iris `DataFrame` and use Seaborn to plot the results:

In [44]:

```
iris['cluster'] = y_gmm
sns.lmplot("PCA1", "PCA2", data=iris, hue='species',
           col='cluster', fit_reg=False);
```



By splitting the data by cluster number, we see exactly how well the GMM algorithm has recovered the underlying label: the *setosa* species is separated perfectly within cluster 1, while there remains a small amount of mixing between *versicolor* and *virginica*. This means that even without an expert to tell us the species labels of the individual flowers, the measurements of these flowers are distinct enough that we could *automatically* identify the presence of these different groups of species with a simple clustering algorithm! This sort of algorithm might further give experts in the field clues as to the relationship between the samples they are observing.

Application: Exploring Hand-written Digits

To demonstrate these principles on a more interesting problem, let's consider one piece of the optical character recognition problem: the identification of hand-written digits. In the wild, this problem involves both locating and identifying characters in an image. Here we'll take a shortcut and use Scikit-Learn's set of pre-formatted digits, which is built into the library.

Loading and visualizing the digits data

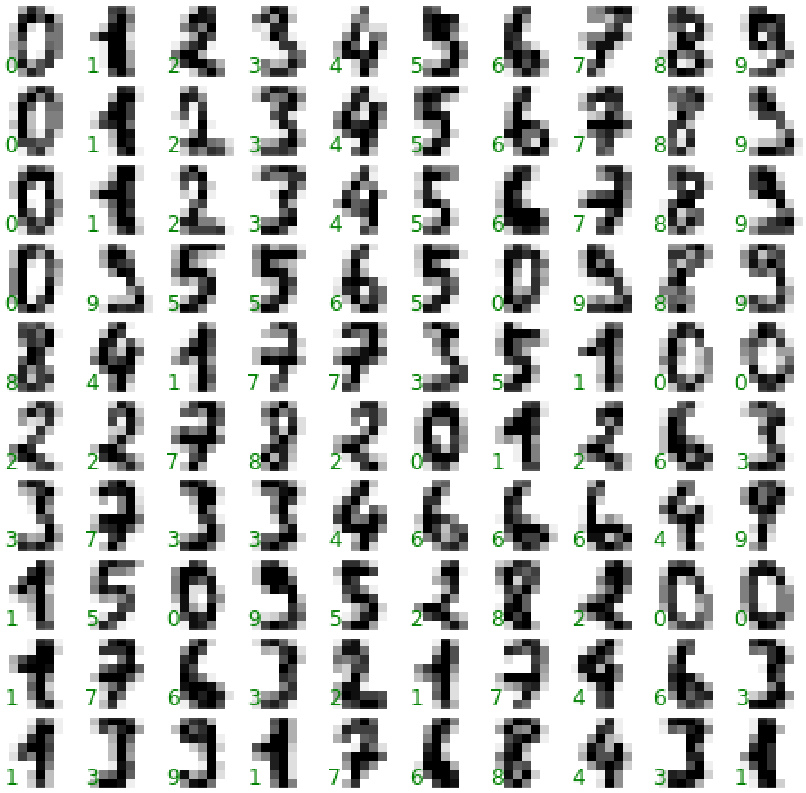
We'll use Scikit-Learn's data access interface and take a look at this data:

```
In [46]: from sklearn.datasets import load_digits
         digits = load_digits()
         digits.images.shape
```

```
Out[46]: (1797, 8, 8)
```

The images data is a three-dimensional array: 1,797 samples each consisting of an 8 × 8 grid of pixels. Let's visualize the first hundred of these:

```
In [47]: import matplotlib.pyplot as plt

         fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                                  subplot_kw={'xticks':[], 'yticks':[]},
                                  gridspec_kw=dict(hspace=0.1, wspace=0.1))

         for i, ax in enumerate(axes.flat):
             ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
             ax.text(0.05, 0.05, str(digits.target[i]),
                     transform=ax.transAxes, color='green')
```



In order to work with this data within Scikit-Learn, we need a two-dimensional, `[n_samples, n_features]` representation. We can accomplish this by treating each pixel in the image as a feature: that is, by flattening out the pixel arrays so that we have a length-64 array of pixel values representing each digit. Additionally, we need the target array, which gives the previously determined label for each digit. These two quantities are built into the digits dataset under the `data` and `target` attributes, respectively:

```
In [48]: X = digits.data
         X.shape
```

```
Out[48]: (1797, 64)
```

```
In  [52]: y = digits.target
          y.shape

Out[52]: (1797,)
```

We see here that there are 1,797 samples and 64 features.

