CS4619: Artificial Intelligence 2

Data Preparation I

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [1]: %load_ext autoreload
%autoreload 2
%matplotlib inline

In [2]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

Introduction
In previous lectures, we rushed straight in to studying learning algorithms. But in reality before you throw learning algorithms at your data, there is the task of **data preparation**. Real data is incomplete (e.g. with values missing), noisy (containing errors) and inconsistent (especially if it comes from multiple sources). We must deal with at least some of these problems before applying any learning algorithms (otherwise, garbage-in-garbage-out). People estimate that data preparation accounts for 60% or more of the work in data analytics.

An incomplete list of data preparation activities (in no particular order):

- **Data cleaning**:
  - handling missing values; smoothing noisy data; removing outliers; correcting erroneous values; ...
- **Data integration**:
  - de-normalizing relational databases and flattening other data structures such as trees and graphs; resolving inconsistencies; identifying and handling duplicates; ...
- **Data transformation**:
  - handling nominal values when only numeric values are allowed; discretizing numeric values; changing character encodings and other data representations; scaling; ...
- **Feature engineering**:
  - augmenting the dataset with new features, either newly-gathered features or ones that are computed from the other features (e.g. aggregates of the others); ...
- **Data reduction**:
  - sampling; feature selection to remove features with low predictive power; reducing the dimensionality of the data through techniques such as Principal Component Analysis; ...

We will only scratch the surface of this huge topic. In particular, we will look at feature engineering and data reduction in later lectures. In this lecture, we will study the following topics:

1. Getting to know your data
2. Detecting anomalies
3. Handling missing values

And in the next lecture, we will study these:

4. Handling nominal-valued features
5. Scaling numeric values

## Getting to Know Your Data

Time spent just exploring the data is always worthwhile — especially if you have access to a domain expert while you are doing so.

To illustrate this lecture, we will use a different version of the Cork property dataset — a version that has more features.
In [3]: # Use pandas to read the CSV file
df = pd.read_csv("dataset-corkB.csv")

print(df.columns)
print(df.shape)

Index([u'flarea', u'type', u'bdrms', u'bthrms', u'floors', u'devment ', u'ber', u'location', u'price'], dtype='object')
(236, 9)

We have 236 examples and 8 features. The 9th column is the dependent variable, the actual selling price of the house.

We should think about the data type of each feature. There are at least the following data types:

- **Numeric-valued:** either real- or integer-valued, such as floor area or number of bedrooms.
- **Nominal-valued:** where there is a finite set of possible values. Often these values are strings. For example, dwelling type is a nominal-valued feature whose values are "Apartment", "Detached", "SemiDetached" or "Terraced". The special case here is, of course, a binary-valued feature, where there are just two values. For example, sex is a nominal-valued feature whose values are (e.g.) 'M' or 'F';
- **Unstructured:**
  - free text
  - media such as images, audio and video

There are other possibilities too. For example, the genre of a movie might be a set-valued feature, i.e. a set of nominal values, e.g. the value of the genre feature for *The Blues Brothers* is {musical, comedy, action}.

The 8 features of the Cork property dataset have the following types:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flarea</td>
<td>numeric</td>
<td>the floor area in square metres</td>
</tr>
<tr>
<td>type</td>
<td>nominal</td>
<td>dwelling type: Apartment, Detached, SemiDetached, Terraced</td>
</tr>
<tr>
<td>bdrms</td>
<td>numeric</td>
<td>the number of bedrooms</td>
</tr>
<tr>
<td>bthrms</td>
<td>numeric</td>
<td>the number of bathrooms</td>
</tr>
<tr>
<td>floors</td>
<td>numeric</td>
<td>the number of floors</td>
</tr>
<tr>
<td>devment</td>
<td>nominal</td>
<td>the type of development: New or SecondHand</td>
</tr>
<tr>
<td>ber</td>
<td>nominal</td>
<td>building energy rating: A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, E1, E2, F, G</td>
</tr>
<tr>
<td>location</td>
<td>nominal</td>
<td>the area of Cork, e.g. Douglas, Glanmire, Wilton,...</td>
</tr>
</tbody>
</table>
In [4]: # Use pandas to see the actual types
df.dtypes

