Data Cleansing - Detecting and fixing defects in data

Dr Rob Collins

Version 6, 31st July 2022

In this session we will look at the first stage of a typical Machine Learning / Data Analytics workflow .. that of importing and 'cleansing' data. In this context 'cleansing' means detecting and 'repairing' defects in data. Defects may be one or more of:

- Missing data
- Data of the wrong type (E.g. strings rather than numbers)
- Data in the wrong range (E.g. an age of a person given as 300 years old)

Instructions for Students

Student activities are indicated in the workbook below with a '**Student Task**' indicator.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. Execute **and experiment with** each of the code blocks to develop and check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

1 Some simple, introductory examples

In this first section we will start with some simple examples. In this case we will build the data ourselves using rather than using real data. This is intended just to get us started using the tools of data cleansing.

We start by loading two libraries that are very popular in Machine Learning. 'Pandas' provides many functions for manipulating tabulated data. 'numpy' provides many numerical algorithms that operate over tables.

In [29]:
```
import pandas as pd
import numpy as np
```

We first create a set of data with a few missing (NaN) values. This example data set will represent the kind of data we often obtain in the real world that has missing data due to failures of data collection or other datacorruption.

This set of data will be created as a Pandas 'DataFrame'. The dataframe will be first filled with random numbers (using the numpy 'random' function). Secondly, a selection of the numbers will be corrupted with 'NaN' (that is, 'not a number') values.

In [30]:
```
df = pd.DataFrame(np.random.randn(8,7))        # Creates a dataframe containing random numbers
df[df > 0.9] = np.nan                          # If a value in the table is greater than 0.9 - make it NaN
df
```

Out[30]:

	0	1	2	3	4	5	6
0	-1.349202	-0.232719	-0.634019	-2.256045	-2.174431	0.089683	-1.602224
1	NaN	NaN	0.026192	0.466882	-0.882645	-0.889865	NaN
2	-1.666124	-0.163158	0.168628	-1.614930	-0.723433	NaN	-0.385234
3	-0.307799	-0.288053	0.695800	-0.490723	-0.124151	-0.621850	-0.318540
4	-0.376620	NaN	-1.073321	-0.580900	0.085371	NaN	0.738210
5	NaN	-0.851670	-0.820428	NaN	-0.569879	-0.898899	-0.844356
6	-0.335798	0.390080	0.260919	0.478434	-0.545533	0.284016	0.413993
7	0.436552	-0.636683	-0.602772	-0.557860	-0.563691	0.382948	-0.330704

You should see that several of the cells are indicated as 'NaN'.

In practice, with very large tables of data, it will be typically impossible to hunt for missing values simply by looking at tabulated data in this way. We therefore need a method for searching for missing values.

Student task : Experiment by generating tables of different sizes above. Experiment by converting more or less of the cells into 'NaN' values.

We can use the following Python code to detect if any missing values exist within the table (of course, we know that they do because we just put them there and we can see them. But this function provides a general method that can applied to large tables)

In [31]:
```
print(df.isnull())
```

```
       0      1      2      3      4      5      6
0  False  False  False  False  False  False  False
1   True   True  False  False  False  False   True
2  False  False  False  False  False   True  False
3  False  False  False  False  False  False  False
4  False   True  False  False  False   True  False
5   True  False  False   True  False  False  False
6  False  False  False  False  False  False  False
7  False  False  False  False  False  False  False
```

That isn't very easy to read .. so maybe it would be better to get a summary of missing data for each column

In [32]:
```
print(df.isnull().sum())
```

```
0    2
1    2
2    0
3    1
4    0
5    2
6    1
dtype: int64
```

Or even a total count of missing data for the whole data set

In [33]:
```
print(df.isnull().sum().sum())
```

```
8
```

1.1 Dropping (Deleting) rows with missing data

One approach to dealing with missing data is to drop (delete) each row that contains a missing value.