Classification on digits

Let's apply a classification algorithm to the digits. As with the Iris data previously, we will split the data into a training and testing set, and fit a Gaussian naive Bayes model:

```
In  [81]: Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)
```

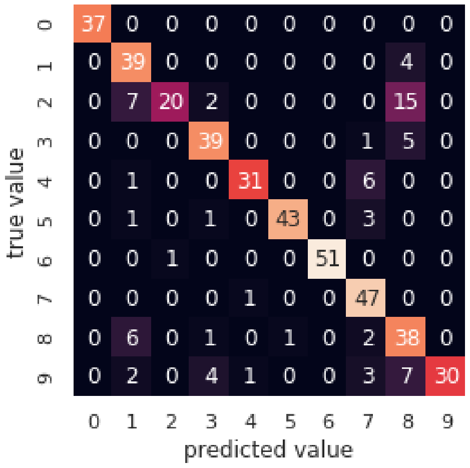
```
In  [82]: from sklearn.naive_bayes import GaussianNB
          model = GaussianNB()
          model.fit(Xtrain, ytrain)
          y_model = model.predict(Xtest)
```

Now that we have predicted our model, we can gauge its accuracy by comparing the true values of the test set to the predictions:

```
In  [83]: from sklearn.metrics import accuracy_score
          accuracy_score(ytest, y_model)

Out[83]: 0.8333333333333334
```

With even this extremely simple model, we find about 80% accuracy for classification of the digits! However, this single number doesn't tell us *where* we've gone wrong—one nice way to do this is to use the *confusion matrix*, which we can compute with Scikit-Learn and plot with Seaborn:

```
In  [84]: from sklearn.metrics import confusion_matrix

          mat = confusion_matrix(ytest, y_model)

          sns.heatmap(mat, square=True, annot=True, cbar=False)
          plt.xlabel('predicted value')
          plt.ylabel('true value');
```

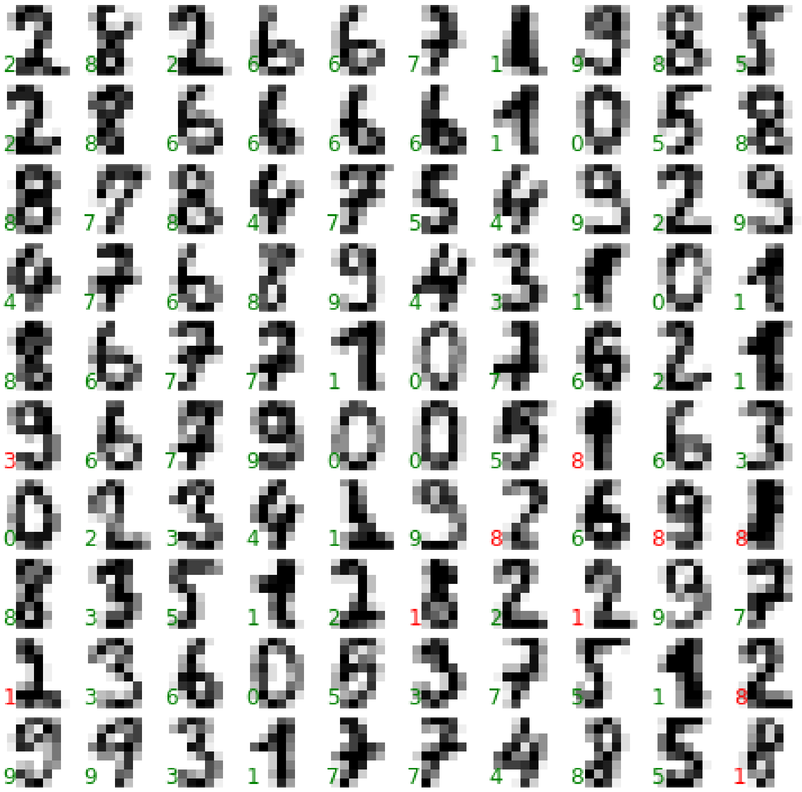


	0	1	2	3	4	5	6	7	8	9
0	37	0	0	0	0	0	0	0	0	0
1	0	39	0	0	0	0	0	0	4	0
2	0	7	20	2	0	0	0	0	15	0
3	0	0	0	39	0	0	0	1	5	0
4	0	1	0	0	31	0	0	6	0	0
5	0	1	0	1	0	43	0	3	0	0
6	0	0	1	0	0	0	51	0	0	0
7	0	0	0	0	1	0	0	47	0	0
8	0	6	0	1	0	1	0	2	38	0
9	0	2	0	4	1	0	0	3	7	30

This shows us where the mis-labeled points tend to be: for example, a large number of twos here are mis-classified as either ones or eights. Another way to gain intuition into the characteristics of the model is to plot the inputs again, with their predicted labels. We'll use green for correct labels, and red for incorrect labels:

In [85]:

```
fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

test_images = Xtest.reshape(-1, 8, 8)

for i, ax in enumerate(axes.flat):
    ax.imshow(test_images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(y_model[i]),
            transform=ax.transAxes,
            color='green' if (ytest[i] == y_model[i]) else 'red')
```



Examining this subset of the data, we can gain insight regarding where the algorithm might be not performing optimally. To go beyond our 80% classification rate, we might move to a more sophisticated algorithm such as support vector machines, random forests or another classification approach.

Summary

In this section we have covered the essential features of the Scikit-Learn data representation, and the estimator API. Regardless of the type of estimator, the same import/instantiate/fit/predict pattern holds. Armed with this information about the estimator API, you can explore the Scikit-Learn documentation and begin trying out various models on your data.