```
Out[8]:    evil  horizon  of  problem  queen
        0     1        0   1        1      0
        1     1        0   0        0      1
        2     0        1   0        1      0
```

There are some issues with this approach, however: the raw word counts lead to features that put too much weight on words that appear very frequently, and this can be suboptimal in some classification algorithms. One approach to fix this is known as *term frequency–inverse document frequency* (*TF–IDF*), which weights the word counts by a measure of how often they appear in the documents. The syntax for computing these features is similar to the previous example:

```
In[9]: from sklearn.feature_extraction.text import TfidfVectorizer
       vec = TfidfVectorizer()
       X = vec.fit_transform(sample)
       pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
```

```
Out[9]:        evil   horizon        of   problem     queen
        0  0.517856  0.000000  0.680919  0.517856  0.000000
        1  0.605349  0.000000  0.000000  0.000000  0.795961
        2  0.000000  0.795961  0.000000  0.605349  0.000000
```

For an example of using TF–IDF in a classification problem, see "In Depth: Naive Bayes Classification" on page 382.

Image Features

Another common need is to suitably encode *images* for machine learning analysis. The simplest approach is what we used for the digits data in "Introducing Scikit-Learn" on page 343: simply using the pixel values themselves. But depending on the application, such approaches may not be optimal.

A comprehensive summary of feature extraction techniques for images is well beyond the scope of this section, but you can find excellent implementations of many of the standard approaches in the Scikit-Image project. For one example of using Scikit-Learn and Scikit-Image together, see "Application: A Face Detection Pipeline" on page 506.

Derived Features

Another useful type of feature is one that is mathematically derived from some input features. We saw an example of this in "Hyperparameters and Model Validation" on page 359 when we constructed *polynomial features* from our input data. We saw that we could convert a linear regression into a polynomial regression not by changing the model, but by transforming the input! This is sometimes known as *basis function regression*, and is explored further in "In Depth: Linear Regression" on page 390.

For example, this data clearly cannot be well described by a straight line (Figure 5-35):

```
In[10]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        x = np.array([1, 2, 3, 4, 5])
        y = np.array([4, 2, 1, 3, 7])
        plt.scatter(x, y);
```

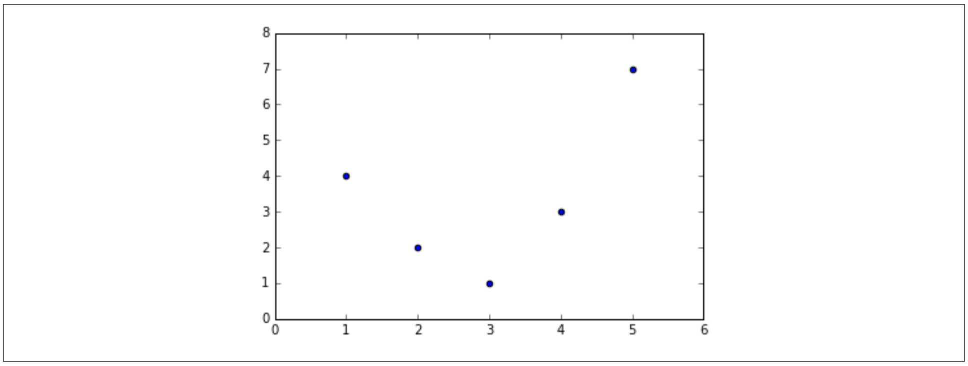

Figure 5-35. Data that is not well described by a straight line

Still, we can fit a line to the data using `LinearRegression` and get the optimal result (Figure 5-36):

```
In[11]: from sklearn.linear_model import LinearRegression
        X = x[:, np.newaxis]
        model = LinearRegression().fit(X, y)
        yfit = model.predict(X)
        plt.scatter(x, y)
        plt.plot(x, yfit);
```

