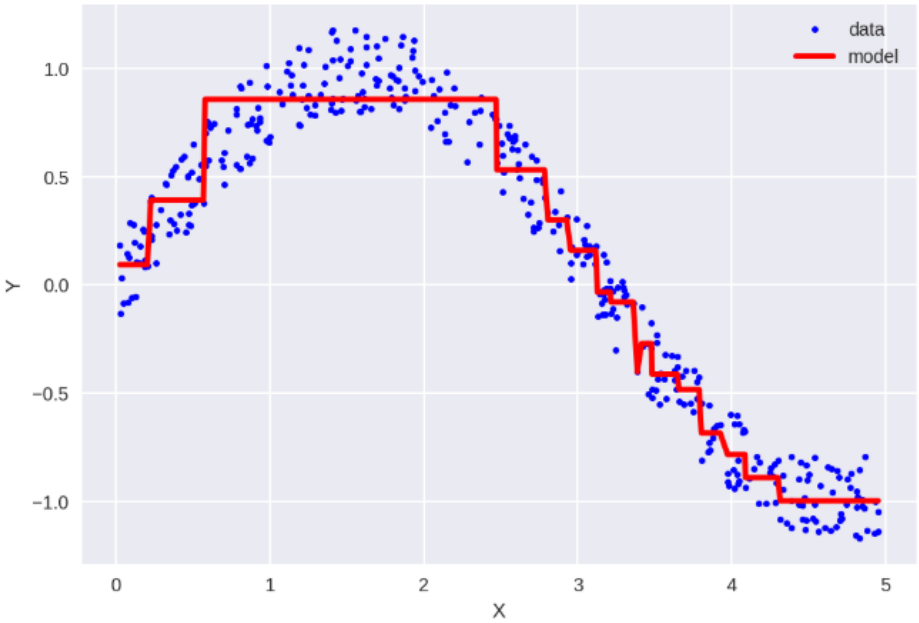
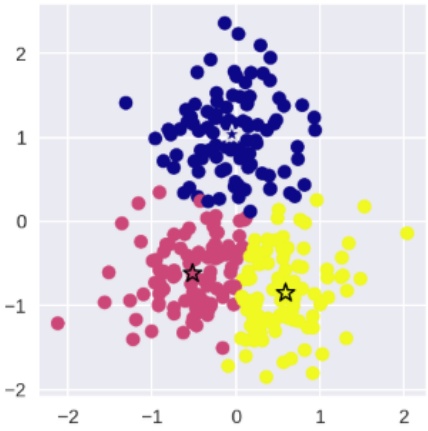
In [1]:
```
%matplotlib inline
import matplotlib
import seaborn as sns
sns.set()
matplotlib.rcParams['figure.dpi'] = 144
```

Introduction to Machine Learning

Machine Learning is a broad collection of tools used to find patterns in existing data and to make predictions about future data.

For example, we can use ML algorithms to approximate an unknown function or to group data into clusters.





Machine Learning is an area of active research and development and has produced many impressive applications:

- Voice and image recognition
- Translation services
- Financial models
- Spam detection
- Self-driving cars

However, Machine Learning is not the answer to every problem, and it has limitations:

- ML models are only as good as the data they train on. If something changes between training and application, or if we're missing important features to start with, then we won't get good results.
- Human intelligence is required to choose an appropriate algorithm (which sometimes includes choosing model parameters) and to evaluate performance. We can run most ML algorithms and get a result with relatively little effort, but it's up to us to make sure the result is meaningful and useful.
- ML algorithms often prioritize performance over explicability. We may train a model that can make good predictions but not be able to explain where those predictions are coming from.

Statistics vs Machine Learning

The exact border between Statistics and Machine Learning is poorly defined, and many statistical methods (e.g. Linear Regression) are considered to be part of the ML toolkit. However, we can identify unique features which are characteristic of each.

