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.

Table 4-4. DataFrame containing categorical string features and integer features

	Categorical Feature	Integer Feature
0	socks	0
1	fox	1
2	socks	2
3	box	1

Using `get_dummies` will only encode the string feature and will not change the integer feature, as you can see in Table 4-5:

In[9]:

```
pd.get_dummies(demo_df)
```

Table 4-5. One-hot-encoded version of the data from Table 4-4, leaving the integer feature unchanged

	Integer Feature	Categorical Feature_box	Categorical Feature_fox	Categorical Feature_socks
0	0	0.0	0.0	1.0
1	1	0.0	1.0	0.0
2	2	0.0	0.0	1.0
3	1	1.0	0.0	0.0

If you want dummy variables to be created for the "Integer Feature" column, you can explicitly list the columns you want to encode using the `columns` parameter. Then, both features will be treated as categorical (see Table 4-6):

In[10]:

```
demo_df['Integer Feature'] = demo_df['Integer Feature'].astype(str)
pd.get_dummies(demo_df, columns=['Integer Feature', 'Categorical Feature'])
```

Table 4-6. One-hot encoding of the data shown in Table 4-4, encoding the integer and string features

	Integer Feature_0	Integer Feature_1	Integer Feature_2	Categorical Feature_box	Categorical Feature_fox	Categorical Feature_socks
0	1.0	0.0	0.0	0.0	0.0	1.0
1	0.0	1.0	0.0	0.0	1.0	0.0
2	0.0	0.0	1.0	0.0	0.0	1.0
3	0.0	1.0	0.0	1.0	0.0	0.0

Binning, Discretization, Linear Models, and Trees

The best way to represent data depends not only on the semantics of the data, but also on the kind of model you are using. Linear models and tree-based models (such as decision trees, gradient boosted trees, and random forests), two large and very commonly used families, have very different properties when it comes to how they work with different feature representations. Let's go back to the `wave` regression dataset that we used in Chapter 2. It has only a single input feature. Here is a comparison of a linear regression model and a decision tree regressor on this dataset (see Figure 4-1):

In[11]:

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

X, y = mglearn.datasets.make_wave(n_samples=100)
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)

reg = DecisionTreeRegressor(min_samples_split=3).fit(X, y)
plt.plot(line, reg.predict(line), label="decision tree")

reg = LinearRegression().fit(X, y)
plt.plot(line, reg.predict(line), label="linear regression")

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

As you know, linear models can only model linear relationships, which are lines in the case of a single feature. The decision tree can build a much more complex model of the data. However, this is strongly dependent on the representation of the data. One way to make linear models more powerful on continuous data is to use *binning* (also known as *discretization*) of the feature to split it up into multiple features, as described here.