Introduction

We continue our study of data preparation with these topics:

- Handling nominal-valued features
- Scaling numeric values

Handling Nominal-Valued Features

Most learning algorithms work only with numeric-valued features. (There are exceptions.) This is true, for example, of the OLS linear regression algorithms that we have looked at; it is less true of kNN, where you can define a special distance function that works on nominal-valued features.

We will look at how to convert nominal-valued features to numeric-valued ones.

Binary-valued features
The simplest case, obviously, is a binary-valued feature. One value is encoded as 0 and the other as 1, e.g. 'M' is 0 and 'F' is 1. In the example below, 'SecondHand' is 0 and 'New' is 1:

In [4]: # Not only does this replace the values, it also changes the data type too :)
   df.replace({'devment': {'SecondHand': 0, 'New': 1}}, inplace=True)

Unordered nominal values

Suppose there are more than two values, e.g. Apartment, Detached, SemiDetached or Terraced. The obvious thing to do is to assign integers to each nominal value, e.g. 0 = Apartment, 1 = Detached, etc.

In pandas, we could do this using the replace function, similar to what we did for the devment feature above.

But often this is not the best encoding. And that's certainly true here. Think of the original values for this feature. It does not make sense to calculate with them: what is Terraced multiplied by SemiDetached? It does not even make sense to compare them: is Apartment greater than Terraced? But if we use a simple numeric encoding like the one above a learning algorithm might mistakenly find such numeric relationships in the data.

Instead, we use one-hot encoding. If the original nominal-valued feature has $p$ values, then we use $p$ binary-valued features: in each example, exactly one of them is set to 1 and the rest are zero.

For example, there are four types of dwelling, so we have four binary-valued features. The first is set to 1 if and only if the type of dwelling is Apartment; the second is set to 1 if and only if the house is Detached; and so on. So for these four features a detached house will have [0, 1, 0, 0] as their values.

Some questions:

- One-hot encoding replaces one nominal-valued feature that has $p$ values by $p$ binary-valued ones — in general, one feature per nominal value. (E.g. type has four values, so we get four binary features.) What is the minimum number of binary-valued features we could use?
- Why don't we use the minimum?
- Although we get $p$ binary features, we only need $p - 1$. How come? (Advanced note: Look up the dummy variable trap to see why this might even be preferable.)
- How might one encode a feature whose values are sets of nominal values (such as the movie genre example given in the previous lecture)?

In practice, it is not uncommon to be given a dataset where a nominal-valued feature has already been encoded numerically, one integer per value. You might be fooled into thinking that the feature is numeric-valued and overlook the need to use one-hot encoding on it. Watch out for this!

So let's see how to do this in pandas:
# The get_dummies() function does the one-hot encoding. Note that NaNs will be replaced by all zeros

```python
one_hot = pd.get_dummies(df['type'], 'type', '_')
# And the next two statements 'replace' the existing feature by the new binary-valued features
# First, drop the existing column
df.drop('type', axis=1, inplace=True)
# Next, concatenate the new columns. This assumes no clash of column names.
df = pd.concat([df, one_hot], axis=1)
```

## Ordered nominal values

Consider the case now of a feature whose values are nominal but where there is an ordering. The `ber` feature in the housing dataset is like this, for example. In this case, G < F < E2 < E1 < D1 ... < A1 Some people would use the phrase 'ordinal-valued' to refer to nominal values that have an ordering.

You might be tempted to use a straightforward numeric encoding, e.g. 0 = G, 1 = F, 2 = E2, 3 = E1, and so on. This encoding preserves the ordering, e.g. that E2 < E1 because 2 < 3. But again this is probably not the best encoding. First, it has become possible to do calculations on the new feature that were not possible on the original, e.g. to multiply a F by a E2 (1 × 2 = 2 — but what does that mean?) Second, the original feature had an ordering on its values but no notion of distance. For example, G < F but you cannot say by how much G is less than F. In the new feature, we have introduced a notion of distance: G is worse than F by 2, and it is 2 worse than E2. So this encoding has added ‘information’ that was not present in the original.

So what should we do?

- We could use one-hot encoding: fifteen binary-valued features. But what’s the weakness of this?
- Another option is to use binary-valued features that represent inequalities. For example, one feature is set to 1 if you have a BER of at least G; another is additionally set to 1 if you have attained at least F, and so on — still fifteen binary-valued features, but no longer mutually exclusive. So, for example, a BER of E2 is converted to the following fifteen binary-valued features:

  `[1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]` and a BER of E1 is converted to

  `[1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`. Unfortunately, I don’t know of an method in a library for doing this — we would have to write our own. So the temptation will be to ‘make do’ with one-hot encoding.

Again watch out for cases where some well-intentioned person has already encoded this kind of feature but using a naive numeric encoding.

## Scaling Numeric Values
Different numeric-valued features often have very different ranges. For example, the values for floor area are going to range from a few tens to a few hundreds of square metres; but the number of bedrooms and bathrooms is going to range from 0 to a dozen or so at most. Often this is a problem for a learning algorithm.

- **OLS linear regression:**
  - The normal equation:
    - Having features with very different ranges of values causes no problem.

- **Gradient descent:**
  - It is a problem in this case. It becomes very difficult to find a value for the learning rate $\alpha$. A 'large' value for $\alpha$ makes big changes in $\beta_1$ (the floor area parameter), with a risk of overshooting the minimum or even of divergence; a 'small' enough value avoids this problem but makes such small changes to $\beta_2$ and $\beta_3$ (the parameters for the number of bedrooms and bathrooms) that convergence may take a very long time.

- **Lasso and Ridge Regression:**
  - It is generally thought to be useful to scale the features, irrespective of how the learner is implemented.

- **kNN:**
  - Features with large ranges will dominate the distance calculations, thus giving features with small ranges negligible influence on finding neighbours.

The solution is to scale the values so that they have similar ranges. We'll discuss several ways to do this. (Note that some people call this normalization, but it is not related to the normalization you may have studied in the context of relational databases.)

### Min-Max Scaling

Suppose we want to scale feature $j$. Let $\text{max}_j$ be the maximum possible value for this feature, which can be supplied by your domain expert. A quick-and-dirty way to scale the values to $[0, 1]$ is to divide each value $x_j$ by $\text{max}_j$:

$$x_j \leftarrow \frac{x_j}{\text{max}_j}$$

For example, Ann tells us that no house will be above 500 square metres and no house will have more than 10 bedrooms. So you divide values for the former by 500 and you divide values for the latter by 10.

Suppose your domain expert also supplies a minimum possible value $\text{min}_j$. Then a slightly improved way to scale to $[0, 1]$ is to subtract the minimum value and divide by the range:

$$x_j \leftarrow \frac{x_j - \text{min}_j}{\text{max}_j - \text{min}_j}$$

For example, Ann tells us that the smallest houses are 40 square metres and the largest are 500 square metres, so we subtract 40 and divide by $500 - 40 = 460$.

### Min-Max Scaling in pandas and scikit-learn

Min-max scaling in pandas is pretty easy:
# bdrms, where min = 1 and max = 10
# Subtract the min
df['bdrms'] -= 1
# Divide by the range, i.e. max - min
df['bdrms'] /= 9

# A quick check
df.describe()

```
Out[6]:

<table>
<thead>
<tr>
<th></th>
<th>flarea</th>
<th>bdrms</th>
<th>bthrms</th>
<th>floors</th>
<th>devment</th>
<th>price</th>
<th>ty</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>233.000000</td>
<td>234.000000</td>
<td>236.000000</td>
<td>233.000000</td>
<td>229.000000</td>
<td>224.000000</td>
<td>2</td>
</tr>
<tr>
<td>mean</td>
<td>145.458247</td>
<td>0.276353</td>
<td>2.161017</td>
<td>1.841202</td>
<td>0.013100</td>
<td>307.714286</td>
<td>0</td>
</tr>
<tr>
<td>std</td>
<td>134.103000</td>
<td>0.136270</td>
<td>1.188386</td>
<td>0.366275</td>
<td>0.113954</td>
<td>374.332421</td>
<td>0</td>
</tr>
<tr>
<td>min</td>
<td>0.371600</td>
<td>0.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.000000</td>
<td>55.000000</td>
<td>0</td>
</tr>
<tr>
<td>25%</td>
<td>81.800000</td>
<td>0.222222</td>
<td>1.000000</td>
<td>2.000000</td>
<td>0.000000</td>
<td>168.000000</td>
<td>0</td>
</tr>
<tr>
<td>50%</td>
<td>109.000000</td>
<td>0.222222</td>
<td>2.000000</td>
<td>2.000000</td>
<td>0.000000</td>
<td>225.000000</td>
<td>0</td>
</tr>
<tr>
<td>75%</td>
<td>167.200000</td>
<td>0.333333</td>
<td>3.000000</td>
<td>2.000000</td>
<td>0.000000</td>
<td>345.000000</td>
<td>0</td>
</tr>
<tr>
<td>max</td>
<td>1301.000000</td>
<td>1.000000</td>
<td>10.000000</td>
<td>2.000000</td>
<td>1.000000</td>
<td>3800.000000</td>
<td>1</td>
</tr>
</tbody>
</table>

8 rows x 10 columns
```

scikit-learn provides a class called `MinMaxScaler`, which does something similar. However:

- Above, we supplied the min (the smallest possible value) and the max (the largest possible value) — presumably we got them from our domain expert.
- In scikit-learn, the min and max are computed from the data: the smallest and largest actual values in the dataset.
- **Question**: What might potentially go wrong by using scikit-learn's approach?

## Demo I: Usefulness of Scaling

We'll show that scaling makes it easier to find a value of $\alpha$ that results in convergence when doing OLS linear regression using Gradient Descent Search. To illustrate, let's use the original version of the Cork property dataset (which has just `flarea`, `bdrms` and `bthrms` features).
Before scaling

Let's try to learn a linear model on the original data (not scaled) with Gradient Descent. See if you can find a value of $\alpha$ that makes it work out:

```python
In [8]: # Use pandas to read the CSV file
df = pd.read_csv("dataset-corkA.csv")

# Insert the extra feature (all ones)
df.insert(loc=0, column='ones', value=1)

# Get the feature-values and the target values into separate numpy arrays of numbers
X = df[['ones', 'flarea', 'bdrms', 'bthrms']].values
y = df['price'].values

# Run the GDS
beta, Jvals = gds_for_ols_linear_regression(X, y, alpha = 0.03, num_iterations = 4000)

# Display the final value of J
Jvals[-1]
```

Out[8]: `nan`
Scale the data

In [9]:

```python
# Use pandas to read the CSV file
df = pd.read_csv("dataset-corkA.csv")

# flarea, where min = 40 and max = 500
# Subtract the min
df['flarea'] -= 40
# Divide by the range, i.e. max - min
df['flarea'] /= 460

# bdrms, where min = 0 and max = 10
# Subtract the min
# df['bdrms'] -= 0
# Divide by the range, i.e. max - min
df['bdrms'] /= 10

# bthrms, where min = 0 and max = 10
# Subtract the min
# df['bthrms'] -= 0
# Divide by the range, i.e. max - min
df['bthrms'] /= 10

# A quick check
df.describe()
```

Out[9]:

<table>
<thead>
<tr>
<th></th>
<th>flarea</th>
<th>bdrms</th>
<th>bthrms</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>224.000000</td>
<td>224.000000</td>
<td>224.000000</td>
<td>224.000000</td>
</tr>
<tr>
<td>mean</td>
<td>0.198869</td>
<td>0.345089</td>
<td>0.213839</td>
<td>307.714286</td>
</tr>
<tr>
<td>std</td>
<td>0.177207</td>
<td>0.121548</td>
<td>0.118796</td>
<td>374.332421</td>
</tr>
<tr>
<td>min</td>
<td>0.002174</td>
<td>0.100000</td>
<td>0.100000</td>
<td>55.000000</td>
</tr>
<tr>
<td>25%</td>
<td>0.093641</td>
<td>0.300000</td>
<td>0.100000</td>
<td>168.000000</td>
</tr>
<tr>
<td>50%</td>
<td>0.147826</td>
<td>0.300000</td>
<td>0.200000</td>
<td>225.000000</td>
</tr>
<tr>
<td>75%</td>
<td>0.246685</td>
<td>0.400000</td>
<td>0.300000</td>
<td>345.000000</td>
</tr>
<tr>
<td>max</td>
<td>0.993478</td>
<td>1.000000</td>
<td>1.000000</td>
<td>3800.000000</td>
</tr>
</tbody>
</table>

8 rows × 4 columns

After scaling

And now it should be much easier to find a value of $\alpha$ that works:
Demo II: Usefulness of Scaling

Now we'll see how useful scaling is for Lasso Regression, Ridge Regression and kNN, again using the original Cork property dataset.

Before scaling

```python
In [10]: # Insert the extra feature (all ones)
df.insert(loc=0, column='ones', value=1)

# Get the feature-values and the target values into separate numpy arrays of numbers
X = df[['ones', 'flarea', 'bdrms', 'bthrms']].values
y = df['price'].values

# Run the GDS
beta, Jvals = gds_for_ols_linear_regression(X, y, alpha = 0.03, num_iterations = 4000)

# Display the final value of J
Jvals[-1]

Out[10]: 30767.643737838611
```

```plaintext
Demo II: Usefulness of Scaling

Now we'll see how useful scaling is for Lasso Regression, Ridge Regression and kNN, again using the original Cork property dataset.

Before scaling

```python
In [11]: # Use pandas to read the CSV file
df = pd.read_csv("dataset-corkA.csv")

# Get the feature-values and the target values into separate numpy arrays of numbers
X = df[['flarea', 'bdrms', 'bthrms']].values
y = df['price'].values

In [12]: from sklearn.cross_validation import cross_val_score
   from sklearn.neighbors import KNeighborsRegressor

knn = KNeighborsRegressor(n_neighbors = 3)
mse_test = np.abs(cross_val_score(knn, X, y, scoring = 'mean_squared_error', cv = 10))
mean_mse_test = np.mean(mse_test)
mean_mse_test

Out[12]: 136650.68151075978

In [13]: from sklearn.linear_model import LassoCV
   lassoCV = LassoCV(cv = 10)
mse_test = np.abs(cross_val_score(lassoCV, X, y, scoring = 'mean_squared_error', cv = 10))
mean_mse_test = np.mean(mse_test)
mean_mse_test

Out[13]: 92299.83116105033
```
In [14]:
from sklearn.linear_model import RidgeCV

ridgeCV = RidgeCV(cv = 10)
rmse_test = np.abs(cross_val_score(ridgeCV, X, y, scoring = 'mean_squared_error', cv = 10))
mean_mse_test = np.mean(rmse_test)
mean_mse_test

Out[14]: 99203.60747200335

Scale the data

In [15]:
# Use pandas to read the CSV file
df = pd.read_csv("dataset-corkA.csv")

# flarea, where min = 40 and max = 500
# Subtract the min
df['flarea'] -= 40
# Divide by the range, i.e. max - min
df['flarea'] /= 460

# bdrms, where min = 0 and max = 10
# Subtract the min
# df['bdrms'] -= 0
# Divide by the range, i.e. max - min
df['bdrms'] /= 10

# bthrms, where min = 0 and max = 10
# Subtract the min
# df['bthrms'] -= 0
# Divide by the range, i.e. max - min
df['bthrms'] /= 10

# Get the feature-values and the target values into separate numpy arrays of numbers
X = df[['flarea', 'bdrms', 'bthrms']].values
y = df['price'].values

After scaling

In [16]:
rmse_test = np.abs(cross_val_score(knn, X, y, scoring = 'mean_squared_error', cv = 10))
mean_mse_test = np.mean(rmse_test)
mean_mse_test

Out[16]: 112618.28019323673

In [17]:
rmse_test = np.abs(cross_val_score(lassoCV, X, y, scoring = 'mean_squared_error', cv = 10))
mean_mse_test = np.mean(rmse_test)
mean_mse_test

Out[17]: 93370.253025557817
We see lower error on the scaled data for kNN and Ridge Regression. Conventional wisdom and intuition is that we should see the same in the case of Lasso Regression. But, we don't! I'm not sure why.

**Standardization**

In some cases, you want feature values not just to have the same range but to have the same mean and even the same variance.

One idea is **mean centering**, where you subtract the mean value of the feature. If you do this to all values, some of the new values will be positive and some will be negative and their mean will be approximately zero. But the features may still have different ranges. So you might still divide by the maximum or the range.

But better still is **standardization**, in which you subtract the mean and divide by the standard deviation:

$$x_j \leftarrow \frac{x_j - \mu_j}{\sigma_j}$$

where $\mu_j$ is the mean of the values for feature $j$ and $\sigma_j$ is their standard deviation.

If you use this, then the mean will be approximately zero and all values will be expressed in terms of how many standard deviations they are from the mean.

**Standardize the Wrong Way**

Here it is done incorrectly on the `flarea` feature!
In [19]:
# Use pandas to read the CSV file
df = pd.read_csv("dataset-corkA.csv")

# flarea
flarea_mean = df['flarea'].mean()
flarea_std = df['flarea'].std()
df['flarea'] -= flarea_mean
df['flarea'] /= flarea_std

# A quick check
df.describe()

Out[19]:
<table>
<thead>
<tr>
<th></th>
<th>flarea</th>
<th>bdrms</th>
<th>bthrms</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>2.24000e+02</td>
<td>224.0</td>
<td>224.0</td>
<td>224.000000</td>
</tr>
<tr>
<td>mean</td>
<td>-8.103637e-17</td>
<td>3.450893</td>
<td>2.138393</td>
<td>307.714286</td>
</tr>
<tr>
<td>std</td>
<td>1.000000e+00</td>
<td>1.215483</td>
<td>1.187962</td>
<td>374.332421</td>
</tr>
<tr>
<td>min</td>
<td>-1.109974e+00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>55.000000</td>
</tr>
<tr>
<td>25%</td>
<td>-5.938135e-01</td>
<td>3.000000</td>
<td>1.000000</td>
<td>168.000000</td>
</tr>
<tr>
<td>50%</td>
<td>-2.880428e-01</td>
<td>3.000000</td>
<td>2.000000</td>
<td>225.000000</td>
</tr>
<tr>
<td>75%</td>
<td>2.698276e-01</td>
<td>4.000000</td>
<td>3.000000</td>
<td>345.000000</td>
</tr>
<tr>
<td>max</td>
<td>4.484065e+00</td>
<td>10.000000</td>
<td>10.000000</td>
<td>3800.000000</td>
</tr>
</tbody>
</table>

8 rows × 4 columns

**Standardization the Correct Way**

The problem with the approach above is the same problem that we discussed when imputing means for missing values: **leakage**.

The solution is to use scikit-learn's `StandardScaler` and `Pipeline` classes.