Statistical models are more likely to depend on distributional assumptions about data (parametric analysis) and more likely to offer theoretical or closed form solutions to relevant optimization problems. By contrast, Machine Learning models generally emphasize flexibility over theory, and often use incremental optimization algorithms like gradient descent.

Data as a Matrix

In order to do Machine Learning, we need to represent our data mathematically. Our convention will be to represent each data set as a matrix (NumPy array) where:

- **rows** correspond the different observations or data points
- **columns** correspond to different *features* (data attributes)

For example, if we have a (very small) data set which has three features f_0, f_1, f_2 and four observations, we can represent this as a 4x3 matrix.

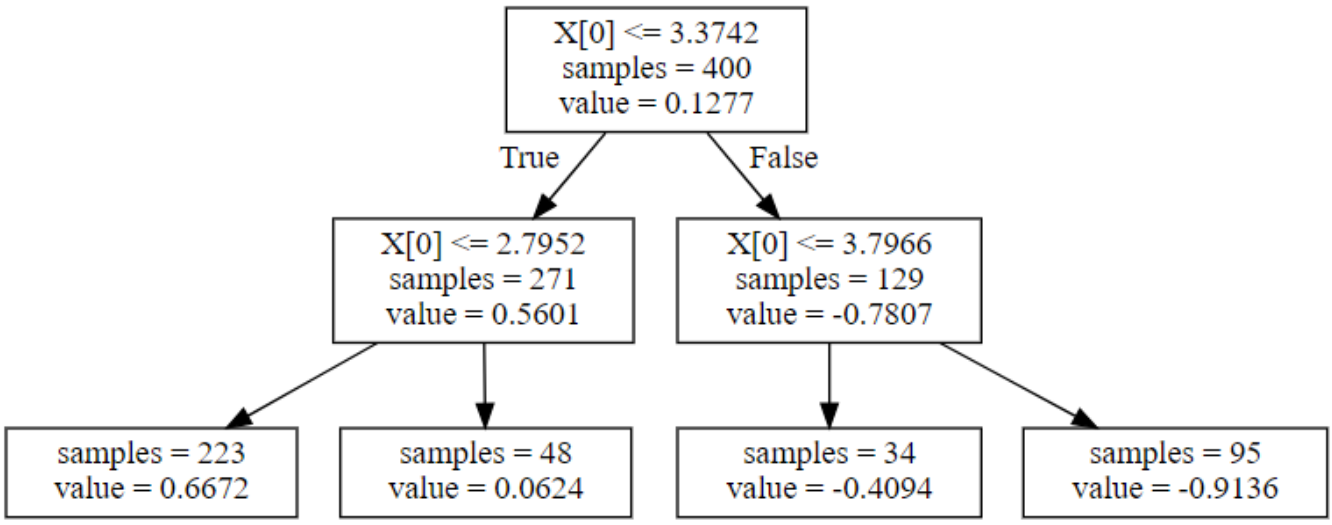
$$X = \begin{pmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \\ x_{30} & x_{31} & x_{32} \end{pmatrix} \begin{matrix} \text{obs 0} \\ \text{obs 1} \\ \text{obs 2} \\ \text{obs 3} \end{matrix}$$

$$f_0 \quad f_1 \quad f_2$$

Models as Functions

Machine Learning **models** are functions which take observations as inputs and return some outputs. Frequently these outputs are predictions, but they might be cluster labels or something else, depending on the application.

For example, a linear model might predict a target variable using the formula $f(x) = 2x + 3$. However, the rule used by a model does not need to have an explicit mathematical representation. In general, it can by any algorithm or procedure. For example, a decision tree model works by dividing data into groups (by iteratively comparing feature values to predetermined split values) and then making a different constant prediction for each group.