To make it clear what is happening here, we will first rename the rows of our dataframe:

In [34]:
```
df.index = ['a','b','c','d','e','f','g','h']
df
```

Out[34]:

	0	1	2	3	4	5	6
a	-1.349202	-0.232719	-0.634019	-2.256045	-2.174431	0.089683	-1.602224
b	NaN	NaN	0.026192	0.466882	-0.882645	-0.889865	NaN
c	-1.666124	-0.163158	0.168628	-1.614930	-0.723433	NaN	-0.385234
d	-0.307799	-0.288053	0.695800	-0.490723	-0.124151	-0.621850	-0.318540
e	-0.376620	NaN	-1.073321	-0.580900	0.085371	NaN	0.738210
f	NaN	-0.851670	-0.820428	NaN	-0.569879	-0.898899	-0.844356
g	-0.335798	0.390080	0.260919	0.478434	-0.545533	0.284016	0.413993
h	0.436552	-0.636683	-0.602772	-0.557860	-0.563691	0.382948	-0.330704

Now we can drop (delete) any rows that contain an 'Nan' value

In [35]:
```
df.dropna()
```

Out[35]:

	0	1	2	3	4	5	6
a	-1.349202	-0.232719	-0.634019	-2.256045	-2.174431	0.089683	-1.602224
d	-0.307799	-0.288053	0.695800	-0.490723	-0.124151	-0.621850	-0.318540
g	-0.335798	0.390080	0.260919	0.478434	-0.545533	0.284016	0.413993
h	0.436552	-0.636683	-0.602772	-0.557860	-0.563691	0.382948	-0.330704

However, as you can probably see, the issue here is that we will often delete a large amount of otherwise useful data.

Thus we will normally require a more efficient way of dealing with missing values. That is what we do in the next section

Student task : Experiment with the sequence of code blocks above. Generate different size tables with more or less Nan values - then delete the rows containing NaNs. Review the impact that this process has in terms of loosing valuable data.

1.2 Fixing missing data by subsituting in zeros ...

One approach to fixing missing data would be to subsitute in some known value, as follows:

In [36]:
```
df.fillna(0, inplace=True)
print(df)
```

```
          0         1         2         3         4         5         6
a -1.349202 -0.232719 -0.634019 -2.256045 -2.174431  0.089683 -1.602224
b  0.000000  0.000000  0.026192  0.466882 -0.882645 -0.889865  0.000000
c -1.666124 -0.163158  0.168628 -1.614930 -0.723433  0.000000 -0.385234
d -0.307799 -0.288053  0.695800 -0.490723 -0.124151 -0.621850 -0.318540
e -0.376620  0.000000 -1.073321 -0.580900  0.085371  0.000000  0.738210
f  0.000000 -0.851670 -0.820428  0.000000 -0.569879 -0.898899 -0.844356
g -0.335798  0.390080  0.260919  0.478434 -0.545533  0.284016  0.413993
h  0.436552 -0.636683 -0.602772 -0.557860 -0.563691  0.382948 -0.330704
```

1.3 Fixing missing data by subsituting in the average of the feature (column)

However, in most cases 'zero' will not be a good substitute value. When we are building models using numerical data, a frequent choice of data-subsitution is to replace missing values in a feature (column) for the mean value of that column. The technical term for 'filling in' missing data like this is 'imputation'.

Since we have already 'fixed' our first data-set, we need to create some more data containing errors:

In [37]:
```
df = pd.DataFrame(np.random.randn(7,6))
df[df > 0.9] = np.nan
df
```

Out[37]:

	0	1	2	3	4	5
0	-0.907146	0.366926	0.245972	0.247959	-0.386248	-0.947796
1	-1.158287	0.518193	NaN	-0.262784	0.686137	-1.483846
2	-0.288352	0.283960	-0.626979	0.508049	0.470081	NaN
3	0.379275	-0.548341	-0.863544	-0.626385	-0.107092	-1.506507
4	-0.520570	-0.681992	0.393395	NaN	-0.231304	0.541660
5	NaN	-0.576182	-2.299372	0.192045	-0.709031	-1.076354
6	0.764776	NaN	-0.849415	NaN	-1.144773	NaN

The following code fixes the first column, by replacing every 'NaN' value, with the mean of the column:

In [38]:
```
df[0].fillna(df[0].mean(), inplace = True)
df
```

Out[38]:

	0	1	2	3	4	5
0	-0.907146	0.366926	0.245972	0.247959	-0.386248	-0.947796
1	-1.158287	0.518193	NaN	-0.262784	0.686137	-1.483846
2	-0.288352	0.283960	-0.626979	0.508049	0.470081	NaN
3	0.379275	-0.548341	-0.863544	-0.626385	-0.107092	-1.506507
4	-0.520570	-0.681992	0.393395	NaN	-0.231304	0.541660
5	-0.288384	-0.576182	-2.299372	0.192045	-0.709031	-1.076354
6	0.764776	NaN	-0.849415	NaN	-1.144773	NaN

We can repeat this operation for every column in the dataframe as follows

In [39]:
```
for column_name in df:
    df[column_name].fillna(df[column_name].mean(), inplace = True)
df
```

Out[39]:

	0	1	2	3	4	5
0	-0.907146	0.366926	0.245972	0.247959	-0.386248	-0.947796
1	-1.158287	0.518193	-0.666657	-0.262784	0.686137	-1.483846
2	-0.288352	0.283960	-0.626979	0.508049	0.470081	-0.894569
3	0.379275	-0.548341	-0.863544	-0.626385	-0.107092	-1.506507
4	-0.520570	-0.681992	0.393395	0.011777	-0.231304	0.541660
5	-0.288384	-0.576182	-2.299372	0.192045	-0.709031	-1.076354
6	0.764776	-0.106239	-0.849415	0.011777	-1.144773	-0.894569

2 Cleaning more complicated data

For this part of the activity we are going to use a much larger data-set. This data has been obtained from Kaggle at the following URL: https://www.kaggle.com/c/sberbank-russian-housing-market/data. This data is also available as part of this tutorial as 'Sberbank Housing Market.csv'.

We will be using the 'seaborne' library to help visualise the dataset: http://seaborn.pydata.org/

In [40]:
```
import seaborn as sns
```

We also need another library to help us plot data. We are selecting some specific functions from that library as well as setting up the style of output:

In [41]:
```
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib
plt.style.use('ggplot')
from matplotlib.pyplot import figure


%matplotlib inline
matplotlib.rcParams['figure.figsize'] = (12,8)
```

First we read in the data-set

In [42]:
```
# read the data
df = pd.read_csv('Sberbank Housing Market.csv')
```

We can determine the size and shape of the data as follows:

In [43]:
```
df.shape
```

Out[43]:
```
(30471, 292)
```

Which should be 30471 rows of data and 292 columns.

Next we can determine the types of each feature in the data-set

In [44]:
```
df.dtypes
```

Out[44]:
```
id                     int64
timestamp             object
full_sq                int64
life_sq              float64
floor                float64
                      ...
mosque_count_5000      int64
leisure_count_5000     int64
sport_count_5000       int64
market_count_5000      int64
price_doc              int64
Length: 292, dtype: object
```

2.1 A quick visualisation of the data

It is often useful to visualise data ahead of any processing. The following enables us to review multiple features and their relationships. The histograms on the diagonal of the following chart show the distribution of specific features. The other cells provide a scatter plots where one feature is plotted against another in an X-Y scatter plot.

First we import a specific function that enables easy creation of the chart we need:

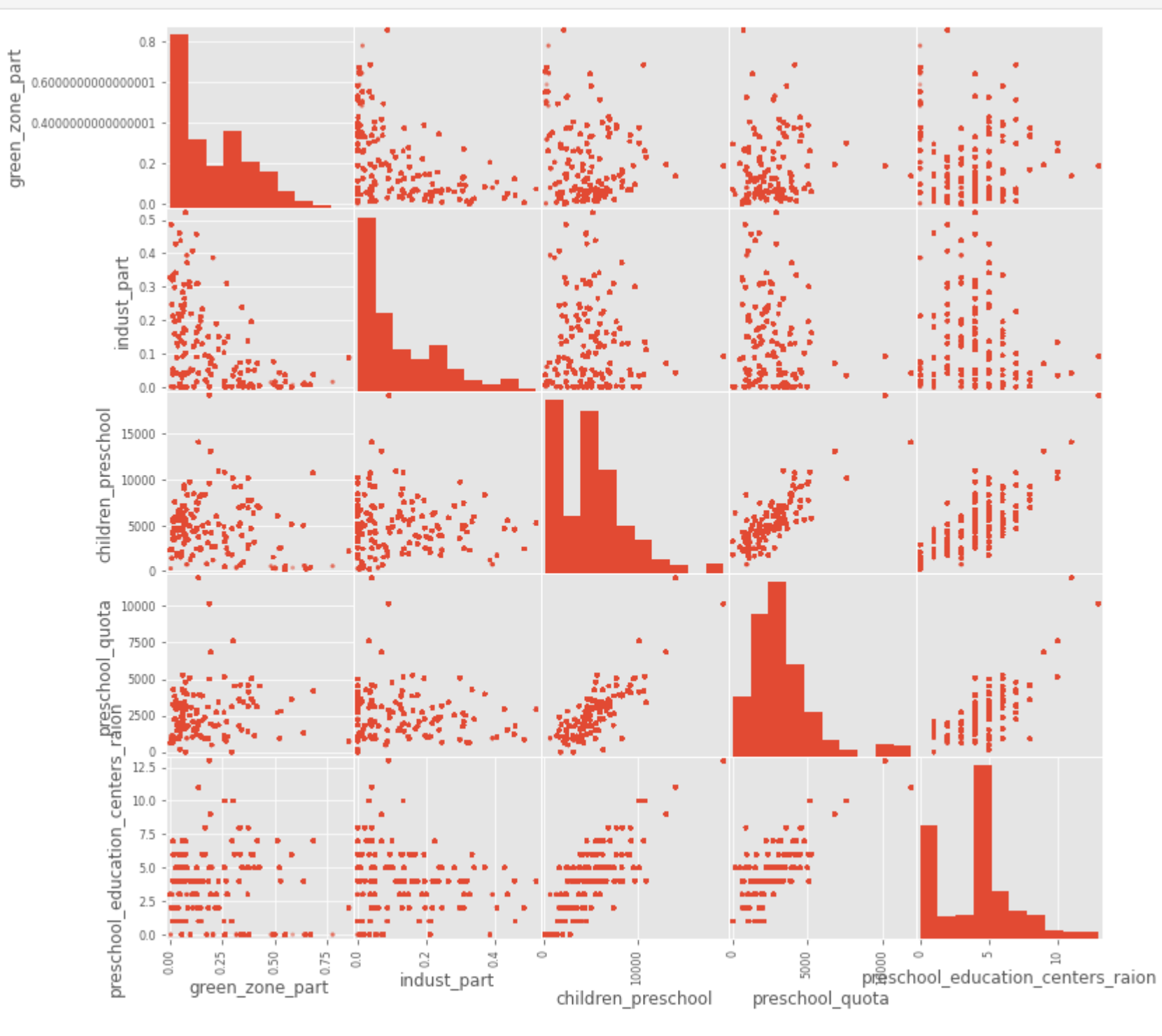
In [45]:
```
from pandas.plotting import scatter_matrix
```

We can't plot the full data-set since it has too many features (292) - the chart would be too large and/or difficult to read. So we need to select a sub-set of the data - in this case columns 15 to 20. These columns were selected after some experimentation simply because they provide a good demonstration of this particular chart.

In [46]:
```
## Step 1
df_fewer_columns = df.iloc[:, 15:20:1]
    # 'iloc' gets the data from integer locations (index down the table).
    # 'loc' uses the labels for rows and can return quite different data:
    # https://stackoverflow.com/questions/31593201/how-are-iloc-and-loc-different
```

Now we create the chart

In [47]:
```
scatter_matrix(df_fewer_columns, figsize=(12, 12));
```



Student task : Experiment by modifying the code block containing the comment '# Step 1' above. The numbers following the 'iloc' command control the range of data shown in the figure. Experiment by displaying different parts of the data-set.

2.2 Identifying missing data

We will use the 'missingno' library as an easy way to visualise missing data in a dataframe. You can find out more about the missingno library here: https://github.com/ResidentMario/missingno

First, as normal, we import the library:

In [48]:
```
import missingno as msno
```

There are many columns in our dataframe. If we plot them all at the same time then the graphics will become very crowded and difficult to read. Therefore it is useful to select smaller slices of the data and view them individually.

