Practical 8: Dimensions in Data

Transformation & Dimensionality Reduction

Table of contents

Complete	Part 1: Foundations	Part 2: Data	Part 3: Analysis	
70%	▯▯▯▯▯▯▯▯	▯▯▯▯▯▯	▯▯▯▯▯▯	8/10

In this session the focus is on MSOA-level Census data from 2011. We're going to explore this as a *possible* complement to the InsideAirbnb data. Although it's not ideal to use 2011 data with scraped from Airbnb this year, we:

1. Have little choice as the 2021 data is only just starting to come through from the Census and the London Data Store hasn't been updated (still!); and
2. Could usefully do a bit of thinking about whether the situation in 2011 might in some way help us to 'predict' the situation now…

Ultimately, however, you don't *need* to use this for your analysis, this practical is intended as a demonstration of how transformation and dimensionality reduction work in practice and the kinds of issues that come up.

> **ℹ ▯ Connections**
>
> There are a *lot* of links across sessions now, as well as some *forward links* to stuff we've not yet covered (see: `pandas.merge`). We'll pick these up as we move through the notebook.

1 Preamble

Let's start with the usual bits of code to ensure plotting works, to import packages and load the data into memory.

```
import os
import re
import numpy as np
import pandas as pd
import geopandas as gpd
import seaborn as sns

import matplotlib.cm as cm
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import PowerTransformer

import umap
from kneed import knee_locator
```

Notice here that we've moved the function from last week's notebook to a separate file `cache.py` from which we import the `cache_data` function. This should give you some ideas about how to work from script -> function -> library effectively.

```
from cache import cache_data
```

2 Loading MSOA Census Data

> **i** Connections
>
> By this point you should be fairly familiar with the UK's census geographies as you'll have encountered them in your GIS module. But in case you need a refresher, here's what the Office for National Statistics says.

> **Tip**
>
> We're going to mix in the London's MSOA 'Atlas' from the London Data Store. I would *strongly* suggest that you have a look around the London Data Store as you develop your thinking for the group assessment – you will likely find useful additional data there!

Once you see how we deal with this MSOA Atlas data you will be in a position to work with any other similarly complex (in terms of the headings and indexes) data set. If you're feeling particularly ambitious you can actually do this *same* work at the LSOA scale using the LSOA Atlas and LSOA boundaries… the process should be the same, though you will have smaller samples in each LSOA than you do in the MSOAs and calculations will take a bit longer to complete. You could also search on the ONS Census web site for data from 2021.

There is a CSV file for the MSOA Atlas that would be easier to work with; however, the Excel file is useful for demonstrating how to work with multi-level indexes (an extension of last week's work). Notice that below we do two new things when reading the XLS file:

1. We have to specify a sheet name because the file contains multiple sheets.
2. We have to specify not just *one* header, we actually have to specify three of them which generates a multi-level index (row 0 is the top-level, row 1 is the second-level, etc.).

2.1 Load MSOA Excel File

> Difficulty level: Low.

You might like to load the cached copy of the file into Excel so that you can see how the next bit works. You can find the rest of the MSOA Atlas here.

```
src_url   = 'https://data.london.gov.uk/download/msoa-atlas/39fdd8eb-e977-4d32-85a4-f65b92f29dcb/msoa-data.x
dest_path = os.path.join('data','msoa')
```

2.1.0.1 Question

```
excel_atlas = pd.read_excel(
    cache_data(src_url, dest_path),
    ???, # Which sheet is the data in?
    header=[0,1,2])          # Where are the column names... there's three of them!
```

2.1.0.2 Answer

```
excel_atlas = pd.read_excel(
    cache_data(src_url, dest_path),
    sheet_name='iadatasheet1', # Which sheet is the data in?
    header=[0,1,2])          # Where are the column names... there's three of them!
```

```
Found data/msoa/msoa-data.xls locally!
    Size is 2 MB (1,979,904 bytes)
```

Notice the format of the output and notice that all of the empty cells in the Excel sheet have come through as `Unnamed: <col_no>_level_<level_no>`:

```
excel_atlas.head(1)
```

	Unnamed: 0_level_0 Unnamed: 0_level_1 MSOA Code	Unnamed: 1_level_0 Unnamed: 1_level_1 MSOA Name	Age Structure (2011 Census) All Ages Unnamed: 2_level_2	 0-15 Unnamed: 3_level_2	 16-29 Unna
0	E02000001	City of London 001	7375.0	620.0	1665.(

```
print(f"Shape of the MSOA Atlas data frame is: {excel_atlas.shape[0]:,} x {excel_atlas.shape[1]:,}")
```

You should get: Shape of the MSOA Atlas data frame is: 984 x 207, but how on earth are you going to access the data?

2.2 Accessing MultiIndexes

Difficulty: Moderate.

The difficulty is conceptual, not technical.

Until now we have understood the pandas index as a single column-like 'thing' in a data frame, but pandas also supports hierarchical and grouped indexes that allow us to interact with data in more complex ways… should we need it. Generally:

- MultiIndex == hierarchical index on *columns*
- DataFrameGroupBy == iterable pseudo-hierarchical index on *rows*

Connections

We'll be looking at **Grouping Data** in much more detail in next week, so the main thing to remember is that grouping is for rows, multi-indexing is about columns.

2.2.1 Direct Access

Of course, one way to get at the data is to use `.iloc[...]` since that refers to columns by *position* and ignores the complexity of the index. Try printing out the the first five rows of the first column using `iloc`:

```
excel_atlas.iloc[???]
```

You should get:

```
0    E02000001
1    E02000002
2    E02000003
3    E02000004
4    E02000005
Name: (Unnamed: 0_level_0, Unnamed: 0_level_1, MSOA Code), dtype: object
```

2.2.2 Named Access

But to do it by name is a little trickier:

```
excel_atlas.columns.tolist()[:5]
```

```
[('Unnamed: 0_level_0', 'Unnamed: 0_level_1', 'MSOA Code'),
 ('Unnamed: 1_level_0', 'Unnamed: 1_level_1', 'MSOA Name'),
 ('Age Structure (2011 Census)', 'All Ages', 'Unnamed: 2_level_2'),
 ('Age Structure (2011 Census)', '0-15', 'Unnamed: 3_level_2'),
 ('Age Structure (2011 Census)', '16-29', 'Unnamed: 4_level_2')]
```

Notice how asking for the first five columns has given us a list of… what exactly?

2.2.2.1 Question

So to get the **same output** by column *name* what do you need to copy from above:

```
excel_atlas.loc[0:5, ???]
```

2.2.2.2 Answer

```
excel_atlas.loc[0:5, ('Unnamed: 0_level_0','Unnamed: 0_level_1','MSOA Code')]
```

```
0    E02000001
1    E02000002
2    E02000003
3    E02000004
4    E02000005
5    E02000007
Name: (Unnamed: 0_level_0, Unnamed: 0_level_1, MSOA Code), dtype: object
```

The answer is *really* awkward, so we're going to look for a better way…

2.2.3 Grouped Access

Despite this, *one* way that MultiIndexes can be useful is for accessing column-slices from a 'wide' dataframe. We can, for instance, select all of the Age Structure columns in one go and it will be *simpler* than what we did above.

```
excel_atlas.loc[0:5, ('Age Structure (2011 Census)')]
```

	All Ages Unnamed: 2_level_2	0-15 Unnamed: 3_level_2	16-29 Unnamed: 4_level_2	30-44 Unnamed: 5_level_2	45-64 Unna
0	7375.0	620.0	1665.0	2045.0	2010.0
1	6775.0	1751.0	1277.0	1388.0	1258.0
2	10045.0	2247.0	1959.0	2300.0	2259.0
3	6182.0	1196.0	1277.0	1154.0	1543.0
4	8562.0	2200.0	1592.0	1995.0	1829.0
5	8791.0	2388.0	1765.0	1867.0	1736.0

2.2.4 Understanding Levels

This works because the MultiIndex tracks the columns using *levels*, with level 0 at the 'top' and level 2 (in our case) at the bottom. These are the unique *values* for the top level ('row 0'):

```
excel_atlas.columns.levels[0]
```

```
Index(['Adults in Employment (2011 Census)', 'Age Structure (2011 Census)',
       'Car or van availability (2011 Census)',
       'Central Heating (2011 Census)', 'Country of Birth (2011)',
       'Dwelling type (2011)', 'Economic Activity (2011 Census)',
       'Ethnic Group (2011 Census)', 'Health (2011 Census)', 'House Prices',
       'Household Composition (2011)', 'Household Income Estimates (2011/12)',
       'Household Language (2011)', 'Households (2011)', 'Incidence of Cancer',
       'Income Deprivation (2010)', 'Land Area', 'Life Expectancy',
       'Lone Parents (2011 Census)', 'Low Birth Weight Births (2007-2011)',
       'Mid-year Estimate totals', 'Mid-year Estimates 2012, by age',
       'Obesity', 'Population Density', 'Qualifications (2011 Census)',
       'Religion (2011)', 'Road Casualties', 'Tenure (2011)',
       'Unnamed: 0_level_0', 'Unnamed: 1_level_0'],
      dtype='object')
```

