

- When the naive assumptions actually match the data (very rare in practice)
- For very well-separated categories, when model complexity is less important
- For very high-dimensional data, when model complexity is less important

The last two points seem distinct, but they actually are related: as the dimension of a dataset grows, it is much less likely for any two points to be found close together (after all, they must be close in *every single dimension* to be close overall). This means that clusters in high dimensions tend to be more separated, on average, than clusters in low dimensions, assuming the new dimensions actually add information. For this reason, simplistic classifiers like naive Bayes tend to work as well or better than more complicated classifiers as the dimensionality grows: once you have enough data, even a simple model can be very powerful.

In Depth: Linear Regression

Just as naive Bayes (discussed earlier in “In Depth: Naive Bayes Classification” on page 382) is a good starting point for classification tasks, linear regression models are a good starting point for regression tasks. Such models are popular because they can be fit very quickly, and are very interpretable. You are probably familiar with the simplest form of a linear regression model (i.e., fitting a straight line to data), but such models can be extended to model more complicated data behavior.

In this section we will start with a quick intuitive walk-through of the mathematics behind this well-known problem, before moving on to see how linear models can be generalized to account for more complicated patterns in data. We begin with the standard imports:

```
In[1]: %matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

Simple Linear Regression

We will start with the most familiar linear regression, a straight-line fit to data. A straight-line fit is a model of the form $y = ax + b$ where a is commonly known as the *slope*, and b is commonly known as the *intercept*.

Consider the following data, which is scattered about a line with a slope of 2 and an intercept of -5 (Figure 5-42):

```
In[2]: rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = 2 * x - 5 + rng.randn(50)
plt.scatter(x, y);
```

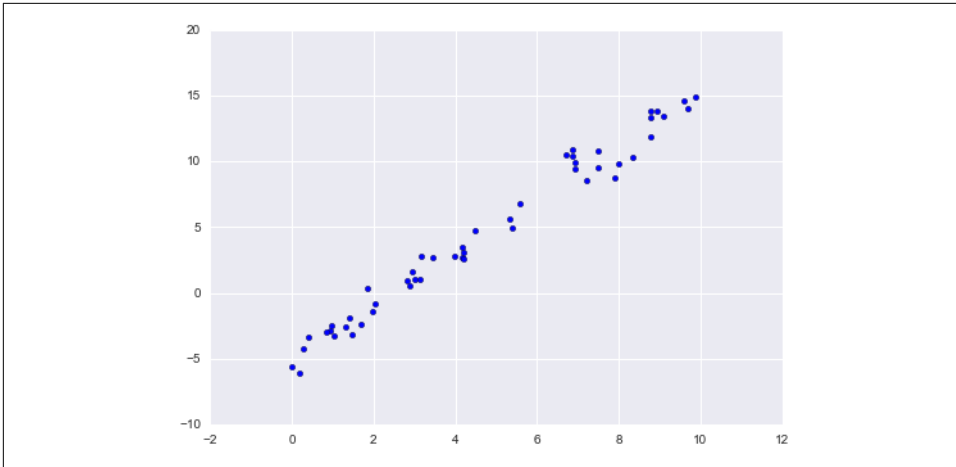


Figure 5-42. Data for linear regression

We can use Scikit-Learn's `LinearRegression` estimator to fit this data and construct the best-fit line (Figure 5-43):

```
In[3]: from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept=True)

model.fit(x[:, np.newaxis], y)

xfit = np.linspace(0, 10, 1000)
yfit = model.predict(xfit[:, np.newaxis])

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

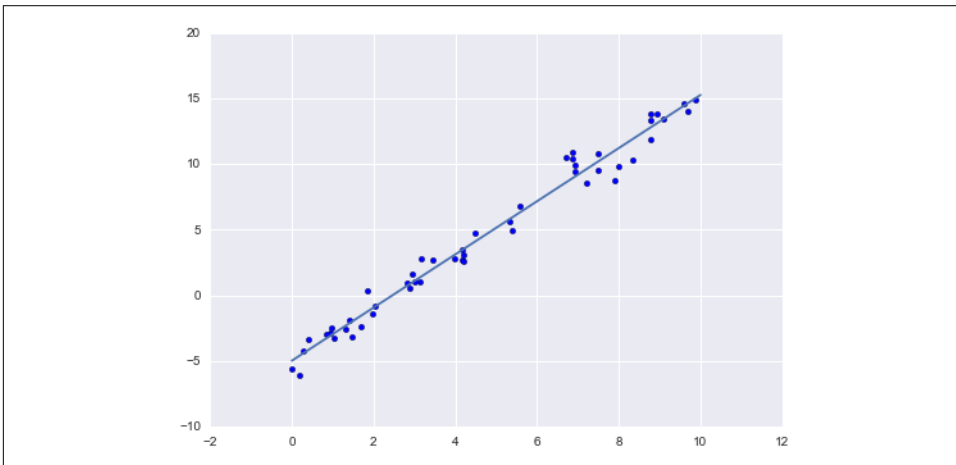


Figure 5-43. A linear regression model

The slope and intercept of the data are contained in the model's fit parameters, which in Scikit-Learn are always marked by a trailing underscore. Here the relevant parameters are `coef_` and `intercept_`:

```
In[4]: print("Model slope: ", model.coef_[0])
       print("Model intercept:", model.intercept_)

Model slope:      2.02720881036
Model intercept: -4.99857708555
```

We see that the results are very close to the inputs, as we might hope.

The `LinearRegression` estimator is much more capable than this, however—in addition to simple straight-line fits, it can also handle multidimensional linear models of the form:

$$y = a_0 + a_1x_1 + a_2x_2 + \cdots$$

where there are multiple x values. Geometrically, this is akin to fitting a plane to points in three dimensions, or fitting a hyper-plane to points in higher dimensions.

The multidimensional nature of such regressions makes them more difficult to visualize, but we can see one of these fits in action by building some example data, using NumPy's matrix multiplication operator:

```
In[5]: rng = np.random.RandomState(1)
       X = 10 * rng.rand(100, 3)
       y = 0.5 + np.dot(X, [1.5, -2., 1.])

       model.fit(X, y)
       print(model.intercept_)
       print(model.coef_)

0.5
[ 1.5 -2.  1. ]
```

Here the y data is constructed from three random x values, and the linear regression recovers the coefficients used to construct the data.

In this way, we can use the single `LinearRegression` estimator to fit lines, planes, or hyperplanes to our data. It still appears that this approach would be limited to strictly linear relationships between variables, but it turns out we can relax this as well.

Basis Function Regression

One trick you can use to adapt linear regression to nonlinear relationships between variables is to transform the data according to *basis functions*. We have seen one version of this before, in the `PolynomialRegression` pipeline used in “[Hyperparameters](#)”