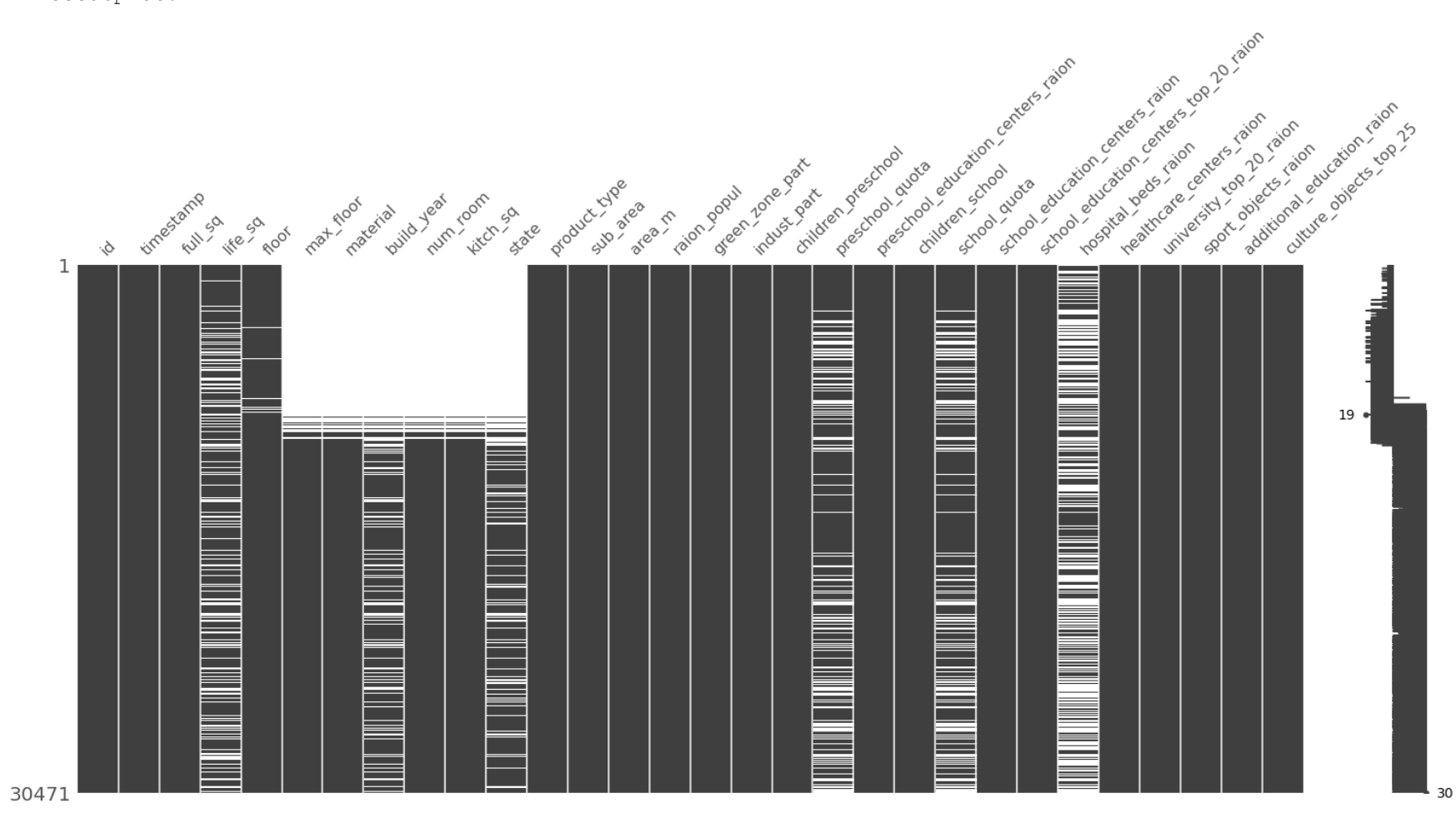
The Pandas 'iloc' function used in the next code block enables a 'slice' of data to be created. Pandas 'iloc' is documented here: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.html

In [49]:
```
# Step 2
df_some_columns = df.iloc[:, 0:30:1]
```

Having selected the columns we want to visualise, we can now plot a graphic that visualises missing data. In this graphic, missing data is shown as white whilst data which is present is shown in black. The graphic helps visualise patterns of missing data.

