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.

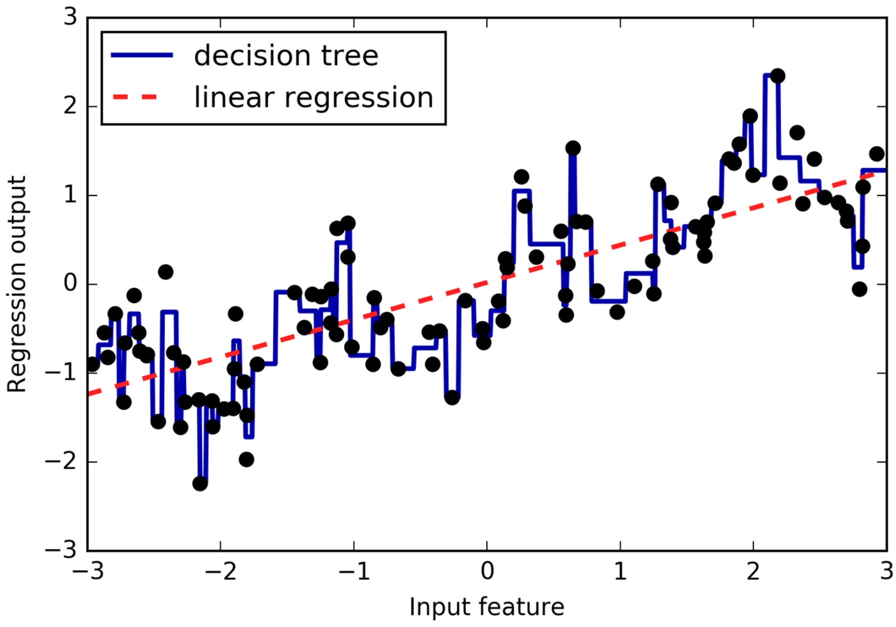

Figure 4-1. Comparing linear regression and a decision tree on the wave dataset

We imagine a partition of the input range for the feature (in this case, the numbers from –3 to 3) into a fixed number of *bins*—say, 10. A data point will then be represented by which bin it falls into. To determine this, we first have to define the bins. In this case, we'll define 10 bins equally spaced between –3 and 3. We use the `np.linspace` function for this, creating 11 entries, which will create 10 bins—they are the spaces in between two consecutive boundaries:

In[12]:

```
bins = np.linspace(-3, 3, 11)
print("bins: {}".format(bins))
```

Out[12]:

```
bins: [-3.  -2.4 -1.8 -1.2 -0.6  0.   0.6  1.2  1.8  2.4  3. ]
```

Here, the first bin contains all data points with feature values –3 to –2.68, the second bin contains all points with feature values from –2.68 to –2.37, and so on.

Next, we record for each data point which bin it falls into. This can be easily computed using the `np.digitize` function:

In[13]:

```
which_bin = np.digitize(X, bins=bins)
print("\nData points:\n", X[:5])
print("\nBin membership for data points:\n", which_bin[:5])
```

Out[13]:

```
Data points:
 [[-0.753]
  [ 2.704]
  [ 1.392]
  [ 0.592]
 [-2.064]]

Bin membership for data points:
 [[ 4]
  [10]
  [ 8]
  [ 6]
 [ 2]]
```

What we did here is transform the single continuous input feature in the `wave` dataset into a categorical feature that encodes which bin a data point is in. To use a `scikit-learn` model on this data, we transform this discrete feature to a one-hot encoding using the `OneHotEncoder` from the `preprocessing` module. The `OneHotEncoder` does the same encoding as `pandas.get_dummies`, though it currently only works on categorical variables that are integers:

In[14]:

```
from sklearn.preprocessing import OneHotEncoder
# transform using the OneHotEncoder
encoder = OneHotEncoder(sparse=False)
# encoder.fit finds the unique values that appear in which_bin
encoder.fit(which_bin)
# transform creates the one-hot encoding
X_binned = encoder.transform(which_bin)
print(X_binned[:5])
```

Out[14]:

```
[[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

Because we specified 10 bins, the transformed dataset `X_binned` now is made up of 10 features:

In[15]:

```
print("X_binned.shape: {}".format(X_binned.shape))
```

Out[15]:

```
X_binned.shape: (100, 10)
```

Now we build a new linear regression model and a new decision tree model on the one-hot-encoded data. The result is visualized in Figure 4-2, together with the bin boundaries, shown as dotted black lines:

In[16]:

```
line_binned = encoder.transform(np.digitize(line, bins=bins))

reg = LinearRegression().fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='linear regression binned')

reg = DecisionTreeRegressor(min_samples_split=3).fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='decision tree binned')
plt.plot(X[:, 0], y, 'o', c='k')
plt.vlines(bins, -3, 3, linewidth=1, alpha=.2)
plt.legend(loc="best")
plt.ylabel("Regression output")
plt.xlabel("Input feature")
```

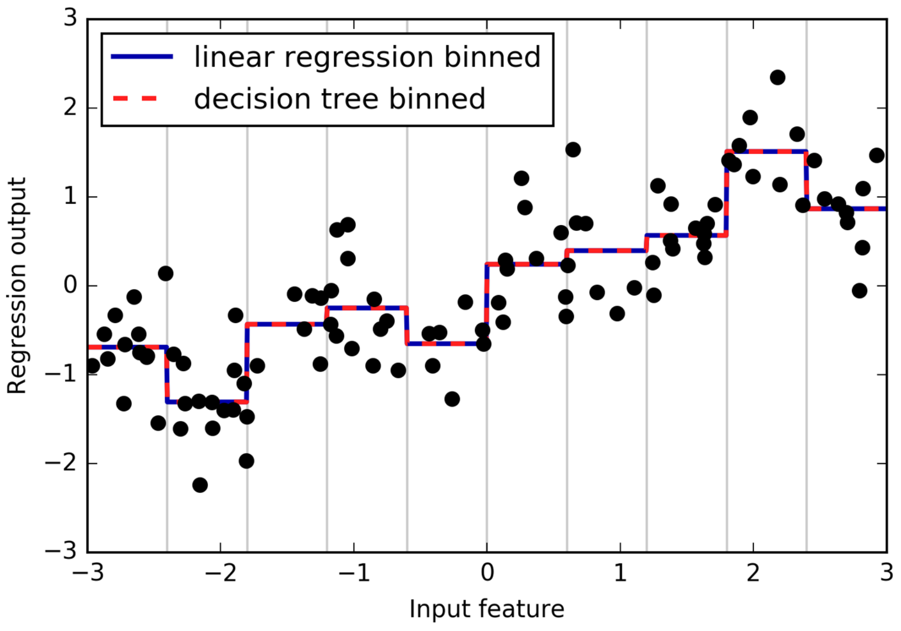

Figure 4-2. Comparing linear regression and decision tree regression on binned features

The dashed line and solid line are exactly on top of each other, meaning the linear regression model and the decision tree make exactly the same predictions. For each bin, they predict a constant value. As features are constant within each bin, any model must predict the same value for all points within a bin. Comparing what the models learned before binning the features and after, we see that the linear model became much more flexible, because it now has a different value for each bin, while the decision tree model got much less flexible. Binning features generally has no beneficial effect for tree-based models, as these models can learn to split up the data anywhere. In a sense, that means decision trees can learn whatever binning is most useful for predicting on this data. Additionally, decision trees look at multiple features at once, while binning is usually done on a per-feature basis. However, the linear model benefited greatly in expressiveness from the transformation of the data.

If there are good reasons to use a linear model for a particular dataset—say, because it is very large and high-dimensional, but some features have nonlinear relations with the output—binning can be a great way to increase modeling power.

Interactions and Polynomials

Another way to enrich a feature representation, particularly for linear models, is adding *interaction features* and *polynomial features* of the original data. This kind of feature engineering is often used in statistical modeling, but it's also common in many practical machine learning applications.

As a first example, look again at Figure 4-2. The linear model learned a constant value for each bin in the `wave` dataset. We know, however, that linear models can learn not only offsets, but also slopes. One way to add a slope to the linear model on the binned data is to add the original feature (the x-axis in the plot) back in. This leads to an 11-dimensional dataset, as seen in Figure 4-3:

In[17]:

```
X_combined = np.hstack([X, X_binned])
print(X_combined.shape)
```

Out[17]:

```
(100, 11)
```

In[18]:

```
reg = LinearRegression().fit(X_combined, y)

line_combined = np.hstack([line, line_binned])
plt.plot(line, reg.predict(line_combined), label='linear regression combined')

for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k')
```