Types of Machine Learning

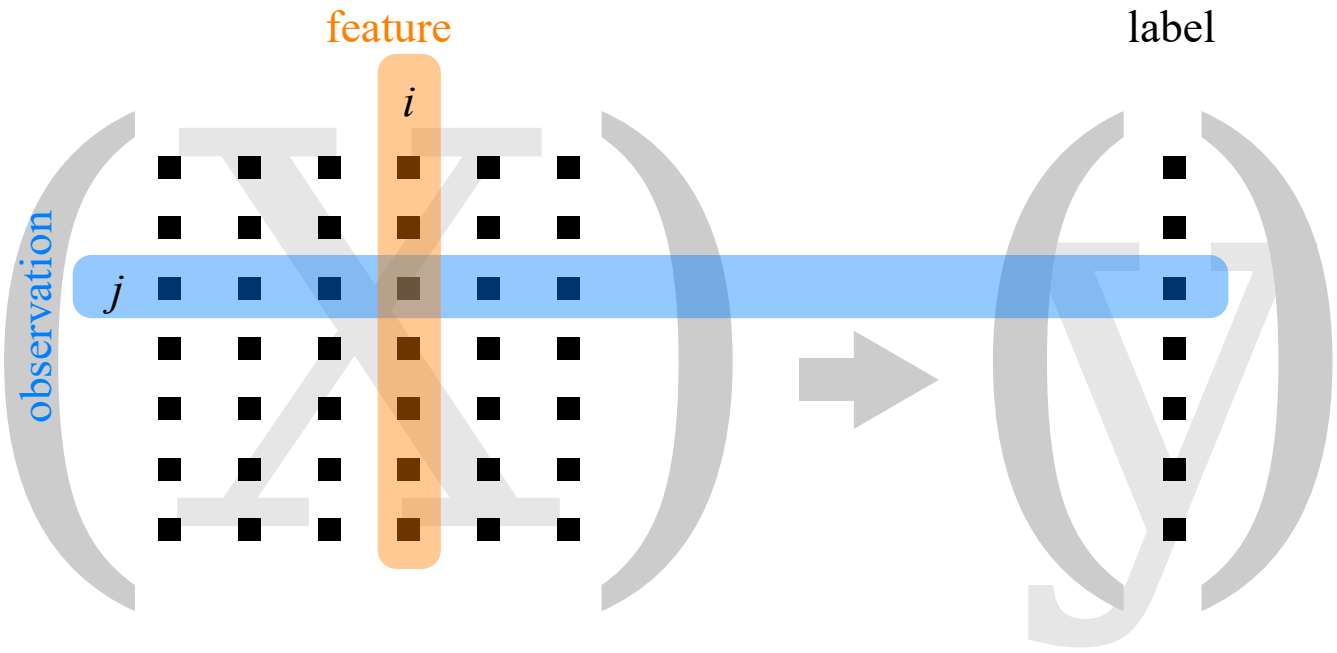
Supervised learning problems are defined by two characteristics:

1. There is a target variable, called the **label**, that we want to predict.
2. True label values are known for some training set.

Just as feature values are typically organized into a matrix $X = (X_{ji})$, label values are typically grouped into a column vector $y = (y_j)$ where y_j is the label value corresponding to observation X_j. Given a training set (X, y), our goal is to build a model f so that

$$f(X_j) \approx y_j$$

for each observation X_j. Then we can use f to make predictions by computing $f(X_{new})$ for each new observation X_{new} where the true label value is unknown.



It may help to think of features as independent variables and labels as dependent variables, although the dependence may be imperfect. Also, note that the label is assumed to be a single value, while there can be many features for each observation. This is because **multi-label** problems can be split into independent tasks by considering each label separately (although it is sometimes better not to do so, e.g. when values are correlated).

There are two broad subclasses of problems within supervised learning:

- **Regression** is when the label values are continuous and real-valued
- **Classification** is when the labels come from a discrete set of values

Unsupervised learning problems are simply those that do not fit into the supervised learning framework described above. Often this means that we are trying to do something other than make a prediction (e.g. Clustering or Dimension Reduction). However, unsupervised learning also includes problems where we want to make a prediction but lack proper label data (e.g. Outlier Detection).

Parameters and Learning

Most supervised learning algorithms have two components:

1. A family of models
2. A method of finding the model in the family that best fits a given data set

We often describe the family of models as a single function with variable parameters. For example, if we apply Linear Regression to a data set with one feature, then we will consider all models of the form $f(x) = ax + b$. Here the coefficients a and b are the parameters. For a decision tree model, the parameters are the split values.

We'll write β as shorthand for a collection of model parameters and f_β to denote the corresponding model. If we define a cost function $C(\beta)$ that measures how well f_β fits the training data (larger values are worse), then choosing the best function from the family $\{f_\beta\}$ becomes an optimization problem. That is, we want to find

$$\text{argmin}_\beta \, C(\beta) \, .$$

This may seem like a useless formalism, but it's often the case that $C(\beta)$ is a differentiable function, in which case we can find an approximate solution using methods like Gradient Descent.

Lets work through a concrete example using some artificial data.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-5,5,30)
y = 2 * np.cos(X + 1) + 0.3*np.random.randn(X.shape[0])
plt.plot(X, y, 'o');
```

Our goal is to approximate y with a function of X. Noting the shape and frequency of the data, a savvy modeler might choose the following family of models (Linear Regression with engineered features):

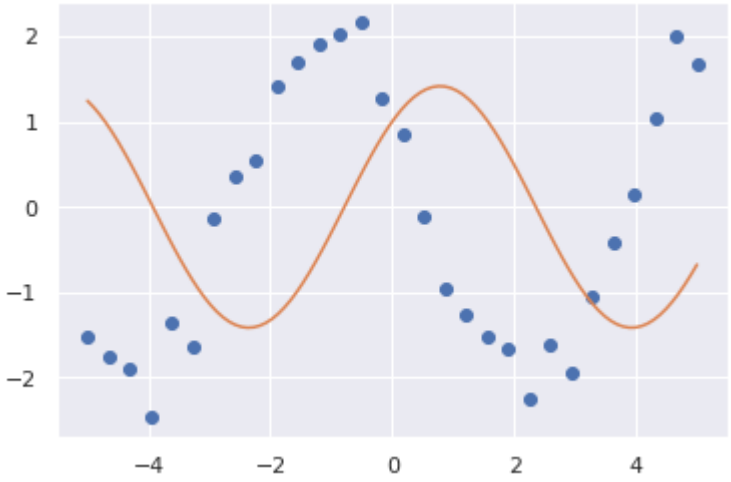
$$f(X) = \beta_0 \cos(X) + \beta_1 \sin(X)$$

Don't worry if this doesn't seem intuitive; most Machine Learning algorithms don't rely quite this heavily on modeler choice.

Let's try $\beta_0 = \beta_1 = 1$ to get a better feeling for what type of models we're considering.

In [3]:
```
X_fine = np.linspace(-5,5,100)
y_model = np.cos(X_fine) + np.sin(X_fine)

plt.plot(X, y, 'o')
plt.plot(X_fine, y_model);
```



Although the shape is correct, we clearly haven't chosen the best model parameters. In order to do this algorithmically, we need to define a cost function so that we have a well defined optimization problem to solve. One simple option is to just use the total error of the predictions, but this won't be differentiable. So instead, we'll use the total squared error

$$C(\beta_0, \beta_1) = \sum_j \left(f(X_j) - y_j\right)^2$$

where f is the function defined above and the sum ranges over all points (X_j, y_j) in the training set.

In [4]:
```
def C(beta):
    errors = beta[0] * np.cos(X) + beta[1] * np.sin(X) - y
    return np.sum(errors**2)
```

It may help to look at a contour plot of this cost function.

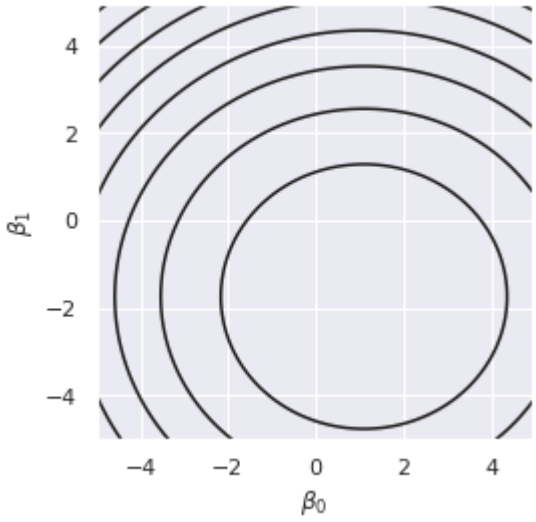
In [5]:
```
import matplotlib.pyplot as plt

m = 5          # maximum value on each beta axis
h = .05        # step size in the mesh
s = 2*int(m/h) # number of grid units on each axis

xx, yy = np.meshgrid(np.arange(-m, m, h),
                     np.arange(-m, m, h))

zz = np.apply_along_axis(C,1,np.c_[xx.ravel(), yy.ravel()]).reshape(s,s)

plt.axes().set_aspect('equal')
plt.contour(xx, yy, zz, 10, colors = 'k')
plt.xlabel(r"$\beta_0$")
plt.ylabel(r"$\beta_1$");
```



Each oval is a level curve, representing points where the cost function has equal value. To minimize the cost function, we need to move to the point in the center, which is roughly at $(1, -1)$. We could solve for this minimum explicitly by finding the unique point where both partial derivatives are equal to 0, but we'll use this as an opportunity to illustrate gradient descent instead.

The idea of gradient descent is to start at a random point and then make incremental movements toward the minimum until we reach a good approximation. The gradient (a generalization of the derivative to multiple dimensions) gives us the direction in which a function is *increasing fastest*. We'll want to go in the opposite direction. Since the gradient itself varies from place to place, we'll just take a small step in that direction, then compute again, and repeat process. This method is simple to implement and can be used even if the cost function is very complicated.

Here the result looks something like this:

In [6]:
```
def grad_C(beta):
    summands = 2 * (beta[0] * np.cos(X) + beta[1] * np.sin(X) - y) * [np.cos(X
), np.sin(X)]
    return np.sum(summands, axis=1)

alpha = 0.01               # parameter controlling step size
beta = np.array([-3., 4.]) # random starting point

betas = []

def gradient_descent_step():
    global beta
    beta -= alpha * grad_C(beta)

def gradient_descent(num_steps):
    for _ in range(num_steps):
        betas.append(beta.copy()) # record the latest value of beta (only need
ed for plotting purposes)
        gradient_descent_step()

gradient_descent(15)

print("beta_0:", beta[0])
print("beta_1:", beta[1])
```

```
beta_0: 1.0689029832196444
beta_1: -1.7208016295219541
```

In [7]:
```
from ipywidgets import interact, IntSlider

betas = np.array(betas)

def plot_grad_desc(num_steps):
    plt.axes().set_aspect('equal')
    plt.contour(xx, yy, zz, 10, colors = 'k')
    plt.plot(betas[:num_steps+1,0],betas[:num_steps+1,1], marker = 'o', marker
size=5)
    plt.xlabel(r"$\beta_0$")
    plt.ylabel(r"$\beta_1$")

interact(plot_grad_desc, num_steps=IntSlider(min=0,max=15,step=1,value=15));
```

Now we've found the parameters that define our best model, and we'll get a much better approximation.

In [8]:
```
X_fine = np.linspace(-5,5,100)
y_model = beta[0]*np.cos(X_fine) + beta[1]*np.sin(X_fine)

plt.plot(X, y, 'o')
plt.plot(X_fine, y_model);
```