In [50]:
```
msno.matrix(df_some_columns)
```

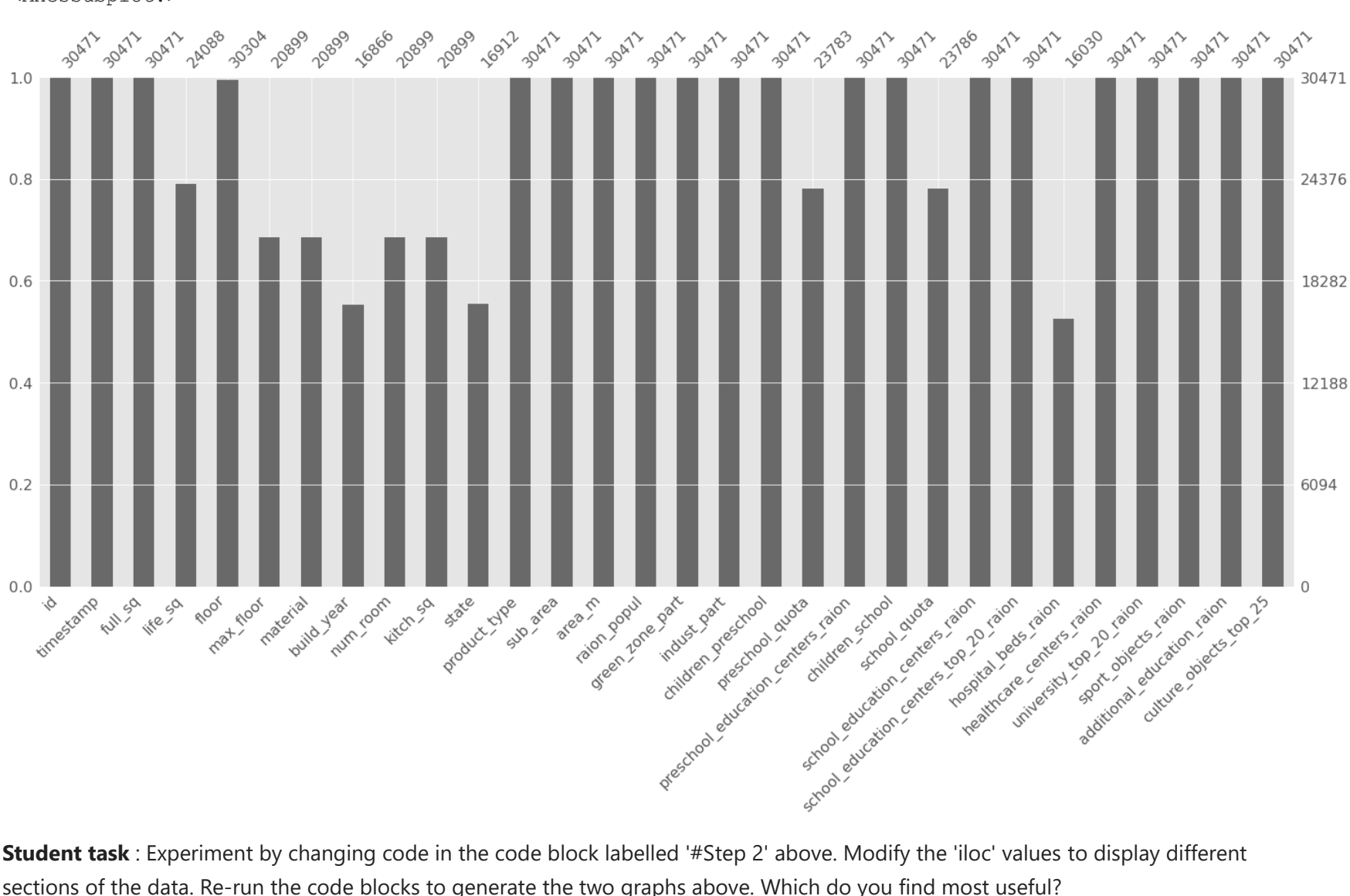
Out[50]:
```
<AxesSubplot:>
```



Next we can draw a bar-chart that counts the number of missing items in each feature:

In [51]:
```
msno.bar(df_some_columns)
```

Out[51]:
```
<AxesSubplot:>
```



Student task : Experiment by changing code in the code block labelled '#Step 2' above. Modify the 'iloc' values to display different sections of the data. Re-run the code blocks to generate the two graphs above. Which do you find most useful?