These are the *values* for those levels across the actual columns in the data frame, notice the repeated 'Age Structure (2011 Census)':

```
excel_atlas.columns.get_level_values(0)[:10]
```

```
Index(['Unnamed: 0_level_0', 'Unnamed: 1_level_0',
       'Age Structure (2011 Census)', 'Age Structure (2011 Census)',
       'Age Structure (2011 Census)', 'Age Structure (2011 Census)',
       'Age Structure (2011 Census)', 'Age Structure (2011 Census)',
       'Age Structure (2011 Census)', 'Mid-year Estimate totals'],
      dtype='object')
```

And here are the *values* for the second level of the index ('row 1' in the Excel file):

```
excel_atlas.columns.get_level_values(1)[:10]
```

```
Index(['Unnamed: 0_level_1', 'Unnamed: 1_level_1', 'All Ages', '0-15', '16-29',
       '30-44', '45-64', '65+', 'Working-age', 'All Ages'],
      dtype='object')
```

By extension, if we drop a level 0 index then *all* of the columns that it supports at levels 1 and 2 are *also* dropped: so when we drop `Mid-year Estimate totals` from level 0 then all 11 of the 'Mid-year Estimate totals (2002…2012)' columns are dropped in one go.

```
excel_atlas[['Mid-year Estimate totals']].head(3)
```

	Mid-year Estimate totals										
	All Ages										
	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012
0	7280.0	7115.0	7118.0	7131.0	7254.0	7607.0	7429.0	7472.0	7338.0	7412.0	7604.0
1	6333.0	6312.0	6329.0	6341.0	6330.0	6323.0	6369.0	6570.0	6636.0	6783.0	6853.0
2	9236.0	9252.0	9155.0	9072.0	9144.0	9227.0	9564.0	9914.0	10042.0	10088.0	10218.0

```
test = excel_atlas.drop(columns=['Mid-year Estimate totals'], axis=1, level=0)

print(f"Excel source had {excel_atlas.shape[1]} columns.")
print(f"Test now has {test.shape[1]} columns.")
```

```
Excel source had 207 columns.
Test now has 196 columns.
```

```
# Tidy up if the variable exists
if 'test' in locals():
    del(test)
```

2.2.5 Questions

- What data type is used for storing/accessing MultiIndexes?
- *Why* is this is the appropriate data type?
- How (conceptually) are the header rows in Excel are mapped on to levels in pandas?

2.3 Tidying Up

> **Difficulty level: Low**
>
> Although there's a lot of dealing with column names.

2.3.1 Dropping Named Levels

There's a *lot* of data in the data frame that we don't need for our Airbnb work, so let's go a bit further with the dropping of column-groups using the MultiIndex.

```
to_drop = ['Mid-year Estimate totals','Mid-year Estimates 2012, by age','Religion (2011)',
        'Land Area','Lone Parents (2011 Census)','Central Heating (2011 Census)','Health (2011 Census)',
        'Low Birth Weight Births (2007-2011)','Obesity','Incidence of Cancer','Life Expectancy',
        'Road Casualties']
tidy = excel_atlas.drop(to_drop, axis=1, level=0)
print(f"Shape of the MSOA Atlas data frame is now: {tidy.shape[0]} x {tidy.shape[1]}")
```

```
Shape of the MSOA Atlas data frame is now: 984 x 111
```

This should drop you down to 984 x 111. Notice below that the multi-level *index* has not changed but the multi-level *values* remaining have!

```
print(f"There are {len(tidy.columns.levels[0].unique())} categories.") # The categories
print(f"But only {len(tidy.columns.get_level_values(0).unique())} values.") # The actual values
```

```
There are 30 categories.
But only 18 values.
```

2.3.2 Selecting Columns using a List Comprehension

Now we need to drop all of the percentages from the data set. These can be found at level 1, though they are specified in a number of different ways so you'll need to come up with a way to find them in the level 1 values using a list comprehension…

I'd suggest looking for: "(%)", "%", and "Percentages". You may need to check both start and end of the string. You could also use a regular expression here instead of multiple tests. Either way *works*, but have a think about the tradeoffs between intelligibility, speed, and what *you* understand…

2.3.2.1 Question

Selection using multiple logical tests:

```
to_drop = [x for x in tidy.columns.get_level_values(1) if (???)]
print(to_drop)
```

Selection using a regular expression:

```
print([x for x in tidy.columns.get_level_values(1) if re.search(???, x)])
```

2.3.2.2 Answer

```
to_drop = [x for x in tidy.columns.get_level_values(1) if (
  x.endswith("(%)") or x.startswith("%") or x.endswith("Percentages") or x.endswith("%"))]
print(to_drop)
```

```
['Percentages', 'Percentages', 'Percentages', 'Percentages', 'Percentages', 'White (%)', 'Mixed/multiple ethnic groups
```

```
print([x for x in tidy.columns.get_level_values(1) if re.search("(?:%|Percentages)", x)])
```

```
['Percentages', 'Percentages', 'Percentages', 'Percentages', 'Percentages', 'White (%)', 'Mixed/multiple ethnic groups
```

With both you should get:

```
['Percentages',
 'Percentages',
 'Percentages',
 'Percentages',
 'Percentages',
 'White (%)',
 'Mixed/multiple ethnic groups (%)',
 'Asian/Asian British (%)',
 'Black/African/Caribbean/Black British (%)',
 'Other ethnic group (%)',
 'BAME (%)',
 'United Kingdom (%)',
 'Not United Kingdom (%)',
 '% of people aged 16 and over in household have English as a main language',
 '% of households where no people in household have English as a main language',
```

'Owned: Owned outright (%)',
'Owned: Owned with a mortgage or loan (%)',
'Social rented (%)',
'Private rented (%)',
'Household spaces with at least one usual resident (%)',
'Household spaces with no usual residents (%)',
'Whole house or bungalow: Detached (%)',
'Whole house or bungalow: Semi-detached (%)',
'Whole house or bungalow: Terraced (including end-terrace) (%)',
'Flat, maisonette or apartment (%)',
'Economically active %',
'Economically inactive %',
'% of households with no adults in employment: With dependent children',
'% living in income deprived households reliant on means tested benefit',
'% of people aged over 60 who live in pension credit households',
'No cars or vans in household (%)',
'1 car or van in household (%)',
'2 cars or vans in household (%)',
'3 cars or vans in household (%)',
'4 or more cars or vans in household (%)']

> **i** Connections
>
> See how regular expressions keep coming baaaaaaaaack? That said, you can also often make use of simple string functions like `startswith` and `endswith` for this problem.

2.3.3 Drop by Level

You now need to drop these columns using the `level` keyword as part of your drop command. You have plenty of examples of how to drop values in place, but I'd suggest *first* getting the command correct (maybe duplicate the cell below and change the code so that the result is saved to a dataframe called `test` before overwriting `tidy`?) and *then* saving the change.

2.3.3.1 Question

```
tidy = tidy.drop(to_drop, axis=1, level=???)
print(f"Shape of the MSOA Atlas data frame is now: {tidy.shape[0]} x {tidy.shape[1]}")
```

2.3.3.2 Answer

```
tidy.drop(to_drop, axis=1, level=1, inplace=True)
print(f"Shape of the MSOA Atlas data frame is now: {tidy.shape[0]} x {tidy.shape[1]}")
```

```
Shape of the MSOA Atlas data frame is now: 984 x 76
```

The data frame should now be 984 x 76. This is a *bit* more manageable though still a *lot* of data columns. Depending on what you decide to do for your final project you might want to revisit some of the columns that we dropped above…

2.3.4 Flattening the Index

Although this ia big improvement, you'll have trouble saving or linking this data to other inputs. The problem is that Level 2 of the multi-index is mainly composed of 'Unnamed' values and so we need to merge it with Level 1 to simplify our data frame, and then merge *that* with level 0...

```
tidy.columns.values[:3]
```

```
array([('Unnamed: 0_level_0', 'Unnamed: 0_level_1', 'MSOA Code'),
      ('Unnamed: 1_level_0', 'Unnamed: 1_level_1', 'MSOA Name'),
      ('Age Structure (2011 Census)', 'All Ages', 'Unnamed: 2_level_2')],
     dtype=object)
```

Let's use code to sort this out!

```
new_cols = []
for c in tidy.columns.values:

  #print(f"Column label: {c}")
  l1 = f"{c[0]}"
  l2 = f"{c[1]}"
  l3 = f"{c[2]}"

  # The new column label
  clabel = ''

  # Assemble new label from the levels
  if not l1.startswith("Unnamed"):
    l1 = l1.replace(" (2011 Census)",'').replace(" (2011)",'').replace("Household ",'').replace("House Prices",'').repla
    l1 = l1.replace('Age Structure','Age').replace("Ethnic Group",'').replace('Dwelling type','').replace('Income Estim
    clabel += l1
  if not l2.startswith("Unnamed"):
    l2 = l2.replace("Numbers",'').replace(" House Price (£)",'').replace("Highest level of qualification: ",'').replace("A
    l2 = l2.replace('At least one person aged 16 and over in household has English as a main language',"1+ English
    clabel += ('-' if clabel != '' else '') + l2
  if not l3.startswith("Unnamed"):
    clabel += ('-' if clabel != '' else '') + l3

  # Replace other commonly-occuring verbiage that inflates column name width
  clabel = clabel.replace('--','-').replace(" household",' hh').replace('Owned: ','')

  #clabel = clabel.replace(' (2011 Census)','').replace(' (2011)','').replace('Sales - 2011.1','Sales - 2012')
  #clabel = clabel.replace('Numbers - ','').replace(' (£)','').replace('Car or van availability','Vehicles')
  #clabel = clabel.replace('Household Income Estimates (2011/12) - ','').replace('Age Structure','Age')

  new_cols.append(clabel)

new_cols
```

```
['MSOA Code',
 'MSOA Name',
 'Age-All Ages',
 'Age-0-15',
 'Age-16-29',
 'Age-30-44',
 'Age-45-64',
 'Age-65+',
 'Age-Working-age',
 'Households-All Households',
 'Composition-Couple hh with dependent children',
 'Composition-Couple hh without dependent children',
 'Composition-Lone parent hh',
 'Composition-One person hh',
 'Composition-Other hh Types',
 'White',
 'Mixed/multiple ethnic groups',
 'Asian/Asian British',
 'Black/African/Caribbean/Black British',
 'Other ethnic group',
 'BAME',
 'Country of Birth-United Kingdom',
 'Country of Birth-Not United Kingdom',
 'Language-1+ English as a main language',
 'Language-None have English as main language',
 'Tenure-Owned outright',
 'Tenure-Owned with a mortgage or loan',
 'Tenure-Social rented',
 'Tenure-Private rented',
 'Household spaces with at least one usual resident',
 'Household spaces with no usual residents',
 'Detached',
 'Semi-detached',
 'Terraced (including end-terrace)',
 'Flat, maisonette or apartment',
 'Population Density-Persons per hectare (2012)',
 'Median-2005',
 'Median-2006',
 'Median-2007',
 'Median-2008',
 'Median-2009',
 'Median-2010',
 'Median-2011',
 'Median-2012',
 'Median-2013 (p)',
 'Sales-2005',
 'Sales-2006',
 'Sales-2007',
 'Sales-2008',
 'Sales-2009',
 'Sales-2010',
 'Sales-2011',
```

```
'Sales-2011.1',
'Sales-2013(p)',
'Qualifications-No',
'Qualifications-Level 1',
'Qualifications-Level 2',
'Qualifications-Apprenticeship',
'Qualifications-Level 3',
'Qualifications-Level 4 and above',
'Qualifications-Other',
'Qualifications-Schoolchildren and full-time students: Age 18 and over',
'Economic Activity-Economically active: Total',
'Economic Activity-Economically active: Unemployed',
'Economic Activity-Economically inactive: Total',
'Economic Activity-Unemployment Rate',
'Adults in Employment-No adults in employment in hh: With dependent children',
'Total Mean hh Income',
'Total Median hh Income',
'Vehicles-No cars or vans in hh',
'Vehicles-1 car or van in hh',
'Vehicles-2 cars or vans in hh',
'Vehicles-3 cars or vans in hh',
'Vehicles-4 or more cars or vans in hh',
'Vehicles-Sum of all cars or vans in the area',
'Vehicles-Cars per hh']
```

Stop

Make sure you understand what is happening here before just moving on to the next thing. Try adding print() statements if it will help it to make sense. This sort of code comes up a *lot* in the real world.

```
tidy.columns = new_cols  # <- Blow away complex index, replace with simple
tidy.head()
```

	MSOA Code	MSOA Name	Age-All Ages	Age-0-15	Age-16-29	Age-30-44
0	E02000001	City of London 001	7375.0	620.0	1665.0	2045.0
1	E02000002	Barking and Dagenham 001	6775.0	1751.0	1277.0	1388.0
2	E02000003	Barking and Dagenham 002	10045.0	2247.0	1959.0	2300.0
3	E02000004	Barking and Dagenham 003	6182.0	1196.0	1277.0	1154.0
4	E02000005	Barking and Dagenham 004	8562.0	2200.0	1592.0	1995.0

You might want to have a look at *what* the code below drops first before just running it… remember that you can pull apart any complex code into pieces:

```
tidy['MSOA Code'].isna()
tidy[tidy['MSOA Code'].isna()].index
```

```
Index([983], dtype='int64')
```

```
tidy.drop(index=tidy[tidy['MSOA Code'].isna()].index, inplace=True)
```

2.4 Add Inner/Outer London Mapping

> ⚠ Difficulty: Moderate, since I'm not giving you many clues.

> ℹ Connections
>
> We touched on lambda functions last week; it's a 'trivial' function that we don't even want to bother defining with def. We also used the lambda function in the context of apply so this is just another chance to remind yourself how this works. This is quite advanced Python, so don't panic if you don't get it right away and have to do some Googling…

We want to add the borough name and a 'subregion' name. We already have the borough name buried in a *separate* column, so step 1 is to extract that from the MSOA Name. Step 2 is to use the borough name as a lookup to the subregion name using a **lambda** function. The format for a lambda function is usually `lambda x: <code that does something with x and returns a value>`. Hint: you've got a dictionary and you know how to use it!

2.4.1 Add Boroughs

We first need to extract the borough names from one of the existing fields in the data frame… a *regex* that does *replacement* would be fastest and easiest: focus on what you *don't* need from the MSOA Name **string** and **replacing** that using a **regex**…

2.4.1.1 Question

```python
tidy['Borough'] = tidy['MSOA Name'].???
tidy.Borough.unique()
```

2.4.1.2 Answer

```python
tidy['Borough'] = tidy['MSOA Name'].str.replace(r' \d+$','',regex=True)
tidy.Borough.unique()
```

```
array(['City of London', 'Barking and Dagenham', 'Barnet', 'Bexley',
       'Brent', 'Bromley', 'Camden', 'Croydon', 'Ealing', 'Enfield',
       'Greenwich', 'Hackney', 'Hammersmith and Fulham', 'Haringey',
       'Harrow', 'Havering', 'Hillingdon', 'Hounslow', 'Islington',
       'Kensington and Chelsea', 'Kingston upon Thames', 'Lambeth',
       'Lewisham', 'Merton', 'Newham', 'Redbridge',
       'Richmond upon Thames', 'Southwark', 'Sutton', 'Tower Hamlets',
       'Waltham Forest', 'Wandsworth', 'Westminster'], dtype=object)
```

You should get:

```
array(['City of London', 'Barking and Dagenham', 'Barnet', 'Bexley',
       'Brent', 'Bromley', 'Camden', 'Croydon', 'Ealing', 'Enfield',
       'Greenwich', 'Hackney', 'Hammersmith and Fulham', 'Haringey',
       'Harrow', 'Havering', 'Hillingdon', 'Hounslow', 'Islington',
       'Kensington and Chelsea', 'Kingston upon Thames', 'Lambeth',
       'Lewisham', 'Merton', 'Newham', 'Redbridge',
       'Richmond upon Thames', 'Southwark', 'Sutton', 'Tower Hamlets',
       'Waltham Forest', 'Wandsworth', 'Westminster'], dtype=object)
```

2.4.2 Map Boroughs to Subregions

And now you need to understand how to *apply* the mapping ot the Borough field using a lambda function. It's fairly straightforward once you know the syntax: just a dictionary lookup. But as usual, you might want to first create a new cell and experiment with the output from the apply function before using it to write the Subregion field of the data frame…

```
mapping = {}
for b in ['Enfield','Waltham Forest','Redbridge','Barking and Dagenham','Havering','Greenwich','Bexley']:
  mapping[b]='Outer East and North East'
for b in ['Haringey','Islington','Hackney','Tower Hamlets','Newham','Lambeth','Southwark','Lewisham']:
  mapping[b]='Inner East'
for b in ['Bromley','Croydon','Sutton','Merton','Kingston upon Thames']:
  mapping[b]='Outer South'
for b in ['Wandsworth','Kensington and Chelsea','Hammersmith and Fulham','Westminster','Camden','City of Londo
  mapping[b]='Inner West'
for b in ['Richmond upon Thames','Hounslow','Ealing','Hillingdon','Brent','Harrow','Barnet']:
  mapping[b]='Outer West and North West'
```

2.4.2.1 Question

```
tidy['Subregion'] = tidy.Borough.apply(???)
```

2.4.2.2 Answer

```
tidy['Subregion'] = tidy.Borough.apply(lambda x: mapping[x])
```

2.4.3 And Save

There's a little snipped of useful code to work out here: we need to check if the clean directory exists in the data directory; if we don't then the tidy.to_parquet() call will fail.

```
if not os.path.exists(os.path.join('data','clean')):
  os.makedirs(os.path.join('data','clean'))
tidy.to_parquet(os.path.join('data','clean','MSOA_Atlas.parquet'))
print("Done.")
```

```
Done.
```

2.4.4 Questions

- What are the advantages to apply and lambda functions over looping and named functions?
- When might you choose a named function over a lambda function?

2.5 Merge Data & Geography

> Difficulty: Low, except for plotting.

> Connections
>
> We'll cover joins (of which a merge is just one type) in the final week's lectures, but between what you'd done in GIS and what we have here there should be enough here for you to start being able to make sense of how they work so that you don't have to wait until Week 10 to think about how this could help you with your Group Assessment.

First, we need to download the MSOA source file, which is a zipped archive of a Shapefile:

```
# Oh look, we can read a Shapefile without needing to unzip it!
msoas = gpd.read_file(
  cache_data('https://github.com/jreades/fsds/blob/master/data/src/Middle_Layer_Super_Output_Areas__Decembe
        os.path.join('data','geo')), driver='ESRI Shapefile')
```

```
Found data/geo/Middle_Layer_Super_Output_Areas__December_2011__EW_BGC_V2-shp.zip locally!
  Size is 7 MB (7,381,177 bytes)
```

2.5.1 Identifying Matching Columns

Looking at the first few columns of each data frame, which one might allow us to link the two files together? You've done this in GIS. *Remember*: the column *names* don't need to match for us to use them in a join, it's the *values* that matter.

```
print(f"Column names: {', '.join(tidy.columns.tolist()[:5])}")
tidy.iloc[:3,:5]
```

```
Column names: MSOA Code, MSOA Name, Age-All Ages, Age-0-15, Age-16-29
```

	MSOA Code	MSOA Name	Age-All Ages	Age-0-15	Age-16-29
0	E02000001	City of London 001	7375.0	620.0	1665.0
1	E02000002	Barking and Dagenham 001	6775.0	1751.0	1277.0
2	E02000003	Barking and Dagenham 002	10045.0	2247.0	1959.0

2.5.2 Merge

One more thing: if you've got more than one choice I'd *always* go with a code over a name because one is intended for matching and other is not...

2.5.2.1 Question

```
gdf = pd.merge(msoas, tidy, left_on=???, right_on=???, how='inner')
gdf = gdf.drop(columns=['MSOA11CD','MSOA11NM','OBJECTID'])

print(f"Final MSOA Atlas data frame has shape {gdf.shape[0]:,} x {gdf.shape[1]}")
```

2.5.2.2 Answer

```
gdf = pd.merge(msoas, tidy, left_on='MSOA11CD', right_on='MSOA Code', how='inner')
gdf = gdf.drop(columns=['MSOA11CD','MSOA11NM','OBJECTID'])

print(f"Final MSOA Atlas data frame has shape {gdf.shape[0]:,} x {gdf.shape[1]}")
```

```
Final MSOA Atlas data frame has shape 983 x 86
```

You should get Final data frame has shape 983 x 86.

2.5.3 Plot Choropleth

Let's plot the median income in 2011 column using the plasma colour ramp... The rest is to show you how to customise a legend.

```
col = 'Median-2011'
fig = gdf.plot(column=???, cmap='???',
        scheme='FisherJenks', k=7, edgecolor='None',
        legend=True, legend_kwds={'frameon':False, 'fontsize':8},
        figsize=(8,7));
plt.title(col.replace('-',' '));

# Now to modify the legend: googling "geopandas format legend"
# brings me to: https://stackoverflow.com/a/56591102/4041902
leg = fig.get_legend()
leg._loc = 3

for lbl in leg.get_texts():
    label_text = lbl.get_text()
    [low, hi] = label_text.split(', ')
    new_text = f'£{float(low):,.0f} - £{float(hi):,.0f}'
    lbl.set_text(new_text)

plt.show();
```

2.5.4 Save

```
gdf.to_geoparquet(os.path.join('data','geo','MSOA_Atlas.geoparquet'))
```

2.5.5 Questions

- Try changing the colour scheme, classification scheme, and number of classes to see if you feel there's a *better* opeion than the one shown above... Copy the cell (click on anywhere outside the code and then hit `C` to copy. Then click on this cell *once*, and hit `V` to paste.

3 Splitting into Test & Train

> **Note**
>
> **🔗 Connections**: Here you will be using a standard approach in Machine Learningknown as _test/tr

A standard approach to Machine Learning, and something that is becoming more widely used elsewhere, is the splitting of a large data into set into testing and training components. Typically, you would take 80-90% of your data to 'train' your algorithm and withold between 10-20% for validation ('testing'). An even 'stricter' approach, in the sense of trying to ensure the robustness of your model against outlier effects, is cross validation such as k-folds cross-validation.

Sci-Kit Learn is probably *the* most important reason Python has become the *de fact* language of data science. Test/train-split is used when to avoid over-fitting when we are trying to **predict** something; so here Sci-Kit Learn *expects* that you'll have an `X` which is your **predictors** (the inputs to your model) and a `y` which is the thing you're **trying to predict** (because: $y = \beta X + \epsilon$).

We're not building a model here (that's for Term 2!) so we'll just 'pretend' that we're trying to predict the price of a listing and will set that up as our `y` data set *so that* we can see how the choice of normalisation/standardisation technique affects the robustness of the model against 'new' data. Notice too that you can pass a data frame directly to Sci-Kit Learn and it will split it for you.

3.1 Reload

> **Tip**
>
> In future 'runs' of this notebook you can now just pick up here and skip all of Task 1.

On subsequent runs of this notebook you might just want to start here!

```
# Notice this handy code: we check if the data is already
# in memory. And notice this is just a list comprehension
# to see what is locally loaded.
if 'gdf' not in locals():
    gdf = gpd.read_parquet(os.path.join('data','geo','MSOA_Atlas.geoparquet'))
print(gdf.shape)
```

```python
categoricals = ['Borough','Subregion']
for c in categoricals:
  gdf[c] = gdf[c].astype('category')
```

3.2 Split

> Difficulty: Low!

For our purposes this is a little bit overkill as you could also use pandas' sample(frac=0.2) and the indexes, but it's useful to see how this works. You use test/train split to get **four** data sets out: the training data gives you two (predictors + target as separate data sets) and the testing data gives you two as well (predictors + target as separate data sets). These are sized accoridng to the test_size specfied in the test_train_split parameters.

```python
from sklearn.model_selection import train_test_split

pdf = gdf['Median-2011'].copy() # pdf for Median *P*rice b/c we need *something*

# df == *data* frame (a.k.a. predictors or independent variables)
# pr == *predicted* value (a.k.a. dependent variable)
# Notice we don't want the median price included in our training data
df_train, df_test, pr_train, pr_test = train_test_split(
        gdf.drop(columns=['Median-2011']), pdf, test_size=0.2, random_state=44)
```

Below you should see that the data has been split roughly on the basis of the test_size parameter.

```python
print(f"Original data size: {gdf.shape[0]:,} x {gdf.shape[1]}")
print(f"  Training data size: {df_train.shape[0]:,} x {df_train.shape[1]} ({(df_train.shape[0]/gdf.shape[0])*100:.0f}%)")
print(f"  Testing data size:  {df_test.shape[0]:,} x {df_test.shape[1]} ({(df_test.shape[0]/gdf.shape[0])*100:.0f}%)")
```

Also notice the indexes of each pair of data sets match:

```python
print(", ".join([str(x) for x in df_train.index[:10]]))
print(", ".join([str(x) for x in pr_train.index[:10]]))
```

3.3 Plot Test/Train Data

> Difficulty: Low, but important!

```python
boros = gpd.read_file(os.path.join('data','geo','Boroughs.gpkg'))

f,axes = plt.subplots(1,2, figsize=(12,5))
df_train.plot(ax=???)
df_test.plot(ax=???)
boros.plot(ax=???, facecolor='none', edgecolor='r', linewidth=.5, alpha=0.4)
boros.plot(ax=???, facecolor='none', edgecolor='r', linewidth=.5, alpha=0.4)
axes[0].set_title('Training Data')
axes[1].set_title('Testing Data');
```

```
axes[0].set_ylim(150000,210000)
axes[1].set_ylim(150000,210000)
axes[0].set_xlim(500000,565000)
axes[1].set_xlim(500000,565000)
axes[1].set_yticks([]);
```

3.3.1 Questions

- Why might it be useful to produce a *map* of a test/train split?
- Why might it matter *more* if you were dealing with user locations or behaviours?

4 Normalisation

The developers of SciKit-Learn define normalisation as "scaling individual samples to have **unit norm**." There are a *lot* of subtleties to this when you start dealing with 'sparse' data, but for the most part it's worthwhile to think of this as a rescaling of the raw data to have similar ranges in order achieve some kind of comparison. This is such a common problem that sklearn offers a range of such (re)scalers including: `MinMaxScaler`.

Let's see what effect this has on the data!

```
# Sets some handy 'keywords' to tweak the Seaborn plot
kwds = dict(s=7,alpha=0.95,edgecolor="none")

# Set the *hue order* so that all plots have the *same*
# colour on the Subregion
ho = ['Inner East','Inner West','Outer West and North West','Outer South','Outer East and North East']
```

4.1 Select Columns

Difficulty: Low.

One thing you'll need to explain is why I keep writing `df[cols+['Subregion']` and why I don't just add it to the `cols` variable at the start? Don't try to answer this now, get through the rest of Tasks 3 and 4 and see what you think.

```
cols = ['Tenure-Owned outright', 'Tenure-Owned with a mortgage or loan',
      'Tenure-Social rented', 'Tenure-Private rented']
```

Answer: one part of the answer is that it makes it easy to change the columns we select without having to remember to keep `Subregion`, but the more important reason is that it allows us to re-use *this* 'definition' of `cols` elsewhere throughout the rest of this practical without needing to remember to *remove* `Subregion`.

```
tr_raw  = df_train[cols+['Subregion']].copy() # train raw
tst_raw = df_test[cols+['Subregion']].copy()  # test raw
```

4.2 Fit to Data

> ⚠ Difficulty: Moderate if you want to understand what reshape is doing.

Fit the training data:

```
from sklearn.preprocessing import MinMaxScaler

# Notice what this is doing! See if you can explain it clearly.
scalers = [???.fit(???[x].values.reshape(-1,1)) for x in cols]
```

4.3 Apply Transformations

> ⚠ Difficulty: Moderate.

Train:

```
tr_normed = tr_raw.copy()

for i, sc in enumerate(scalers):
    # Ditto this -- can you explain what this code is doing
    tr_normed[cols[i]] = sc.transform(df_???[cols[i]].values.reshape(-1,1))
```

Test:

```
tst_normed = tst_raw.copy()

for i, sc in enumerate(scalers):
    tst_normed[cols[i]] = sc.transform(df_???[cols[i]].values.reshape(-1,1))
```

> **Note**
>
> **🔗 Connections**: You don't _have_ to fully understand the next section, but if you are able to get you

Check out the properties of tst_normed below. If you've understood what the MinMaxScaler is doing then you should be able to spot something unexpected in the transformed *test* outputs. If you've *really* understood this, you'll see why this result is problematic for *models*. *Hint*: one way to think of it is an issue of **extrapolation**.

```
for c in cols:
    print(f"  Minimum: {tst_normed[c].min():.4f}")
    ???
```

4.4 Plot Distributions

> ⚠ Difficulty: Moderate.

```
tr_raw.columns     = [re.sub('(-|/)','\n',x) for x in tr_raw.columns.values]
tst_raw.columns    = [re.sub('(-|/)','\n',x) for x in tst_raw.columns.values]
tr_normed.columns  = [re.sub('(-|/)','\n',x) for x in tr_normed.columns.values]
tst_normed.columns = [re.sub('(-|/)','\n',x) for x in tst_normed.columns.values]
```

```
sns.pairplot(data=tr_raw, hue='Subregion', diag_kind='kde', corner=True, plot_kws=kwds, hue_order=ho);
```

```
sns.pairplot(data=tr_normed, hue='Subregion', diag_kind='kde', corner=True, plot_kws=kwds, hue_order=ho);
```

4.4.1 Questions

- Why do I keep writing `df[cols+['Subregion']`? Why I don't just add Subregions to the `cols` variable at the start?
- What has changed between the two plots (of `tr_raw` and `tr_normed`)?
- What is the *potential* problem that the use of the transformer fitted on `tr_normed` to data from `tst_normed` might cause? *Hint*: this is why I asked you to investigate the data in the empty code cell above.
- Can you explain what this is doing: `[MinMaxScaler().fit(df_train[x].values.reshape(-1,1)) for x in cols]`?
- Can you explain what *this* is doing: `sc.transform(df_test[cols[i]].values.reshape(-1,1))`?

5 Standardisation

Standardisation is typically focussed on rescaling data to have a mean (or median) of 0 and standard deviation or IQR of 1. That these approaches are conceptually tied to the idea of symmetric, unimodal data such as that encountered in the standard normal distribution. Rather confusingly, many data scientists will use standardisation and normalisation largely interchangeably!

```
col = 'Vehicles-No cars or vans in hh'
tr  = df_train[[col]].copy()
tst = df_test[[col]].copy()
```

5.1 Z-Score Standardisation

> Difficulty: Low.

```
stsc = StandardScaler().fit(tr[col].values.reshape(-1,1))
```

```
tr[f"Z. {col}"]  = stsc.transform(???)
tst[f"Z. {col}"] = stsc.transform(???)
```

5.2 Inter-Quartile Standardisation

Difficulty: Low.

```
rs = ???(quantile_range=(25.0, 75.0)).fit(???)

tr[f"IQR. {col}"] = rs.transform(???)
tst[f"IQR. {col}"] = rs.transform(???)
```

5.3 Plot Distributions

Note

🔗 Connections: The point of these next plots is simply to show that *linear* transformations (whic

Difficulty: Low.

```
sns.jointplot(data=tr, x=f"{col}", y=f"Z. {col}", kind='kde'); # hex probably not the best choice
```

```
sns.jointplot(data=tr, x=f"{col}", y=f"IQR. {col}", kind='kde'); # hex probably not the best choice
```

```
sns.jointplot(data=tr, x=f"Z. {col}", y=f"IQR. {col}", kind='hex'); # hex probably not the best choice
```

Perhaps a little more useful…

```
ax = sns.kdeplot(tr[f"Z. {col}"])
sns.kdeplot(tr[f"IQR. {col}"], color='r', ax=ax)
plt.legend(loc='upper right', labels=['Standard', 'Robust']) # title='Foo'
ax.ticklabel_format(useOffset=False, style='plain')
ax.set_xlabel("Standardised Value for No cars or vans in hh")
```

5.3.1 Questions?

- Can you see the differences between these two rescalers?
- Can you explain why you might want to choose one over the other?

6 Non-Linear Transformations

Note

🔗 Connections: Now *these* transformations are not directly revsersible because they change the

So transformations are useful when a data series has features that make comparisons or analysis difficult, or that affect our ability to intuit meaningful difference. By manipulating the data using one or more mathematical operations we can sometimes make it more *tractable* for subsequent analysis. In other words, it's all about the *context* of our data.

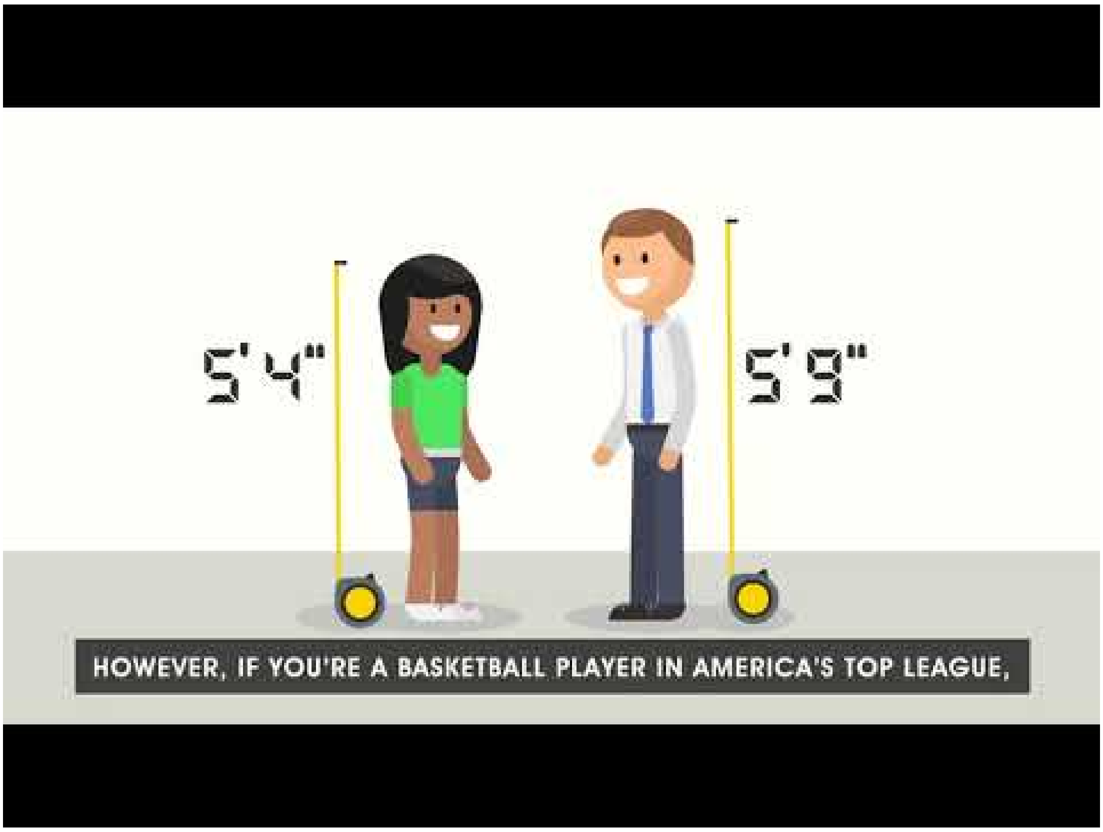

Figure 1: How tall is tall?