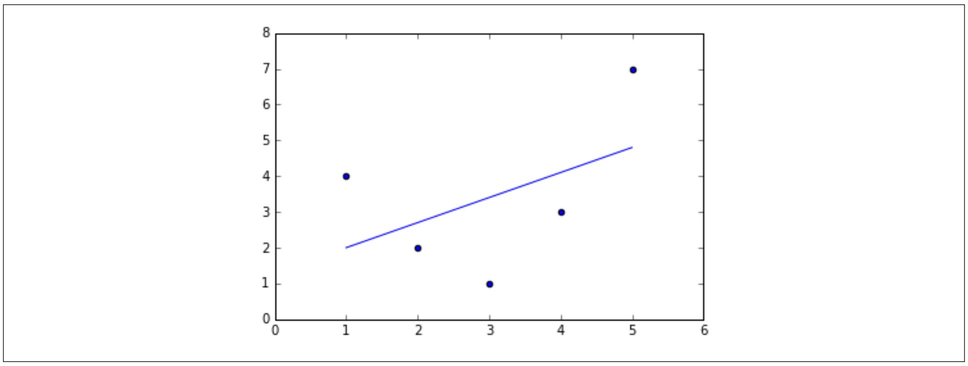

Figure 5-36. A poor straight-line fit

It's clear that we need a more sophisticated model to describe the relationship between *x* and *y*. We can do this by transforming the data, adding extra columns of features to drive more flexibility in the model. For example, we can add polynomial features to the data this way:

```
In[12]: from sklearn.preprocessing import PolynomialFeatures
        poly = PolynomialFeatures(degree=3, include_bias=False)
        X2 = poly.fit_transform(X)
        print(X2)
```

```
[[   1.    1.    1.]
 [   2.    4.    8.]
 [   3.    9.   27.]
 [   4.   16.   64.]
 [   5.   25.  125.]]
```

The derived feature matrix has one column representing x, and a second column representing x^2, and a third column representing x^3. Computing a linear regression on this expanded input gives a much closer fit to our data (Figure 5-37):

```
In[13]: model = LinearRegression().fit(X2, y)
        yfit = model.predict(X2)
        plt.scatter(x, y)
        plt.plot(x, yfit);
```

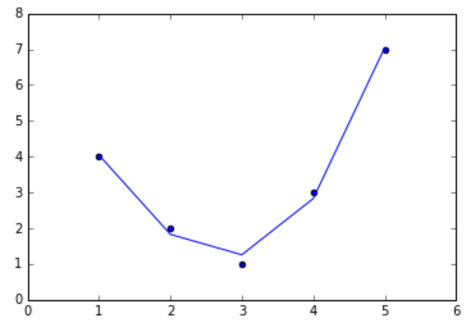

Figure 5-37. A linear fit to polynomial features derived from the data

This idea of improving a model not by changing the model, but by transforming the inputs, is fundamental to many of the more powerful machine learning methods. We explore this idea further in "In Depth: Linear Regression" on page 390 in the context of *basis function regression*. More generally, this is one motivational path to the powerful set of techniques known as *kernel methods*, which we will explore in "In-Depth: Support Vector Machines" on page 405.

Imputation of Missing Data

Another common need in feature engineering is handling missing data. We discussed the handling of missing data in `DataFrame`s in "Handling Missing Data" on page 119, and saw that often the `NaN` value is used to mark missing values. For example, we might have a dataset that looks like this:

```
In[14]: from numpy import nan
        X = np.array([[ nan, 0,   3  ],
                      [ 3,   7,   9  ],
                      [ 3,   5,   2  ],
                      [ 4,   nan, 6  ],
                      [ 8,   8,   1  ]])
        y = np.array([14, 16, -1,  8, -5])
```

When applying a typical machine learning model to such data, we will need to first replace such missing data with some appropriate fill value. This is known as *imputation* of missing values, and strategies range from simple (e.g., replacing missing values with the mean of the column) to sophisticated (e.g., using matrix completion or a robust model to handle such data).

The sophisticated approaches tend to be very application-specific, and we won't dive into them here. For a baseline imputation approach, using the mean, median, or most frequent value, Scikit-Learn provides the `Imputer` class:

```
In[15]: from sklearn.preprocessing import Imputer
        imp = Imputer(strategy='mean')
        X2 = imp.fit_transform(X)
        X2

Out[15]: array([[ 4.5,  0. ,  3. ],
                [ 3. ,  7. ,  9. ],
                [ 3. ,  5. ,  2. ],
                [ 4. ,  5. ,  6. ],
                [ 8. ,  8. ,  1. ]])
```

We see that in the resulting data, the two missing values have been replaced with the mean of the remaining values in the column. This imputed data can then be fed directly into, for example, a `LinearRegression` estimator:

```
In[16]: model = LinearRegression().fit(X2, y)
        model.predict(X2)

Out[16]:
array([ 13.14869292,  14.3784627 ,  -1.15539732,  10.96606197,  -5.33782027])
```

Feature Pipelines

With any of the preceding examples, it can quickly become tedious to do the transformations by hand, especially if you wish to string together multiple steps. For example, we might want a processing pipeline that looks something like this: