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 following formula:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b > 0$$

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"`. Clearly we need to represent our data in some different way when applying logistic regression. The next section will explain how we can overcome this problem.

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 the formula for linear binary classification (and for all other models in `scikit-learn`), and we can represent any number of categories by introducing one new feature per category, as described here.

Let's say for the `workclass` feature we have possible values of `"Government Employee"`, `"Private Employee"`, `"Self Employed"`, and `"Self Employed Incorporated"`. To encode these four possible values, we create four new features, called `"Government Employee"`, `"Private Employee"`, `"Self Employed"`, and `"Self Employed Incorporated"`. A 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 in Table 4-2. 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.

Table 4-2. Encoding the workclass feature using one-hot encoding

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

The one-hot encoding we use is quite similar, but not identical, to the dummy encoding used in statistics. For simplicity, we encode each category with a different binary feature. In statistics, it is common to encode a categorical feature with k different possible values into k–1 features (the last one is represented as all zeros). This is done to simplify the analysis (more technically, this will avoid making the data matrix rank-deficient).

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

In[2]:

```
import pandas as pd
# The file has no headers naming the columns, so we pass header=None
# and provide the column names explicitly in "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',
           'income'])
# For illustration purposes, we only select some of the columns
data = data[['age', 'workclass', 'education', 'gender', 'hours-per-week',
             'occupation', 'income']]
# IPython.display allows nice output formatting within the Jupyter notebook
display(data.head())
```

Table 4-3 shows the result.

Table 4-3. The first five rows of the adult dataset

	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:

In[3]:

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

Out[3]:

```
 Male      21790
 Female    10771
Name: gender, dtype: int64
```

We can see that there are exactly two values for gender in this dataset, `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. The `get_dummies` function automatically transforms all columns that have object type (like strings) or are categorical (which is a special `pandas` concept that we haven't talked about yet):

In[4]:

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

Out[4]:

```
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', 'workclass_ Never-worked', 'workclass_ Private',
  'workclass_ Self-emp-inc', 'workclass_ Self-emp-not-inc',
  'workclass_ State-gov', 'workclass_ Without-pay', 'education_ 10th',
  'education_ 11th', 'education_ 12th', 'education_ 1st-4th',
   ...
  'education_ Preschool', 'education_ Prof-school', 'education_ Some-college',
  'gender_ Female', 'gender_ Male', 'occupation_ ?',
  'occupation_ Adm-clerical', 'occupation_ Armed-Forces',
  'occupation_ Craft-repair', 'occupation_ Exec-managerial',
  'occupation_ Farming-fishing', 'occupation_ Handlers-cleaners',
  ...
  'occupation_ Tech-support', 'occupation_ Transport-moving',
  'income_ <=50K', 'income_ >50K']
```

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:

In[5]:

```
data_dummies.head()
```

Out[5]:

	age	hours-per-week	workclass_ ?	workclass_ Federal-gov	workclass_ Local-gov	...	occupation_ Tech-support	occupation_ Transport-moving	income_ <=50K	income_ >50K
0	39	40	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	1.0	0.0
2	38	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
3	53	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
4	28	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0

```
5 rows × 46 columns
```

We can now 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.

Be careful: column indexing in `pandas` includes the end of the range, so `'age':'occupation_ Transport-moving'` is inclusive of `occupation_ Transport-moving`. This is different from slicing a NumPy array, where the end of a range is not included: for example, `np.arange(11)[0:10]` doesn't include the entry with index 10.

In this case, we extract 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:

In[6]:

```
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: {}".format(X.shape, y.shape))
```

Out[6]:

```
X.shape: (32561, 44)  y.shape: (32561,)
```

Now the data is represented in a way that `scikit-learn` can work with, and we can proceed as usual:

In[7]:

```
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("Test score: {:.2f}".format(logreg.score(X_test, y_test)))
```

Out[7]:

```
Test score: 0.81
```

In this example, we called `get_dummies` on a `DataFrame` containing both the training and the test data. This is important to ensure categorical values are represented in the same way in the training set and the test set.

Imagine we have the training and test sets in two different `DataFrames`. If the `"Private Employee"` value for the `workclass` feature does not appear in the test set, `pandas` will assume there are only three possible values for this feature and will create only three new dummy features. Now our training and test sets have different numbers of features, and we can't apply the model we learned on the training set to the test set anymore. Even worse, imagine the `workclass` feature has 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` will create two new dummy features, so the encoded `DataFrames` will have the same number of features. However, the two dummy features have entirely different meanings in the training and test sets. The column that means `"Government Employee"` for the training set would encode `"Self Employed"` for the test set.

If we built a machine learning model on this data it would work very badly, because it would assume the columns mean the same things (because they are in the same position) when in fact they mean very different things. To fix this, either call `get_dummies` on a `DataFrame` that contains both the training and the test data points, or make sure that the column names are the same for the training and test sets 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 encoded as integers. For example, imagine the census data in the `adult` dataset was collected using a questionnaire, 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 will contain numbers from 0 to 8, instead of strings like `"Private"`, and it won't be immediately obvious to someone looking at the table representing the dataset whether they should treat this variable as continuous or categorical. Knowing that the numbers indicate employment status, however, it is clear that these are very distinct states and should not be modeled by a single continuous variable.

Categorical features are often encoded using integers. That they are numbers doesn't mean that they should necessarily be treated as continuous features. 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 in the `workclass` example), the feature must be treated as discrete. For other cases, like five-star ratings, the better encoding depends on the particular task and data and which machine learning algorithm is used.

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, let's create a `DataFrame` object with two columns, one containing strings and one containing integers:

`In[8]:`

```
# 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']})
display(demo_df)
```

Table 4-4 shows the result.