

Practical 12: Grouping Data


Aggregation, Classification & Clustering

Table of contents

1	Preamble	2
2	Load Data	5
3	Aggregate Listings by MSOA	7
4	Pivot Tables & 'Wide Data'	12
5	First K-Means Clustering	15
6	Second K-Means Clustering	17
7	DBSCAN	22
8	Self-Organising Maps	26
9	Classification	26

A common challenge in data analysis is how to group observations in a data set together in a way that allows for generalisation: *this* group of observations are similar to one another, *that* group is dissimilar to this group. Sometimes we have a *label* that we can use as part of the process (in which case we're doing **classification**), and sometimes we don't (in which case we're doing **clustering**). But what defines similarity and difference? There is no *one* answer to that question and so there are many different ways to cluster or classify data, each of which has strengths and weaknesses that make them more, or less, appropriate in different contexts.

Note

 **Connections:** This practical pulls together many topics covered in other modules, and many of the ideas covered elsewhere in *this* module: clustering, reproducibility, dimensionality reduction... but, above all, this practical is about the importance of **judgement**. Do *not* take what we've done here as the ONE RIGHT WAY: a number of these results are questionable at best because we haven't developed or defined an underlying hypothesis informed by a critical appraisal of the data. You should be *much* more selective in how you deploy the data and the algorithms, as last week's [session on dimensionality](#) should have shown.

1 Preamble

```
import warnings # This suppresses some meaningless errors from Seaborn and Pandas
warnings.simplefilter(action='ignore', category=FutureWarning)
```

```
import numpy as np
import pandas as pd
import geopandas as gpd
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib as mpl
import re
import os

from matplotlib.colors import ListedColormap

# All of these are potentially useful, though
# not all have been used in this practical --
# I'd suggest exploring the use of different
# Scalers/Transformers as well as clustering
# algorithms...
from sklearn.neighbors import NearestNeighbors
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler, PowerTransformer
from sklearn.cluster import KMeans, DBSCAN, OPTICS
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import silhouette_samples, silhouette_score

import random
random.seed(42) # For reproducibility
np.random.seed(42) # For reproducibility

# Make numeric display a bit neater
pd.set_option('display.float_format', lambda x: '{:,.2f}'.format(x))
```


1.0.1 Initialise the Scaler(s)

Remember that you can set up the `sklearn` transformers in advance, and then fit them before transform-ing them.

```
mms = MinMaxScaler(feature_range=(-1,1))
stds = StandardScaler()
rbs = RobustScaler()
pts = PowerTransformer()
```

1.0.2 Set Up Plotting Functions

Note

 **Connections:** Here's an example of how you can use a function to do something a little more complex than just locally save some data. This is still largely a kind of 'stub', but if you are going to be producing a lot of plots of London why not automate away some of the pain of producing a good-looking basemap *each time*: use a function to apply the formatting and then just return `fig` and `ax` as if you'd done this all yourself.

```
def plt_ldn(w, b):
    """
    Creates a new figure of a standard size with the
    water (w) and boundary (b) layers set up for easy
    plotting. Right now this function assumes that you're
    looking at London, but you could parameterise it in
    other ways ot allow it to work for other areas.

    w: a water layer for London
    b: a borough (or other) boundary layer for London
    """
    fig, ax = plt.subplots(1, figsize=(14, 12))
    w.plot(ax=ax, color='#79aef5', zorder=2)
    b.plot(ax=ax, edgecolor='#cc2d2d', facecolor='None', zorder=3)
    ax.set_xlim([502000, 563000])
    ax.set_ylim([155000, 201500])
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.spines['bottom'].set_visible(False)
    ax.spines['left'].set_visible(False)
    return fig, ax

#####
# These may no longer be relevant because of changes to geopandas API

def default_cmap(n, outliers=False):
    cmap = mpl.cm.get_cmap('viridis_r', n)
    colors = cmap(np.linspace(0,1,n))
    if outliers:
        gray = np.array([225/256, 225/256, 225/256, 1])
        colors = np.insert(colors, 0, gray, axis=0)
    return ListedColormap(colors)

# mappable = ax.collections[-1] if you add the geopandas
# plot last.
def add_colorbar(mappable, ax, cmap, norm, breaks, outliers=False):
    cb = fig.colorbar(mappable, ax=ax, cmap=cmap, norm=norm,
                      boundaries=breaks,
                      extend=('min' if outliers else 'neither'),
```

```

        spacing='uniform',
        orientation='horizontal',
        fraction=0.05, shrink=0.5, pad=0.05)
cb.set_label("Cluster Number")

```

1.0.3 Set up Caching Function

```

import os
from requests import get
from urllib.parse import urlparse

def cache_data(src:str, dest:str) -> str:
    """Downloads and caches a remote file locally.

    The function sits between the 'read' step of a pandas or geopandas
    data frame and downloading the file from a remote location. The idea
    is that it will save it locally so that you don't need to remember to
    do so yourself. Subsequent re-reads of the file will return instantly
    rather than downloading the entire file for a second or n-th itme.

    Parameters
    -----
    src : str
        The remote *source* for the file, any valid URL should work.
    dest : str
        The *destination* location to save the downloaded file.

    Returns
    -----
    str
        A string representing the local location of the file.
    """

    url = urlparse(src) # We assume that this is some kind of valid URL
    fn = os.path.split(url.path)[-1] # Extract the filename
    dfn = os.path.join(dest,fn) # Destination filename

    # Check if dest+filename does *not* exist --
    # that would mean we have to download it!
    if not os.path.isfile(dfn) or os.path.getsize(dfn) < 1:

        print(f"{dfn} not found, downloading!")

        # Convert the path back into a list (without)
        # the filename -- we need to check that directories
        # exist first.
        path = os.path.split(dest)

        # Create any missing directories in dest(ination) path

```

```

# -- os.path.join is the reverse of split (as you saw above)
# but it doesn't work with lists... so I had to google how
# to use the 'splat' operator! os.makedirs creates missing
# directories in a path automatically.
if len(path) >= 1 and path[0] != '':
    os.makedirs(os.path.join(*path), exist_ok=True)

# Download and write the file
with open(dfn, "wb") as file:
    response = get(src)
    file.write(response.content)

print('Done downloading...')

else:
    print(f"Found {dfn} locally!")

return dfn

```

2 Load Data

2.1 London Data Layers

 Difficulty: Low.

```

spath = 'https://github.com/jreades/fsds/blob/master/data/src/' # source path
ddir = os.path.join('data', 'geo') # destination directory
water = gpd.read_file( cache_data(spath+'Water.gpkg?raw=true', ddir) )
boros = gpd.read_file( cache_data(spath+'Boroughs.gpkg?raw=true', ddir) )
green = gpd.read_file( cache_data(spath+'Greenspace.gpkg?raw=true', ddir) )

msoas = gpd.read_file( cache_data('http://orca.casa.ucl.ac.uk/~jreades/data/MSOA-2011.gpkg') )
msoas = msoas.to_crs(epsg=27700)

# I don't use this in this practical, but it's a
# really useful data set that gives you 'names'
# for MSOAs that broadly correspond to what most
# Londoners would think of as a 'neighbourhood'.
msoa_nms = gpd.read_file( cache_data('http://orca.casa.ucl.ac.uk/~jreades/data/MSOA-2011-nms.gpkg') )
msoa_nms = msoa_nms.to_crs(epsg=27700)
print("Done.")

```

2.2 Reduced Dimensionality MSOA Data

💡 Difficulty: Low.

You should have this locally from last week, but just in case...

```
host = 'http://orca.casa.ucl.ac.uk'
path = '~jreades/data'
rddf = gpd.read_parquet( cache_data(f'{host}/{path}/Reduced_Dimension_Data.geoparquet')
print(f"Data frame is {rddf.shape[0]:,} x {rddf.shape[1]:,}")
```

You should have: Data frame is 983 x 93.

And below you should see both the components and the dimensions from last week's processing.

```
rddf.iloc[0:3, -7:]
```

I get the results below, but note that the **Dimension values** may be slightly different:

	Component 5	Component 6	Component 7	Borough	Dimension 1	Dimension 2	Subregion
E02000001	1.44	3.95	-1.52	City of Lon- don	7.74	3.36	Inner West
E02000002	-0.28	0.89	0.26	Barking and Da- gen- ham	2.04	7.59	Outer East and North East
E02000003	-0.11	1.12	0.83	Barking and Da- gen- ham	2.20	6.87	Outer East and North East

2.3 Listings Data

💡 Difficulty: Low.

Let's also get the listings data from a few weeks back:

```
# Set download URL
ymd = '2024-06-14'
host = 'https://orca.casa.ucl.ac.uk'
```

```
url = f'{host}/~jreades/data/{ymd}-listings.geoparquet'
```

```
listings = gpd.read_parquet( cache_data(url, ddir) )  
listings = listings.to_crs(epsg=27700)  
print(f"Data frame is {listings.shape[0]:,} x {listings.shape[1]:,}")
```


You should have: Data frame is 85,134 x 31.

And a quick plot of the price to check:

```
listings.plot(???, cmap='plasma', scheme='quantiles', k=10,  
              markersize=.5, alpha=0.15, figsize=(10,7));
```

3 Aggregate Listings by MSOA

3.1 Join Listings to MSOA

 **Difficulty: Medium-to-hard.**

First, let's link all this using the MSOA Geography that we created last week and a mix or merge and sjoin!

Note

Connections: Notice a few things going on here! We are calling `gpd.sjoin`

```
# Before the spatial join  
listings.columns
```

```
msoa_listings = gpd.sjoin(???, msoas.drop(  
    columns=['MSOA11NM', 'LAD11CD', 'LAD11NM', 'RGN11CD', 'RGN11NM',  
            'USUALRES', 'HHOLDRES', 'COMESTRES', 'POPDEN', 'HHOLDRES',  
            'AVHHOLDSZ']), predicate='???').drop(  
    columns=['latitude', 'longitude', 'index_right']  
)
```


```
# All we've added is the MSOA11CD  
msoa_listings.columns
```

All being well you should now have:

```
Index(['listing_url', 'last_scraped', 'name', 'description', 'host_id',  
      'host_name', 'host_since', 'host_location', 'host_is_superhost',  
      'host_listings_count', 'host_total_listings_count',  
      'host_verifications', 'property_type', 'room_type', 'accommodates',  
      'bathrooms_text', 'bedrooms', 'beds', 'amenities', 'price',
```

```
'minimum_nights', 'maximum_nights', 'availability_365',
'number_of_reviews', 'first_review', 'last_review',
'review_scores_rating', 'reviews_per_month', 'geometry', 'MSOA11CD'],
dtype='object')
```

3.2 Price by MSOA

 Difficulty: Medium.

Let's calculate the median price by MSOA... Notice that we have to specify the column we want after the `groupby` so the we don't get the median of *every* column returned

Note


`**🔗 Connections**`: I find ``groupby`` to be a complex operation and often need a co

```
# *m*soa *l*istings *g*rouped by *p*rice
mlgp = msoa_listings.groupby('???')['price'].agg('???')
mlgp.head()
```

You should get something like:

```
MSOA11CD
E02000001    170.00
E02000002     97.00
E02000003     80.00
E02000004     54.00
E02000005    100.00
Name: price, dtype: float64
```

3.3 Room Type by MSOA

 Difficulty: Medium.

Now let's calculate the count of room types by MSOA and compare the effects of `reset_index` on the outputs below. And notice too that we can assign the aggregated value to a column name!

```
# *m*soa *l*istings *g*rouped *c*ount
mlgc = msoa_listings.groupby(['???', '???'], observed=False).listing_url.agg(Count='?')
mlgc.head()
```

You should get something resembling this:


MSOA11CD	room_type	Count
E02000001	Entire home/apt	466
	Hotel room	0
	Private room	61
	Shared room	1
E02000002	Entire home/apt	4

```
# *m*soa *l*istings *g*rouped *c*ount *r*eset index
mlgr = msoa_listings.groupby(['???', '???'], observed=False).listing_url.agg(Count='
mlgr.head()
```

You should get something like:

	MSOA11CD	room_type	Count
0	E02000001	Entire home/apt	466
1	E02000001	Hotel room	0
2	E02000001	Private room	61
3	E02000001	Shared room	1
4	E02000002	Entire home/apt	4

3.4 Price by Room Type

 Difficulty: Hard.

But perhaps median price/room type would make more sense? And do we want to retain values where there are no listings? For example, there are no hotel rooms listed for E02000001, how do we ensure that these *NAs are dropped*?

```
# *m*soa *l*istings *g*rouped *r*oom *p*rice
mlgrp = msoa_listings.???(???, observed=True
                           )['price'].agg('???').reset_index()
mlgrp.head()
```

You should get something like:

	MSOA11CD	room type	price
0	E02000001	Entire home/apt	177.00
2	E02000001	Private room	100.00
3	E02000001	Shared room	120.00
4	E02000002	Entire home/apt	117.00
6	E02000002	Private room	42.00

3.5 Explore Outlier Per-MSOA Prices

⚠ Difficulty: Medium.

Are there MSOAs what *look* like they might contain erroneous data?

3.5.1 Plot MSOA Median Prices

```
mlgp.hist(bins=200);
```

3.5.2 Examine Listings from High-Priced MSOAs

Careful, this is showing the *listings* from MSOAs whose median price is above \$300/night:

```
msoa_listings[
    msoa_listings.MSOA11CD.isin(mlgp[mlgp > 300].index)
].sort_values(by='price', ascending=False).head(7)[
    ['price', 'room_type', 'name', 'description']
]
```

Some of these look legi (4, 5, and... 8 bedroom 'villas?'), though not every one...

And how about these?

```
msoa_listings[
    (msoa_listings.MSOA11CD.isin(mlgp[mlgp > 300].index)) & (msoa_listings.room_type
).sort_values(by='price', ascending=False).head(7)[
    ['price', 'room_type', 'property_type', 'name', 'description']
]
```

If we wanted to be rigorous then we'd have to investigate further: properties in Mayfair and Westminster *are* going to be expensive, but are these plausible nightly prices? In some cases, yes. In others...


```
msoa_listings[
    (msoa_listings.MSOA11CD.isin(mlgp[mlgp < 100].index)) & (msoa_listings.room_type
).sort_values(by='price', ascending=False).head(7)[
    ['price', 'room_type', 'name', 'description']
]
```

On the whole, let's take a *guess* that there are a small number of implausibly high prices for individual units that aren't in very expensive neighbourhoods and that these are either erroneous/deliberately incorrect, or represent a price that is not per-night.

Note

`**🔗 Connections**`: What's the right answer here? There isn't one. You could proba

3.5.3 Filter Unlikely Listings

 Difficulty: Hard.

See if you can filter out these less likely listings on the following criteria:

1. Listings are priced above \$300/night AND
2. Room type is not 'Entire home/apt' AND
3. Listings do *not* contain the words: suite, luxury, loft, stunning, prime, historic, or deluxe.

I found 901 rows to drop this way.

```
target_regex = r'(?:(suite|luxury|loft|stunning|prime|historic|deluxe|boutique) '
to_drop = msoa_listings[
    (???) &
    (???) &
    ~((???(target_regex, flags=re.IGNORECASE, regex=True, na=True)))
print(f"Have found {to_drop.shape[0]:,} rows to drop on the basis of unlikely per ni

to_drop.sort_values(by='price', ascending=False)[['price', 'room_type', 'name', 'descri
```

3.5.4 Plot Unlikely Listings

Here we use the `plt_ldn` function – notice how it's designed to return `f, ax` in the same way that `plt.subplots` (which we're already familiar with) does!

```
f, ax = plt_ldn(???, ???)
to_drop.plot(column='price', markersize=10, alpha=0.7, cmap='viridis', ax=ax);
```

3.5.5 ... And Drop

Some might be legitimate, but I'm feeling broadly ok with the remainder.

```
cleaned = msoa_listings.drop(index=to_drop.???)
print(f"Cleaned data has {cleaned.shape[0]:,} rows.")
```

After this I had 84,308 rows.

I would normally, at this point, spend quite a bit of time validating this cleaning approach, but right now we're going to take a rough-and-ready approach.

3.5.6 Questions

- What data type did Task 2.2 return?
- What is the function of `reset_index()` in Task 2.3 and when might you choose to reset (or not)?


4 Pivot Tables & 'Wide Data'

The `group_by` operation is *one* way to organise and aggregate our data, but pivot tables are a *second* common way to achieve this. We typically use a pivot table to go from long to wide data frames – it's often seen as one of Excel's main benefits, but Pandas can do that too!

Note

`**Connections**`: Notice that a pivot table is just a different kind of aggregate

4.1 Create Pivot Table

 Difficulty: Hard.

We can make use of the pivot table function to generate counts by MSOA in a 'wide' format.

```
pivot = cleaned.groupby(
    ['MSOA11CD', 'room_type'], observed=False
).listing_url.agg(Count='count').reset_index().pivot(
    index='???', columns=['???'], values=['???'])
pivot.head(3)
```

The formatting will look a tiny bit different, but you should get something like this:

				Count
room_type	Entire home/apt	Hotel room	Private room	Shared room
MSOA11CD				
E02000001	466	0	55	1
E02000002	4	0	2	0
E02000003	12	0	13	0

4.2 Check Counts

💡 Difficulty: Low.

```
pivot.sum()
```

Just to reassure you that the pivot results 'make sense':

```
print(cleaned[cleaned.room_type=='Entire home/apt'].listing_url.count())
print(cleaned[cleaned.room_type=='Private room'].listing_url.count())
```

4.3 Tidy & Normalise

💡 Difficulty: Low.

My instinct at this point is that, looking at the pivot table, we see quite different levels of Airbnb penetration and it is hard to know how handle this difference: share would be unstable because of the low counts in some places and high counts in others; a derived variable that tells us something about density or mix could be interesting (e.g. HHI or LQ) but wouldn't quite capture the pattern of mixing.

4.3.1 Tidy

Personally, based on the room type counts above I think we can drop Hotel Rooms and Shared Rooms from this since the other two categories are so dominant.

```
# Flatten the column index
pivot.columns = ['Entire home/apt', 'Hotel room', 'Private room', 'Shared room']
# Drop the columns
pivot.drop(???, inplace=True)
pivot.head()
```

You should have only the **Entire home/apt** and **Private room** columns now.

4.3.2 Normalise

```
pivot_norm = pd.DataFrame(index=pivot.index)
for c in pivot.columns.to_list():
    # Power Transform
    pivot_norm[c] = pts.???(pivot[c].to_numpy().reshape(???,???)

pivot_norm.head()
```


You should have something like:

	Entire home/apt	Private room
MSOA11CD		
E02000001	2.20	1.06
E02000002	-1.29	-1.85

4.3.3 Plot

```
pnm = pd.merge(msoas.set_index('MSOA11CD'), pivot_norm, left_index=True, right_index=True)
pnm.plot(column='Entire home/apt', cmap='viridis', edgecolor='none', legend=True, fi
```

4.4 PCA

 Difficulty: Moderate, though you might find the questions hard.

You can merge the output of this next step back on to the `rddf` data frame as part of a clustering process, though we'd really want to do some more thinking about what this data *means* and what transformations we'd need to do in order to make them *meaningful*.

For instance, if we went back to last week's code, we could have appended this InsideAirbnb data *before* doing the dimensionality reduction, or we could apply it now to create a new measure that could be used as a separate part of the clustering process together with the reduced dimensionality of the demographic data.

4.4.1 Perform Reduction

```
pcomp = PCA(n_components=???, random_state=42)
rd     = pcomp.???(pivot_norm)
print(f"The explained variance of each component is: {'', '.join([f'{x*100:.2f}%'] for x in pcomp.explained_variance_ratio_))
```

Take the first component and convert to a series to enable the merge:

```
airbnb_pca = pd.DataFrame(
    {'Airbnb Component 1': mms.fit_transform(rd[:,1]).reshape(-1,1)}.res
    index=pivot.index)

airbnb_pca.head()
```

You should have something like: | | Airbnb Component 1 | | :-- | --: | | **MSOA11CD**
| | | **E02000001** | 0.47 | | **E02000002** | 0.19

```
pcanm = pd.merge(msoas.set_index('MSOA11CD'), airbnb_pca, left_index=True, right_index=True)
pcanm.plot(column='Airbnb Component 1', cmap='viridis', edgecolor='none', legend=True, fi
```

4.4.2 Write to Data Frame

```
# Result Set from merge
rs = pd.merge(rddf, airbnb_pca, left_index=True, right_index=True)
```

Grab the PCA, UMAP, and Airbnb outputs for clustering and append rescaled price:

```
# Merge the reduced dimensionality data frame with the PCA-reduced Airbnb data
# to create the *cluster*data*frame
cldf = pd.merge(rddf.loc[:, 'Component 1:'], airbnb_pca,
               left_index=True, right_index=True)

# Append median price from cleaned listings grouped by MSOA too!
s1 = cleaned.groupby(by='MSOA11CD').price.agg('median')
cldf['median_price'] = pd.Series(np.squeeze(mms.fit_transform(s1.values.reshape(-1,1)))

# Append mean price from cleaned listings grouped by MSOA too!
s2 = cleaned.groupby(by='MSOA11CD').price.agg('mean')
cldf['mean_price'] = pd.Series(np.squeeze(mms.fit_transform(s2.values.reshape(-1,1)))

cldf.drop(columns=['Subregion', 'Borough'], inplace=True)

cldf.head()
```

4.4.3 Questions

- Have a think about why you might want to keep the Airbnb data separate from the MSOA data when doing PCA (or any other kind of dimensionality reduction)!
- Why *might* it be interesting to add *both* mean and median MSOA prices to the clustering process? Here's a hint (but it's very subtle): `sns.jointplot(x=s1, y=s2, s=15, alpha=0.6)`

5 First K-Means Clustering

5.1 Perform Clustering

 Difficulty: Low.

```
c_nm = 'KMeans' # Clustering name
k_pref = ??? # Number of clusters
```

```
kmeans = KMeans(n_clusters=k_pref, n_init=25, random_state=42).fit(cldf.drop(columns=
```

Here are the results:

```
print(kmeans.labels_) # The results
```

5.2 Save Clusters to Data Frame

💡 Difficulty: Low.

5.2.1 Write Series and Assign

Now capture the labels (i.e. clusters) and write them to a data series that we store on the result set `df` (`rs`):

```
rs[c_nm] = pd.Series(kmeans.labels_, index=cldf.index)
```

5.2.2 Histogram of Cluster Members

How are the clusters distributed?

```
sns.histplot(data=???, x=c_nm, bins=k_pref);
```

5.2.3 Map Clusters

And here's a map!

```
fig, ax = plt_ldn(water, boros)
fig.suptitle(f"{c_nm} Results (k={k_pref})", fontsize=20, y=0.92)
rs.plot(column=???, ax=ax, linewidth=0, zorder=0, categorical=???, legend=True);
```


5.2.4 Questions

- What critical assumption did we make when running this analysis?
- Why did I *not* use the UMAP dimensions here?
- Why do we have the `c_nm= 'kMeans'` when we *know* what kind of clustering we're doing?

- Does this look like a *good* clustering?

6 Second K-Means Clustering

6.1 What's the 'Right' Number of Clusters?

 Difficulty: Moderate.

There's more than one way to find the 'right' number of clusters. In Singleton's *Geo-computation* chapter they use WCSS to pick the 'optimal' number of clusters. The idea is that you plot the average WCSS for each number of possible clusters in the range of interest (2...n) and then look for a 'knee' (i.e. kink) in the curve. The principle of this approach is that you look for the point where there is declining benefit from adding more clusters. The problem is that there is always some benefit to adding more clusters (the perfect clustering is $k=n$), so you don't always see a knee.

Another way to try to make the process of selecting the number of clusters a little less arbitrary is called the silhouette plot and (like WCSS) it allows us to evaluate the 'quality' of the clustering outcome by examining the distance between each observation and the rest of the cluster. In this case it's based on Partitioning Around the Medoid (PAM).

Either way, to evaluate this in a systematic way, we want to do multiple k-means clusterings for multiple values of k and then we can look at which gives the best results...

```
kcldf = cldf.drop(columns=['Dimension 1','Dimension 2'])
```

6.1.1 Repeated Clustering

Let's try clustering across a wider range. Because we repeatedly re-run the clustering code (unlike with Hierarchical Clustering) this can take a few minutes. I got nearly 5 minutes on a M2 Mac.

```
%%time

# Adapted from: http://scikit-learn.org/stable/auto\_examples/cluster/plot\_kmeans\_sil

x = []
y = []

# For resolutions of 'k' in the range 2..40
for k in range(2,41):
```

```
#####
# Do the clustering using the main columns
kmeans = KMeans(n_clusters=k, n_init=25, random_state=42).fit(kcldf)

# Calculate the overall silhouette score
silhouette_avg = silhouette_score(kcldf, kmeans.labels_)

y.append(k)
x.append(silhouette_avg)

print('.', end='')
```

6.1.2 Plot Silhouette Scores

```
print()
print(f"Largest silhouette score was {max(x):6.4f} for k={y[x.index(max(x))]}")

plt.plot(y, x)
plt.gca().xaxis.grid(True);
plt.gcf().suptitle("Average Silhouette Scores");
```

Warning

****⚠ Note**:** Had we used the UMAP dimensions here you'd likely see more instability

We can use the largest average silhouette score to determine the ‘natural’ number of clusters in the data, but that that’s only if we don’t have any kind of underlying theory, other empirical evidence, or even just a reason for choosing a different value... Again, we’re now getting in areas where your judgement and your ability to communicate your rationale to readers is the key thing.

6.2 Final Clustering

Difficulty: Low.

So although we should probably pick the largest silhouette scores, that’s $k=3$ which kind of defeats the purpose of clustering in the first place. In the absence of a *compelling* reason to pick 2 or 3 clusters, let’s have a closer look at the *next* maximum silhouetted score:

6.2.1 Perform Clustering

```
k_pref=???
```

```
#####
# Do the clustering using the main columns
kmeans = KMeans(n_clusters=k_pref, n_init=25, random_state=42).fit(kcldf)

# Convert to a series
s = pd.Series(kmeans.labels_, index=kcldf.index, name=c_nm)

# We do this for plotting
rs[c_nm] = s

# Calculate the overall silhouette score
silhouette_avg = silhouette_score(kcldf, kmeans.labels_)

# Calculate the silhouette values
sample_silhouette_values = silhouette_samples(kcldf, kmeans.labels_)
```

6.2.2 Plot Diagnostics

```
#####
# Create a subplot with 1 row and 2 columns
fig, (ax1, ax2) = plt.subplots(1, 2)
fig.set_size_inches(9, 5)

# The 1st subplot is the silhouette plot
# The silhouette coefficient can range from -1, 1
ax1.set_xlim([-1.0, 1.0]) # Changed from -0.1, 1

# The (n_clusters+1)*10 is for inserting blank space between silhouette
# plots of individual clusters, to demarcate them clearly.
ax1.set_ylim([0, kcldf.shape[0] + (k_pref + 1) * 10])

y_lower = 10

# For each of the clusters...
for i in range(k_pref):
    # Aggregate the silhouette scores for samples belonging to
    # cluster i, and sort them
    ith_cluster_silhouette_values = \
        sample_silhouette_values[kmeans.labels_ == i]

    ith_cluster_silhouette_values.sort()

    size_cluster_i = ith_cluster_silhouette_values.shape[0]
    y_upper = y_lower + size_cluster_i
```

```

# Set the color ramp
color = plt.cm.Spectral(i/k_pref)
ax1.fill_betweenx(np.arange(y_lower, y_upper),
                  0, ith_cluster_silhouette_values,
                  facecolor=color, edgecolor=color, alpha=0.7)

# Label the silhouette plots with their cluster numbers at the middle
ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))

# Compute the new y_lower for next plot
y_lower = y_upper + 10 # 10 for the 0 samples

ax1.set_title("The silhouette plot for the clusters.")
ax1.set_xlabel("The silhouette coefficient values")
ax1.set_ylabel("Cluster label")

# The vertical line for average silhouette score of all the values
ax1.axvline(x=silhouette_avg, color="red", linestyle="--", linewidth=0.5)

ax1.set_yticks([]) # Clear the yaxis labels / ticks
ax1.set_xticks(np.arange(-1.0, 1.1, 0.2)) # Was: [-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1

# 2nd Plot showing the actual clusters formed --
# we can only do this for the first two dimensions
# so we may not see fully what is causing the
# resulting assignment
colors = plt.cm.Spectral(kmeans.labels_.astype(float) / k_pref)
ax2.scatter(kcldf[kcldf.columns[0]], kcldf[kcldf.columns[1]],
            marker='.', s=30, lw=0, alpha=0.7, c=colors)

# Labeling the clusters
centers = kmeans.cluster_centers_

# Draw white circles at cluster centers
ax2.scatter(centers[:, 0], centers[:, 1],
            marker='o', c="white", alpha=1, s=200)

for i, c in enumerate(centers):
    ax2.scatter(c[0], c[1], marker='$%d$' % i, alpha=1, s=50)

ax2.set_title("Visualization of the clustered data")
ax2.set_xlabel("Feature space for the 1st feature")
ax2.set_ylabel("Feature space for the 2nd feature")

plt.suptitle(("Silhouette results for KMeans clustering "
             "with %d clusters" % k_pref),
             fontsize=14, fontweight='bold')

plt.show()

```

Warning

****⚠ Stop****: Make sure that you understand how the silhouette plot and value work, and

6.2.3 Map Clusters

```
fig, ax = plt_ldn(water, boros)
fig.suptitle(f"{c_nm} Results (k={k_pref})", fontsize=20, y=0.92)
rs.plot(column=c_nm, ax=ax, linewidth=0, zorder=0, categorical=True, legend=True);
```

6.3 'Representative' Centroids

 **Difficulty**: Moderate since, conceptually, there's a lot going on.

To get a sense of how these clusters differ we can try to extract 'representative' centroids (mid-points of the multi-dimensional cloud that constitutes a cluster). In the case of k-means this will work quite well since the clusters are explicitly built around mean centroids. There's also a k-medoids clustering approach built around the median centroid.

These are columns that we want to suppress from our sample:

```
to_suppress=['OBJECTID', 'BNG_E', 'BNG_N', 'LONG', 'LAT',
             'Shape__Are', 'Shape__Len', 'geometry', 'Component 1',
             'Component 2', 'Component 3', 'Component 4', 'Component 5',
             'Component 6', 'Component 7', 'Dimension 1', 'Dimension 2',
             'Airbnb Component 1']
```

Take a sample of the full range of numeric columns:

```
cols = random.sample(rs.select_dtypes(exclude='object').drop(columns=to_suppress).columns, 10)
print(cols)
```

Calculate the mean of these columns for each cluster:

```
# Empty data frame with the columns we'll need
centroids = pd.DataFrame(columns=cols)

# For each cluster...
for k in sorted(rs[c_nm].unique()):
    print(f"Processing cluster {k}")

    # Select rows where the cluster name matches the cluster number
    clust = rs[rs[c_nm]==k]

    # Append the means to the centroids data frame
    centroids.loc[k] = clust[cols].mean()
```

```
centroids
```

```
centroids_long = pd.DataFrame(columns=['Variable','Cluster','Std. Value'])
for i in range(0,len(centroids.index)):
    row = centroids.iloc[i,:]
    for r in row.index:
        d = pd.DataFrame({'Variable':r, 'Cluster':i, 'Std. Value':row[r]}, index=[1])
        centroids_long = pd.concat([centroids_long, d], ignore_index=True)
```

```
g = sns.FacetGrid(centroids_long, col="Variable", col_wrap=3, height=3, aspect=1.5,
g = g.map(plt.bar, "Cluster", "Std. Value")
```

Note


Connections: The above centroid outputs are a way to think about how each c

7 DBSCAN

For what it's worth, I've had *enormous* trouble with DBSCAN and this kind of data. I don't think it deals very well with *much* more than three dimensions, so the flexibility to not have to specify the number of clusters is balanced with a density-based approach that is severely hampered by high-dimensional distance-inflation.

```
# Drop the PCA dimensions
cldf2 = cldf.loc[:, 'Dimension 1:'].copy()
for c in [x for x in cldf.columns.to_list() if x.startswith('Dimension ')]:
    cldf2[c] = pd.Series(np.squeeze(mms.fit_transform(cldf2[c].to_numpy()).reshape(-1))
cldf2.head()
```

7.1 Work out the Neighbour Distance


 **Difficulty: Moderate.**

We normally look for some kind of 'knee' to set the distance.

```
nbrs = NearestNeighbors(n_neighbors=6).fit(cldf2)
distances, indices = nbrs.kneighbors(cldf2)

distances = np.sort(distances, axis=0)
distances = distances[:,1]
```

7.2 Derive Approximate Knee


 Difficulty: Low.

```
from kneed import knee_locator

kn = knee_locator.KneeLocator(np.arange(distances.shape[0]), distances, S=12,
                             curve='convex', direction='increasing')
print(f"Knee detected at: {kn.knee}")
kn.plot_knee()
kn.plot_knee_normalized()

print(f"Best guess at epsilon for DBSCAN is {distances[kn.knee]:0.4f}")
```

7.3 Explore Epsilons

 Difficulty: Moderate.

There are two values that need to be specified: `eps` and `min_samples`. Both seem to be set largely by trial and error, though we can use the above result as a target. It's easiest to set `min_samples` first since that sets a floor for your cluster size and then `eps` is basically a distance metric that governs how far away something can be from a cluster and still be considered part of that cluster.

7.3.1 Iterate Over Range

 Caution

Warning: Depending on the data volume, this next step may take quite a lot of

```
%%time

c_nm = 'DBSCAN'

# Make numeric display a bit neater
pd.set_option('display.float_format', lambda x: '{:,.4f}'.format(x))

el = []

max_clusters = 10
cluster_count = 1

iters = 0
```

```

for e in np.arange(0.025, 0.76, 0.01): # <- You might want to adjust these!

    if iters % 25==0: print(f"{iters} epsilons explored.")

    # Run the clustering
    dbs = DBSCAN(eps=e, min_samples=cldf2.shape[1]+1).fit(cldf2)

    # See how we did
    s = pd.Series(dbs.labels_, index=cldf2.index, name=c_nm)

    row = [e]
    data = s.value_counts()

    for c in range(-1, max_clusters+1):
        try:
            if np.isnan(data[c]):
                row.append(None)
            else:
                row.append(data[c])
        except KeyError:
            row.append(None)

    el.append(row)
    iters+=1

edf = pd.DataFrame(el, columns=['Epsilon']+["Cluster " + str(x) for x in list(range(
# Make numeric display a bit neater
pd.set_option('display.float_format', lambda x: '{:,.2f}'.format(x))

print("Done.")

```

7.3.2 Examine Clusters

```
edf.head() # Notice the -1 cluster for small epsilons
```

```

epsilon_long = pd.DataFrame(columns=['Epsilon','Cluster','Count'])

for i in range(0,len(edf.index)):
    row = edf.iloc[i,:]
    for c in range(1,len(edf.columns.values)):
        if row[c] != None and not np.isnan(row[c]):
            d = pd.DataFrame({'Epsilon':row[0], 'Cluster':f"Cluster {c-2}", 'Count':
            epsilon_long = pd.concat([epsilon_long, d], ignore_index=True)

epsilon_long['Count'] = epsilon_long.Count.astype(float)


```


7.3.3 Plot Cluster Sizes

One of the really big problems with DBSCAN and this kind of data is that you have no *practical* way of specifying epsilon (whereas if you were doing walkability analysis then you could cluster on walking distance!). So you can look at the data (as above) to get a reasonable value, but look what the output below shows about the stability of the clusters for different values of epsilon!

```
fig, ax = plt.subplots(figsize=(12,8))
sns.lineplot(data=epsilon_long, x='Epsilon', y='Count', hue='Cluster');
plt.vlines(x=distances[kn.knee], ymin=0, ymax=epsilon_long.Count.max(), color=(1, .7, .7));
plt.gcf().suptitle(f"Cluster sizes for various realisations of Epsilon");
plt.tight_layout()
```

7.4 Final Clustering

 Difficulty: Moderate.

###: Perform Clustering

Use the value from kneed...

```
dbs = DBSCAN(eps=distances[kn.knee], min_samples=cldf2.shape[1]+1).fit(cldf2.values)
s = pd.Series(dbs.labels_, index=cldf2.index, name=c_nm)
rs[c_nm] = s
print(s.value_counts())
```

###: Map Clusters

```
fig, ax = plt_ldn(water, boros)
fig.suptitle(f"{c_nm} Results", fontsize=20, y=0.92)
rs.plot(column=c_nm, ax=ax, linewidth=0, zorder=0, legend=True, categorical=True);
```

7.4.1 'Representative' Centroids

```
to_suppress=['OBJECTID', 'BNG_E', 'BNG_N', 'LONG', 'LAT',
             'Shape__Are', 'Shape__Len', 'geometry', 'Component 1',
             'Component 2', 'Component 3', 'Component 4', 'Component 5',
             'Component 6', 'Component 7', 'Dimension 1', 'Dimension 2',
             'Airbnb Component 1']
```

Take a sample of the full range of numeric columns:

```
cols = random.sample(rs.select_dtypes(exclude='object').drop(columns=to_suppress).columns, 10)
print(cols)
```

Calculate the mean of these columns for each cluster:

```

# Empty data frame with the columns we'll need
centroids = pd.DataFrame(columns=cols)

# For each cluster...
for k in sorted(rs[c_nm].unique()):
    print(f"Processing cluster {k}")

    # Select rows where the cluster name matches the cluster number
    clust = rs[rs[c_nm]==k]

    # Append the means to the centroids data frame
    centroids.loc[k] = clust[cols].mean()

# Drop the unclustered records (-1)
centroids.drop(labels=[-1], axis=0, inplace=True)
centroids

centroids_long = pd.DataFrame(columns=['Variable', 'Cluster', 'Std. Value'])
for i in range(0, len(centroids.index)):
    row = centroids.iloc[i, :]
    for r in row.index:
        d = pd.DataFrame({'Variable': r, 'Cluster': i, 'Std. Value': row[r]}, index=[1])
        centroids_long = pd.concat([centroids_long, d], ignore_index=True)

g = sns.FacetGrid(centroids_long, col="Variable", col_wrap=3, height=3, aspect=1.5,
g = g.map(plt.bar, "Cluster", "Std. Value")

```

8 Self-Organising Maps

SOMs offer a third type of clustering algorithm. They are a relatively 'simple' type of neural network in which the 'map' (of the SOM) adjusts to the data. We're not going to do this in *this* practical, but the main thing is that, unlike the above approaches, SOMs build a 2D map of a higher-dimensional space and use this as a mechanism for subsequently clustering the raw data. In this sense there is a conceptual link between SOMs and PCA or t-SNE or UMAP. They are used quite a lot for text-clustering using keywords (where you have high-dimensionality).


There are [a lot of SOM implementations in Python](#) but the one I used to use, called SOMPY, appears to have [been abandoned](#).

9 Classification

And now for something completely different! This section is **completely optional**, but I thought that you might find it helpful to have a look at how *supervised* learning (classification) differs from *unsupervised* learning (clustering). Here we're going to perform a fairly straightforward classification: predicting the `room_type` for

randomly-selected listings. Of course we know the true answer, but this is for demonstration purposes!

9.1 Additional Setup

 Difficulty: Hard, as I've left out quite a bit of code.

9.1.1 Import Libraries

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix
from sklearn.inspection import permutation_importance
```

9.1.2 Set Up Data

I'm taking a fairly brutal approach here: anything that is not inherently numeric is gone (bye, bye, text), and I'm not bothering to convert implicitly numeric values either: dates could be converted to 'months since last review', for instance, while amenities could be One-Hot Encoded after some pruning of rare amenities. This leaves us with a much smaller number of columns to feed *in* to the classifier.

```
print(f"Cleaned columns: {' '.join(cleaned.columns.to_list())}")
classifier_in = cleaned.drop(columns=['listing_url', 'last_scraped', 'name', 'description',
                                     'host_name', 'host_location', 'property_type',
                                     'bathrooms_text', 'amenities', 'geometry', 'MSR',
                                     'host_since', 'first_review', 'last_review',
                                     'host_verifications', 'review_scores_rating',
                                     'reviews_per_month'])
```

9.1.3 Remove NAs

Not all classifiers have this issue, but some will struggle to make predictions (or not be able to do so at all) if there are NAs in the data set. The classifier we're using can't deal with NAs, so we have to strip these out, but before we do let's check the effect:

```
classifier_in.isna().sum()
```

We can safely drop these now, and you should end up with about 54,000 rows to work with.

```

classifier_in = classifier_in.dropna(axis=0, how='any')
print(f"Now have {classifier_in.shape[0]:,} rows of data to work with (down from {cl

```

```

print()
print(f"Classifier training columns: {'', '.join(classifier_in.columns.to_list())}."
classifier_in.head()

```

9.1.4 Remap Non-Numeric Columns

We do still have a couple of non-numeric columns to deal with: booleans and the thing we're actually trying to predict (the room type)!

```

classifier_in['host_is_superhost'] = classifier_in.host_is_superhost.replace({True:1

```

```

le = LabelEncoder()
classifier_in['room_class'] = le.fit_transform(classifier_in.room_type)

```


A quick check: we should only have one type per class and vice versa.

```


classifier_in.groupby(by=['room_type', 'room_class']).host_id.agg('count').reset_inde

```

9.2 Random Forest Classification

 Difficulty: Hard.

We're going to use a [Random Forest Classifier](#) but the nice thing about `sklearn` is that you can quite easily swap in other classifiers if you'd like to explore further. This is one *big* advantage of Python over R in my book: whereas R tends to get new algorithms *first*, they are often implemented independently by many people and you can end up with incompatible data structures that require a lot of faff to reorganise for a different algorithm. Python is a bit more 'managed' and the dominance of `numpy` and `sklearn` and `pandas` means that people have an incentive to contribute to this library or, if it's genuinely novel, to create an implementation that works *like* it would if it were part of `sklearn`!

 Note

`**🔗 Connections**`: So here's an `_actual_` Machine Learning implementation, but yo

9.2.1 Train/Test Split

```
train, test = train_test_split(classifier_in, test_size=0.2, random_state=42)
print(f"Train contains {train.shape[0]:,} records.")
print(f"Test contains {test.shape[0]:,} records.")
```

```
y_train = train.room_class
X_train = train.drop(columns=['room_class', 'room_type'])

y_test = test.room_class
X_test = test.drop(columns=['room_class', 'room_type'])
```

9.2.2 Classifier Setup


```
rfc = RandomForestClassifier(
    max_depth=8,
    min_samples_split=7,
    n_jobs=4,
    random_state=42
)
```

9.2.3 Fit and Predict

```
rfc.fit(X_train, y_train)
```

```
y_hat = rfc.predict(X_test)
```

9.3 Validate

 Difficulty: Hard.

9.3.1 Confusion Matrix

```
c_matrix = pd.DataFrame(confusion_matrix(y_test, y_hat))
c_matrix.index = le.inverse_transform(c_matrix.index)
c_matrix.columns = le.inverse_transform(c_matrix.columns)
c_matrix
```

9.3.2 Feature Importance

Compare the Random Forest's built-in 'feature importance' with Permutation Feature Importance as [documented here](#).

Note

****Connections**:** This next section is the reason you shouldn't blindly run ML a

```
mdi_importances = pd.Series(
    rfc.feature_importances_, index=rfc.feature_names_in_
).sort_values(ascending=True)

ax = mdi_importances.plot.barh()
ax.set_title("Random Forest Feature Importances (MDI)")
ax.figure.tight_layout()
```

9.3.3 Permutation Feature Importance

```
result = permutation_importance(
    rfc, X_test, y_test, n_repeats=10, random_state=42, n_jobs=2
)
```

```
sorted_importances_idx = result.importances_mean.argsort()
importances = pd.DataFrame(
    result.importances[sorted_importances_idx].T,
    columns=X_test.columns[sorted_importances_idx],
)

ax = importances.plot.box(vert=False, whis=10)
ax.set_title("Permutation Importances (Test Set)")
ax.axvline(x=0, color="k", linestyle="--")
ax.set_xlabel("Decrease in accuracy score")
ax.figure.tight_layout()
```

9.4 Shapely Values

Shapely values are a big part of [explainable AI](#) and they work (very broadly) by permuting the data to explore how sensitive the predictions made by the model are to the results that you see. For these we need to install two libraries: `shap` (to do the heavy lifting) and `slicer` to deal with the data.

9.4.1 Install Libraries

We should now have this already available in Podman, but just in case...

```
try:
    import shap
except ModuleNotFoundError:
    ! pip install slicer shap
    import shap
```

9.4.2 Check for Data Types

You are looking for anything *other* than `int64` or `float64` for the most part. Boolean should be fine, but pandas' internal, nullable integer type will give you a `ufunc` error.

```
X_test.info()
```

```
X_test['beds'] = X_test.beds.astype('int')
```

9.4.3 Plot Partial Dependence

```
shap.partial_dependence_plot(
    "price", rfc.predict, X_test, ice=False,
    model_expected_value=True, feature_expected_value=True
)
```

9.4.4 Calculate Shapely Values

This can take a *long* time: 4-5 hours (!!!) without developing a *strategy* for tackling it. See the [long discussion here](#). I've taken the approach of subsetting the data substantially (the model is already trained so it won't impact the model's predictions) with a 20% fraction of the test data and an explainer sample of 5%. On my laptop the 'Permutation explainer' stage took about 14 minutes, but your results may obviously be rather different.

```
Xsample = shap.utils.sample(X_test.sample(frac=0.2, random_state=41), 10)
explainer = shap.Explainer(rfc.predict, Xsample)
```

Now we calculate the shap values for the 5% sample from `X_test`.

Caution

****Warning**:** This next block is the one that takes a long time to run. I got between

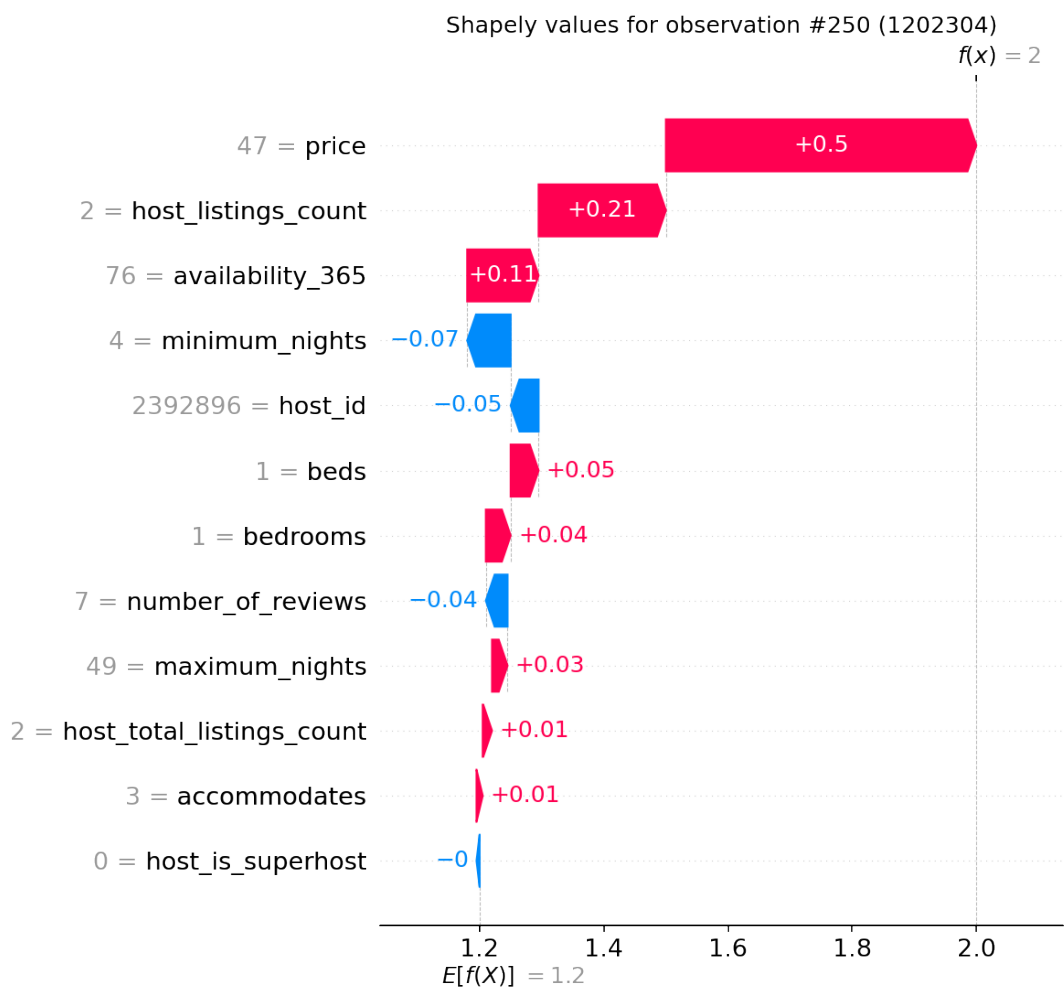
```
%time
shap_values = explainer(X_test.sample(frac=.05, random_state=42))
```

9.4.5 Single Observation

Now you can take any random record (`sample_ind`) and produce a shap plot to show the role that each attribute played in its classification. Note that getting these plots to save required [some searching on GitHub](#).

```
sample_ind=250
shap.plots.waterfall(shap_values[sample_ind], max_display=14, show=False);
plt.title(f"Shapely values for observation #{sample_ind} ({X_test.sample(frac=.05, r",
plt.tight_layout()
#plt.savefig('practical-09-waterfall.png', dpi=150)
```

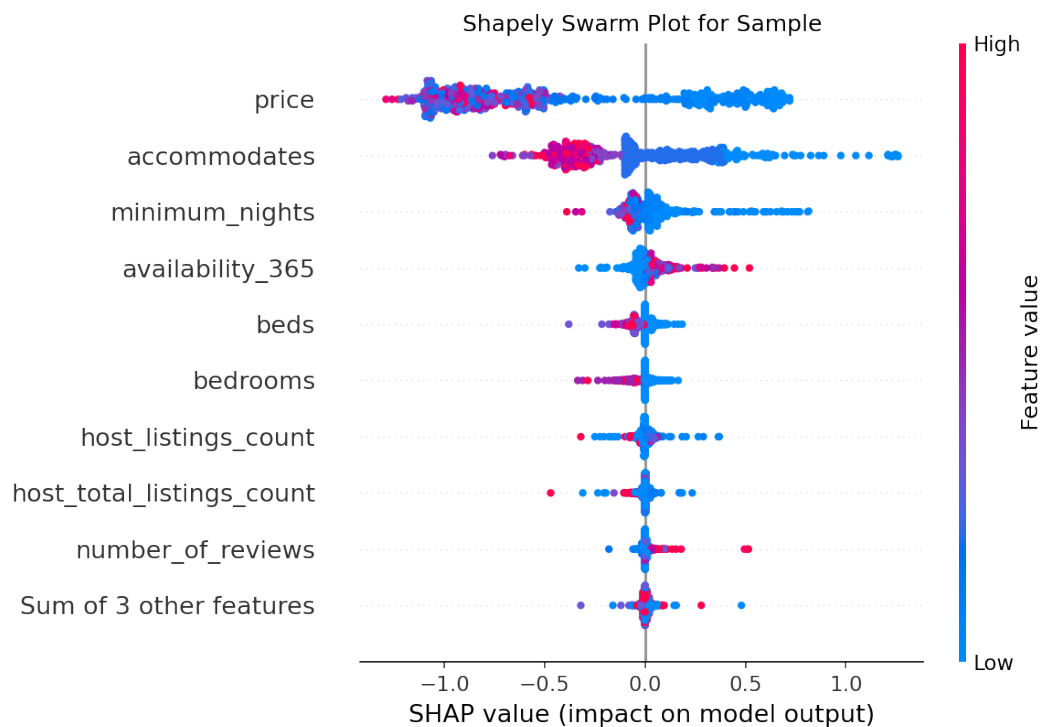
Figure 1: Shapely Feature Plot for Feature 250



9.4.6 All Observations

```
shap.plots.beeswarm(shap_values, show=False)
plt.title(f"Shapely Swarm Plot for Sample")
plt.tight_layout()
plt.savefig('practical-09-swarm.png', dpi=150)
```


Figure 2: Shapely Swarm Plot



9.5 Wrap-Up

- Find the appropriate eps value: [Nearest Neighbour Distance Functions](#) or [Inter-event Distance Functions](#)
- [Clustering Points](#)
- [Regionalisation algorithms with Agglomerative Clustering](#)

You've reached the end, you're done...

Er, no. This is barely scratching the surface! I'd suggest that you go back through the above code and do three things: 1. Add a lot more comments to the code to ensure that really have understood what is going on. 2. Try playing with some of the parameters (e.g. my thresholds for skew, or non-normality) and seeing how your results change. 3. Try outputting additional plots that will help you to understand the *quality* of your clustering results (e.g. what *is* the makeup of cluster 1? Or 6? What has it picked up? What names would I give these clusters?).

If all of that seems like a lot of work then why not learn a bit more about machine learning before calling it a day?

See: [Introduction to Machine Learning with Scikit-Learn](#).