Mirador Analytics - Data Science Task

1 k - Anonymity Analysis

a) Randomly generate a dataset (dataframe) with eight columns and 50,000 rows. Each column should be a categorical variable (of arbitrary name) with three levels (of arbitrary names) in approximately equal proportions.

We first define a dictionary, whose keys and values will be used to generate the dataframe. Then, using `numpy.random.choice` we can sample 50,000 times (with replacement) from the dictionary values and assign the sample to the dataframe using the dictionary keys as the column names. To ensure that the sample is repeatable, we also fix the random seed.

Creating the DataFrame

```python
import pandas as pd
import numpy as np

dictionary = dict({
    'A':['a1','a2','a3'],
    'B':['b1','b2','b3'],
    'C':['c1','c2','c3'],
    'I':['i1','i2','i3'],
    'J':['j1','j2','j3'],
    'K':['k1','k2','k3'],
    'X':['x1','x2','x3'],
    'Y':['y1','y2','y3']
})

np.random.seed(92)

def categorical_df_from_dict(d: dict, size: int) -> pd.DataFrame:
    """Creates a dataframe with columns = d.keys, where each column is populated
    ↪  with values randomly sampled from d.values."""

    df = pd.DataFrame()

    for key, val in d.items():

        df[key] = np.random.choice(val, size)

    return df

df = categorical_df_from_dict(dictionary, size=50000)
```

The first five rows of the resulting dataframe can be seen in Table 1.

Table 1: The first five rows of the dataframe `df`.

A	B	C	I	J	K	X	Y
a3	b2	c3	i1	j3	k3	x3	y1
a3	b1	c3	i2	j1	k3	x1	y2
a1	b1	c3	i3	j2	k3	x1	y3
a2	b3	c2	i1	j1	k1	x1	y1
a2	b3	c3	i2	j1	k3	x1	y1

b) Verify that the proportions of each value are similar for each of the eight columns.

Since the values are sampled with equal probability, we expect the distribution of the samples to be uniform. That is, we should see approximately 33% representation of each variable. This can be confirmed by looping over the columns in `df`, counting the unique values and dividing by the length of `df`.

Table 2: Representation of variables in each column

	a	b	c	i	j	j	x	y
1	0.33394	0.33356	0.33194	0.33266	0.33590	0.33884	0.33333	0.33054
2	0.33490	0.33686	0.33228	0.33426	0.33598	0.32884	0.32944	0.33614
3	0.33116	0.32958	0.33578	0.33308	0.32812	0.33232	0.33756	0.33332

Table 2 shows the proportions of each variable and was obtained by collecting the output of the code below.

Verifying Proportions

```
def verify_proportions(df: pd.DataFrame) -> None:
    """Prints the proportions of values in all columns in the dataframe df."""

    for col in df:
        vals, counts = np.unique(df[col], return_counts=True)

        print(vals)
        print(counts/sum(counts))

verify_proportions(df)
```

c) How many unique rows (i.e., permutations of category levels) are possible?

There are 6561 possible permutations, given by n^r, where n is the number of unique values in each column and r is the number of columns. Note that this only holds when there are an equal number of unique values in each column. A more general solution would be to use an n-fold cartesian product, which we make use of below to print the first five permutations.

Displaying Permutations

```
import itertools

def print_permutations(n: int) -> None:
    """Prints the first n permutations from itertools.product."""

    for num, perm in enumerate(itertools.product('123', repeat=8)):
        perm = list(perm)
        letters = ['a','b','c','i','j','k','x','y']
        for x, y in enumerate(perm):
            perm[x] = letters[x]+y
        print(perm)
        if num >= n-1:
            break

print_permutations(n=5)
```

Executing the above code results in the following output:

```
['a1', 'b1', 'c1', 'i1', 'j1', 'k1', 'x1', 'y1']
['a1', 'b1', 'c1', 'i1', 'j1', 'k1', 'x1', 'y2']
['a1', 'b1', 'c1', 'i1', 'j1', 'k1', 'x1', 'y3']
['a1', 'b1', 'c1', 'i1', 'j1', 'k1', 'x2', 'y1']
['a1', 'b1', 'c1', 'i1', 'j1', 'k1', 'x2', 'y2']
```

d) Produce a table and appropriate graph which show the frequencies (numbers of groups) by permutation group sizes up to group size of 12. That is, how many groups are unique combinations (group size = 1), how many groups are made up of a pair of matching combinations (group size = 2), how many groups are made up three the same, etc?

First, let's compute the frequencies of each permutation using `.groupby()` and `.size()`, then remove any where the group size is greater than 12 using `.query()`.

Creating the Frequency Table

```
unique_perms = (df
    .groupby(df.columns.tolist(), as_index=False)
    .size()
    .query('size <= 12')
)
display(unique_perms.groupby('size').count().iloc[:,0])
```

This gives us all unique permutations in `df`, with an extra column `size`, recording how many times each permutation was observed. Grouping again by size and counting the values gives us the frequencies seen in Table 3.

Table 3: Size frequency table

Group Size	No of Groups
1	24
2	122
3	220
4	453
5	625
6	863
7	974
8	989
9	766
10	580
11	398
12	230

From this we can produce the histogram displayed in Figure 1.

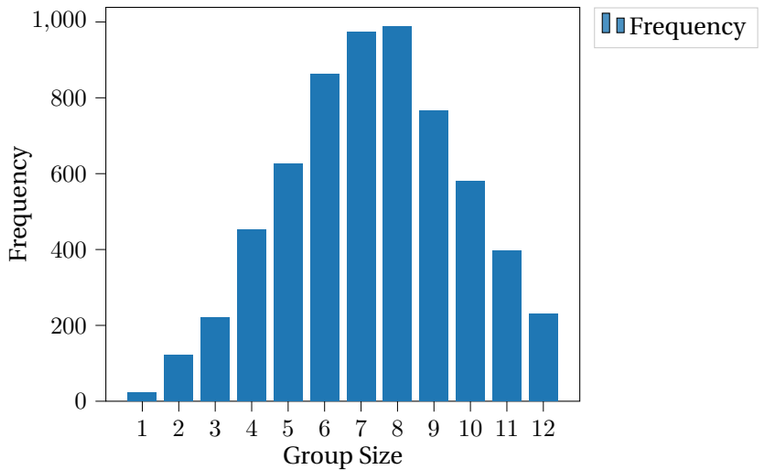

Figure 1: A histogram of the group size frequencies.

The code to produce Figure 1 is below:

Creating the Frequency Plot

```
import matplotlib.pyplot as plt

sizes = unique_perms.loc[:, 'size']

def plot_frequency_hist(x: pd.Series) -> tuple:
    """Creates the frequency histogram plot of the series x."""

    fig, ax = plt.subplots(1, 1, figsize=(10,5))

    # define bins and binwidth
    bins = np.arange(min(x.unique()), min(x.unique()) + len(x.unique()) + 1)
    binwidth = 0.8

    # plot hist
    plt.hist(x.values, bins=bins-0.5*binwidth, width=binwidth)

    # formatting
    ax.set_xlabel('Group Size'); ax.set_ylabel('Frequency')
    ax.set_xlim([min(x.unique()) - 1, min(x.unique()) + len(x.unique())])
    ax.set_xticks(bins[:-1])
    ax.set_xticklabels(labels=bins[:-1],rotation=45)

    return fig, ax

fig, ax = plot_frequency_hist(sizes)

plt.show()
```

e) Comment upon the distribution of group sizes in d).

This particular sample appears to follow a zero-truncated poisson distribution with a slight positive skew (obscured by the removal of sizes > 12). The distribution can be approximated using the normal distribution as shown in Figure 2.

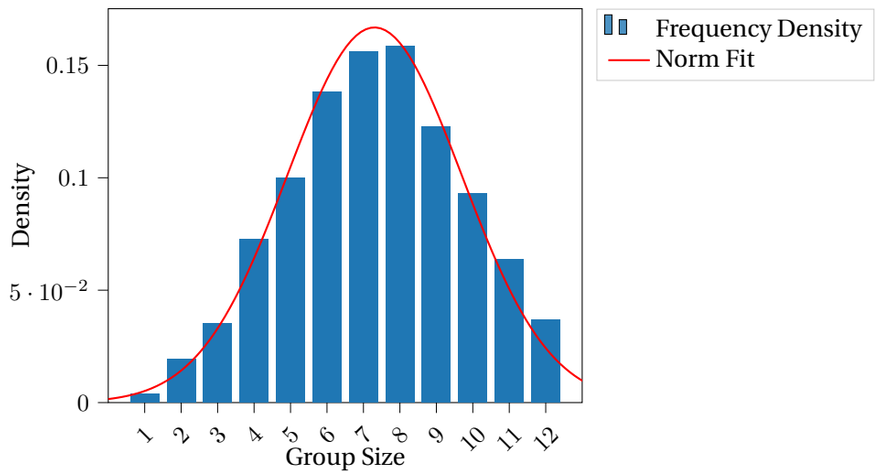

Figure 2: A gaussian fitted to the frequency density of group sizes.

Creating a Frequency Density Plot and Fitting a Distribution

```
import scipy.stats as st

def hist_fit(x: pd.Series, dists: list) -> tuple:
    """Creates density histogram and fits the distributions in dists."""

    fig, ax = plt.subplots(1, 1, figsize=(10,5))

    # define bins and binwidth
    bins = np.arange(min(x.unique()), min(x.unique()) + len(x.unique()) + 1)
    binwidth = 0.8

    # plot density
    ax.hist(x.values, bins=bins - 0.5*binwidth, width=binwidth, density=True,
    ↪  label='Frequency Density')

    for dist_name in dists:
        xmin, xmax = ax.get_xlim()
        xx = np.linspace(xmin, xmax, 100)

        dist = getattr(st, dist_name)
        params = dist.fit(x)

        p = dist.pdf(xx, *params)
        ax.plot(xx, p, linewidth=2, label=dist_name.title() + ' Distribution
        ↪  Fit')

    # formatting
    ax.set_xlabel('Group Size')
    ax.set_ylabel('Density')
    ax.set_xlim([min(x.unique()) - 1, min(x.unique()) + 1 + len(x.unique())])
    ax.set_xticks(bins[:-1])
    ax.set_xticklabels(labels=bins[:-1], rotation=45)
    plt.legend()

    return fig, ax

# experiment with other continuous distributions by adding them to the list
fig, ax = hist_fit(sizes, dists=['norm'])

plt.show()
```

f) If your random variables were, in fact, meaningful information on individuals, which group sizes are of most concern from a privacy perspective?

Since the goal of k-anonymity is to maximise k, we should be most concerned about the smaller group sizes, as it would be easier to match the variables to a specific individual or group of individuals.

g) Consider the effect of missing data in the dataset you created in Part a). How might this complicate the production of a frequency table of group sizes in Part d)?

The effect of missing data is something that we can test for quite easily by creating a function to randomly assign `np.nan` to values in our categorical dataframe. A function to do this is given below and replaces 20% by default.

Creating Missing Values

```
def create_missing_values(df: pd.DataFrame, pct_missing: float=0.2) ->
↪  pd.DataFrame:
    """ Chooses random rows and columns and replaces the values with np.nan
    ↪  until the desired portion of missing data is met. """

    n = int(pct_missing*df.shape[0]*df.shape[1])

    # define all possible row/column index combinations
    all_rc_pairs = np.array(list(itertools.product(range(0,len(df)),
    ↪  range(0,8))))

    # randomly select a subset of size n (no replacement)
    idx = np.random.choice(len(all_rc_pairs), replace=False, size=n)
    subset_rc_pairs = all_rc_pairs[idx]

    # replace with np.nan
    for x, y in subset_rc_pairs:
        df.iat[x, y] = np.nan

    return df
```

Re-applying the process described in 1. d) we observe that `np.nan` values are removed by default during the `.groupby()` step. This leaves us with fewer unique permutations and smaller group sizes. Alternatively, setting `dropna=False` introduces a fourth option for each column, which in turn increases the total number of unique combinations but lowers the probability of seeing the same permutation more than once, again leading to smaller group sizes. The existence of these pseudo-group sizes is undesireable, as any individuals that were legitimately at risk of having their privacy compromised are now buried amongst the new cases with low group size, making it more difficult to properly assess the anonymity of the data.

h) Imagine the code that you wrote for Part d) was to be deployed in an automated system that Mirador's customers could use independently, on potentially large volumes of their own data. Describe how you might deploy the code, and what additional considerations you might have or any changes to the code you might make. Note: it is not necessary to provide another version of the code created for d).

If the customers are to use the tool independently then there are a number of deployment options. For example, you could create a GUI (PyQt, tkinter), or dashboard (Plotly/Dash), or if the end user is expected to have a technical background then you could supply a `requirements.txt` file, or Docker environment and have them run a script manually. Ideally, clear and easy-to-follow documentation on how to use the tool would also be supplied.

We'd have to enforce a list of acceptable data formats e.g. `.csv`, `.xlsx`, and consider the possibility that the data will contain non-categorical variables such as height or weight. If such values are allowed, then the software will highlight to the user that their data has severe privacy concerns, as continuous values are much more likely to be unique, resulting in a low k-value. The alternative is to filter them out or educate the client in binning the variables or removing them entirely prior to analysis.

With regards to processing speed, assuming that we're no longer filtering out group sizes > 12, as a minimum we could re-run our code with more samples and measure the time it takes to produce the frequency table and graph. This would test its ability to process larger volumes of data. Table 4 below shows the typical times we could expect, running on an Intel i9 9900k @ 5 GHz. However, without knowing the time constraints and dimensions of the data we're dealing with it would be difficult to comment on whether these speeds are fast enough. If the software is expected to be run locally on the client's computer, then it would be worth re-testing on a lower-spec machine.

Table 4: Computation times (seconds) for different sample sizes (thousands).

Samples (k)	Time to Filter (s)	Time to Plot (s)
50	0.5	0.1
500	0.2	0.9
5000	2.1	2.3
50000	52	7.1

As you increase the number of samples in the data, you increase the probability of seeing more unique group sizes. This will become a problem when plotting the frequency histogram, as each group size is currently represented as a single bar on the chart and too many would flood the chart leading to overlapping xticklabels. To solve this we may want to implement an automated system to bin the higher group sizes, as it's the smaller ones we're most interested in.

It would also be beneficial to write some unit tests to import some sample files, checking that everything is computed and displayed correctly. This would help to ensure that the core functionality of the system remains in tact, should any changes be made to the code at a later date.

2 Postcodes and Privacy

In the US, 5-digit Zip codes are usually rounded to 3-digits when anonymizing health data, so knowledge of the Zip code doesn't allow small groups to be identified. Even then, there are some 3-digit codes that have fewer than 20,000 residents, and the advice is to lump these together under a new code (000).

Looking forward to how GDPR may affect data handling in the UK, might a similar approach be possible here? In answering this, use the data below. You might want to include some examples of any postcodes which could be problematic. Write it up as a mini report to inform the decision of a privacy offcer.

UK population by postcode data (28 MB) found here:

`www.nomisweb.co.uk/output/census/2011/Postcode_Estimates_Table_1.csv.`

In the UK, postcodes are alphanumeric with lengths ranging from six to eight characters (including a space). The postcode is divided into two parts separated by a single space: the outward code and the inward code respectively. The outward code includes the area and district, and the inward code contains the sector and unit. Some examples of UK postcodes and their structure can be seen in Table 5.

Table 5: Structure of UK postcodes, with examples.

Postcode			
Outward code		Inward code	
Area	District	Sector	Unit
TD	6	9	EF
AL	10	0	AB
B	1	1	BA

The following subsections will cover two possible ways to anonymize UK postcodes: the outward code and the area.

2.1 Area Code

We will perform our analysis of the postcodes using Python, starting by importing the file.

Importing the CSV

```
import pandas as pd

postcodes = pd.read_csv('./postcodes.csv')
```

Upon initial inspection of the format of the postcodes, we notice that a strict character length of 7 is enforced. This means that postcodes where the outward code is four characters long have no spaces and those where the area code is a single character have a double space. Therefore simply splitting on the space character will not work.

Splitting the Postcodes

```
def split_postcode(s: str) -> tuple:
    """
    Splits postcode into outward and inward.
    """
    # replace double spaces in postcodes where area is only a single letter
    s = s.replace("  "," ")

    if " " not in s:
        outward = s[:4]
        inward = s[-3:]
    elif " " in s:
        outward, inward = s.split(" ")

    return outward, inward

# apply the above function to create outward and inward columns
postcodes[['Outward','Inward']] = postcodes.apply(lambda x:
↪  split_postcode(x['Postcode']), axis=1, result_type='expand')

# use regex to capture the first one or two letters in the outward code
postcodes['Area'] = postcodes['Outward'].apply(lambda s:
↪  re.match(r"([A-Z]{1,2})", s).groups()[0])
```

Instead, we use the above function, which replaces double spaces with single spaces, ensuring all postcodes should have either a single space, or no space at all. Those that have a single space can be split on the space character, and those with no space at all can be split based on the first four and last three characters. Finally, the area can be extracted by using regex to capture the first one or two letters in the outward code.

Low Population Area Codes

```
area_grouped = (
    postcodes
    .groupby(['Area'])
    .sum()
    .sort_values('Total')
)

display(area_grouped.query('Total < 20000'))
```

Grouping by area code and counting the number of residents in each area reveals that there are just two area codes that do not meet the 20,000 residents threshold. Namely, DG and TD which both lie on the England-Scotland border, highlighted in Figure 3, which uses a traffic light system to indicate population. You can download the UK shapefile from https://www.opendoorlogistics.com/wp-content/uploads/Data/UK-postcode-boundaries-Jan-2015.zip.

Table 6: Area codes with fewer than 20,000 residents.

Index	Area Code	Total	Males	Females	Occupied_Households
0	DG	65	30	35	26
1	TD	18331	8838	9493	8262

Short of merging with the neighbouring English areas, the best we can do here is to combine these into a single 'Scottish' category. It's worth noting however that the original source of the data states that the postcode file is for England and Wales only, which means that these areas have a much higher population than is indicated by the file.

We can therefore conclude that area code is a suitable substitute for English/Welsh postcodes with regards to anonymisation. However, the problem with this approach is that, for some applications, the area code might be too general. If, for example, the data is to be used to study patient health vs proximity to a motorway or other source of pollution, the area code becomes less useful. Here, the outward code might provide more insight.

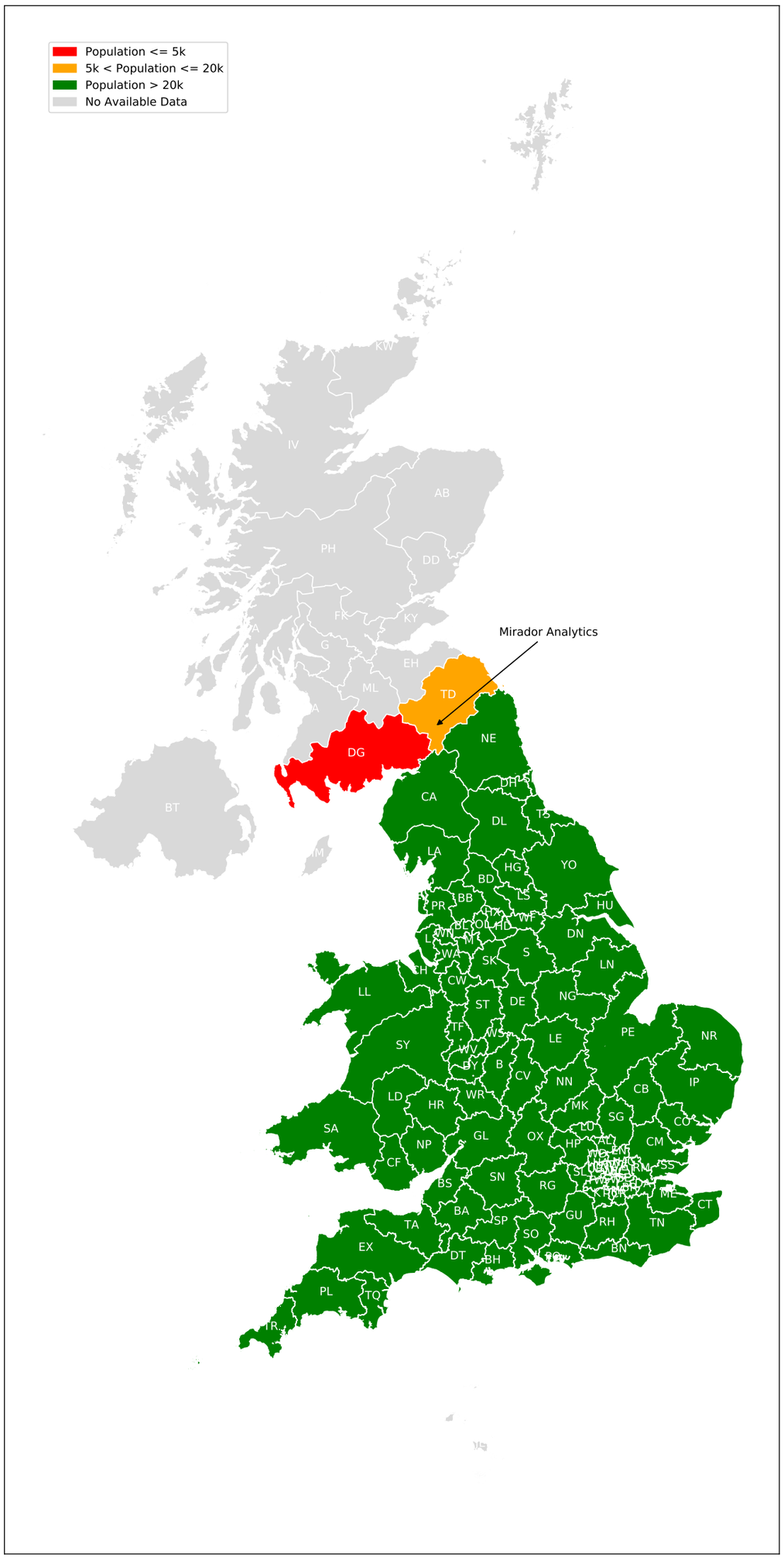

Figure 3: Population in England, Wales and Scotland (partial) by area using 2011 census data.

2.2 Outward Code

With the outbound code computed from the previous subsection, the code below can be used to display the outbound codes with fewer than 20,000 residents, the results of which can be found in Table 7.

Low Population Outward Codes

```
outward_grouped = (
    postcodes
    .groupby('Outward')
    .sum()
    .sort_values('Total')
    .reset_index()
)

display(outward_grouped.query('Total < 20000'))
```

Table 7: Outward codes with fewer than 20,000 residents.

Index	Outward Code	Total	Males	Females	Occupied_Households
0	UB11	2	1	1	1
1	EC2N	5	4	1	4
2	EC3M	6	4	2	6
3	DG14	7	4	3	4
4	N1C	7	3	4	3
⋮	⋮	⋮	⋮	⋮	⋮
1005	BS31	19955	9596	10359	8590
1006	GL53	19968	9673	10295	8456
1007	L30	19983	9392	10591	8525
1008	BH1	19992	10717	9275	10265
1009	IG7	19996	9595	10401	7790

We see that there are 1010 outward codes with fewer than 20,000 residents, resulting in large areas of red and amber in Figure 4, particularly in Scotland and Wales. The approach of grouping these into a new code (e.g. 000) is extreme and sacrifices a lot of geographical information.

2.3 Mixed Approach

An alternative would be to map outward codes with insufficient population to their area codes from Section 2.1, leaving the rest unchanged. This can be achieved using the code below:

Mixing Outward Code and Area

```
# find outward codes with small population
bad_outward = outward_grouped.query('Total < 20000')['Outward']

# extract area code from bad_outward
bad_outward_area = [re.match(r"([A-Z]{1,2})", s).groups()[0] for s in
↪  bad_outward]

# create a dictionary from bad_outward and respective areas
outward_area_dict = dict(zip(bad_outward, bad_outward_area))

# outward codes with sufficient population map back to their original values
for outward_code in postcodes['Outward'].unique():
    if outward_code not in outward_area_dict.keys():
        outward_area_dict[outward_code] = outward_code

# use dictionary to create new column called Outward_Area_Mix
postcodes['Outward_Area_Mix'] = postcodes['Outward'].map(outward_area_dict)

# check the results
(postcodes
    .groupby('Outward_Area_Mix')
    .sum()
    .sort_values('Total')
    .query('Total < 20000')
)
```

The results are given in Table 8, and show a significant improvement over using just the outward code. The mixed approach offers detailed geographical information where possible, and falls back to a sensible alternative otherwise. The downside is that there are still areas from the mixed approach which still do not meet the 20k threshold.

Table 8: Outward code & area code mixed with fewer than 20,000 residents.

Outward_Area_Mix	Total	Males	Females	Occupied_Households
DG	65	30	35	26
SR	5336	2890	2446	2595
CR	10220	4947	5273	4063
UB	14338	6939	7399	5980
PR	15387	8587	6800	5518
EN	17203	8343	8860	6855
SL	17531	8631	8900	7109
TD	18331	8838	9493	8262
HG	18374	8942	9432	7812
SM	18559	8826	9733	7266
DH	19570	9447	10123	8681
WN	19742	9978	9764	9073

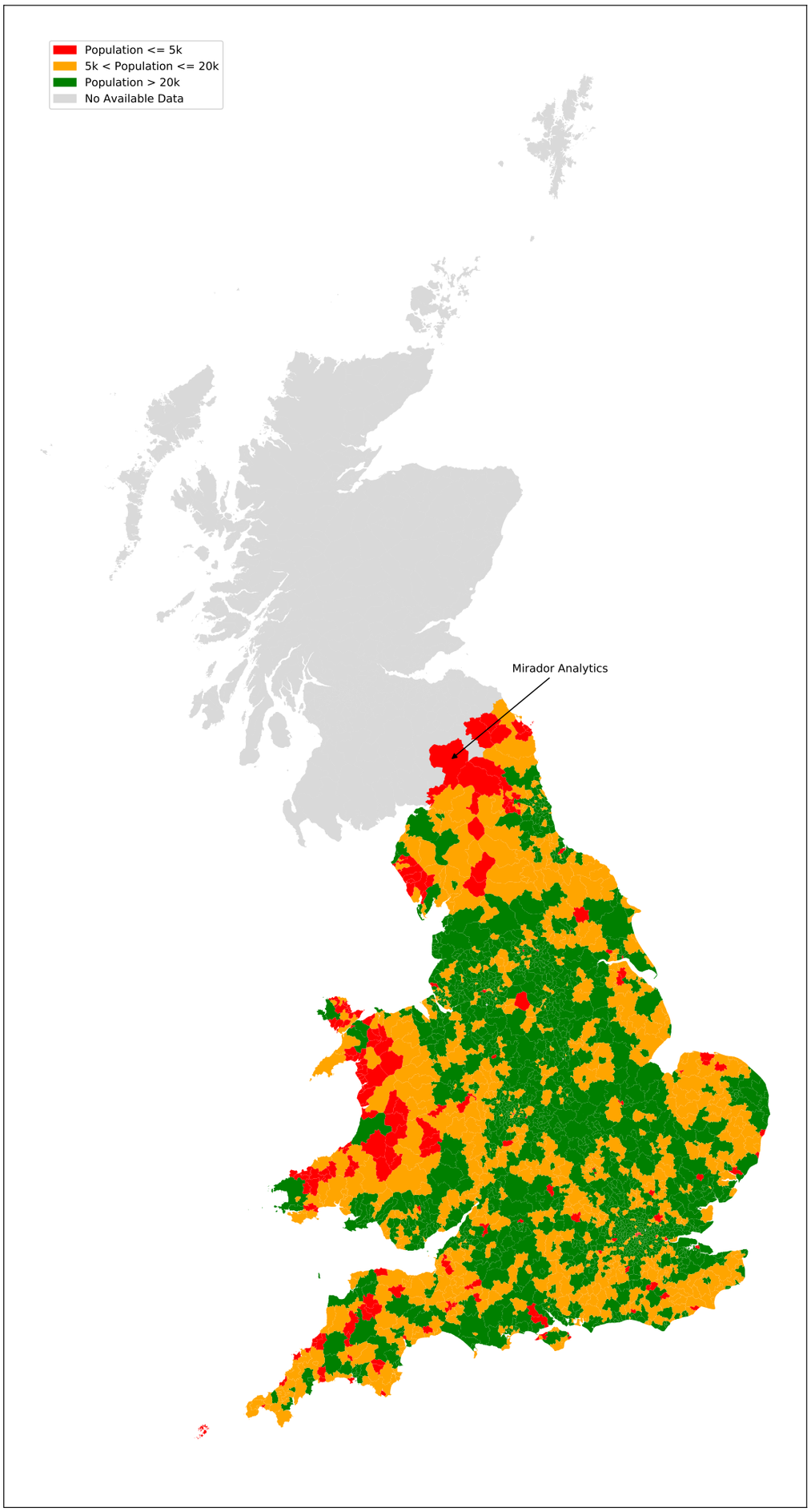

Figure 4: Population in England, Wales and Scotland (partial) by outward code using 2011 census data.