Out[4]:
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>flarea</td>
<td>float64</td>
<td>type</td>
<td>object</td>
<td>bdrms</td>
</tr>
<tr>
<td>bthrms</td>
<td>int64</td>
<td>floors</td>
<td>float64</td>
<td>devment</td>
</tr>
<tr>
<td>ber</td>
<td>object</td>
<td>location</td>
<td>object</td>
<td>price</td>
</tr>
</tbody>
</table>

dtype: object

In [5]: # Use pandas to get summary statistics
   # In the latest version of pandas, df.describe(include='all') should describe the entire dataframe
   # For older versions, we must look at the numeric-valued columns and nominal-valued columns separately
   # By default, df.describe() will summarize the numeric-valued columns (if there are any)
df.describe()

Out[5]:

<table>
<thead>
<tr>
<th></th>
<th>flarea</th>
<th>bdrms</th>
<th>bthrms</th>
<th>floors</th>
<th>price</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>224.000000</td>
</tr>
<tr>
<td>mean</td>
<td>145.458247</td>
<td>3.487179</td>
<td>2.161017</td>
<td>1.841202</td>
<td>307.714286</td>
</tr>
<tr>
<td>std</td>
<td>134.103000</td>
<td>1.226428</td>
<td>1.188386</td>
<td>0.366275</td>
<td>374.332421</td>
</tr>
<tr>
<td>min</td>
<td>0.371600</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>55.000000</td>
</tr>
<tr>
<td>25%</td>
<td>81.800000</td>
<td>3.000000</td>
<td>1.000000</td>
<td>2.000000</td>
<td>168.000000</td>
</tr>
<tr>
<td>50%</td>
<td>109.000000</td>
<td>3.000000</td>
<td>2.000000</td>
<td>2.000000</td>
<td>225.000000</td>
</tr>
<tr>
<td>75%</td>
<td>167.200000</td>
<td>4.000000</td>
<td>3.000000</td>
<td>2.000000</td>
<td>345.000000</td>
</tr>
<tr>
<td>max</td>
<td>1301.000000</td>
<td>10.000000</td>
<td>10.000000</td>
<td>2.000000</td>
<td>3800.000000</td>
</tr>
</tbody>
</table>

Even from what we have done so far, we might already be able to detect some problems:

- Are there features with missing values?
- Are there features with anomalous values?

In [6]: # And now we'll ask it explicitly to summarize the other columns
df[['type', 'devment', 'ber', 'location']].describe()

Out[6]:

<table>
<thead>
<tr>
<th></th>
<th>type</th>
<th>devment</th>
<th>ber</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>236</td>
<td>229</td>
<td>236</td>
<td>236</td>
</tr>
<tr>
<td>unique</td>
<td>4</td>
<td>2</td>
<td>12</td>
<td>38</td>
</tr>
<tr>
<td>top</td>
<td>Detached</td>
<td>SecondHand</td>
<td>C3</td>
<td>CityCentre</td>
</tr>
<tr>
<td>freq</td>
<td>77</td>
<td>226</td>
<td>28</td>
<td>45</td>
</tr>
</tbody>
</table>
Detecting Anomalies

Most datasets are noisy.

- People supply incorrect data; e.g. see the *flarea* example below.
- They make spelling mistakes. In nominal-valued data, this can lead to a proliferation of values, where there should be fewer. For example, "Sunday's Well", "Sundays' Well", "Sundays Well", "Sunday Wells", "Sundays Wells", "Sundays well" will, from the algorithm's point of view, all be different locations.
- They make typing errors.
- Software that has been instrumented to measure, e.g., user actions (number of clicks, response times, etc.) may be buggy.
- Measuring equipment may malfunction.
- ...and so on.

And this noise can affect the feature values or the values of the dependent variable.

What will you do? Well, if it matters a lot, you can do lots of checking. There are also some automatic methods that can help with anomaly detection based on probability distributions — but we don't have time to examine these.

A simple-minded approach for numeric-valued features is to do some graphing. You might be able to detect some noisy values visually, which you can then check with your domain expert.

Here, for the new version of the Cork property dataset, is a graph of the non-missing values of *flarea*:
# You might be worried about the missing flareas and prices.
# In this case, there's no need to worry. When there's an empty string in the CSV file, pandas stores these as NaN, # which is a float. And in a scatter plot, matplotlib ignores any points whose $x = \text{NaN}$ or whose $y = \text{NaN}$.

```python
x = df['flarea'].values
y = df['price'].values

fig = plt.figure()
plt.title("Training set")
plt.xlabel("Floor area (sq metres)")
plt.xlim(0, 1800)
plt.ylabel("Price (000 euros)")
plt.ylim(0, 1000)
xvals = np.linspace(0, 1800, 2)
plt.scatter(x, y, color = 'green')
plt.show()
```

It's clear that three of the values are anomalous. It's almost certain that the person who supplied them put the decimal point in the wrong place. These three values need to be corrected, or the examples should be deleted.

```python
In [8]: df = df[(df['flarea'].isnull()) | ((df['flarea'] > 10) & (df['flarea'] < 1000))]
    df.reset_index(drop=True, inplace=True)

    # Check the anomalies were filtered out
    df.shape

Out[8]: (233, 9)
```

Questions:

- Correcting or deleting anomalous values typically needs to happen before handling missing values. Why? (After reading the next section, make sure you know why this is.)
- Simple graphs like the one above will not reveal all anomalous data. What kinds of anomalies will fail to show up?
Handling Missing Values

Many datasets contain examples that are missing the values of some of the features. In a text file such as a CSV file, these missing values are usually indicated by using a special value:

- For a nominal-valued feature, a missing value may be encoded by the empty string, or a string containing one or more whitespace characters, or a dash, or ‘?’ or ‘UNK’, or ‘N/A’, ...
- For a numeric-valued feature, missing values may be encoded using the same strings that we have just mentioned, or an out-of-range number might be used (0, -1,...).

Values may be missing for at least the following reasons:

- Inapplicable feature: the value is missing because it does not apply to this object. E.g. a feature that records how many weeks pregnant someone is only applies to people who are pregnant. E.g. if you are feeding mice for ten weeks in a lab experiment and measuring their weight at the end of each week, then you will have ten features for each mouse; but if a mouse dies before the end of the experiment one or more of its ten features will have missing values.
- Value not recorded: there is a value but it is not recorded in the dataset. E.g. a person filling in a survey refuses to give his/her sex or income. E.g. a student does not know what CAO points s/he attained. E.g. the mouse survives but we forget to weigh it or the measuring equipment malfunctions in week 4, so that value is missing.
- Status unknown: the value is missing but we don't know which of the previous two is the case. E.g. the mouse's weight for week 10 is missing: is it because the mouse died at the end of week 9 (so the value for week 10 is inapplicable) or is it because we forgot to measure it?

But there can be domain-specific reasons why values are missing. For example, a student assignment may be scored out of ten (including zero, meaning the work was submitted but was worthless). But you might want to distinguish the following cases where the value is missing: the student did not submit (for no good reason), the student did not submit for good reason (e.g. illness) so the assignment is written off, or the student plagiarised the work.

If you're really lucky, the dataset might use different symbols for these different cases.

Most learning algorithms will expect a value for each feature. (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 could define a clever distance function that handles missing values in some special way.

So, how can we handle missing values? We'll look at some of the options.

Finding Missing Values in Pandas

As we explored our data, we found various clues that suggested that in this version of the Cork property dataset the flaira, bdrms, floors, devment and price have missing values.

When pandas reads a CSV file that contains empty strings, it represents them as the float NaN — even if the column is nominal-valued! It provides the isnull, notnull and count functions for testing for these. (If a value is NaN or None, then isnull returns True.)
In [9]: `df['flarea'].isnull`

Out[9]:  0  False
        1  False
        2  False
        3   True
        4  False
        5  False
        6  False
        7  False
        8  False
        9  False
       10  False
       11  False
       12  False
       13  False
       14  False
       ...  False
       218  False
       219  False
       220  False
       221  False
       222  False
       223  False
       224  False
       225  False
       226  False
       227  False
       228  False
       229  False
       230  False
       231  False
       232  False

Name: flarea, Length: 233, dtype: bool

In [10]: `# Count non-nulls`
   `df['price'].count`

Out[10]: 221

Suppose your CSV files doesn't use the empty string for missing values; it uses some other special string such as 'UNK', 'N/A' or '?' instead. (When this is the case and pandas reads in the CSV file, it will result in a numeric-valued feature being stored as strings.) You must determine what special string is being used. My advice is to replace these special strings by NaN right from the outset.

Now all missing values are represented by NaN. Let's look at what to do about them.

Revisit the Data Collection Process

In a few cases, it might be possible to return to the data collection process and, perhaps with some cost and effort, obtain some of the missing values.

Assuming this is not possible, let's look in more detail at the other options. Bear in mind that they should be used with care, taking into account as far as possible what the missing values signify.
Delete Examples

If only a small proportion of the examples in the dataset contain features with missing values, then you could simply delete those *examples*.

For example, where missing values are recorded using the empty string, we can do this in Python:

```
In [11]: # Delete examples whose flarea is NaN
    df.dropna(subset=['flarea'], inplace=True)
    df.reset_index(drop=True, inplace=True)

    # Check the examples with missing values were removed
    df.shape
```

Out[11]: (230, 9)

**Question:**
- In which 'column' is a missing value particularly problematic? If you can't revisit the data collection process, you will probably delete these examples.

Delete Features

If one or two of the features are plagued by missing values, then we could simply delete those *features*.

```
In [12]: # E.g. pretend that the BER feature has so many missing values that
   it is hardly worth using it
   # We can delete it either using del
   # del df['ber']
   # or using drop
   df.drop('ber', axis=1, inplace=True)

   # Check the feature was removed
   df.shape
```

Out[12]: (230, 8)

This is OK if the features that you remove are not likely to be very predictive.

Impute Values the Wrong Way
The final approach is to **impute** a value, which simply means filling in the missing values. For numeric-valued features, you can use the mean (or median) of the values that are not missing, e.g. the average floor area. For nominal-valued features, you can use the mode (i.e. the most frequent of the values that are not missing).

You might instead have some domain-specific way of imputing values that you can agree with your domain expert. Returning again to the example of the weights of mice, if a mouse is missing the value for week 4 but its values for weeks 3 and 5 are present, then you could take the average of its week 3 and 5 values, rather than the average of the week 4 values for all the other mice.

Here's one way to impute the mean of the non-missing values for a numeric-valued feature such as `bdrms` and `devment`:

```python
In [13]: # You could get the mean from the describe() function, but we have made changes to df since we ran describe()
   # so the mean may have also changed.
   # Safer is to compute the mean afresh using df['bdrms'].mean() and fill the missing values (nan) by the mean:
   df.fillna({'bdrms': df['bdrms'].mean(), 'devment': df['devment'].mode()[0]}, inplace=True)

   # Check the missing values have been replaced by counting non-nulls
   df['bdrms'].count(), df['devment'].count()
```

```
Out[13]: (230, 230)
```

Now how the code above is more complicated for the nominal-valued feature. In what way and why?

**Impute Values the Correct Way**

The code in the previous section gives a quick-and-dirty solution. But it is not really correct.

Suppose we are going to split the dataset into training and test sets. The solution above is an example of **leakage**, which we have discussed before. What is leakage?

In a future lecture, we will look at a better solution, where we will use scikit-learn's **Imputer** class, which will also necessitate use of scikit-learn's **Pipeline** class.