2.3 Removing features that have insufficient data

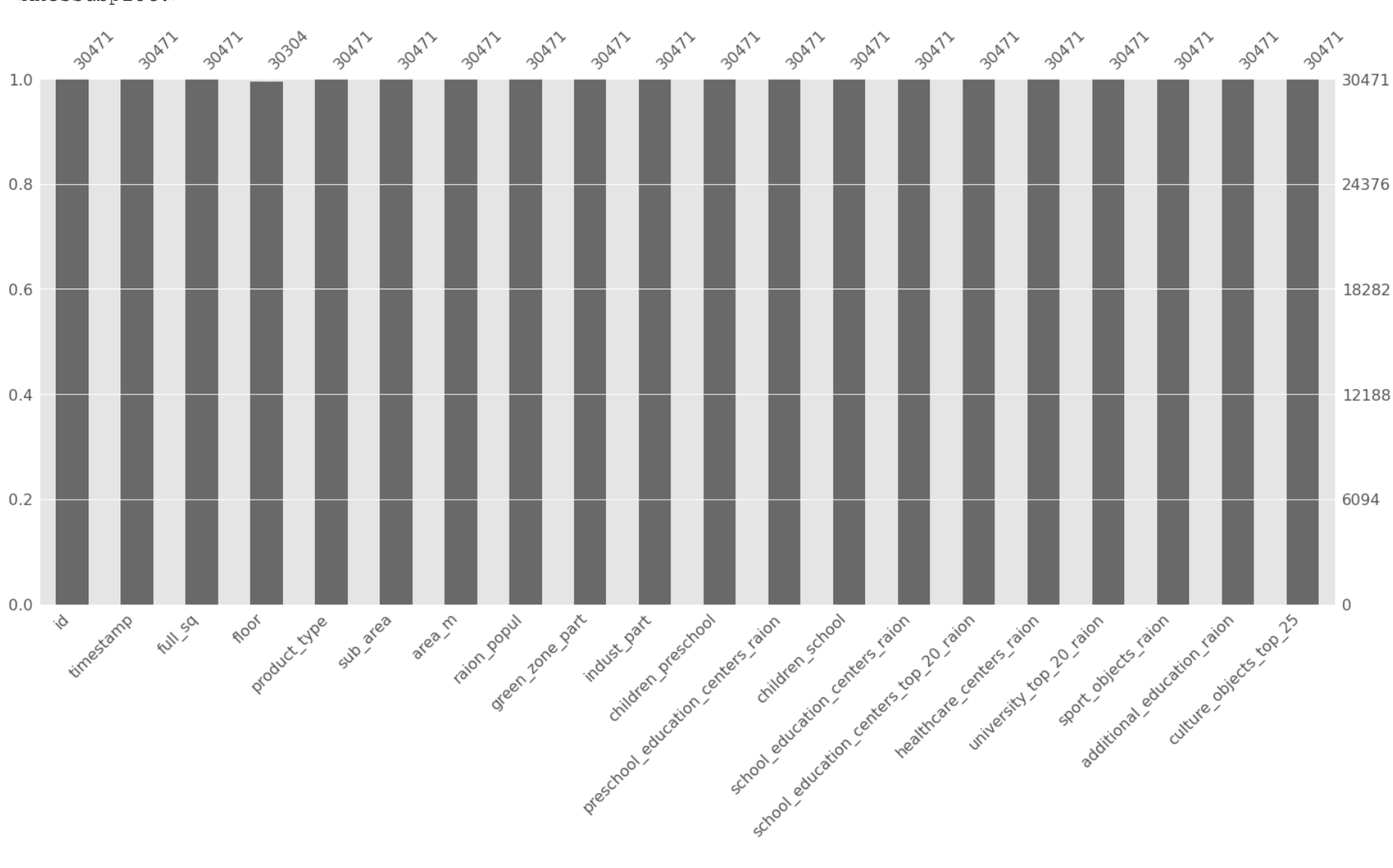
You may decide to drop (delete) features with less than a given fraction of missing data. For example, here we drop all features with less than 80% of the data available:

In [52]:
```
limitPer = len(df_some_columns) * .80    # 80% of the number of columns
df_some_columns.dropna(axis='columns', thresh = limitPer, inplace = True)
```

It is instructive to plot the bar-chart above again after removing features with insufficient data:

In [53]:
```
msno.bar(df_some_columns)
```

Out[53]:
```
<AxesSubplot:>
```

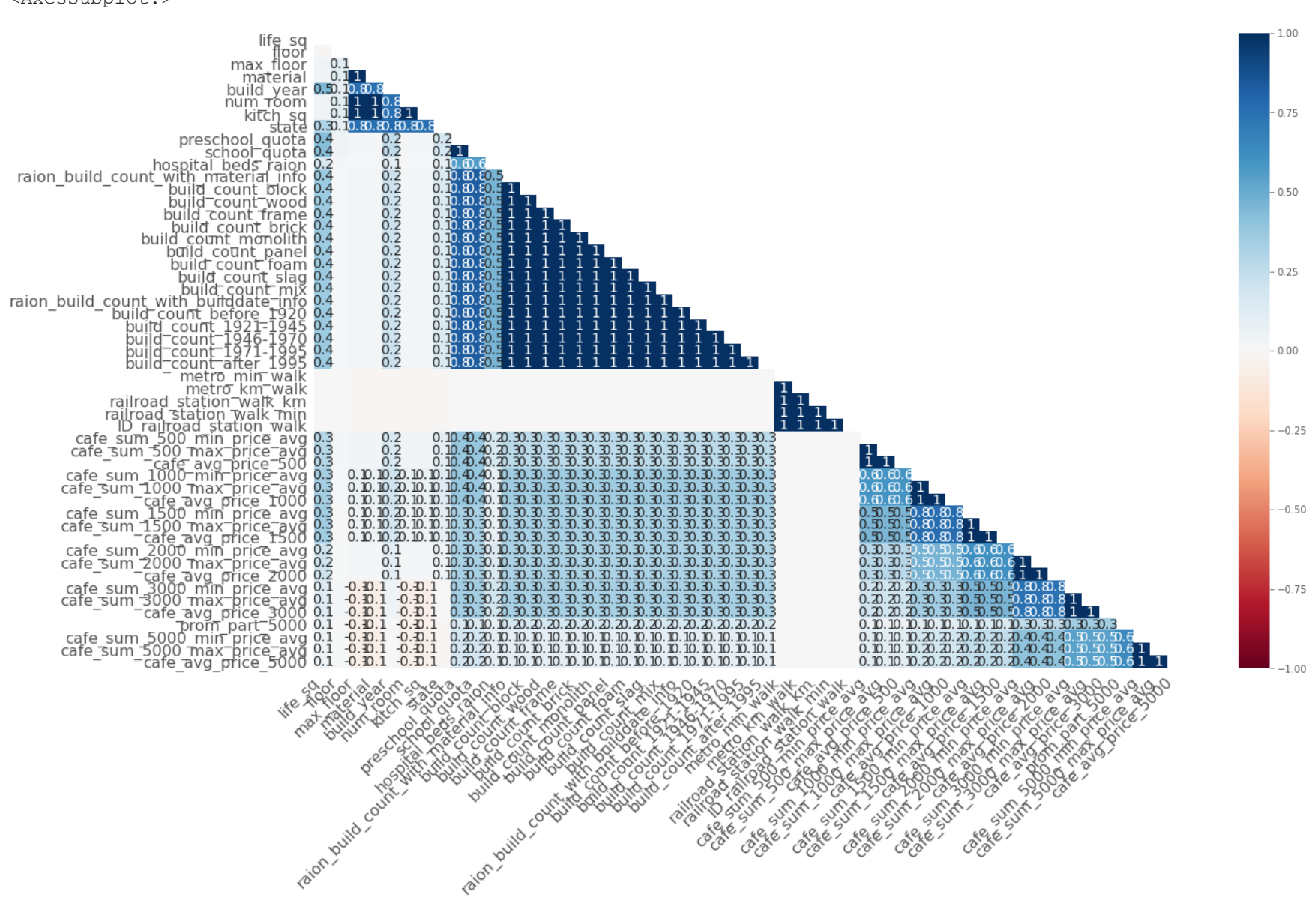


2.4 Is there a correlation in missing data?

We can look for correlations in missing data. Sometimes we find that having data available in one feature (column) is linked with (correlated to) data being present in another feature. Thus, in the following graphic, dark blue squares indicate a high correlation: Data being available in the first column is well correlated with daa being available in the second column.

In [54]:
```
msno.heatmap(df)
```

Out[54]:
```
<AxesSubplot:>
```



3 Conclusion

In this session we have demonstrated some common tools for visualising and pre-processing data ahead of building Machine Learning models.

You should experiment with the above code blocks to gain familiarity with them. For example, you might consider applying the same functions to different data sets and/or modifying the functions above.

In [ ]: