Practical 6: Instance-Based Learning

1 Overview

The aim of these lab exercises is to give you some hands-on experience with two advanced forms of **clustering**: Instance-based modelling, which lets you deal with nominal (categorical) values as well as numeric; and Gaussian Mixture modelling, a probabilistic form of clustering.

2 Getting Started

For this lab, you will use some data provided. Download from the KEATS page the zip archive containing the data for this practical. You will find one data file in the archive, called **house-prices.csv**. This data set contains a sample of house prices from a US metropolitan area. The data contains the following attributes:

attribute	*data type*	*description*
`HomeID`	int64	unique property identifier
`Price`	int64	selling price
`SqFt`	int64	size in square feet
`Bedrooms`	int64	number of bedrooms
`Bathrooms`	int64	number of bathrooms
`Offers`	int64	number of offers made on the property since it was listed
`Brick`	object	indication of whether the house is made of brick ('Yes' if it is or 'No' otherwise)
`Neighborhood`	object	the location of the house relative to the city centre

$\rightarrow$ **Write a Python script** that reads in the data from the file, **house-prices.csv**.

Explore the data:

- How many instances are there?
- How many attributes are there?
- What are the names of the attributes?
- What are the data types of the attributes?

3 Build a Distance Matrix

First, we'll build a simple distance matrix by hand.

$\rightarrow$ **Extend your Python script** so that it contains the following:

- A function that returns the squared difference between two numeric scalars:

$$d = (v_0 - v_1)^2$$

- A function that returns the distance between two nominal scalars:

```
if      $(v0 == v1)$
            $d = 0$
else
            $d = 1$
```

- A function that computes the Euclidean distance between two instances, X_0 and X_1, by taking the square root of the sum of the distances between each attribute value:

$$d = \sqrt{\sum_{i=0}^{N} d(X_0[i], X_1[i])}$$

  where N is the number of attributes in each instance

→ Now use your distance functions to compute the pairwise distance between all instances in the data set.

Here are some hints to make this process a bit faster and easier:

- The distance between an instance and itself is always 0, i.e., $d(X_0, X_0) = 0$.
- The distance between two instances is symmetrical, i.e., $d(X_0, X_1) = d(X_1, X_0)$.
- Ignore the `HomeID` field because it is not an attribute that will be useful for comparing distances between instances.

→ Plot a **heatmap** illustrating the distance matrix between all instances in the data set.

A heatmap is a mesh where the colour of each cell indicates the distance between the instances with indexes corresponding to the cell's row and column number. Usually, a heatmap is printed with a gradient colour map so that the largest distances are one extreme of the colour map and the smallest distances are at the other extreme.

My Euclidean distance heatmap is shown in Figure 1a.

→ Extend your code to compute the Manhattan distance between instances.

You will need to write a couple of additional functions:

- A function that returns the absolute value of the difference between two numeric scalars:

$$d = |v_0 - v_1|$$

- A function that computes the Manhattan distance between two instances, X_0 and X_1, by taking the sum of the distances between each attribute value:

$$d = \sum_{i=0}^{N} d(X_0[i], X_1[i])$$

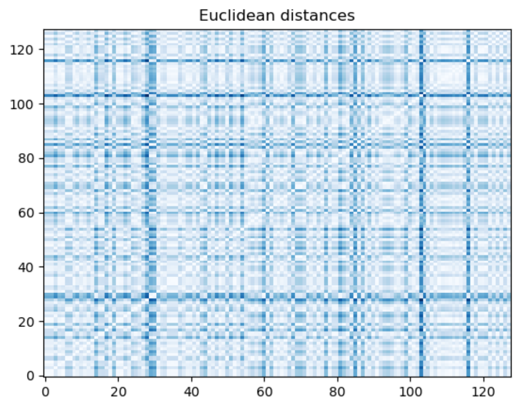
  where N is the number of attributes in each instance

Plot another heatmap, this time showing the Manhattan distances. Mine is shown in Figure 1b.

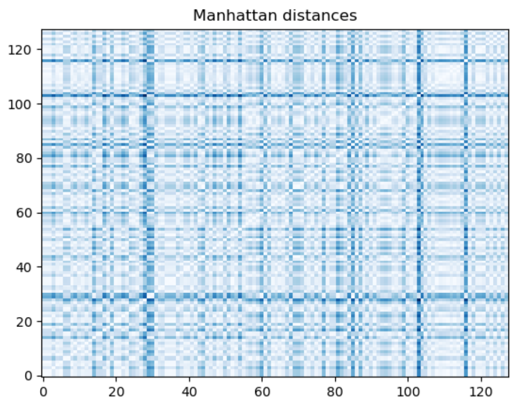
→ You will likely notice that the two heatmaps in Figure 1 are pretty similar to each other. This is because Euclidean distance and Manhattan distance produce values that are similar, relatively, across the data set.

You can check your values by printing them all. Or a quicker and easier way is to compute some statistics about the distances. My values are:

```
Euclidean: mean distance = 30055.35 (23007.5269), minimum distance = 0.00,
    maximum distance = 142102.48
Manhattan: mean distance = 30291.34 (23057.3367), minimum distance = 0.00,
    maximum distance = 142945.00
```



(a) Euclidean distance



(b) Manhattan distance

Figure 1: The distance between all instances in the data set, computed manually. Darker values indicate larger distances between instances and lighter values indicate smaller distances. The diagonal is white, because all those distances are 0.

4 Build an Instance-Based Model

Use the distance matrix computed in part 3 to facilitate an instance-based model.

Suppose you are looking to buy a house in this area. Here are the attributes you desire in your house:

SqFt	=	2050
Bedrooms	=	2
Bathrooms	=	1
Offers	=	2
Brick	=	'No'
Neighborhood	=	'East'

→ **Find the 5 nearest neighbours** to your desired house. Compute the average price of those neighbours to give yourself an idea of how much you would have to spend for this house.

Hint: Leave out the `Price` attribute when computing the nearest neighbours, since you are not using that as a criteria for finding the closest instances.

5 Numeric Variation

Sometimes (as you'll see in part 6) you need to have all the attributes represented numerically. You can continue to ignore `HomeID` field, so you only need to encode two fields: `Brick` and `Neighborhood`.

The first one (`Brick`) is easy, because it is Boolean: either 'Yes' or 'No'. It is common practice to encode `False` values as 0 and `True` values as 1 (which has to do with the logic of binary switches, but we don't need to go into that here). So encode the `Brick` values in the data set so that 'No' values are 0 and 'Yes' values are 1.

The second one (`Neighborhood`) is less obvious. You could encode 'North', 'South', 'East', 'West' as 0, 1, 2 and 3. Then the distance between pairs of these values would really be pretty meaningless. But since the values are indicative of the location of the house with respect to the city centre, we can encode them using a more intuitive encoding: 'East' $= 0$, 'North' $= 90$', 'West' $= 180$ and 'South' $= 270$. If we assume that the city centre is at the origin of a 2D grid, then we could encode each of the compass directions as the angle from the city centre. This would have some real meaning with respect to distance, even though it is approximate angular distance—at least it is more meaningful than $0 \ldots$.

→ **Write a new Python script in which you encode the data set** so that all attributes are numeric.

Re-compute the distance matrices using both Euclidean and Manhattan distances, and plot the corresponding heatmaps. Mine are shown in Figure 2. Again, they don't look very different from each other or from those in Figure 1. First, remember that the heatmaps are plotting relative distances[1], so even if the absolute distance values are different from one method to another, the relative differences between the methods are insignificant.

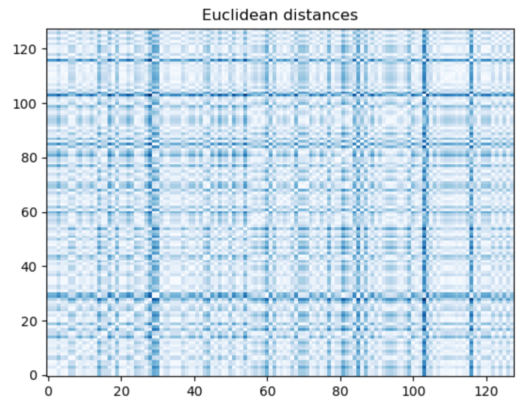
Second, the fields which are now encoded are only two out of 7 fields. Since all fields are equally weighted (in our examples so far), and some of the values will be quite large (e.g., Price), the differences are negligible.

The statistics are not that different:

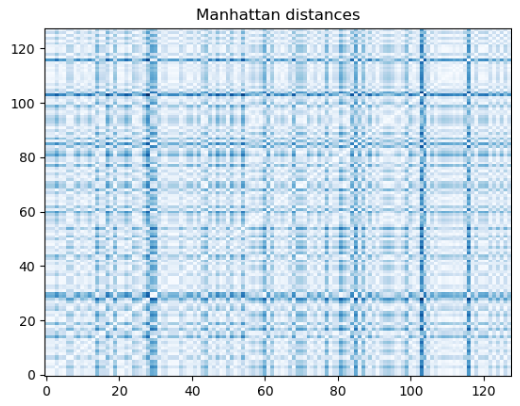
```
Euclidean: mean distance = 30055.83 (23007.1237), minimum distance = 0.00,
    maximum distance = 142102.51
Manhattan: mean distance = 30369.84 (23075.8881), minimum distance = 0.00,
    maximum distance = 143034.00
```

→ **Compute the 5 nearest neighbours** again, this time using your numeric encoding. Are your results different?

[1]Note that you could, technically, normalise all your distance variations in these lab exercises and apply the same (now absolute) colour map to each plot, if you wanted to show absolute difference values in your heatmaps.



(a) Euclidean distance



(b) Manhattan distance

Figure 2: The distance between all instances in the data set, encoded as numeric only and computed manually. Darker values indicate larger distances between instances and lighter values indicate smaller distances.

6 Build a K-Nearest Neighbours Model

Now that you've built an instance-based model by hand, you can now use the one provided by **scikit-learn**, in particular the **NearestNeighbors** package:

`https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html#sklearn.neighbors.NearestNeighbors`

The example code, below, shows how to call it:

```
import sklearn.neighbors as neighbors
nn = neighbors.NearestNeighbors( n_neighbors=K, algorithm='kd_tree', metric='
    euclidean' )
nn.fit( X )
dist, ind = nn.kneighbors( X, return_distance=True )
```

Here are the arguments and values you need to pay attention to:

- The n_neighbors argument in the constructor indicates how many *nearest neighbours* you want your model to consider. If you set that to M, the number of instances in the data set, then you'll get a large model. If you know that you'll only ever be interested in, say, 5 nearest neighbours, then you can set that value to 5 (or whatever number is relevant).
- The metric='euclidean' argument specifies which measure you wish to use in computing the distance between instances. There are quite a few options available, but we'll stick with Euclidean for now, so that you can compare your results to those from part 3.
- The X argument is the $M \times N$ encoded data set, where M is the number of instances and N is the number of attributes. Remember to leave out `HomeID`.
- The values returned by the nn.kneighbors() function are, respectively, the distances between the instances in the first argument (X) and the data set used to build the model (i.e., the argument

to nn. fit (), which in our example is the same) in ascending order from the nearest neighbour to furthest, and the indices of those neighbours in the first argument (X).

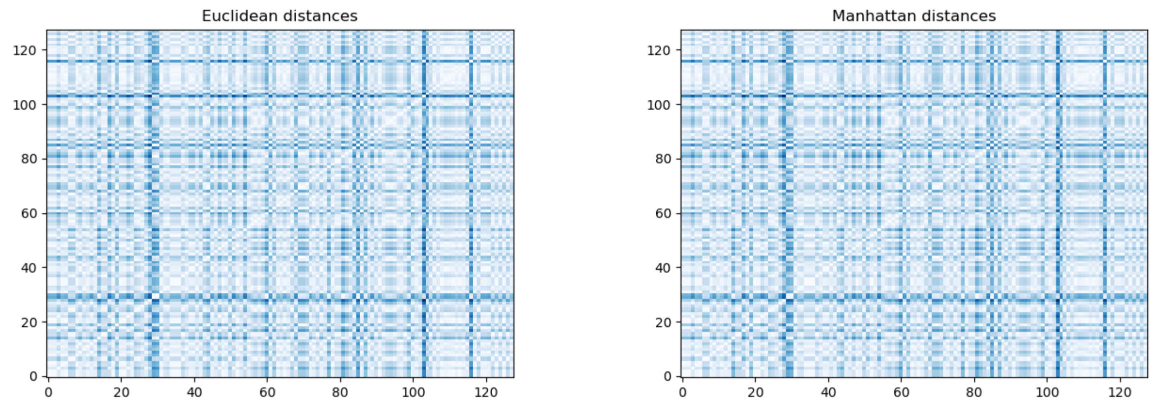
→ **Build a kNN model using scikit** and get a list of the distances between all instances in the data set. Use that list to construct heatmaps, as you did earlier.

→ **Try setting the metric argument** to metric='manhattan' and repeat the step above.

My heatmaps are shown in Figure 3. These should not look different from Figure 2.

And your statistics should be the same as in part 5:

```
Euclidean: mean distance = 30055.83 (23007.1237), minimum distance = 0.00,
    maximum distance = 142102.51
Manhattan: mean distance = 30369.84 (23075.8881), minimum distance = 0.00,
    maximum distance = 143034.00
```



(a) Euclidean distance (b) Manhattan distance

¿

Figure 3: The distance between all instances in the data set, encoded as numeric only and computed using kNN. Darker values indicate larger distances between instances and lighter values indicate smaller distances.

→ **Compute the 5 nearest neighbours** again, this time using scikit.

In order to do this, you initialise and fit the model the same way as before, but you call the function nn.kneighbors() with a different first argument:

```
dist, ind = nn.kneighbors( my_home, return_distance=True )
```

where my_home is one instance representing the all-numeric encoding of your desired home, as described at the end of part 4.

Note that you can leave out `Price` in the model altogether, to make things easier (which means that you'll actually initialise and fit the model slightly differently because you'll remove one attribute from the model). And because you know here that you are only interested in the 5 nearest neighbours, you could also change K to 5 before you initialise the model.

7 Build a Gaussian Mixture Model

For this last exercise, you will build a simple **Gaussian Mixture Model (GMM)** using **sklearn**:

`https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html#sklearn.mixture.GaussianMixture`

→ **Prepare your data.** To keep things simple, we'll create a 2-attribute version of the housing data set: using `Price` and `SqFt`. Plot the values to see what they look like, as in Figure 4a.

→ **Initialise and fit the GMM.** The sample code below shows how to initialise the GMM and fit it to your data. I used $K = 2$ components. The argument `X` to `gmm.fit()` is the 2-dimensional array of price and square footage data.

```
import sklearn.mixture as mixture
gmm = mixture.GaussianMixture( n_components=K, covariance_type='spherical' )
gmm.fit( X )
```

→ **Plot predictions.** You can now use your GMM to assign the points to clusters. First, you can find the centres of the clusters by looking at the values of the model parameter `gmm.means_`. My values are listed below (attribute 0 is `Price` and attribute 1 is `SqFt`):

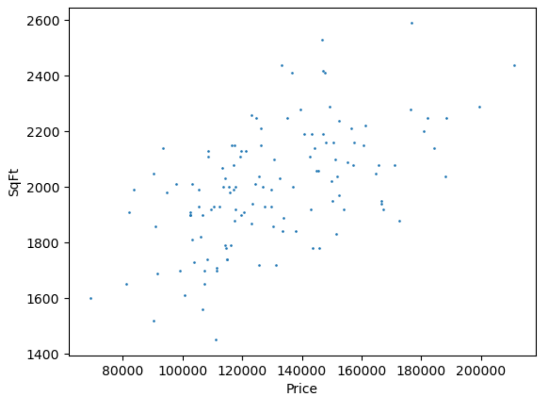
```
component 0: attribute 0, mean=111950.768129 attribute 1, mean=1917.275083
component 1: attribute 0, mean=156374.170409 attribute 1, mean=2118.425409
```

You can use the `gmm.predict()` function to return a list of the most likely component for each instance in the data set:

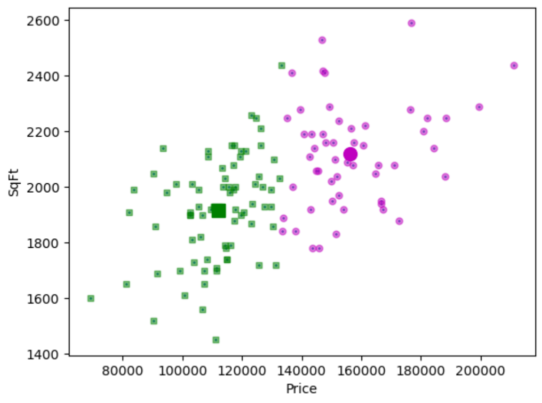
```
labels = gmm.predict( X )
```

This will return a one-dimensional array of length M, where M is the number of instances in the data set. The return `labels` are either 0 or 1, indicating the component corresponding to each instance.

Plot the predictions, as I have in Figure 4b. I used `alpha=0.5` when I plotted the labels, so that the instances can be seen as well.



(a) Raw data



(b) Components

Figure 4: Sample Gaussian Mixture Model

→ **How good is your model?** You can use the gmm.score() function that is part of the **scikit** package:

```
#-compute log-likelihood score (maximise)
ll_score = gmm.score( X )
```

This produces the **log-likelihood score** for the model. Remember from Lecture 5 that we want to **maximise** the log-likelihood score.

You can also use the other metrics we used for clustering in last week's practical:

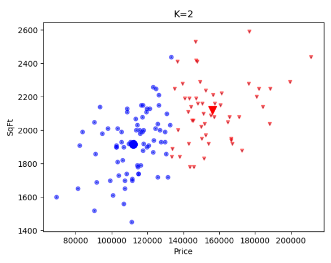
```
#-compute silhouette score (maximise)
sc_score = metrics.silhouette_score( X, labels, metric='euclidean' )

#-compute calinski-harabaz score (maximise)
ch_score = metrics.calinski_harabaz_score( X, labels )
```

Which value of K gives you the best and worst scores for this data set?

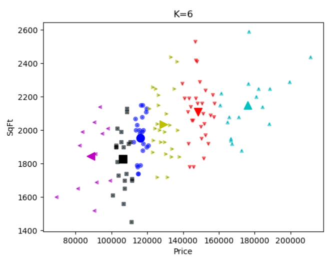
My results are shown in Table 1 and some plots are in Figure 5.

K	*log-likelihood (maximise)*	*silhouette (maximise)*	*calinski-harabaz (maximise)*
2	-21.98	**0.6057**	272.84
3	-21.65	0.5358	233.63
4	-21.37	0.5317	355.31
5	-21.07	0.5455	393.81
6	-20.96	0.5377	456.40
7	-20.68	0.5366	508.21
8	-20.61	0.5473	581.46
9	**-20.50**	0.5338	**634.15**

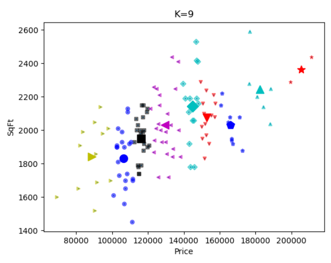
Table 1: Sample scores for different numbers of GMM components (K). The best scores in each category are highlighted in **bold**. The worst scores in each category are underlined.



(a) K=2



(b) K=6



(c) K=9

Figure 5: Sample GMM results with different numbers of components (K).