CS4618: Artificial Intelligence I

Datasets

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
```

In [3]:

```
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin

from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler

# Class, for use in pipelines, to select certain columns from a DataFrame and co
nvert to a numpy array
# From A. Geron: Hands-On Machine Learning with Scikit-Learn & TensorFlow, O'Rei
lly, 2017
# Modified by Derek Bridge to allow for casting in the same ways as pandas.DataF
rame.astype
class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names, dtype=None):
        self.attribute_names = attribute_names
        self.dtype = dtype
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        X_selected = X[self.attribute_names]
        if self.dtype:
            return X_selected.astype(self.dtype).values
        return X_selected.values
```

Features

- Suppose we want to store data about objects, such as houses
- **Features** describe the houses, e.g.
  - $flarea$: the total floor area (in square metres)
  - $bdrms$: the number of bedrooms
  - $bthrms$: the number of bathrooms
- A particular house has **values** for the features
  - e.g. your house: $flarea = 114, bdrms = 3, bthrms = 2$
- Then we can represent a house using a vector
  - e.g. your house: $\begin{bmatrix} 114 \\ 3 \\ 2 \end{bmatrix}$

  We will always use n to refer to the number of features, e.g. above $n = 3$

Examples

- Suppose we collect a **dataset** containing data about lots of houses, e.g.:

$$\begin{bmatrix} 114 \\ 3 \\ 2 \end{bmatrix} \begin{bmatrix} 92.9 \\ 3 \\ 2 \end{bmatrix} \begin{bmatrix} 171.9 \\ 4 \\ 3 \end{bmatrix} \begin{bmatrix} 79 \\ 3 \\ 1 \end{bmatrix}$$

- Each member of this dataset is called an **example**, and we will use m to refer to the number of examples, e.g. above $m = 4$

Dataset notation

- We will use a *superscript* to index the examples
  - $\boldsymbol{x}^{(i)}$ will be the ith example
  - The first example in the dataset is $\boldsymbol{x}^{(1)}$, the second is $\boldsymbol{x}^{(2)}$, $\ldots$, the last is $\boldsymbol{x}^{(m)}$ (Note, we index from 1)
  - We're writing the superscript in parentheses to make it clear that we are using it for indexing. It is not 'raising to a power'. If we want to raise to a power, we will drop the parentheses.
- We will use a *subscript* to index the features (again starting from 1)
- Class exercise. Using the dataset on the previous slide
  - what is $\boldsymbol{x}_2^{(1)}$?
  - what is $\boldsymbol{x}_1^{(2)}$?

Dataset as a matrix

- We can represent a dataset $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$ as a $m \times n$ matrix $\boldsymbol{X}$ as follows:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^{(1)} & \boldsymbol{x}_2^{(1)} & \ldots & \boldsymbol{x}_n^{(1)} \\ \boldsymbol{x}_1^{(2)} & \boldsymbol{x}_2^{(2)} & \ldots & \boldsymbol{x}_n^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{x}_1^{(m)} & \boldsymbol{x}_2^{(m)} & \ldots & \boldsymbol{x}_n^{(m)} \end{bmatrix}$$

- Note how each example becomes a *row* in $\boldsymbol{X}$
- You can think of row i as the transpose of $\boldsymbol{x}^{(i)}$
- For the example dataset, we get

$$\boldsymbol{X} = \begin{bmatrix} 114 & 3 & 2 \\ 92.9 & 3 & 2 \\ 171.9 & 4 & 3 \\ 79 & 3 & 1 \end{bmatrix}$$

Cork Property Prices Dataset

- At the beginning of November 2014, I scraped a dataset of property prices for Cork city from www.daft.ie
- They are in a CSV file. Each line in the file is an example, representing one house
- Hence, each line of the file contains the feature-values for the floor area, number of bedrooms, number of bathrooms, and several other features that we will ignore for now
- We will use the pandas library
  - to read the dataset from the csv file into what pandas calls a DataFrame
  - to explore the dataset: looking at values, computing summary statistics, plotting graphs…
- But then we will use the scikit-learn library
  - we will create 'pipelines' to transform the data
  - typically the first step in every pipeline will convert the pandas DataFrame to a numpy 2D array
  - typically the next step in the pipeline will prepare the data (e.g. scale it)
  - typically the last step in the pipeline will do something interesting: clustering, regression, classification,…

Using pandas to Read and Explore the Data

In [4]:

```
# Use pandas to read the CSV file into a DataFrame
df = pd.read_csv("datasets/dataset_corkA.csv")
```

In [5]:

```
# The dimensions
df.shape
```

Out[5]:

```
(207, 9)
```

In [6]:

```
# The features
df.columns
```

Out[6]:

```
Index(['flarea', 'type', 'bdrms', 'bthrms', 'floors', 'devment', 'be
r',
       'location', 'price'],
      dtype='object')
```

In [7]:

```
# The datatypes
df.dtypes
```

Out[7]:

```
flarea      float64
type         object
bdrms         int64
bthrms        int64
floors        int64
devment      object
ber          object
location     object
price         int64
dtype: object
```

In [8]:

```
# Summary statistics
df.describe(include="all")
```

Out[8]:

	flarea	type	bdrms	bthrms	floors	devment	ber
count	207.000000	207	207.000000	207.000000	207.000000	207	207
unique	NaN	4	NaN	NaN	NaN	2	12
top	NaN	Semi-detached	NaN	NaN	NaN	SecondHand	G
freq	NaN	65	NaN	NaN	NaN	204	25
mean	128.094686	NaN	3.434783	2.106280	1.826087	NaN	NaN
std	73.970582	NaN	1.232390	1.185802	0.379954	NaN	NaN
min	41.800000	NaN	1.000000	1.000000	1.000000	NaN	NaN
25%	82.650000	NaN	3.000000	1.000000	2.000000	NaN	NaN
50%	106.000000	NaN	3.000000	2.000000	2.000000	NaN	NaN
75%	153.650000	NaN	4.000000	3.000000	2.000000	NaN	NaN
max	497.000000	NaN	10.000000	10.000000	2.000000	NaN	NaN

In [9]:

```
# A few of the examples
df.head(3)
```

Out[9]:

	flarea	type	bdrms	bthrms	floors	devment	ber	location	price
0	497.0	Detached	4	5	2	SecondHand	B2	Carrigrohane	975
1	83.6	Detached	3	1	1	SecondHand	D2	Glanmire	195
2	97.5	Semi-detached	3	2	2	SecondHand	D1	Glanmire	225

Using a scikit-learn Pipeline

- This pipeline will contain only one step: a class for selecting certain features (columns) from a pandas DataFrame, and converting to a numpy array (which is what scikit-learn uses)
- Normally, a pipeline will contain more than one step (see later examples)

In [10]:

```
# The features we want to select
features = ["flarea", "bdrms", "bthrms"]

# Create the pipeline
pipeline = Pipeline([
        ("selector", DataFrameSelector(features))
    ])
```

In [11]:

```
# Run the pipeline
pipeline.fit(df)
X = pipeline.transform(df)
```

In [12]:

```
# Let's take a look at a few rows in X - to show you that we now have a 2D numpy
 array
X[:3]
```

Out[12]:

```
array([[ 497. ,    4. ,    5. ],
       [  83.6,    3. ,    1. ],
       [  97.5,    3. ,    2. ]])
```

Similarity & Distance

- In AI, we often want to know how *similar* one object is to another
  - E.g. how similar is my house to yours
  - E.g. which house in our dataset is most similar to yours
- In fact, here we are instead going to measure how *different* they are using a **distance function**
  - (N.B. This is not about geographical distance)
- Let $\boldsymbol{x}$ be one vector of feature values and $\boldsymbol{x}'$ be another
- Simplest is to measure their **Euclidean distance**:

$$d(\boldsymbol{x}, \boldsymbol{x}') = \sqrt{(\boldsymbol{x}_1 - \boldsymbol{x}'_1)^2 + (\boldsymbol{x}_2 - \boldsymbol{x}'_2)^2 + \ldots + (\boldsymbol{x}_n - \boldsymbol{x}'_n)^2}$$

  or, more concisely:

$$d(\boldsymbol{x}, \boldsymbol{x}') = \sqrt{\sum_{j=1}^{n} (\boldsymbol{x}_j - \boldsymbol{x}'_j)^2}$$

- Euclidean distance has a minimum value of 0 (meaning identical) but no maximum value (depends on your data)
- Class exercise. What is the Euclidean distance between $\boldsymbol{x} = \begin{bmatrix} 100 \\ 1 \\ 4 \end{bmatrix}$ and $\boldsymbol{x}' = \begin{bmatrix} 100 \\ 5 \\ 1 \end{bmatrix}$?

Euclidean Distance in numpy

- It has a nice vectorized implementation (no loop!) using numpy:

In [13]:

```
def euc(x, xprime):
    return np.sqrt(np.sum((x - xprime)**2))
```

In [14]:

```
# Example
your_house = np.array([114.0, 3, 2])
my_house = np.array([107.0, 3, 1])

euc(your_house, my_house)
```

Out[14]:

```
7.0710678118654755
```

- We can compute the distance between your house and all the houses in X
- (We have to write a loop here, because our `euc` function is not vectorized)

In [15]:

```
dists = [euc(your_house, x) for x in X]
```

In [16]:

```
# Just to show you, here are the first 3 distances
dists[:3]
```

Out[16]:

```
[383.01305460780316, 30.4164429215515, 16.5]
```

In [17]:

```
# Even better, we can, with one line of code, find the most similar house
np.min([euc(your_house, x) for x in X])
```

Out[17]:

```
1.5620499351813331
```

In [18]:

```
# Even better again, we can find which house is the most similar
np.argmin([euc(your_house, x) for x in X])
```

Out[18]:

```
25
```

In [19]:

```
# Best of all, we can display the most similar house
df.ix[np.argmin([euc(your_house, x) for x in X])]
```

Out[19]:

```
flarea              115.2
type         Semi-detached
bdrms                   4
bthrms                  2
floors                  2
devment        SecondHand
ber                    D2
location          Douglas
price                 385
Name: 25, dtype: object
```

Problems with Euclidean distance

- There are at least two problems with Euclidean distance (and many other distance measures too):
  - Features with different scales
  - The curse of dimensionality (next lecture)

Scaling Numeric Values

- Different numeric-valued features often have very different ranges
  - E.g. 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
- When computing the Euclidean distance, features with large ranges will dominate the distance calculations, thus giving features with small ranges negligible influence.
- E.g., consider your house $\boldsymbol{x} = \begin{bmatrix} 114 \\ 3 \\ 2 \end{bmatrix}$ and two others, $\boldsymbol{y} = \begin{bmatrix} 119 \\ 3 \\ 2 \end{bmatrix}$ and $\boldsymbol{z} = \begin{bmatrix} 114 \\ 7 \\ 2 \end{bmatrix}$.
  - *Intuitively*, which house is more similar to yours, $\boldsymbol{y}$ or $\boldsymbol{z}$?
  - Now compute the Euclidean distances
  - According to these distances, which house is more similar to yours?
- The solution is to **scale** (or 'normalize') the values so that they have similar ranges
- We'll discuss two ways to do this:
  - Min-max sclaing
  - Standardization

Min-Max Scaling

- Suppose we want to scale feature j
- Let 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 $\boldsymbol{x}_j$ by max_j:
$$\boldsymbol{x}_j \leftarrow \frac{\boldsymbol{x}_j}{max_j}$$
    - E.g. suppose no house will be above 500 square metres
    - So you divide values by 500
- Suppose your domain expert also supplies a minimum possible value min_j
- Then a slightly improved way to scale to $[0, 1]$ is to subtract the minimum value and divide by the range:
$$\boldsymbol{x}_j \leftarrow \frac{\boldsymbol{x}_j - min_j}{max_j - min_j}$$
    - Suppose the smallest houses are 40 square metres and the largest are 500 square metres
    - So we subtract 40 and divide by $500 - 40$

  This is called **min-max scaling**

Min-Max Scaling in scikit-learn

- scikit-learn provides a class called `MinMaxScaler`, which does something similar:
    - Above, we said we should use the smallest *possible* value and 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?
- We can include the scaler as a step in our pipeline

In [20]:

```
# The features we want to select
features = ["flarea", "bdrms", "bthrms"]

# Create the pipeline
pipeline = Pipeline([
        ("selector", DataFrameSelector(features)),
        ("scaler", MinMaxScaler())
    ])
```

In [21]:

```
# Run the pipeline
pipeline.fit(df)
X = pipeline.transform(df)
```

In [22]:

```
# Let's take a look at a few rows in X
X[:3]
```

Out[22]:

```
array([[ 1.        ,  0.33333333,  0.44444444],
       [ 0.09182777,  0.22222222,  0.        ],
       [ 0.1223638 ,  0.22222222,  0.11111111]])
```

In [23]:

```
# Let's scale your house too
# Don't try to understand or copy this code - it's a hack that you won't need
your_house_df = pd.DataFrame([{"flarea":114.0, "bdrms":3, "bthrms":2}])
your_house_scaled = pipeline.transform(your_house_df)[0]
your_house_scaled
```

Out[23]:

```
array([ 0.1586116 ,  0.22222222,  0.11111111])
```

In [24]:

```
# To see what effect this has had, let's see which house is most similar to your
s
np.argmin([euc(your_house_scaled, x) for x in X])
```

Out[24]:

```
23
```

In [25]:

```
# Let's look at its features
df.ix[np.argmin([euc(your_house_scaled, x) for x in X])]
```

Out[25]:

```
flarea              112.4
type         Semi-detached
bdrms                   3
bthrms                  2
floors                  2
devment        SecondHand
ber                    C2
location        Blackrock
price                 225
Name: 23, dtype: object
```

Standardization

- In some cases, you don't want feature values 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 better still is **standardization**, in which you subtract the mean and divide by the standard deviation:
  $$\boldsymbol{x}_j \leftarrow \frac{\boldsymbol{x}_j - \mu_j}{\sigma_j}$$
  where μ_j is the mean of the values for feature j and σ_j is their standard deviation
- If you use this, then the mean will be approximately zero, the standard deviation will be 1

Standardization in scikit-learn

- scikit-learn provides a class called `StandardScaler`
- It uses means and standard deviations that it calculates from your dataset (statisticians would say that it should use the population mean and standard deviation, but these are generally not known)
- We can include the scaler as a step in our pipeline

In [26]:

```
# The features we want to select
features = ["flarea", "bdrms", "bthrms"]

# Create the pipeline
pipeline = Pipeline([
        ("selector", DataFrameSelector(features)),
        ("scaler", StandardScaler())
    ])
```

In [27]:

```
# Run the pipeline
pipeline.fit(df)
X = pipeline.transform(df)
```

In [28]:

```
# Let's take a look at a few rows in X
X[:3]
```

Out[28]:

```
array([[ 4.99927973,  0.45974713,  2.4462228 ],
       [-0.6029769 , -0.35365164, -0.93520037],
       [-0.41460881, -0.35365164, -0.08984458]])
```