Preamble

Table of contents

Complete	Part 1: Foundations	Part 2: Data	Part 3: Analysis	
90%	▯▯▯▯▯▯▯▯	▯▯▯▯▯▯	▯▯▯▯▯▯	10/10

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. But what defines similarity and difference? There is no *one* answer to that question and so there are many different ways to cluster data, each of which has strengths and weaknesses that make them more, or less, appropriate in different contexts.

> **Note**
>
> ▯ **Connections**:

```
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 requests
import zipfile
import re
import os
import pickle as pk

from io import BytesIO, StringIO
from os.path import join as pj
from pathlib import Path
import matplotlib as mpl
from matplotlib.colors import ListedColormap

import sklearn
from sklearn.neighbors import NearestNeighbors
from sklearn.manifold import TSNE
```

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import PowerTransformer, RobustScaler, StandardScaler
from sklearn.cluster import KMeans, DBSCAN, OPTICS
from esda.adbscan import ADBSCAN

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))
```

0.1 Load GeoData for Display

```
# Load Water GeoPackage
w_path = os.path.join('data','geo','Water.gpkg')
if not os.path.exists(w_path):
    water = gpd.read_file('https://github.com/jreades/i2p/blob/master/data/src/Water
    water.to_file(w_path)
    print("Downloaded Water.gpkg file.")
else:
    water = gpd.read_file(w_path)

# Boroughs GeoPackage
b_path = os.path.join('data','geo','Boroughs.gpkg')
if not os.path.exists(b_path):
    boroughs = gpd.read_file('https://github.com/jreades/i2p/blob/master/data/src/Bo
    boroughs.to_file(b_path)
    print("Downloaded Boroughs.gpkg file.")
else:
    boroughs = gpd.read_file(b_path)
```

Useful Functions for Plotting

```
def plt_ldn(w=water, b=boroughs):
    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

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")
```

0.2 Loading Data

Load the Listings Data

Feel free to download this manually and load it locally rather that loading via the URL:

```
fdf = pd.read_csv(url, compression='gzip', low_memory=False)
```

```
fdf.columns
```

```
fdf[fdf.host_listings_count == 0][['host_total_listings_count','calculated_host_list
```

```
url = 'https://github.com/jreades/i2p/blob/master/data/clean/2020-08-24-listings.csv
df = pd.read_csv(url, compression='gzip', low_memory=False,
                 usecols=['room_type','calculated_host_listings_count','availability
print(f"Data frame is {df.shape[0]:,} x {df.shape[1]}")
```

You should have: `Data frame is 74,120 x 7`.

Aggregating Listings by MSOA

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

```
msoas = gpd.read_file(os.path.join('data','geo','London_MSOAs.gpkg'), driver='GPKG')
```

```
gdf = gpd.GeoDataFrame(df,
      geometry=gpd.points_from_xy(df['longitude'], df['latitude'], crs='epsg:4326'))
gdf = gdf.to_crs('epsg:27700')
```

```
# ml == MSOA Listings
ml = gpd.sjoin(gdf, msoas[['MSOA11CD','geometry']], op='within').drop(columns=
    ['latitude','longitude','index_right','geometry']
)
ml = ml[~(ml.room_type.isin(['Hotel room','Shared room']))]
ml.head()
```

```
ax = ml.calculated_host_listings_count.hist(bins=1500)
ax.set_xlim([0,50]);
```

```
ml['multihost'] = False
ml.loc[ml.calculated_host_listings_count>=3,'multihost'] = True
```

```
ax = ml.price.hist(bins=5000)
ax.set_xlim([0,1000]);
```

```
for p in range(300, 1050, 50):
    print(f"Percent of listings above ${p:,}/night: {(ml[ml.price > p].price.count()
```

```
mlg = ml[ml.price <= 700].groupby(
    ['MSOA11CD','room_type','multihost']
)['price'].agg(Mean='mean', Count='count').reset_index()
mlg.head()
```

```
rs = RobustScaler(quantile_range=[2.5,97.5])
```

You should see wide ranges of counts by roomt type in the first MSOA alone: the largest number of listings is for Entire home/apt but there are 242 multi-host listings compared to 'just' 156 non-multis. I believe this is the City of London though, which is quite unusual for a MSOA and should probably be treated as an outlier in most cases.

```
mlgp = mlg.pivot(index='MSOA11CD', columns=['room_type','multihost'], values=['Mean'
mlgp.head()
```

```
mlgp[['Mean']].head()
```

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

```
mlgp[['Mean']].head()
```

```
mlgp[['Count']].head()
```

```
total = mlgp[['Count']].sum(axis=1)
```

```
for c in mlgp[['Count']].columns.values:
    mlgp[c] = rs.fit_transform( (mlgp[c]/total).values.reshape(-1,1) )

mlgp[['Count']].head()
```

```
cols = []
for c in mlgp.columns.values:
    l0 = c[0]
    l1 = c[1]
    l2 = "Multi-host" if c[2] else "Single host"

    colname = ""

    if l0=='MSOA11CD':
        cols.append(l0)
    elif l0=='Mean':
        cols.append(f"{l1} {l2} Mean")
    elif l0=='Count':
        cols.append(f"{l1} {l2} Listings")
    else:
        raise Exception("Sorry, please specify how to handle this column")

print(cols)
mlgp.columns = cols
mlgp.set_index('MSOA11CD', inplace=True)
mlgp.head()
```

0.3 PCA

```
from sklearn.decomposition import PCA

max_components = mlgp.shape[0] if mlgp.shape[0] < mlgp.shape[1] else mlgp.shape[1]

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

pca.fit(mlgp)

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

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

```
for i in range(0, len(explained_variance)):
    print(f"Component {i+1:>2} accounts for {explained_variance[i]*100:>2.2f}% of va
```

```
keep_n_components = 6

# 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(mlgp)
```

```
mlgpca = pd.DataFrame(index=mlgp.index)

for i in range(0,len(X_train.T)):
    mlgpca[f"Component {i+1}"] = X_train.T[i]
```

```
mlgpca.sample(3)
```

0.4 Clustering Data Frame

```
cldf = mlgpca.copy()
```

```
cols_to_plot = np.random.choice(cldf.columns.values, 3, replace=False)
print("Plotting cols: " + ", ".join(cols_to_plot))
rs = cldf.copy()
```

```
c_nm   = 'KMeans' # Clustering name
k_pref = 4 # Number of clusters

kmeans = KMeans(n_clusters=k_pref, n_init=20, random_state=42).fit(cldf) # The proce

print(kmeans.labels_) # The results

# Add it to the data frame
rs[c_nm] = pd.Series(kmeans.labels_, index=cldf.index)

# How are the clusters distributed?
rs[c_nm].hist(bins=k_pref)

# Going to be a bit hard to read if
# we plot every variable against every
# other variables, so we'll just pick a few
sns.set(style="white")
sns.pairplot(rs,
             vars=cols_to_plot,
             hue=c_nm, markers=".", height=3, diag_kind='kde')
```

```
cgdf = pd.merge(msoas, rs, on='MSOA11CD')
```

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

del(cgdf)
```

One More Thing...

There's just one little problem: what assumption did I make when I started this k-means cluster analysis? It's a huge one, and it's one of the reasons that k-means clustering can be problematic when used naively...

STOP. What critical assumption did we make when running this analysis? The 'Right' Number of Clusters Again, there's more than one way to skin this cat. In Geocomputation 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...

Let's try it for a range of values...

```
# Adapted from: http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_sil
from sklearn.metrics import silhouette_samples, silhouette_score

text = []

for k in range(2,16):
    # Debugging
    print("Cluster count: " + str(k))

    ##############
    # Do the clustering using the main columns
    clusterer = KMeans(n_clusters=k, n_init=20, random_state=42)
    cluster_labels = clusterer.fit_predict(cldf)

    # Calculate the overall silhouette score
    silhouette_avg = silhouette_score(cldf, cluster_labels)
    text = text + [f"For k={k} the average silhouette_score is: {silhouette_avg:6.4f

    # Calculate the silhouette values
    sample_silhouette_values = silhouette_samples(cldf, cluster_labels)
```

```
#############
# 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, cldf.shape[0] + (k + 1) * 10])

y_lower = 10

# For each of the clusters...
for i in range(k):
    # Aggregate the silhouette scores for samples belonging to
    # cluster i, and sort them
    ith_cluster_silhouette_values = \
        sample_silhouette_values[cluster_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
    #cmap  = cm.get_cmap("Spectral")
    color = plt.cm.Spectral(i/k)
    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="--")

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(cluster_labels.astype(float) / k)
    ax2.scatter(cldf[cldf.columns[0]], cldf[cldf.columns[1]], marker='.', s=30, lw=0
                c=colors)

    # Labeling the clusters
    centers = clusterer.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 analysis for KMeans clustering on sample data "
                  "with n_clusters = %d" % k),
                 fontsize=14, fontweight='bold')

    plt.show()

print("\n".join(text))
```

Interpreting the Results

STOP. Make sure that you understand how the silhouette plot and value work, and why your results may diverge from mine.

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.

```
c_nm   = 'KMeans'
k_pref = 7
kmeans = KMeans(n_clusters=k_pref, n_init=75, random_state=42).fit(cldf)

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

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

```
cgdf = pd.merge(msoas, rs, on='MSOA11CD')

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

del(cgdf)
```

'Representative' Centroids 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 will since the clusters are explicitly built around mean centroids. There's also a k-medoids clustering approach built around the median centroid.

```
centroids = None
for k in sorted(rs[c_nm].unique()):
    print(f"Processing cluster {k}")

    clmsoas = rs[rs[c_nm]==k]
    if centroids is None:
        centroids = pd.DataFrame(columns=clmsoas.columns.values)
    centroids = centroids.append(clmsoas.mean(), ignore_index=True)

odf = pd.DataFrame(columns=['Variable','Cluster','Std. Value'])
for i in range(0,len(centroids.index)):
    row = centroids.iloc[i,:]
    c_index = list(centroids.columns.values).index(c_nm)
    for c in range(0,c_index):
        d = {'Variable':centroids.columns[c], 'Cluster':row[c_index], 'Std. Value':r
        odf = odf.append(d, ignore_index=True)

g = sns.FacetGrid(odf, col="Variable", col_wrap=3, height=3, aspect=1.5, margin_titl
g = g.map(plt.plot, "Cluster", "Std. Value", marker=".")
```

DBScan

Of course, as we've said above k-means is just one way of clustering, DBScan is another. Unlike k-means, we don't need to specify the number of clusters in advance. Which sounds great, but we still need to specify other parameters (typically, these are known as hyperparameters because they are about specifying parameters that help the aglorithm to find the right solution... or final set of parameters!) and these can have a huge impact on our results!

Find a Reasonable Value for Epsilon

Before we an use DBSCAN it's useful to find a good value for Epsilon. We can look for the point of maximum 'curvature' in a nearest neigbhours plot. Which seems to be in the vicinity of 0.55. Tips on selecting min_pts can be found here.

```
neigh = NearestNeighbors(n_neighbors=2)
nbrs = neigh.fit(cldf)
distances, indices = nbrs.kneighbors(cldf)
```

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

```
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(0.5, 2.5, 0.01):

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

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

    # See how we did
    s = pd.Series(dbs.labels_, index=cldf.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.")
```

```
odf = 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 = {'Epsilon':row[0], 'Cluster':f"Cluster {c-2}", 'Count':row[c]}
            odf = odf.append(d, ignore_index=True)
```

```
odf['Count'] = odf.Count.astype(float)
```

```
xmin = odf[odf.Cluster=='Cluster 0'].Epsilon.min()
xmax = odf[(odf.Cluster=='Cluster -1') & (odf.Count < cldf.shape[0]/5)].Epsilon.min(

fig, ax = plt.subplots(figsize=(12,8))
ax.set_xlim([xmin,xmax])
sns.lineplot(data=odf, x='Epsilon', y='Count', hue='Cluster');
```

```
e = 0.835
dbs = DBSCAN(eps=e, min_samples=cldf.shape[1]+1).fit(cldf.values)
s = pd.Series(dbs.labels_, index=cldf.index, name=c_nm)
rs[c_nm] = s
print(s.value_counts())
```

```
cgdf = pd.merge(msoas, rs, on='MSOA11CD')

fig, ax = plt_ldn()
fig.suptitle(f"{c_nm} Results", fontsize=20, y=0.92)

cgdf.plot(column=c_nm, ax=ax, linewidth=0, zorder=0, legend=True, categorical=True)

del(cgdf)
```

```
from sompy.sompy import SOMFactory
```

```
cldf.columns.values
```

```
c_nm = 'SOM'

sm = SOMFactory().build(
    cldf.values, mapsize=(10,15),
    normalization='var', initialization='random', component_names=cldf.columns.value
sm.train(n_job=4, verbose=False, train_rough_len=2, train_finetune_len=5)
```

```
topographic_error  = sm.calculate_topographic_error()
quantization_error = np.mean(sm._bmu[1])
print("Topographic error = {0:0.5f}; Quantization error = {1:0.5f}".format(topograph
```

```
from sompy.visualization.mapview import View2D
view2D = View2D(10, 10, "rand data", text_size=10)
view2D.show(sm, col_sz=4, which_dim="all", denormalize=True)
```

```
from sompy.visualization.bmuhits import BmuHitsView
vhts = BmuHitsView(15, 15, "Hits Map", text_size=8)
vhts.show(sm, anotate=True, onlyzeros=False, labelsize=9, cmap="plasma", logaritmic=
```

```
from sompy.visualization.hitmap import HitMapView

k_val = 5
sm.cluster(k_val)
hits  = HitMapView(15, 15, "Clustering", text_size=14)
a     = hits.show(sm)
```

```
# Get the labels for each BMU
# in the SOM (15 * 10 neurons)
clabs = sm.cluster_labels

# Project the data on to the SOM
# so that we get the BMU for each
# of the original data points
bmus  = sm.project_data(cldf.values)

# Turn the BMUs into cluster labels
# and append to the data frame
s = pd.Series(clabs[bmus], index=rs.index, name=c_nm)

rs[c_nm] = s
```

```
cgdf = pd.merge(msoas, rs, on='MSOA11CD')

fig, ax = plt_ldn()
fig.suptitle(f"{c_nm} Results", fontsize=20, y=0.92)

cgdf.plot(column=c_nm, ax=ax, linewidth=0, zorder=0, legend=True, categorical=True)

del(cgdf)
```

```
centroids = None
for k in sorted(rs[c_nm].unique()):
    print(f"Processing cluster {k}")

    clsoas = rs[rs[c_nm]==k]
    if centroids is None:
        centroids = pd.DataFrame(columns=clsoas.columns.values)
    centroids = centroids.append(clsoas.mean(), ignore_index=True)
```

```
odf = pd.DataFrame(columns=['Variable','Cluster','Std. Value'])
for i in range(0,len(centroids.index)):
    row = centroids.iloc[i,:]
    c_index = list(centroids.columns.values).index(c_nm)
    for c in range(0,c_index):
        d = {'Variable':centroids.columns[c], 'Cluster':row[c_index], 'Std. Value':r
        odf = odf.append(d, ignore_index=True)

g = sns.FacetGrid(odf, col="Variable", col_wrap=3, height=3, aspect=1.5, margin_titl
g = g.map(plt.plot, "Cluster", "Std. Value", marker=".")

del(odf, centroids)
```

0.4.1 Which Clustering Approach is Right?

The reason that there is no 'right' approach to clustering is that it all depends on what you're trying to accomplish and how you're *reasoning* about your problem. The image below highlights the extent to which the different clustering approaches in sklearn can produce different results – and this is only for the *non-geographic* algorithms!

Note: for geographically-aware clustering you need to look at PySAL.

To think about this in a little more detail:

- If I run an online company and I want to classify my customers on the basis of their product purchases, then I probably don't care much about where they are, only about what they buy, and so my clustering approach doesn't need to take geography into account. I might well *discover* that many of my most valuable customers live in a few areas, but that is a finding, not a factor, in my research.
- Conversely, if I am looking for cancer clusters then I might well care a *lot* about geography because I want to make sure that I don't overlook an important cluster of cases because it's 'hidden' inside an area with lots of people who don't have cancer. In that case, I want my clusters to take geography into account. That approach might classify an area with a smaller proportion of cancer patients as part of a 'cancer cluster' but that's because it is still significant *because* of the geography.

So you can undertake a spatial analysis using *either* approach, it just depends on the role that you think geography should play in producing the clusters in the first place. We'll see this in action today!

Ensure Plotting Output

```
import matplotlib as mpl
mpl.use('TkAgg')
%matplotlib inline
```

Importing the Libraries

```
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 requests
import zipfile
import re
import os
import pickle as pk

from io import BytesIO, StringIO
from os.path import join as pj
from pathlib import Path
import matplotlib as mpl
from matplotlib.colors import ListedColormap

import sklearn
sklv = int(sklearn.__version__.replace(".",""))
if sklv < 210:
    print("SciKit-Learn verion is: " + sklearn.__version__)
    print("The OPTICS part of this notebook relies on a version >= 0.21.0")

from sklearn.neighbors import NearestNeighbors
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from sklearn.preprocessing import PowerTransformer
from sklearn import preprocessing
from sklearn import cluster

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))
```

```
src = 'https://github.com/kingsgeocomp/applied_gsa/blob/master/data/Census.zip?raw=t
dst = os.path.join('analysis','Census.zip')

if not os.path.exists(dst):
    if not os.path.exists(os.path.dirname(dst)):
        os.makedirs(os.path.dirname(dst))

    print("Downloading...")
    r = requests.get(src, stream=True)

    with open(dst, 'wb') as fd:
        for chunk in r.iter_content(chunk_size=128):
```

```
            fd.write(chunk)
else:
    print("File already downloaded.")

print("Done.")
```

0.4.2 Loading the NomisWeb Data

You may need to make a few adjustments to the path to get the data loaded on your own computer. But notice what we're now able to do here: using the `zipfile` library we can extract a data file (or any other file) from the Zip archive without even having to open it. Saves even more time *and* disk space!

```
z = zipfile.ZipFile(os.path.join('analysis','Census.zip'))
z.namelist()
```

We're going to save each data set to a separate data frame to make it easier to work with during cleaning. But note that this code is fairly flexible since we stick each new dataframe in a dictionary (`d`) where we can retrieve them via an iterator.

```
raw   = {}
clean = {}
total_cols = 0

for r in range(0, len(z.namelist())):

    m  = re.search("(?:-)([^\.]+)", z.namelist()[r])
    nm = m.group(1)

    print("Processing {0} file: ".format(nm))

    with z.open(z.namelist()[r]) as f:

        if z.namelist()[r] == '99521530-Activity.csv':
            raw[nm] = pd.read_csv(BytesIO(f.read()), header=7, skip_blank_lines=True
        else:
            raw[nm] = pd.read_csv(BytesIO(f.read()), header=6, skip_blank_lines=True

    print("\tShape of dataframe is {0} rows by {1} columns".format(raw[nm].shape[0],
    total_cols += raw[nm].shape[1]

print("There are {0} columns in all.".format(total_cols))
```

```
nm  = 'Occupation'
url = 'https://github.com/jreades/urb-studies-predicting-gentrification/raw/master/d

print(f"Processing {nm}")
df = pd.read_csv(url, header=7, skip_blank_lines=True, compression='gzip', low_memor
mapping = {
```

```
    '1. Managers, directors and senior officials':'Managers',
    '2. Professional occupations':'Professionals',
    '3. Associate professional and technical occupations':'Associates',
    '4. Administrative and secretarial occupations':'Administrative',
    '5. Skilled trades occupations':'Skilled trades',
    '6. Caring, leisure and other service occupations':'Caring and Leisure',
    '7. Sales and customer service occupations':'Customer Service',
    '8. Process plant and machine operatives':'Operatives',
    '9. Elementary occupations':'Elementary'
}
df.rename(columns=mapping, inplace=True)
df.drop(['2011 super output area - lower layer','All categories: Occupation'], axis=

raw[nm] = df
df.sample(3, random_state=42)
```

```
nm  = 'Income'
url  = 'https://data.london.gov.uk/download/household-income-estimates-small-areas/7

print(f"Processing {nm}")
df = pd.read_csv(url, encoding='latin-1')[['Code','Median 2011/12']]
df['Median Income'] = df['Median 2011/12'].str.replace('£','').str.replace(',','').a
df.drop('Median 2011/12', axis=1, inplace=True)

raw[nm] = df
df.sample(3, random_state=42)
```

```
nm  = 'Housing'
url = 'https://data.london.gov.uk/download/average-house-prices/9a92fbaf-c04e-498a-9

print(f"Processing {nm}")
df = pd.read_csv(url, encoding='latin-1', low_memory=False)

df['Borough'] = df.Area.str.replace(' [0-9A-Z]{4}$','')
df.drop('Area', axis=1, inplace=True)

df = df[ (df.Year=='Year ending Dec 2011') & (df.Measure=='Median') ][['Code','Value

# Note: not all have a value for this year!
df['Median House Price'] = df.Value.str.replace(':','-1').astype(float)

la = df.groupby('Borough')
la_prices = pd.DataFrame(la['Median House Price'].median())

df = df.join(la_prices, how='inner', on='Borough', rsuffix='_la')

df.loc[df['Median House Price'] < 50000,'Median House Price'] =  df[df['Median House
df.drop(['Value','Borough','Median House Price_la'], inplace=True, axis=1)

raw[nm] = df
```

```
df.sample(3, random_state=42)
```

0.4.3 ONS Boundary Data

We also need to download the LSOA boundary data. A quick Google search on "2011 LSOA boundaries" will lead you to the Data.gov.uk portal. The rest is fairly straightforward: * We want 'generalised' because that means that they've removed some of the detail from the boundaries so the file will load (and render) more quickly. * We want 'clipped' because that means that the boundaries have been clipped to the edges of the land (e.g. the Thames; the 'Full' data set splits the Thames down the middle between adjacent LSOAs).

Saving Time

Again, in order to get you started more quickly I've already created a 'pack' for you. However, note that the format of this is a GeoPackage, this is a fairly new file format designed to replace ESRI's antique Shapefile format, and it allows us to include all kinds of useful information as part of the download as well as doing away with the need to unzip a download first! So here we load the data directly into a geopandas dataframe:

```
src = 'https://github.com/kingsgeocomp/applied_gsa/raw/master/data/London%20LSOAs.gp

gdf = gpd.read_file(src)
print("Shape of LSOA file: {0} rows by {1} columns".format(gdf.shape[0], gdf.shape[1
gdf.columns = [x.lower() for x in gdf.columns.values]
gdf.set_index('lsoa11cd', drop=True, inplace=True)
gdf.sample(4)
```

Error!

Depending on your version of GDAL/Fiona, you may not be able to read the GeoPackage file directly. In this case you will need to replace the code above with the code below for downloading and extracting a Shapefile from a Zip archive:

```
src = 'https://github.com/kingsgeocomp/applied_gsa/blob/master/data/Lower_Layer_Supe
dst = os.path.join('analysis','LSOAs.zip')
zpd = 'analysis'

if not os.path.exists(dst):
    if not os.path.exists(os.path.dirname(dst)):
        os.makedirs(os.path.dirname(dst))

    r = requests.get(src, stream=True)

    with open(dst, 'wb') as fd:
        for chunk in r.iter_content(chunk_size=128):
            fd.write(chunk)
```

```
if not os.path.exists(zpd):
    os.makedirs(os.path.dirname(zpd))

zp = zipfile.ZipFile(dst, 'r')
zp.extractall(zpd)
zp.close()

gdf = gpd.read_file(os.path.join('analysis','lsoas','Lower_Layer_Super_Output_Areas_
gdf.crs = {'init' :'epsg:27700'}
print(\"Shape of LSOA file: {0} rows by {1} columns\".format(gdf.shape[0], gdf.shape
gdf.set_index('lsoa11cd', drop=True, inplace=True)
gdf.sample(4)
```

You can probably see why I'm a big fan of GeoPackages when they're available!

0.4.4 Other Sources of Data

If you're more interested in US Census data then there's a nice-looking (though I haven't used it) wrapper to the Census API. And Spielman and Singleton have done some work on large-scale geodemographic clustering of U.S. Census geographies.

0.4.5 Direct Downloads

These are already clean, so we can just copy them over.

```
for t in ['Occupation','Housing','Income']:
    raw[t].rename(columns={'Code':'mnemonic'}, inplace=True)
    print(raw[t].columns)
    clean[t] = raw[t]
```

0.4.6 Dwellings

From dwellings we're mainly interested in the housing type since we would expect that housing typologies will be a determinant of the types of people who live in an area. We *could* look at places with no usual residents as well, or explore the distribution of shared dwellings, but this is a pretty good start.

```
t = 'Dwellings'
raw[t].columns
```

```
# Select the columns we're interested in analysing
selection = ['mnemonic',
    'Whole house or bungalow: Detached',
    'Whole house or bungalow: Semi-detached',
    'Whole house or bungalow: Terraced (including end-terrace)',
    'Flat, maisonette or apartment: Purpose-built block of flats or tenement',
```

```
    'Flat, maisonette or apartment: Part of a converted or shared house (including b
    'Flat, maisonette or apartment: In a commercial building'
]

# Drop everything *not* in the selection
clean[t] = raw[t].drop(raw[t].columns[~np.isin(raw[t].columns.values,selection)].val

mapping = {}
for c in selection[1:]:
    m  = re.search("^(?:[^\:]*)(?:\:\s)?([^\(]+)", c)
    nm = m.group(1).strip()
    #print("Renaming '{0}' to '{1}'".format(c, nm))
    mapping[c] = nm

clean[t].rename(columns=mapping, inplace=True)

clean[t].sample(5, random_state=42)
```

0.4.7 Age

Clearly, some areas have more young people, some have older people, and some will be composed of families. A lot of these are going to be tied to 'lifestage' and so will help us to understand something about the types of areas in which they live.

```
t = 'Age'
raw[t].columns
```

```
# Select the columns we're interested in analysing
selection = ['mnemonic',
    'Age 0 to 14',
    'Age 15 to 24',
    'Age 25 to 44',
    'Age 45 to 64',
    'Age 65+'
]

# Derived columns
raw[t]['Age 0 to 14']  = raw[t]['Age 0 to 4'] + raw[t]['Age 5 to 7'] + raw[t]['Age 8
raw[t]['Age 15 to 24'] = raw[t]['Age 15'] + raw[t]['Age 16 to 17'] + raw[t]['Age 18
raw[t]['Age 25 to 44'] = raw[t]['Age 25 to 29'] + raw[t]['Age 30 to 44']
raw[t]['Age 45 to 64'] = raw[t]['Age 45 to 59'] + raw[t]['Age 60 to 64']
raw[t]['Age 65+']      = raw[t]['Age 65 to 74'] + raw[t]['Age 75 to 84'] + raw[t]['A

# Drop everything *not* in the selection
clean[t] = raw[t].drop(raw[t].columns[~np.isin(raw[t].columns.values,selection)].val

clean[t].sample(5, random_state=42)
```

0.4.8 Ethnicity

We might also think that the balance of ethnic groups might impact a categorisation of LSOAs in London.

```
t = 'Ethnicity'
raw[t].columns
```

```
# Select the columns we're interested in analysing
selection = ['mnemonic',
    'White',
    'Mixed/multiple ethnic groups',
    'Asian/Asian British',
    'Black/African/Caribbean/Black British',
    'Other ethnic group'
]

# Drop everything *not* in the selection
clean[t] = raw[t].drop(raw[t].columns[~np.isin(raw[t].columns.values,selection)].val

clean[t].sample(5, random_state=42)
```

0.4.9 Rooms

Let's next incorporate the amount of space available to each household.

```
t = 'Rooms'
raw[t].columns
```

```
# Select the columns we're interested in analysing
selection = ['mnemonic',
    'Occupancy rating (bedrooms) of -1 or less',
    'Occupancy rating (rooms) of -1 or less',
    'Average household size',
#    'Average number of bedrooms per household',
#    'Average number of rooms per household',
]

# Drop everything *not* in the selection
clean[t] = raw[t].drop(raw[t].columns[~np.isin(raw[t].columns.values,selection)].val

clean[t].sample(5, random_state=42)
```

0.4.10 Vehicles

Car ownership and use is also known to be a good predictor of social and economic 'status': Guy Lansley's article on the DLVA's registration database offers a useful perpective on the usefulness of this approach.

```
t = 'Vehicles'
raw[t].columns
```

```
# Select the columns we're interested in analysing
selection = ['mnemonic',
    'No cars or vans in household',
    '1 car or van in household',
    '2 cars or vans in household',
    '3 or more cars or vans in household'
]

# Calculate a new column
raw[t]['3 or more cars or vans in household'] = raw[t]['3 cars or vans in household'

# Drop everything *not* in the selection
clean[t] = raw[t].drop(raw[t].columns[~np.isin(raw[t].columns.values,selection)].val

clean[t].sample(5, random_state=42)
```

0.4.11 Tenure

Ownership structure is another categorisation predictor.

```
t = 'Tenure'
raw[t].columns
```

```
# Select the columns we're interested in analysing
selection = ['mnemonic',
    'Owned',
    'Social rented',
    'Private rented',
    'Shared ownership (part owned and part rented)'
]

# Drop everything *not* in the selection
clean[t] = raw[t].drop(raw[t].columns[~np.isin(raw[t].columns.values,selection)].val

clean[t].rename(columns={'Shared ownership (part owned and part rented)':'Shared own

clean[t].sample(5, random_state=42)
```

0.4.12 Qualifications

You can find out a bit more about qualifications here.

```
t = 'Qualifications'
raw[t].columns
```

```
# Select the columns we're interested in analysing
selection = ['mnemonic',
    'Highest level of qualification: Below Level 3 qualifications',
    'Highest level of qualification: Level 3 and above qualifications',
    'Highest level of qualification: Other qualifications'
]

# Derive a new aggregate field for 'didn't complete HS'
raw[t]['Highest level of qualification: Below Level 3 qualifications'] = \
    raw[t]['No qualifications'] + \
    raw[t]['Highest level of qualification: Level 1 qualifications'] + \
    raw[t]['Highest level of qualification: Level 2 qualifications'] + \
    raw[t]['Highest level of qualification: Apprenticeship']

raw[t]['Highest level of qualification: Level 3 and above qualifications'] = \
    raw[t]['Highest level of qualification: Level 3 qualifications'] + \
    raw[t]['Highest level of qualification: Level 4 qualifications and above']

# Drop everything *not* in the selection
clean[t] = raw[t].drop(raw[t].columns[~np.isin(raw[t].columns.values,selection)].val

mapping = {}
for c in selection[1:]:
    m  = re.search("^(?:[^\:]*)(?:\:\s)?([^\(]+)", c)
    nm = m.group(1).strip()
    #print("Renaming '{0}' to '{1}'".format(c, nm))
    mapping[c] = nm

clean[t].rename(columns=mapping, inplace=True)

clean[t].sample(5, random_state=42)
```

0.4.13 Activity

```
t = 'Activity'
raw[t].columns
```

```
# Select the columns we're interested in analysing
selection = ['mnemonic',
    'Economically active: In employment',
    'Economically active: Unemployed',
    'Economically active: Full-time student',
#    'Economically inactive: Retired',
    'Economically inactive: Looking after home or family',
    'Economically inactive: Long-term sick or disabled',
#    'Economically inactive: Other'
]
```

```
# Drop everything *not* in the selection
clean[t] = raw[t].drop(raw[t].columns[~np.isin(raw[t].columns.values,selection)].val

mapping = {}
for c in selection[1:]:
    m  = re.search("^(?:[^\:]*)(?:\:\s)?([^\(]+)", c)
    nm = m.group(1).strip()
    #print("Renaming '{0}' to '{1}'".format(c, nm))
    mapping[c] = nm

clean[t].rename(columns=mapping, inplace=True)

clean[t].sample(5, random_state=42)
```

0.4.14 Standardisation with SKLearn

Let's try standardising the data now:

```
# Here's how we can rescale and transform data easily
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import MinMaxScaler
```

```
random.seed(42)
t   = random.sample(population=list(clean.keys()), k=1)[0]
col = random.sample(population=list(clean[t].columns.values[1:]), k=1)[0]
print(f"Looking at {col} column from {t}.")
```

Here's the 'original' distribution:

```
plt.rcParams['figure.figsize']=(7,3)
sns.distplot(clean[t][col], kde=False)
```

Here's the version that has been re-scaled (standardised) using Min/Max rescaling:

```
plt.rcParams['figure.figsize']=(7,3)
sns.distplot(preprocessing.minmax_scale(clean[t][col].values.reshape(-1,1)), kde=Fal
```

Here's a version that has been robustly rescaled:

```
plt.rcParams['figure.figsize']=(7,3)
sns.distplot(preprocessing.robust_scale(clean[t][col].values.reshape(-1,1), quantile
```

And here's a version that has been Power Transformed... spot the difference!

```
sns.distplot(
    preprocessing.power_transform(clean[t][col].values.reshape(-1,1), method='yeo-jo
```

Combining transformation *and* rescaling:

```
sns.distplot(
    preprocessing.robust_scale(
        preprocessing.power_transform(
            clean[t][col].values.reshape(-1,1), method='yeo-johnson'
        ), quantile_range=[5.0, 95.0]
    ),
    kde=False
)
```

```
# Set up a new dictionary for the transforms
transformed = {}

transformer = preprocessing.PowerTransformer()
scaler      = preprocessing.RobustScaler(quantile_range=[5.0, 95.0])
#scaler      = preprocessing.MinMaxScaler()

# Simple way to drop groups of data we don't want...
suppress       = set(['Rooms','Vehicles'])

for k in set(clean.keys()).difference(suppress):
    print(f"Transforming {k}")
    df = clean[k].copy(deep=True)
    df.set_index('mnemonic', inplace=True)

    # For rescale and transforming everything when the operations
    # apply to each Series separately you can do it as a 1-liner like this:
    #df[df.columns] = scaler.fit_transform(transformer.fit_transform(df[df.columns]))

    # To calculate within-*column* proportions it's like this:
    #for c in df.columns.values:
    #    df[c] = scaler.fit_transform( (df[c]/df[c].max() ).values.reshape(-1, 1) )

    # To calculate within-*group* proportions it's like this:
    if k in ['Housing','Income','Rooms']:
        df[df.columns] = scaler.fit_transform( df[df.columns] )
    else:
        df['sum'] = df[list(df.columns)].sum(axis=1)
        for c in df.columns.values:
            if c == 'sum':
                df.drop(['sum'], axis=1, inplace=True)
            else:
                df[c] = scaler.fit_transform( (df[c]/df['sum']).values.reshape(-1, 1

    #print(df.sample(5, random_state=42))
    transformed[k] = df
```

0.4.15 Creating the Single Data Set

Now that we've converted everything to percentages, it's time to bring the data together! We'll initialise the data frame using the first matching data set, and then iterate over the rest, merging the data frames as we go.

```
matching = list(transformed.keys())
print("Found the following data sets:\n\t" + ", ".join(matching))

# Initialise the data frame simply by grabbing the
# very first existing data frame and copying it
# directly (SCaled Data Frame == scdf)
scdf  = transformed[matching[0]].copy()
lsoac = clean[matching[0]].copy()

for m in range(1, len(matching)):
    scdf  = scdf.merge(transformed[matching[m]], how='inner', left_on='mnemonic', ri
    lsoac = lsoac.merge(clean[matching[m]], how='inner', left_on='mnemonic', right_o

scdf.to_csv(os.path.join('data','Scaled_and_Transformed.csv.gz'), compression='gzip'
lsoac.to_csv(os.path.join('data','Cleaned.csv.gz'), compression='gzip')
```

```
print("Shape of full data frame is {0} by {1}".format(scdf.shape[0], scdf.shape[1]))
```

With luck you still have 4,835 rows, but now you have rather fewer than 88 columns.

```
random.seed(42)
cols_to_plot = random.sample(population=list(scdf.columns.values), k=3)
print("Columns to plot: " + ", ".join(cols_to_plot))
```

```
# The data as it is now...
sns.set(style="whitegrid")
sns.pairplot(lsoac,
             vars=cols_to_plot,
             markers=".", height=3, diag_kind='kde')
```

```
# The data as it is now...
sns.set(style="whitegrid")
sns.pairplot(scdf,
             vars=cols_to_plot,
             markers=".", height=3, diag_kind='kde')
```

STOP. Making sure that you understand how and why this results differns from the *same* plot above.

:::

Right, so you can see that rescaling the dimension hasn't *actually* changed the relationships within each dimension, or even between dimensions, but it has changed the overall range so that the the data is broadly re-centered on 0 but we *still* have the original outliers from the raw data. You could *also* do IQR standardisation (0.25

and 0.75) with the percentages, but in those cases you would have *more* outliers and then *more* extreme values skewing the results of the clustering algorithm.

Freeing Up Memory

We now have quite a few variables/datasets in memory, so it's a good idea to free up some RAM by getting rid of anything we no longer need...

```
in_scope  = set([x for x in dir() if not x.startswith('_')])
to_delete = set(['raw','clean','transformed','col','k','c','lsoac','scdf'])
z = list(in_scope.intersection(to_delete))
del(z)
```

Clustering Your Data

OK, we're finally here! It's time to cluster the cleaned, normalised, and standardised data set! We're going to start with the best-known clustering technique (k-means) and work from there... Don't take my word for it, here are the 5 Clustering Techniques Every Data Scientist Should Know. This is also a good point to refer back to some of what we've been doing (and it's a good point to potentially disagree with me!) since clustering in high dimensions can be problematic (*i.e.* the more dimensions the worse the Euclidean distance gets as a cluster metric).

The effectiveness of clustering algorithms is usually demonstrated using the 'iris data' – it's available by default with both Seaborn and SciKit-Learn. This data doesn't usually need normalisation but it's a good way to start looking at the data across four dimensions and seeing how it varies and why some dimensions are 'good' for clustering, while others are 'not useful'...

Unfortunately, our data is a *lot* messier and has many more dimensions (>25) than this.

:::

- Find the appropriate eps value: Nearest Neighbour Distance Functions

Brief Discussion

In the practical I've followed the *Geocomputation* approach of basically converting everything to a share (percentage) and then clustering on that. This is *one* way to approach this problem, but there are *many* others. For instance, many people might skip the percentages part and apply robust rescaling (`sklearn.preprocessing.RobustScaler`) using centering and quantile standardisation (the 5th and 95th, for example) instead. And possibly using a normalising transformation (such as a Power Transform) as well.

I would also consider using PCA on groups of related variables (*e.g.* the housing features as a group, the ethnicity features as a group, etc.) and then take the first few eigenvalues from each group and cluster on all of those together. This would remove quite a bit of the correlation between variables and still allow us to perform hierarchical and other types of clustering on the result. It might also do a better job of preserving outliers.

Create an Output Directory and Load the Data

```
o_dir = os.path.join('outputs','clusters')
if os.path.isdir(o_dir) is not True:
    print("Creating '{0}' directory.".format(o_dir))
    os.makedirs(o_dir)
```

```
df = pd.read_csv(os.path.join('data','Scaled_and_Transformed.csv.gz'))
df.rename(columns={'mnemonic':'lsoacd'}, inplace=True)
df.set_index('lsoacd', inplace=True)
df.describe()
```

```
df.sample(3, random_state=42)
```

Grab Borough Boundaries and Water Courses

Note: if reading these GeoPackages gives you errors then you will need to comment out the following two lines from the `plt_ldn` function immediately below:

```
    w.plot(ax=ax, color='#79aef5', zorder=2)
    b.plot(ax=ax, edgecolor='#cc2d2d', facecolor='None', zorder=3)
```

```
# Load Water GeoPackage
w_path = os.path.join('data','Water.gpkg')
if not os.path.exists(w_path):
    water = gpd.read_file('https://github.com/kingsgeocomp/applied_gsa/raw/master/da
    water.to_file(w_path)
    print("Downloaded Water.gpkg file.")
else:
    water = gpd.read_file(w_path)

# Boroughs GeoPackage
b_path = os.path.join('data','Boroughs.gpkg')
if not os.path.exists(b_path):
    boroughs = gpd.read_file('https://github.com/kingsgeocomp/applied_gsa/raw/master
    boroughs.to_file(b_path)
    print("Downloaded Boroughs.gpkg file.")
else:
    boroughs = gpd.read_file(b_path)
```

Useful Functions for Plotting

```
def plt_ldn(w=water, b=boroughs):
    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

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")
```

Select 4 Columns to Plot

```
random.seed(42)
cols_to_plot = random.sample(population=list(df.columns.values), k=4)
print("Columns to plot: " + ", ".join(cols_to_plot))
```

Storing Results

```
result_set = None

def add_2_rs(s, rs=result_set):
    if rs is None:
        # Initialise
        rs = pd.DataFrame()
    rs[s.name] = s
    return rs
```

0.4.16 K-Means

Importing the Library

```
from sklearn.cluster import KMeans
#help(KMeans)
```

The next few code blocks may take a while to complete, largely because of the `pairplot` at the end where we ask Seaborn to plot every dimension against every other dimension *while* colouring the points according to their cluster. I've reduced the plotting to just three dimensions, if you want to plot all of them, then just replace the array attached to `vars` with `main_cols`, but you have to bear in mind that that is plotting 4,300 points *each* time it draws a plot... and there are 81 of them! It'll take a while, but it *will* do it, and try doing that in Excel or SPSS?

A First Cluster Analysis

```
c_nm   = 'KMeans' # Clustering name
k_pref = 6 # Number of clusters

# Quick sanity check in case something hasn't
# run successfully -- these muck up k-means
cldf = df.drop(list(df.columns[df.isnull().any().values].values), axis=1)

kmeans = KMeans(n_clusters=k_pref, n_init=20, random_state=42, n_jobs=-1).fit(cldf)

print(kmeans.labels_) # The results

# Add it to the data frame
cldf[c_nm] = pd.Series(kmeans.labels_, index=df.index)

# How are the clusters distributed?
cldf[c_nm].hist(bins=k_pref)

# Going to be a bit hard to read if
# we plot every variable against every
# other variables, so we'll just pick a few
sns.set(style="white")
sns.pairplot(cldf,
             vars=cols_to_plot,
             hue=c_nm, markers=".", height=3, diag_kind='kde')
```

```
cgdf = gdf.join(cldf, how='inner')

breaks = np.arange(0,cldf[c_nm].max()+2,1)
cmap   = default_cmap(len(breaks))

norm    = mpl.colors.BoundaryNorm(breaks, cmap.N)

fig, ax = plt_ldn()
fig.suptitle(f"{c_nm} Results (k={k_pref})", fontsize=20, y=0.92)
cgdf.plot(column=c_nm, ax=ax, cmap=cmap, norm=norm, linewidth=0, zorder=0)
```

```
add_colorbar(ax.collections[-1], ax, cmap, norm, breaks)

del(cgdf)
```

One More Thing...

There's just *one* little problem: what assumption did I make when I started this *k*-means cluster analysis? It's a huge one, and it's one of the reasons that *k*-means clustering *can* be problematic when used naively...

STOP. What critical assumption did we make when running this analysis?

:::

The 'Right' Number of Clusters

Again, there's more than one way to skin this cat. In *Geocomputation* 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...

Let's try it for the range 3-9.

```
# Adapted from: http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_sil
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score

cldf = df.drop(list(df.columns[df.isnull().any().values].values), axis=1)

text = []

for k in range(3,10):
    # Debugging
    print("Cluster count: " + str(k))

    #############
    # Do the clustering using the main columns
    clusterer = KMeans(n_clusters=k, n_init=15, random_state=42, n_jobs=-1)
    cluster_labels = clusterer.fit_predict(cldf)
```

```
    # Calculate the overall silhouette score
    silhouette_avg = silhouette_score(cldf, cluster_labels)
    text = text + [f"For k={k} the average silhouette_score is: {silhouette_avg:6.4f

    # Calculate the silhouette values
    sample_silhouette_values = silhouette_samples(cldf, cluster_labels)

    #############
    # 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, cldf.shape[0] + (k + 1) * 10])

    y_lower = 10

    # For each of the clusters...
    for i in range(k):
        # Aggregate the silhouette scores for samples belonging to
        # cluster i, and sort them
        ith_cluster_silhouette_values = \
            sample_silhouette_values[cluster_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
        #cmap  = cm.get_cmap("Spectral")
        color = plt.cm.Spectral(i/k)
        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="--")

    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(cluster_labels.astype(float) / k)
    ax2.scatter(cldf[cldf.columns[0]], cldf[cldf.columns[1]], marker='.', s=30, lw=0
                c=colors)

    # Labeling the clusters
    centers = clusterer.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 analysis for KMeans clustering on sample data "
                  "with n_clusters = %d" % k),
                 fontsize=14, fontweight='bold')

    plt.show()

print("\n".join(text))

del(cldf)
```

Interpreting the Results

STOP. Make sure that you understand how the silhouette plot and value work, and why your results may diverge from mine.

:::

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.

Final Clustering

Let's repeat the clustering process *one more time* using the silhouette score as a guide and then map it.

```
#| scrolled: true
c_nm = 'KMeans'
# Quick sanity check in case something hasn't
# run successfully -- these muck up k-means
cldf = df.drop(list(df.columns[df.isnull().any().values].values), axis=1)

k_pref = 4
kmeans = KMeans(n_clusters=k_pref, n_init=75, random_state=42).fit(cldf)

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

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

# We do this to keep track of the results
result_set=add_2_rs(s)
```

Mapping Results

```
cgdf = gdf.join(cldf, how='inner')

breaks = np.arange(0,cldf[c_nm].max()+2,1)
cmap   = default_cmap(len(breaks))

norm    = mpl.colors.BoundaryNorm(breaks, cmap.N)

fig, ax = plt_ldn()
fig.suptitle(f"{c_nm} Results (k={k_pref})", fontsize=20, y=0.92)
cgdf.plot(column=c_nm, ax=ax, cmap=cmap, norm=norm, linewidth=0, zorder=0)

add_colorbar(ax.collections[-1], ax, cmap, norm, breaks)

plt.savefig(os.path.join(o_dir,f"{c_nm}-{k_pref}.png"), dpi=200)
del(cgdf)
```

To make sense of whether this is a 'good' result, you might want to visit datashine or think back to last year when we examined the NS-SeC data.

You could also think of ways of plotting how these groups differ. For instance...

'Representative' Centroids

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 will since the clusters are explicitly built around

mean centroids. There's also a *k*-medoids clustering approach built around the median centroid.

```
centroids = None
for k in sorted(cldf[c_nm].unique()):
    print(f"Processing cluster {k}")

    clsoas = cldf[cldf[c_nm]==k]
    if centroids is None:
        centroids = pd.DataFrame(columns=clsoas.columns.values)
    centroids = centroids.append(clsoas.mean(), ignore_index=True)

odf = pd.DataFrame(columns=['Variable','Cluster','Std. Value'])
for i in range(0,len(centroids.index)):
    row = centroids.iloc[i,:]
    c_index = list(centroids.columns.values).index(c_nm)
    for c in range(0,c_index):
        d = {'Variable':centroids.columns[c], 'Cluster':row[c_index], 'Std. Value':r
        odf = odf.append(d, ignore_index=True)

g = sns.FacetGrid(odf, col="Variable", col_wrap=3, height=3, aspect=1.5, margin_titl
g = g.map(plt.plot, "Cluster", "Std. Value", marker=".")

del(odf, centroids, cldf)
```

0.4.17 DBScan

Of course, as we've said above *k*-means is just one way of clustering, DBScan is another. Unlike *k*-means, we don't need to specify the number of clusters in advance. Which sounds great, but we still need to specify *other* parameters (typically, these are known as *hyperparameters* because they are about specifying parameters that help the aglorithm to find the right solution... or final set of parameters!) and these can have a huge impact on our results!

Importing the Library

```
from sklearn.cluster import DBSCAN
#?DSCAN
```

Find a Reasonable Value for Epsilon

Before we an use DBSCAN it's useful to find a good value for Epsilon. We can look for the point of maximum 'curvature' in a nearest neigbhours plot. Which seems to be in the vicinity of 0.55. Tips on selecting `min_pts` can be found here.

```
# Quick sanity check in case something hasn't
# run successfully -- these muck up k-means
cldf = df.drop(list(df.columns[df.isnull().any().values].values), axis=1)
```

```
neigh = NearestNeighbors(n_neighbors=2)
nbrs = neigh.fit(cldf)
distances, indices = nbrs.kneighbors(cldf)

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

Exploration

There are two values that need to be specified: `eps` and `min_samples`. Both seem to be set largely by trial and error. 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.

WARNING. This next step may take quite a lot of time since we are iterating through many, many values of Epsilon to explore how the clustering result changes and how well this matches up with (or doesn't) the graph above.

:::

```
c_nm = 'DBSCAN'

# Quick sanity check in case something hasn't
# run successfully -- these muck up k-means
cldf = df.drop(list(df.columns[df.isnull().any().values].values), axis=1)

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

# There's an argument for making min_samples = len(df.columns)+1

el  = []

max_clusters  = 10
cluster_count = 1

iters = 0

for e in np.arange(0.15, 1.55, 0.01):

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

    # Run the clustering
    dbs = DBSCAN(eps=e, min_samples=12, n_jobs=-1).fit(cldf.values)

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

```
    row = [e]
    data = s.value_counts()
    for c in range(-1, max_clusters+1):
        try:
            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(
edf.sample(random_state=42)

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

print("Done.")
```

```
odf = 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 not np.isnan(row[c]):
            d = {'Epsilon':row[0], 'Cluster':f"Cluster {c-2}", 'Count':row[c]}
            odf = odf.append(d, ignore_index=True)
```

```
xmin = odf[odf.Cluster=='Cluster 0'].Epsilon.min()
xmax = odf[(odf.Cluster=='Cluster -1') & (odf.Count < cldf.shape[0]/5)].Epsilon.min(

fig, ax = plt.subplots(figsize=(12,8))
ax.set_xlim([xmin,xmax])
sns.lineplot(data=odf, x='Epsilon', y='Count', hue='Cluster')
```

Final Clustering

```
e = 0.835
dbs = DBSCAN(eps=e, min_samples=12, n_jobs=-1).fit(cldf.values)
s = pd.Series(dbs.labels_, index=cldf.index, name=c_nm)
cldf[c_nm] = s
result_set=add_2_rs(s)
print(s.value_counts())
```

```
cgdf = gdf.join(cldf, how='inner')

breaks = np.arange(cldf[c_nm].min(),cldf[c_nm].max()+2,1)
cmap   = default_cmap(len(breaks), outliers=True)
```

```
norm   = mpl.colors.BoundaryNorm(breaks, cmap.N, clip=False)

fig, ax = plt_ldn()
fig.suptitle(f"{c_nm} Results", fontsize=20, y=0.92)

cgdf.plot(column=c_nm, ax=ax, cmap=cmap, norm=norm, linewidth=0, zorder=0, legend=Fa

add_colorbar(ax.collections[-1], ax, cmap, norm, breaks, outliers=True)

plt.savefig(os.path.join(o_dir,f"{c_nm}.png"), dpi=200)
del(cgdf)
```

'Representative' Centroids

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). For algorithms other than *k*-means it may be better to use medians than means.

```
centroids = None
for k in sorted(cldf[c_nm].unique()):
    print(f"Processing cluster {k}")

    clsoas = cldf[cldf[c_nm]==k]
    if centroids is None:
        centroids = pd.DataFrame(columns=clsoas.columns.values)
    centroids = centroids.append(clsoas.mean(), ignore_index=True)

odf = pd.DataFrame(columns=['Variable','Cluster','Std. Value'])
for i in range(0,len(centroids.index)):
    row = centroids.iloc[i,:]
    c_index = list(centroids.columns.values).index(c_nm)
    for c in range(0,c_index):
        d = {'Variable':centroids.columns[c], 'Cluster':row[c_index], 'Std. Value':r
        odf = odf.append(d, ignore_index=True)

# Drop outliers
odf = odf[odf.Cluster >= 0]

g = sns.FacetGrid(odf, col="Variable", col_wrap=3, height=3, aspect=1.5, margin_titl
g = g.map(plt.plot, "Cluster", "Std. Value", marker=".")

del(odf, centroids)
```

0.4.18 OPTICS Clustering

This is a fairly new addition to `sklearn` and is similar to DBSCAN in that there are very few (if any) parameters to specify. This means that we're making fewer assumptions about the nature of any clustering in the data. It also allows us to have outliers that don't get assigned to *any* cluster. The focus is mainly on local density, so in some

sense it's more like a geographically aware clustering approach, but applied in the data space, not geographical space.

Importing the Library

```
from sklearn.cluster import OPTICS
```

Final Clustering

WARNING. This next step may take quite a lot of time since the algorithm is making far fewer assumptions about the structure of the data. On a 2018 MacBook Pro with 16GB of RAM it took about 5 minutes.

:::

```
c_nm = 'Optics'

# Can try to set this from DBSCAN results
e = 0.9850

# Quick sanity check in case something hasn't
# run successfully -- these muck up k-means
cldf = df.drop(list(df.columns[df.isnull().any().values].values), axis=1)

import math

# Run the clustering
opt = OPTICS(min_samples=len(df.columns)+1, max_eps=math.ceil(e * 100)/100, n_jobs=-

# See how we did
s = pd.Series(opt.labels_, index=cldf.index, name=c_nm)
cldf[c_nm] = s
result_set=add_2_rs(s)

# Distribution
print(s.value_counts())
```

Mapping Clustering Results

WARNING. My sense is that these results are a bit rubbish: the majority of items are assigned to *one cluster*??? I've tried PCA on the standardised data and that made little difference. This should also have worked *better* but it seems that a small number of LSOAs are so utterly different that the more sophisticated clustering algorithm effectively 'chokes' on them.

:::

```
cgdf = gdf.join(cldf, how='inner')

breaks = np.arange(cldf[c_nm].min(),cldf[c_nm].max()+2,1)
```

```
cmap   = default_cmap(len(breaks), outliers=True)
norm   = mpl.colors.BoundaryNorm(breaks, cmap.N, clip=False)

fig, ax = plt_ldn()
fig.suptitle(f"{c_nm} Results", fontsize=20, y=0.92)

cgdf.plot(column=c_nm, ax=ax, cmap=cmap, norm=norm, linewidth=0, zorder=0, legend=Fa

add_colorbar(ax.collections[-1], ax, cmap, norm, breaks, outliers=True)

plt.savefig(os.path.join(o_dir,f"{c_nm}.png"), dpi=200)
del(cgdf)
```

'Representative' Centroids

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). For algorithms other than *k*-Means it may be better to use medians, not means.

```
centroids = None
for k in sorted(cldf[c_nm].unique()):
    print(f"Processing cluster {k}")

    clsoas = cldf[cldf[c_nm]==k]
    if centroids is None:
        centroids = pd.DataFrame(columns=clsoas.columns.values)
    centroids = centroids.append(clsoas.mean(), ignore_index=True)

odf = pd.DataFrame(columns=['Variable','Cluster','Std. Value'])
for i in range(0,len(centroids.index)):
    row = centroids.iloc[i,:]
    c_index = list(centroids.columns.values).index(c_nm)
    for c in range(0,c_index):
        d = {'Variable':centroids.columns[c], 'Cluster':row[c_index], 'Std. Value':r
        odf = odf.append(d, ignore_index=True)

g = sns.FacetGrid(odf, col="Variable", col_wrap=3, height=3, aspect=1.5, margin_titl
g = g.map(plt.plot, "Cluster", "Std. Value", marker=".")

del(odf, centroids)
```

STOP. Aside from the fact that we should probably reduce the number of dimensions on which we're clustering, what about the process of selecting variables (a.k.a. feature selection) might have led to the result that our results are a bit crap? *Hint: how did we decide what to keep and what to drop, and is this a robust approach?*

:::

0.4.19 HDBSCAN

Not implemented, but you could give it a try after installing the package:

```
conda activate <your environment name here>
conda install -c conda-forge sklearn-contrib-hdbscan
```

Then it should be something like:

```
import hdbscan
clusterer = hdbscan.HDBSCAN()
# HDBSCAN(algorithm='best', alpha=1.0, approx_min_span_tree=True,
#     gen_min_span_tree=False, leaf_size=40, memory=Memory(cachedir=None),
#     metric='euclidean', min_cluster_size=5, min_samples=None, p=None)
clusterer.fit(<data>)
clusterer.labels_
```

0.4.20 Hierarchical Clustering

Probably not appropriate as it tends to be confused by noise.

0.4.21 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 going to see how this works over the next few code blocks, 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 tSNE (another form of dimensionality reduction).

(Re)Installing SOMPY

WARNING. The maintainers of the main SOMPY library are fairly inactive, so we've had to write our own version that fixes a few Python3 bugs, but this means that it can't be installed the 'usual' way without also having Git installed. Consequently, I have left the output from SOMPY in place so that you can see what it will produce *even if you cannot successfully install SOMPY during this practical*

To work out if there is an issue, check to see if the `import` statement below gives you errors:

```
from sompy.sompy import SOMFactory
```

If this import has failed with a warning about being unable to find SOM or something similar, then you will need to *re*-install SOMPY using a fork that I created on our Kings GSA GitHub account. For *that* to work, you will need to ensure that you have `git` installed.

If the following Terminal command (which should also work in the Windows Terminal) does not give you an error then `git` is already installed:

```
git --version
```

To install `git` on a Mac is fairly simple. Again, from the Terminal issue the following command:

```
xcode-select --install
```

This installation may take some time over eduroam since there is a lot to download.

Once that's complete, you can move on to installing SOMPY from our fork. On a Mac this is done on the Terminal with:

```
conda activate <your kernel name here>
pip install -e git+git://github.com/kingsgeocomp/SOMPY.git#egg=SOMPY
conda deactivate
```

On Windows you probably drop the `conda` part of the command.

Training the SOM

We are going to actually train the SOM using the input data. This is where you specify the input parameters that have the main effect on the clustering results.

```
from sompy.sompy import SOMFactory
```

```
c_nm = 'SOM'

# Quick sanity check in case something hasn't
# run successfully -- these muck up k-means
cldf = df.drop(list(df.columns[df.isnull().any().values].values), axis=1)

sm = SOMFactory().build(
    cldf.values, mapsize=(10,15),
    normalization='var', initialization='random', component_names=cldf.columns.value
sm.train(n_job=4, verbose=False, train_rough_len=2, train_finetune_len=5)
```

How good is the fit?

```
topographic_error  = sm.calculate_topographic_error()
quantization_error = np.mean(sm._bmu[1])
print("Topographic error = {0:0.5f}; Quantization error = {1:0.5f}".format(topograph
```

How do the results look?

```
from sompy.visualization.mapview import View2D
view2D = View2D(10, 10, "rand data", text_size=10)
view2D.show(sm, col_sz=4, which_dim="all", denormalize=True)
```

```
plt.savefig(os.path.join(o_dir, f"{c_nm}-Map.png"), dpi=200)
```

Here's What I Got

WARNING. These are the results from the approach that is closest to the one outlined in *Geocomputation*.

:::

How many data points were assigned to each BMU?

```
from sompy.visualization.bmuhits import BmuHitsView
vhts = BmuHitsView(15, 15, "Hits Map", text_size=8)
vhts.show(sm, anotate=True, onlyzeros=False, labelsize=9, cmap="plasma", logaritmic=
plt.savefig(os.path.join(o_dir,f"{c_nm}-BMU Hit View.png"), dpi=200)
```

BMU Hit Map

WARNING. These are the results from the approach that is closest to the one outlined in *Geocomputation*.

:::

How many clusters do we want and where are they on the map?

```
from sompy.visualization.hitmap import HitMapView

k_val = 5
sm.cluster(k_val)
hits  = HitMapView(15, 15, "Clustering", text_size=14)
a     = hits.show(sm)
plt.savefig(os.path.join(o_dir,f"{c_nm}-Hit Map View.png"), dpi=200)
```

Clustering the BMUs

WARNING. These are the results from the approach that is closest to the one outlined in *Geocomputation*.

:::

Finally, let's get the cluster results and map them back on to the data points:

```
# Get the labels for each BMU
# in the SOM (15 * 10 neurons)
clabs = sm.cluster_labels

try:
    cldf.drop(c_nm,inplace=True,axis=1)
except KeyError:
    pass
```

```
# Project the data on to the SOM
# so that we get the BMU for each
# of the original data points
bmus  = sm.project_data(cldf.values)

# Turn the BMUs into cluster labels
# and append to the data frame
s = pd.Series(clabs[bmus], index=cldf.index, name=c_nm)

cldf[c_nm] = s
result_set = add_2_rs(s)
```

```
cgdf = gdf.join(cldf, how='inner')

breaks = np.arange(cldf[c_nm].min(),cldf[c_nm].max()+2,1)
cmap   = default_cmap(len(breaks))
norm   = mpl.colors.BoundaryNorm(breaks, cmap.N, clip=False)

fig, ax = plt_ldn()
fig.suptitle(f"{c_nm} Results", fontsize=20, y=0.92)

cgdf.plot(column=c_nm, ax=ax, cmap=cmap, norm=norm, linewidth=0, zorder=0, legend=Fa

add_colorbar(ax.collections[-1], ax, cmap, norm, breaks)

plt.savefig(os.path.join(o_dir,f"{c_nm}.png"), dpi=200)
del(cgdf)
```

Result!

WARNING. These are the results from the approach that is closest to the one outlined in *Geocomputation*.

:::

Representative Centroids

```
centroids = None
for k in sorted(cldf[c_nm].unique()):
    print(f"Processing cluster {k}")

    clsoas = cldf[cldf[c_nm]==k]
    if centroids is None:
        centroids = pd.DataFrame(columns=clsoas.columns.values)
    centroids = centroids.append(clsoas.mean(), ignore_index=True)

odf = pd.DataFrame(columns=['Variable','Cluster','Std. Value'])
for i in range(0,len(centroids.index)):
    row = centroids.iloc[i,:]
```

```
    c_index = list(centroids.columns.values).index(c_nm)
    for c in range(0,c_index):
        d = {'Variable':centroids.columns[c], 'Cluster':row[c_index], 'Std. Value':r
        odf = odf.append(d, ignore_index=True)

g = sns.FacetGrid(odf, col="Variable", col_wrap=3, height=3, aspect=1.5, margin_titl
g = g.map(plt.plot, "Cluster", "Std. Value", marker=".")

del(odf, centroids)
```

0.5 Wrap-Up

- Find the appropriate eps value: Nearest Neighbour Distance Functions or Interevent Distance Functions
- Clustering Points
- Regionalisation algorithms with Aglomerative 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 clsuters?).

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.