From above, we know the *Median Income* data are *not* normally distributed, but can we work out what distribution best represents *Median Income*? This can be done by comparing the shape of the histogram to the shapes of theoretical distributitions. For example:

- the log-normal distribution
- the exponential distribution
- the Poisson distribution (for non-continuous data)

From looking at those theoretical distributions, we might make an initial guess as to the type of distribution. There are actually *many* other distributions encountered in real life data, but these ones are particuarly common. A wider view of this would be that quantile and power transformations are ways of preserving the rank of values but lose many of the other features of the relationships that might be preserved by, for instance, the standard scaler.

In the case of Median Income, taking a log-transform of the data might make it *appear* more normal: you do **not** say that a transformation *makes* data more normal, you say either that 'it allows us to treat the data as normally distributed' or that 'the transformed data follows a log-normal distribution'.

6.1 The Normal Distribution

⚠ Difficulty: Moderate.

Z-scores are often associated with the normal distribution because their interpretation implicitly assumes a normal distribution. Or to put it another way… You can always calculate z-scores for your data (it's just a formula applied to data points), but their *intuitive meaning* is lost if your data don't have something like a normal distribution (or follow the 68-95-99.7 rule).

But… what if our data are non-normal? Well, Just because data are non-normal doesn't mean z-scores can't be calculated (we already did that above); we just have to be careful what we do with them… and sometimes we should just avoid them entirely.

6.1.1 Creating a Normal Distribution

Below is a function to create that theoretical normal distribution. See if you can understand what's going and add comments to the code to explain what each line does.

```
def normal_from_dist(series):
    mu = ???          # mean of our data
    sd = ???          # standard deviation of our data
    n  = ???          # count how many observations are in our data
    s = np.random.normal(???, ???, ???)  #use the parameters of the data just calculated to generate n random numl
    return s                #return this set of random numbers
```

To make it easier to understand what the function above is doing, let's use it! We'll use the function to plot both a distribution plot with both histogram and KDE for our data, and then add a *second* overplot distplot to the same fig showing the theoretical normal distribution (in red). We'll do this in a loop for each of the three variables we want to examine.

6.1.2 Visual Comparisons

Looking at the output, which of the variables has a roughly normal distribution? Another way to think about this question is, for which of the variables are the mean and standard deviation *most* appropriate as measures of centrality and spread?

Also, how would you determine the *meaning* of some of the departures from the normal distribution?

```
selection = [x for x in df_train.columns.values if x.startswith('Composition')]

for c in selection:
    ax = sns.kdeplot(df_train[c])
    sns.kdeplot(normal_from_dist(df_train[c]), color='r', fill=True, ax=ax)
    plt.legend(loc='upper right', labels=['Observed', 'Normal']) # title='Foo'
    ax.ticklabel_format(useOffset=False, style='plain')
    if ax.get_xlim()[1] > 999999:
        plt.xticks(rotation=45)
    plt.show()
```

6.1.3 Questions

- Which, if any, of the variables has a roughly normal distribution? Another way to think about this question is, for which of the variables are the mean and standard deviation *most* appropriate as measures of centrality and spread?
- How might you determine the *significance* of some of the departures from the normal distribution?

6.2 Logarithmic Transformations

> ⚠ Difficulty: Moderate.

To create a new series in the data frame containing the natural log of the original value it's a similar process to what we've done before, but since pandas doesn't provide a log-transform operator (i.e. you can't call `df['MedianIncome'].log()`) we need to use the `numpy` package since pandas data series are just numpy arrays with some fancy window dressing that makes them even *more* useful.

Let's perform the transform then compare to the un-transformed data. Comment the code below to ensure that you understand what it is doing.

6.2.1 Apply and Plot

```
cols = ['Median-2012','Total Mean hh Income']

for m in cols:
  s  = df_train[m] # s == series
  ts = ???.???(s)   # ts == transformed series

  ax = sns.kdeplot(s)
  sns.kdeplot(normal_from_dist(s), color='r', fill=True, ax=ax)
  plt.legend(loc='upper right', labels=['Observed', 'Normal']) # title also an option
  plt.title("Original Data")

  ### USEFUL FORMATTING TRICKS ###
  # This turns off scientific notation in the ticklabels
  ax.ticklabel_format(useOffset=False, style='plain')
  # Notice this snippet of code
  ax.set_xlabel(ax.get_xlabel() + " (Raw Distribution)")
  # Notice this little code snippet too
  if ax.get_xlim()[1] > 999999:
    plt.xticks(rotation=45)

  plt.show()

  ax = sns.kdeplot(ts)
  sns.kdeplot(normal_from_dist(ts), color='r', fill=True, ax=ax)
  plt.legend(loc='upper right', labels=['Observed', 'Normal'])
  ax.ticklabel_format(useOffset=False, style='plain')
```

```
    ax.set_xlabel(ax.get_xlabel() + " (Logged Distribution)")
    if ax.get_xlim()[1] > 999999:
        plt.xticks(rotation=45)
    plt.title("Log-Transformed Data")
    plt.show()
```

Hopefully, you can see that the transformed data do indeed look 'more normal'; the peak of the red and blue lines are closer together and the blue line at the lower extreme is also closer to the red line, but we can check this by seeing what has happened to the z-scores.

6.3 Power Transformations

⚠ Difficulty: Moderate.

```
cols = ['Median-2012','Total Mean hh Income']
pt = ???(method='yeo-johnson')

for m in cols:
    s  = df_train[m] # s == series
    ts = pt.fit_transform(s.values.reshape(-1,1))
    print(f"Using lambda (transform 'exponent') of {pt.lambdas_[0]:0.5f}")

    ax = sns.kdeplot(ts.reshape(-1,))

    sns.kdeplot(normal_from_dist(???), color='r', fill=True, ax=ax)
    plt.legend(loc='upper right', labels=['Observed', 'Normal'])
    ax.ticklabel_format(useOffset=False, style='plain')
    ax.set_xlabel(m + " (Transformed Distribution)")
    if ax.get_xlim()[1] > 999999: # <-- What does this do?
        plt.xticks(rotation=45)
    plt.title("Power-Transformed Data")
    plt.show();
```

7 Principal Components Analysis

ℹ Note

🔗 Connections: This is all about _dimensionality_ and the different ways that we can reduce dime

Now we're going to ask the question: how can we represent our data using a smaller number of components that capture the variance in the original data. You should have covered PCA in Quantitative Methods.

7.0.1 Optional Reload

Use this is your data gets messy...

```
gdf = gpd.read_parquet(os.path.join('data','geo','MSOA_Atlas.geoparquet')).set_index('MSOA Code')
print(gdf.shape)
```

```
categoricals = ['Borough','Subregion']
for c in categoricals:
    gdf[c] = gdf[c].astype('category')
```

7.1 Calculating Shares

> Difficulty: Hard.

Sadly, there's no transformer to work this out for you automatically, but let's start by converting the raw population and household figures to shares so that our later dimensionality reduction steps aren't impacted by the size of the MSOA.

```
gdf[['Age-All Ages','Households-All Households']].head(5)
```

7.1.1 Specify Totals Columns

```
total_pop = gdf['Age-All Ages']
total_hh  = gdf['Households-All Households']
total_vec = gdf['Vehicles-Sum of all cars or vans in the area']
```

7.1.2 Specify Columns for Pop or HH Normalisation

```
pop_cols  = ['Age-', 'Composition-', 'Qualifications-', 'Economic Activity-', 'White', 'Mixed/multiple',
             'Asian/Asian British', 'Black/African', 'BAME', 'Other ethnic',
             'Country of Birth-']
hh_cols   = [???, ???, ???, 'Detached', 'Semi-detached', 'Terraced', 'Flat, ']
```

```
popre = re.compile(r'^(?:' + "|".join(pop_cols) + r')')
hhre  = re.compile(r'^(?:' + "|".join(???) + r')')
```

7.1.3 Apply to Columns

```
tr_gdf = gdf.copy()
tr_gdf['Mean hh size'] = tr_gdf['Age-All Ages']/tr_gdf['Households-All Households']

for c in gdf.columns:
    print(c)
    if popre.match(c):
        print("  Normalising by total population.")
        tr_gdf[c] = gdf[c]/???
    elif ???.match(???):
        print("  Normalising by total households.")
        tr_gdf[c] = gdf[c]/???
    elif c.startswith('Vehicles-') and not c.startswith('Vehicles-Cars per hh'):
        print("  Normalising by total vehicles.")
```

```
        tr_gdf[c] = gdf[c]/???
    else:
        print("  Passing through.")
```

7.2 Removing Columns

> ⚠ Difficulty: Moderate.

To perform dimensionality we can only have numeric data. In theory, categorical data can be converted to numeric and retained, but there are two issues:

1. Nominal data has no *innate* order so we *can't* convert > 2 categories to numbers and have to convert them to One-Hot Encoded values.
2. A binary (i.e. One-Hot Encoded) variable will account for a *lot* of variance in the data because it's only two values they are 0 and 1!

So in practice, it's probably a good idea to drop categorical data if you're planning to use PCA.

7.2.1 Drop Totals Columns

```
pcadf = tr_gdf.drop(columns=['Age-All Ages', 'Households-All Households',
                    'Vehicles-Sum of all cars or vans in the area'])
pcadf = pcadf.set_index('MSOA Code')
```

7.2.2 Drop Non-Numeric Columns

```
pcadf.select_dtypes(['category','object']).columns
```

```
pcadf.drop(columns=pcadf.select_dtypes(['category','object']).columns.to_list(), inplace=True)
pcadf.drop(columns=['BNG_E','BNG_N','geometry', 'LONG', 'LAT','Shape__Are', 'Shape__Len'], inplace=True)
```

```
pcadf.columns
```

7.3 Rescale & Reduce

> ⚠ Difficulty: Moderate.

In order to ensure that our results aren't dominated by a single scale (e.g. House Prices!) we need to rescale all of our data. You could easily try different scalers as well as a different parameters to see what effect this has on your results.

7.3.1 Robustly Rescale

Set up the Robust Rescaler for inter-decile standardisation: 10th and 90th quantiles.

```
rs = ???

for c in pcadf.columns.values:
    pcadf[c] = rs.fit_transform(pcadf[c].values.reshape(-1, 1))
```

7.3.2 PCA Reduce

```
from sklearn.decomposition import PCA

pca = PCA(n_components=50, whiten=True)

pca.fit(pcadf)

explained_variance = pca.explained_variance_ratio_
singular_values = pca.singular_values_
```

7.3.3 Examine Explained Variance

```
x = np.arange(1,???)
plt.plot(x, explained_variance)
plt.ylabel('Share of Variance Explained')
plt.show()

for i in range(0, 20):
    print(f"Component {i:>2} accounts for {explained_variance[i]*100:>2.2f}% of variance")
```

You should get that Component 0 accounts for 31.35% of the variance and Component 19 accounts for 0.37%.

###: How Many Components?

There are a number of ways that we could set a threshold for dimensionality reduction: - The most common is to look for the 'knee' in the Explained Variance plot above. That would put us at about 5 retained components. - Another is to just keep all components contributing more than 1% of the variance. That would put us at about 10 components. - You can also (I discovered) look to shuffle the data and repeatedly perform PCA to build confidence intervals. I have not implemented this (yet).

In order to *do* anything with these components we need to somehow reattach them to the MSOAs. So that entails taking the transformed results (X_train and X_test)

```
kn = knee_locator.KneeLocator(x, explained_variance,
                              curve='convex', direction='decreasing',
                              interp_method='interp1d')
print(f"Knee detected at: {kn.knee}")
kn.plot_knee()
```

```python
keep_n_components = 7

# If we weren't changing the number of components we
# could re-use the pca object created above.
pca = PCA(n_components=keep_n_components, whiten=True)

X_train = pca.fit_transform(???)

# Notice that we get the _same_ values out,
# so this is a *deterministic* process that
# is fully replicable (allowing for algorithmic
# and programming language differences).
print(f"Total explained variance: {pca.explained_variance_ratio_.sum()*100:2.2f}%")
for i in range(0, keep_n_components):
    print(f"  Component {i:>2} accounts for {pca.explained_variance_ratio_[i]*100:>5.2f}% of variance")

# Notice...
print(f"X-train shape: {len(X_train)}")
print(f"PCA df shape: {pcadf.shape[0]}")
# So each observation has a row in X_train and there is
# 1 column for each component. This defines the mapping
# of the original data space into the reduced one
print(f"Row 0 of X-train contains {len(X_train[0])} elements.")
```

7.4 Components to Columns

> ⚠ Difficulty: Moderate.

You could actually do this more quickly (but less clearly) using X_train.T to transpose the matrix!

```python
for i in range(0,keep_n_components):
    s = pd.Series(X_train[:,???], index=pcadf.???)
    pcadf[f"Component {i+1}"] = s

pcadf.sample(3).iloc[:,-10:-4]
```

7.5 (Re)Attaching GeoData

> ⚠ Difficulty: Moderate.

```python
msoas = gpd.read_file(os.path.join('data','geo','Middle_Layer_Super_Output_Areas__December_2011__EW_BGC_V
msoas = msoas.set_index('MSOA11CD')
print(msoas.columns)

msoas.head(1)

pcadf.head(1)
```

```
gpcadf = pd.merge(msoas.set_index(['MSOA11CD'], drop=True), pcadf, left_index=True, right_index=True, how='inn
print(f"Geo-PCA df has shape {gpcadf.shape[0]} x {gpcadf.shape[1]}")
```

You should get PCA df has shape 983 x 89.

```
gpcadf['Borough'] = gpcadf.MSOA11NM.apply(???)
```

7.6 Map the First n Components

> ⚠ Difficulty: Moderate.

How would you automate this so that the loop creates one plot for each of the first 3 components? How do you interpret these?

```
for comp in [f"Component {x}" for x in range(1,3)]:
  ax = gpcadf.plot(column=???, cmap='plasma',
      scheme='FisherJenks', k=7, edgecolor='None', legend=True, figsize=(9,7));
  boros.plot(ax=ax, edgecolor='w', facecolor='none', linewidth=1, alpha=0.7)
  ax.set_title(f'PCA {comp}')
```

Your first component map should look something like this:

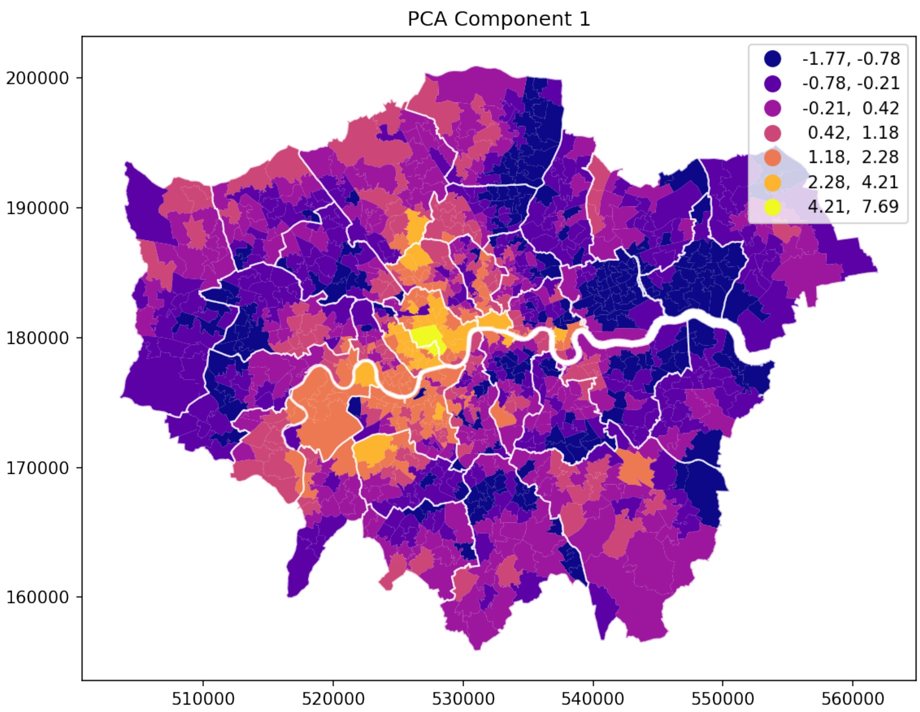

Figure 2: PCA Component 1

8 UMAP

UMAP is a non-linear dimensionality reduction technique. Technically, it's called *manifold* learning: imagine being able to roll a piece of paper up in more than just the 3rd dimension…). As a way to see if there is structure in your data this is a *much* better technique than one you might encounter in many tutorials: *t-SNE*. It has to do with how the two techniques 'learn' the manifold to use with your data.

8.1 UMAP on Raw Data

> Difficulty: Hard.

```python
from umap import UMAP

# You might want to experiment with all
# 3 of these values -- it may make sense
# to package a lot of this up into a function!
keep_dims=2
rs=42

u = UMAP(
  n_neighbors=25,
  min_dist=0.01,
  n_components=keep_dims,
  random_state=rs)

X_embedded = u.fit_transform(???)
print(X_embedded.shape)
```

8.2 Write to Data Frame

> Difficulty: Low.

Can probably also be solved using X_embedded.T.

```python
for ix in range(0,X_embedded.shape[1]):
  print(ix)
  s = pd.Series(X_embedded[:,???], index=pcadf.???)
  gpcadf[f"Dimension {ix+1}"] = s
```

8.3 Visualise!

> Difficulty: Low.

```python
rddf = gpcadf.copy() # Reduced Dimension Data Frame
```

8.3.1 Simple Scatter

```
f,ax = plt.subplots(1,1,figsize=(8,6))
sns.scatterplot(x=rddf[???], y=rddf[???], hue=rddf['Borough'], legend=False, ax=ax)
```

8.3.2 Seaborn Jointplot

That is *suggestive* of there being struccture in the data, but with 983 data points and 33 colours it's hard to make sense of what the structure *might* imply. Let's try this again using the Subregion instead and taking advantage of the Seaborn visualisation library's `jointplot` (joint distribution plot):

```
rddf['Subregion'] = rddf.Borough.apply(lambda x: mapping[x])
```

```
# Sets some handy 'keywords' to tweak the Seaborn plot
kwds = dict(s=7,alpha=0.95,edgecolor="none")
# Set the *hue order* so that all plots have some colouring by Subregion
ho = ['Inner East','Inner West','Outer West and North West','Outer South','Outer East and North East']
```

```
g = sns.jointplot(data=rddf, x=???, y=???, height=8,
            hue=???, hue_order=ho, joint_kws=kwds)
g.ax_joint.legend(loc='upper right', prop={'size': 8});
```

Your jointplot should look like this:

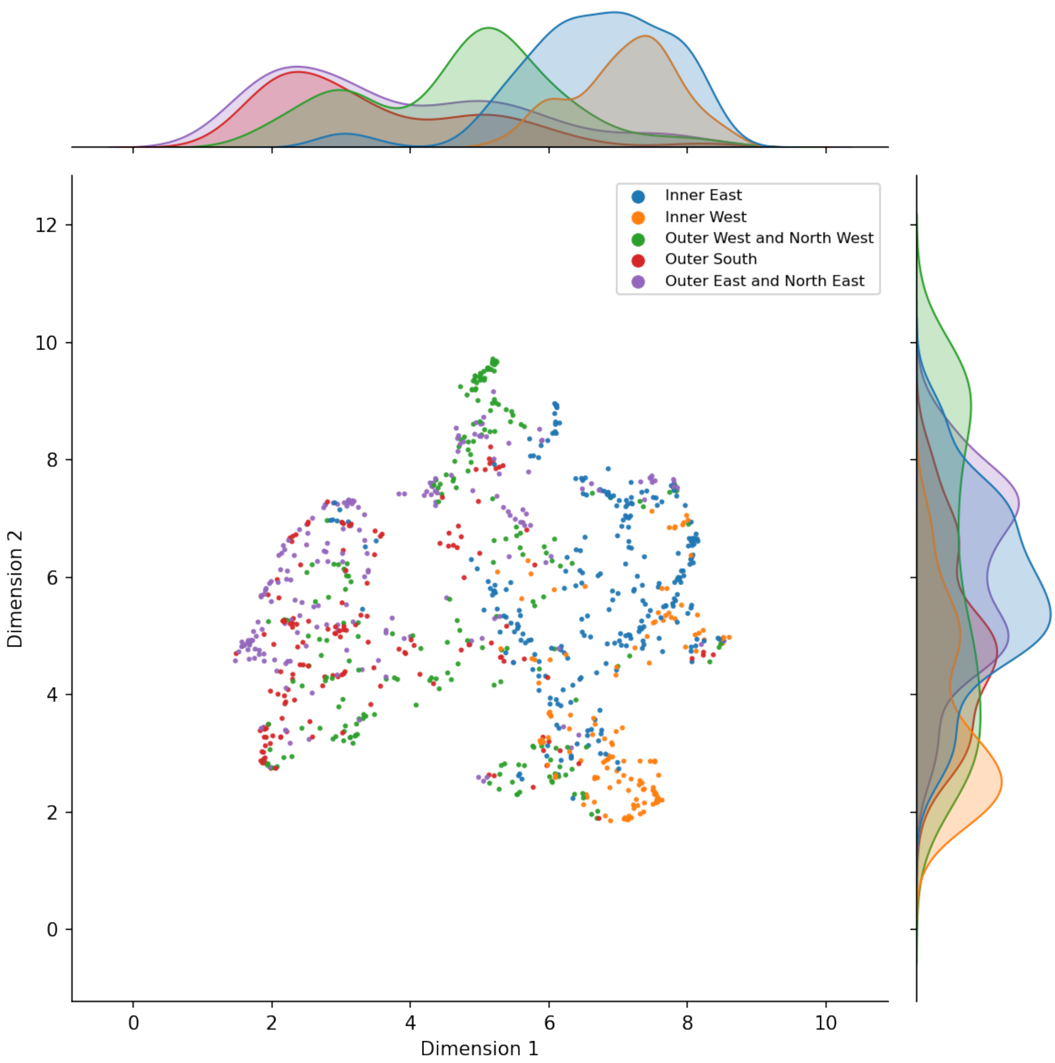

Figure 3: UMAP Jointplot

What do you make of this?

Maybe let's give this one last go splitting the plot out by subregion so that we can see how these vary:

```
for r in rddf.Subregion.unique():
    g = sns.jointplot(data=rddf[rddf.Subregion==r], x='Dimension 1', y='Dimension 2',
            hue='Borough', joint_kws=kwds)
    g.ax_joint.legend(loc='upper right', prop={'size': 8});
    g.ax_joint.set_ylim(0,15)
    g.ax_joint.set_xlim(0,15)
    plt.suptitle(r)
```

We can't unfortunately do any clustering at this point to create groups from the data (that's next week!) so for now note that there are several large-ish groups (in terms of membership) and few small ones picked up by t-SNE. Alos note that there is strong evidence of some incipient structure: Inner East and West largely clump together, while Outher East and Outer South also seem to group together, with Outer West being more distinctive. If you look back at the PCA Components (especially #1) you might be able to speculate about some reasons for this! Please note: this is *only* speculation at this time!

Next week we'll also add the listings data back in as part of the picture!

8.4 Map the n Dimensions

> Difficulty: Low.

```
for comp in [f"Dimension {x}" for x in range(1,3)]:
    f, ax = plt.subplots(1,1,figsize=(12,8))
    rddf.plot(???);
    boros.plot(edgecolor='w', facecolor='none', linewidth=1, alpha=0.7, ax=ax)
    ax.set_title(f'UMAP {comp}')
```

Your first dimension map should look something like this:

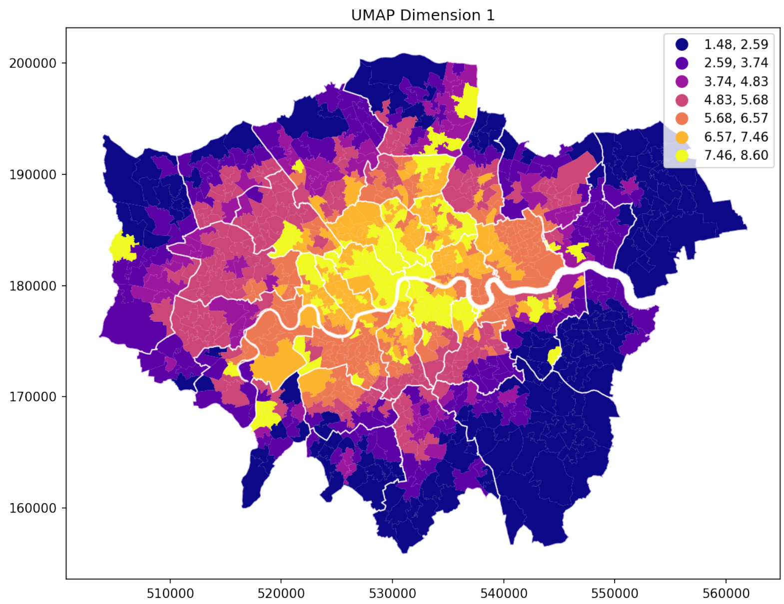

Figure 4: UMAP Dimension 1

8.5 And Save

```
rddf.to_parquet(os.path.join('data','clean','Reduced_Dimension_Data.geoparquet'))
```

8.5.1 Questions

- How would you compare/contrast PCA components with UMAP dimensions? Why do they not seem to show the same thing even though *both* seem to show *something*?

- What might you do with the output of either the PCA or UMAP processes?

8.6 Credits!

8.6.0.1 Contributors:

The following individuals have contributed to these teaching materials: Jon Reades (j.reades@ucl.ac.uk).

8.6.0.2 License

8.6.0.3 Potential Dependencies:

This notebook may depend on the following libraries: pandas, geopandas, sklearn, matplotlib, seaborn