data_cleaning

April 13, 2026

1 Tutorial on Data Cleaning

This tutorial demonstrates various data cleaning stages on a sample astronomical dataset.

1) Setting up the environment
2) Downloading and importing the dataset
3) Inspect and visualize the dataset content
4) Run some consistency checks on the dataset
5) Detect and handle missing data values
6) Detect outliers

The original dataset is the `Stellar Classification Dataset`, available here:

- SDSS DR19: https://www.kaggle.com/datasets/randelmaglaya/sdss-dr19-2025/data

For this tutorial, however, we will use a modified "dirtier" version of the dataset, in which we artificially added corrupted, missing data, and outliers/anomalies.

2 Configuring the environment

2.1 Module installation

We'll begin by installing the necessary Python modules for this tutorial.

[79]:
```
import os

# - Install modules from requirements.txt if present
if os.path.isfile("requirements.txt"):
  print("Installing modules from local requirements.txt file ...")
  %pip install -q -r requirements.txt
else:
  print("Installing modules ...")

  %pip install -q gdown
  %pip install -q matplotlib seaborn[stats] missingno
  %pip install -q pandas scikit-learn umap-learn[plot]

  # - Create requirements file
  %pip freeze > requirements.txt
```

```
Installing modules …
Note: you may need to restart the kernel to use updated packages.
Note: you may need to restart the kernel to use updated packages.
Note: you may need to restart the kernel to use updated packages.
Note: you may need to restart the kernel to use updated packages.
```

2.2 Import modules

Next, we import the essential modules needed throughout the tutorial.

[3]:
```
############################
##   STANDARD MODULES
############################
import os
from pathlib import Path
import shutil
import gdown
import random
import numpy as np
from numpy import percentile
import matplotlib.pyplot as plt
import seaborn as sns
sns.reset_defaults()

############################
##   ML MODULES
############################
import pandas as pd
from pandas import read_csv
import missingno as msno
from sklearn.feature_selection import VarianceThreshold
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import SimpleImputer, IterativeImputer
from sklearn.neighbors import LocalOutlierFactor
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import RobustScaler
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
import umap, umap.plot
from umap import UMAP
```

```
/home/riggi/Software/venvs/usc-c-ai-school-2026/lib/python3.10/site-
packages/numba/np/ufunc/dufunc.py:346: NumbaWarning: Compilation requested for
previously compiled argument types ((uint32,)). This has no effect and perhaps
indicates a bug in the calling code (compiling a ufunc more than once for the
same signature
  warnings.warn(msg, errors.NumbaWarning)
/home/riggi/Software/venvs/usc-c-ai-school-2026/lib/python3.10/site-
```

```
packages/numba/np/ufunc/dufunc.py:346: NumbaWarning: Compilation requested for
previously compiled argument types ((uint32,)). This has no effect and perhaps
indicates a bug in the calling code (compiling a ufunc more than once for the
same signature
  warnings.warn(msg, errors.NumbaWarning)
/home/riggi/Software/venvs/usc-c-ai-school-2026/lib/python3.10/site-
packages/numba/np/ufunc/dufunc.py:346: NumbaWarning: Compilation requested for
previously compiled argument types ((uint32,)). This has no effect and perhaps
indicates a bug in the calling code (compiling a ufunc more than once for the
same signature
  warnings.warn(msg, errors.NumbaWarning)
```

2.3 Set random seeds

Let's define a function to set random numpy seeds to make random generation reproducible.

[4]:
```python
def set_seed(seed=42):
  """ Set random seed """
  random.seed(seed)
  np.random.seed(seed)

# - Set the seed
set_seed(1)
```

2.4 Project folders

We create a working directory `rundir` to run the tutorial in.

[5]:
```python
topdir= os.getcwd()
rundir= os.path.join(topdir, "run-data_cleaning")
path = Path(rundir)
path.mkdir(parents=True, exist_ok=True)
```

3 Dataset

For this tutorial, we will use the **Stellar Classification Dataset - SDSS DR19**.

This dataset contains **100,000** observations of sources taken by the SDSS (Sloan Digital Sky Survey) in various optical bands. SDSS photometric data are observed through five filters, u, g, r, i, and z. Every observation is described by **17 feature columns** and **1 class column** which identifies it to be either a **star**, **galaxy** or **quasar**.

1) `obj_ID`: Object Identifier, the unique value that identifies the object in the image catalog used by the CAS
2) `alpha`: Right Ascension angle (at J2000 epoch)
3) `delta`: Declination angle (at J2000 epoch)
4) `u`: Ultraviolet filter in the photometric system
5) `g`: Green filter in the photometric system
6) `r`: Red filter in the photometric system

7) `i`: Near Infrared filter in the photometric system
8) `z`: Infrared filter in the photometric system
9) `run_ID`: Run Number used to identify the specific scan
10) `rerun_ID`: Rerun Number to specify how the image was processed
11) `cam_col`: Camera column to identify the scanline within the run
12) `field_ID`: Field number to identify each field
13) `spec_obj_ID`: Unique ID used for optical spectroscopic objects (this means that 2 different observations with the same spec_obj_ID must share the output class)
14) `class`: object class (galaxy, star or quasar object)
15) `redshift`: redshift value based on the increase in wavelength
16) `plate`: plate ID, identifies each plate in SDSS
17) `MJD`: Modified Julian Date, used to indicate when a given piece of SDSS data was taken
18) `fiber_ID`: fiber ID that identifies the fiber that pointed the light at the focal plane in each observation

3.1 Dataset Download

Next, we download the dataset from Google Drive and unzip it in the main folder.

[7]:
```
# - Download dataset
def download_data(fname, url, destdir, force=False):
  """ Download dataset """

  # - Download dataset (if not previously downloaded)
  fname_full= os.path.join(destdir, fname)
  if force or not os.path.isfile(fname_full):
    print(f"Downloading data from url {url} ...")
    gdown.download(url, fname, quiet=False)
    print("DONE!")

  # - Moving data to destdir
  if os.path.isfile(fname) and os.path.isdir(destdir):
    print(f"Moving data {fname} to dir %s ..." % (destdir))
    shutil.move(fname, fname_full)

# - Set dataset URL & paths
dataset_name= 'star_classification.csv'
dataset_name_fullpath= os.path.join(rundir, dataset_name)
#dataset_url= 'https://drive.google.com/uc?
 ↪export=download&id=1RH1v7q2bukXV4BNQhp30EsOdncxgIrwk' # DR19
dataset_url= 'https://drive.google.com/uc?
 ↪export=download&id=1rFoh30_zNdDJJ_tdnEdVpepbATOpvx5D' # DR19 dirty

# - Download dataset
download_data(dataset_name, dataset_url, rundir, force=True)
```

```
Downloading data from url
https://drive.google.com/uc?export=download&id=1rFoh30_zNdDJJ_tdnEdVpepbATOpvx5D
```

```
…
Downloading…
From:
https://drive.google.com/uc?export=download&id=1rFoh30_zNdDJJ_tdnEdVpepbATOpvx5D
To: /home/riggi/Analysis/MLProjects/INAF-USC-C-
AI/school2026/notebooks/star_classification.csv
100%|

       | 15.7M/15.7M [00:03<00:00, 4.54MB/s]

DONE!
Moving data star_classification.csv to dir /home/riggi/Analysis/MLProjects/INAF-
USC-C-AI/school2026/notebooks/run-data_cleaning …
```

3.2 Dataset load

Let's load the downloaded dataset as a Pandas data frame.

[8]:
```
print(f"Loading data from file {dataset_name_fullpath} ...")
try:
  df= pd.read_csv(dataset_name_fullpath, sep=',')
  print(df)
  #print(df.head())
  #print(df.tail())
except UnicodeDecodeError:
  print(f"Unicode decoding error when loading data file {dataset_name_fullpath}!
 ↪")
```

```
Loading data from file /home/riggi/Analysis/MLProjects/INAF-USC-C-
AI/school2026/notebooks/run-data_cleaning/star_classification.csv …
                    obj_ID       alpha      delta         u         g  \
0       1237652947459637379   17.941809  -9.816818      -inf  17.43624
1       1237671123217940946  122.127448   7.336519  18.55875  16.56873
2       1237672026783416538  336.686095  39.020594  15.23208  15.55459
3       1237663542064119850  311.369182  -1.009367  18.65634  17.48205
4       1237648702973542435  199.516504  -1.123175  18.00832  16.83256
…                       …         …         …       …         …
101995  1237655126082846840  184.088675   5.459447  19.31630  17.96200
101996  1237664853114159283  232.163943  27.435908  19.40644  18.14540
101997  1237658492286599185  196.790855   9.552958  18.82735  18.42521
101998  1237657595152171182  127.857481  40.956219       NaN  14.48068
101999  1237679502705492124  353.260324  18.903515  18.83230  17.84337

                r         i         z  run_ID  rerun_ID  cam_col  field_ID  \
0       16.86528  16.46371  16.28979    1740       301        3       140
1       15.59954  15.17670  14.83782    5972       301        2        42
2       16.01454  16.35339  16.65512    6182       301        5       221
```

```
3       17.07172  16.90119  16.83375    4207       301        1        47
4       16.44511  16.31481  16.25952     752       301        1       374
…          …         …        …          …         …        …         …
101995  17.41003  17.21149  17.06680    2247       301        5       156
101996  17.66140  17.26682  17.06125    4512       301        3       219
101997  18.55614  18.37914  18.43999    3031       301        3       509
101998  13.46446  12.84834  12.43901    2822       301        4       156
101999  17.48526  17.32426  17.24340    7923       301        2       139

                spec_obj_ID   class  redshift  plate    MJD  fiber_ID
0         742133002246580224  GALAXY  0.151069    659  52199       600
1        1977214428508088320  GALAXY  0.100066   1756  53080       488
2         2949888066266884096    STAR  0.000030   2620  54397       110
3         1105708516682786816    STAR -0.000120    982  52466       272
4         3723431889416069120    STAR  0.000271   3307  54970       294
…                          …      …         …     …     …         …
101995    3661477981311836160    STAR  0.000209   3252  54893       187
101996    2079541321168611328  GALAXY  0.071405   1847  54176        15
101997    2018748218841524224     QSO  0.574060   1793  53883        35
101998    1007757150862206976  GALAXY  0.024419    895  52581       279
101999    8559234419877238784    STAR -0.000628   7602  56954       521

[102000 rows x 18 columns]
```

3.3 Dataset inspect

Let's inspect the loaded data frame.

[9]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 102000 entries, 0 to 101999
Data columns (total 18 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   obj_ID       102000 non-null  int64
 1   alpha        102000 non-null  float64
 2   delta        102000 non-null  float64
 3   u            100067 non-null  float64
 4   g            100473 non-null  float64
 5   r            101179 non-null  float64
 6   i            101183 non-null  float64
 7   z            100478 non-null  float64
 8   run_ID       102000 non-null  int64
 9   rerun_ID     102000 non-null  int64
 10  cam_col      102000 non-null  int64
 11  field_ID     102000 non-null  int64
 12  spec_obj_ID  102000 non-null  uint64
```

```
 13  class         102000 non-null  object
 14  redshift      102000 non-null  float64
 15  plate         102000 non-null  int64
 16  MJD           102000 non-null  int64
 17  fiber_ID      102000 non-null  int64
dtypes: float64(8), int64(8), object(1), uint64(1)
memory usage: 14.0+ MB
```

We remove the columns we are not interested in for this tutorial, leaving only band fluxes, redshift and class information.

[10]:
```
columns_to_be_removed= ["alpha", "delta", "run_ID", "rerun_ID", "cam_col",␣
 ↪"field_ID", "spec_obj_ID", "plate", "MJD", "fiber_ID"]
df.drop(columns_to_be_removed, inplace=True, axis=1)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 102000 entries, 0 to 101999
Data columns (total 8 columns):
 #   Column    Non-Null Count   Dtype
---  ------    --------------   -----
 0   obj_ID    102000 non-null  int64
 1   u         100067 non-null  float64
 2   g         100473 non-null  float64
 3   r         101179 non-null  float64
 4   i         101183 non-null  float64
 5   z         100478 non-null  float64
 6   class     102000 non-null  object
 7   redshift  102000 non-null  float64
dtypes: float64(6), int64(1), object(1)
memory usage: 6.2+ MB
```

We compute summary statistics for selected column left.

[11]:
```
df.describe(include = 'all')
```

```
/home/riggi/Software/venvs/usc-c-ai-school-2026/lib/python3.10/site-
packages/numpy/_core/_methods.py:52: RuntimeWarning: invalid value encountered
in reduce
  return umr_sum(a, axis, dtype, out, keepdims, initial, where)
/home/riggi/Software/venvs/usc-c-ai-school-2026/lib/python3.10/site-
packages/numpy/_core/_methods.py:52: RuntimeWarning: invalid value encountered
in reduce
  return umr_sum(a, axis, dtype, out, keepdims, initial, where)
/home/riggi/Software/venvs/usc-c-ai-school-2026/lib/python3.10/site-
packages/pandas/core/nanops.py:1016: RuntimeWarning: invalid value encountered
in subtract
  sqr = _ensure_numeric((avg - values) ** 2)
/home/riggi/Software/venvs/usc-c-ai-school-2026/lib/python3.10/site-
```

```
packages/pandas/core/nanops.py:1016: RuntimeWarning: invalid value encountered
in subtract
  sqr = _ensure_numeric((avg - values) ** 2)
/home/riggi/Software/venvs/usc-c-ai-school-2026/lib/python3.10/site-
packages/numpy/_core/_methods.py:52: RuntimeWarning: invalid value encountered
in reduce
  return umr_sum(a, axis, dtype, out, keepdims, initial, where)
```

[11]:

	obj_ID	u	g	r	i	\
count	1.020000e+05	1.000670e+05	1.004730e+05	1.011790e+05	1.011830e+05	
unique	NaN	NaN	NaN	NaN	NaN	
top	NaN	NaN	NaN	NaN	NaN	
freq	NaN	NaN	NaN	NaN	NaN	
mean	1.237663e+18	NaN	NaN	inf	inf	
std	7.277021e+12	NaN	NaN	NaN	NaN	
min	1.237646e+18	-inf	-inf	9.823509e+00	9.337089e+00	
25%	1.237658e+18	1.821624e+01	1.685172e+01	1.619863e+01	1.586763e+01	
50%	1.237662e+18	1.887664e+01	1.751741e+01	1.689665e+01	1.660286e+01	
75%	1.237667e+18	1.927474e+01	1.805712e+01	1.759093e+01	1.734983e+01	
max	1.237681e+18	inf	inf	inf	inf	

	z	class	redshift
count	1.004780e+05	102000	102000.000000
unique	NaN	12	NaN
top	NaN	GALAXY	NaN
freq	NaN	52290	NaN
mean	NaN	NaN	0.170501
std	NaN	NaN	0.437256
min	-inf	NaN	-0.004136
25%	1.562012e+01	NaN	0.000003
50%	1.642687e+01	NaN	0.046372
75%	1.723462e+01	NaN	0.095634
max	inf	NaN	7.011245

The above output tells that the loaded dataset has NaNs and corrupted values we will need to handle. We will do it later in this tutorial. Below, we re-compute the summary statistics on completely observed data.

[12]:
```
# - Compute data summary stats on observed numerical data
df_obs= df.replace([np.inf, -np.inf], np.nan, inplace=False).dropna()
df_num = df_obs.select_dtypes(include=['number'])
stats= df_num.describe()

# - Compute median on numerical columns
median = df_num.median()
median_df= pd.DataFrame({'median':median}).T

# - Compute skewness & kurtosis on numerical columns
```

```
skewness = df_num.skew()
kurtosis = df_num.kurtosis() # excess kurtosis (e.g. =0 for normal distr)
skewness_df = pd.DataFrame({'skew':skewness}).T
kurtosis_df = pd.DataFrame({'kurt':kurtosis}).T

# - Add to stats
stats= pd.concat([stats, median_df, kurtosis_df, skewness_df])
print(stats)
```

```
              obj_ID             u             g             r             i  \
count   9.401500e+04  94015.000000  94015.000000  94015.000000  94015.000000
mean    1.237663e+18     18.640590     17.408388     16.874920     16.617437
std     7.270947e+12      0.828296      0.980850      1.147699      1.224351
min     1.237646e+18     10.611810      9.668339      9.823509      9.337089
25%     1.237658e+18     18.217755     16.854630     16.194750     15.864575
50%     1.237662e+18     18.874770     17.517100     16.890930     16.598180
75%     1.237667e+18     19.272685     18.054830     17.582845     17.340730
max     1.237681e+18     19.599960     19.974990     31.990100     32.141470
median  1.237662e+18     18.874770     17.517100     16.890930     16.598180
kurt   -1.027052e-01      2.565488      1.704239      3.355190      3.116639
skew    2.841035e-01     -1.391816     -0.724179     -0.307974     -0.097547

                   z      redshift
count   94015.000000  94015.000000
mean       16.465977      0.170003
std         1.276510      0.436879
min         8.947795     -0.004136
25%        15.622765      0.000003
50%        16.427940      0.046307
75%        17.229170      0.095608
max        30.017040      7.011245
median     16.427940      0.046307
kurt        1.171445     22.182731
skew        0.147275      4.102528
```

Flux and redshift variables are moderately negatively skewed, peaked and with heavy tails (kurtosis>0).

4 Data Integrity Check

Let's run first some integrity check on the loaded data frame.

4.1 Checking for nans/inf

Let's see if the loaded data frame contains any missing values (empty column value) or +-inf numerical values.

Below, we count the number of NaN entries.

[13]:
```python
def dump_nan_counts(df):
  total_rows = len(df)
  total_nan_count = df.isnull().sum().sum()
  print("--> Total NaN count:", total_nan_count)

  column_nan_count = df.isnull().sum()
  print("--> NaN count per column")
  for col in df.columns:
    count = column_nan_count[col]
    percent = (count / total_rows) * 100
    print(f"{col}: {count} ({percent:.2f}%)")

dump_nan_counts(df)
```

```
--> Total NaN count: 6620
--> NaN count per column
obj_ID: 0 (0.00%)
u: 1933 (1.90%)
g: 1527 (1.50%)
r: 821 (0.80%)
i: 817 (0.80%)
z: 1522 (1.49%)
class: 0 (0.00%)
redshift: 0 (0.00%)
```

Below, we count the number of inf entries.

[14]:
```python
def dump_inf_counts(df):
  total_rows = len(df)
  is_infinite = df.isin([np.inf, -np.inf])
  counts_dict= is_infinite.stack().value_counts().to_dict()
  total_inf_count= 0 if True not in counts_dict else counts_dict[True]
  print("--> Total Inf count:", total_inf_count)

  print("--> Inf count per column:")
  for col in is_infinite.columns:
    counts_dict= is_infinite[col].value_counts().to_dict()
    counts_inf= 0 if True not in counts_dict else counts_dict[True]
    percent = (counts_inf / total_rows) * 100
    print(f"{col}: {counts_inf} ({percent:.2f}%)")

dump_inf_counts(df)
```

```
--> Total Inf count: 1642
--> Inf count per column:
obj_ID: 0 (0.00%)
u: 407 (0.40%)
g: 302 (0.30%)
```

```
r: 315 (0.31%)
i: 314 (0.31%)
z: 304 (0.30%)
class: 0 (0.00%)
redshift: 0 (0.00%)
```

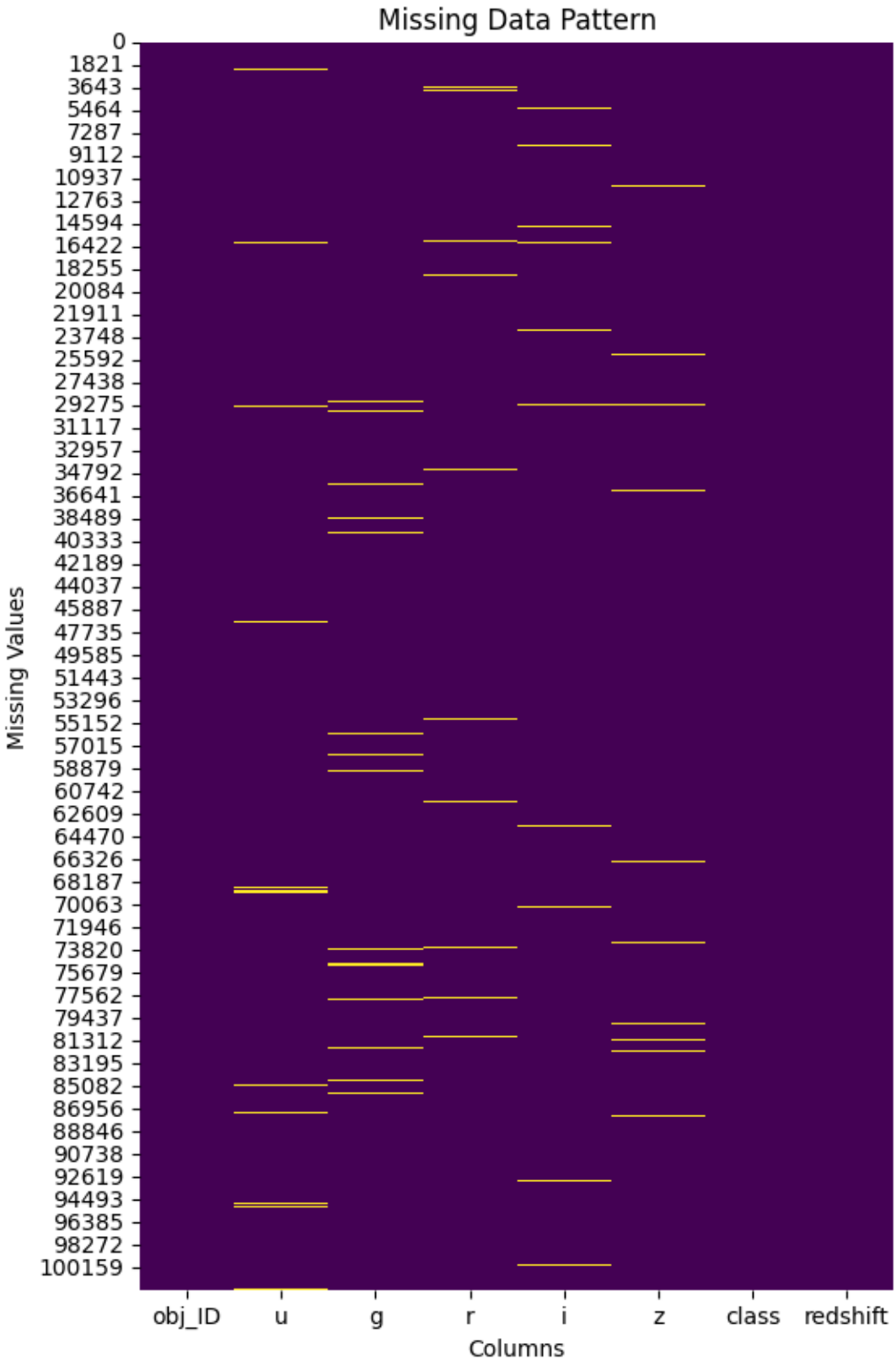
Let's convert the inf values into NaNs from the rest of the tutorial.

[15]:
```
df.replace([np.inf, -np.inf], np.nan, inplace=True)
dump_nan_counts(df)
```

```
--> Total NaN count: 8262
--> NaN count per column
obj_ID: 0 (0.00%)
u: 2340 (2.29%)
g: 1829 (1.79%)
r: 1136 (1.11%)
i: 1131 (1.11%)
z: 1826 (1.79%)
class: 0 (0.00%)
redshift: 0 (0.00%)
```

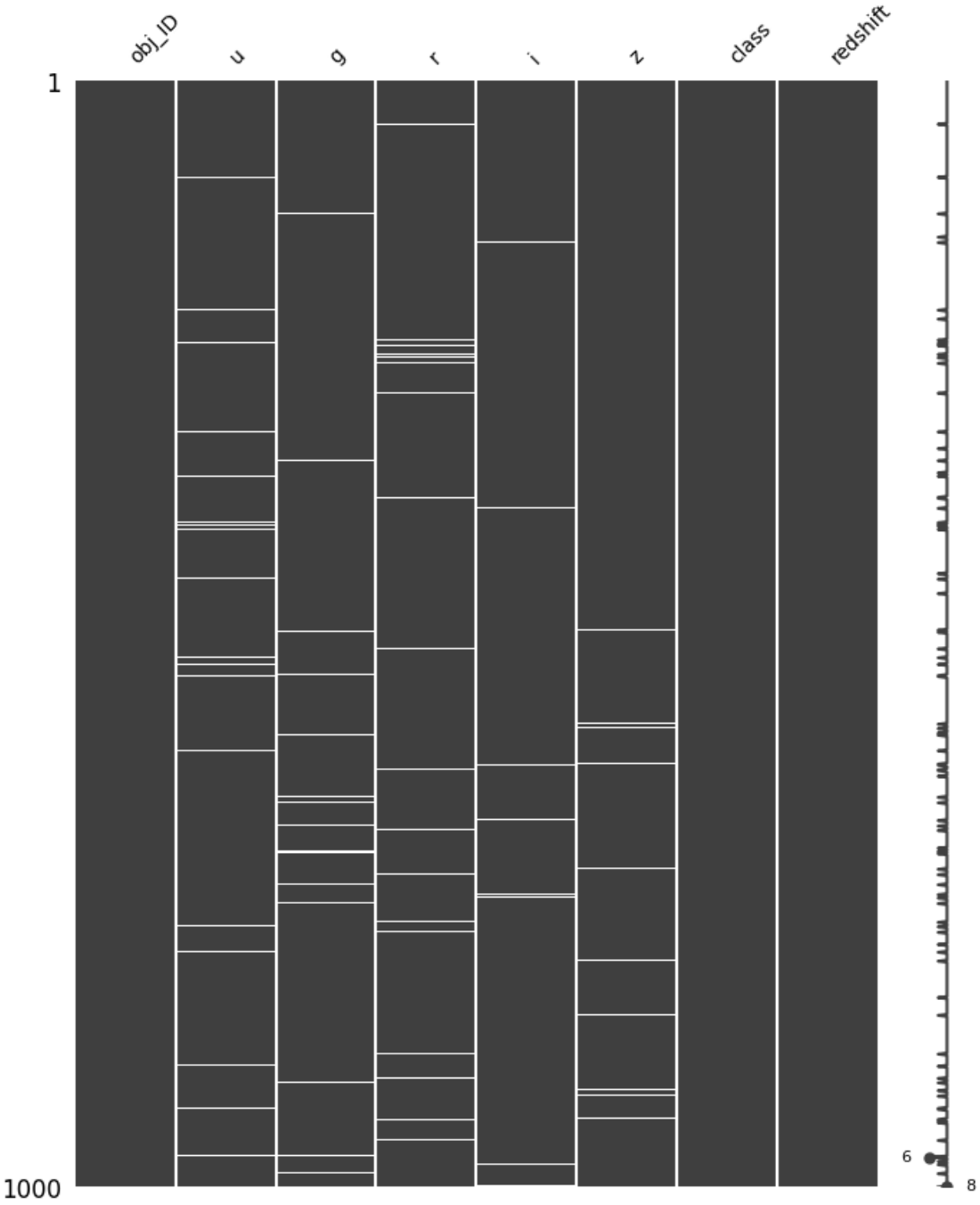
Below, we produce a heatmap of missing values. It can be useful to display and inspect possible patterns in the missingness.

[100]:
```
plt.figure(figsize=(6,10))
sns.heatmap(df.isna(), cbar=False, cmap='viridis')
plt.title("Missing Data Pattern")
plt.xlabel('Columns', fontsize = 10)
plt.ylabel('Missing Values', fontsize = 10)
plt.show()
```


Missing Data Pattern
Missing Values
0
1821
3643
5464
7287
9112
10937
12763
14594
16422
18255
20084
21911
23748
25592
27438
29275
31117
32957
34792
36641
38489
40333
42189
44037
45887
47735
49585
51443
53296
55152
57015
58879
60742
62609
64470
66326
68187
70063
71946
73820
75679
77562
79437
81312
83195
85082
86956
88846
90738
92619
94493
96385
98272
100159
obj_ID
u
g
r
i
z
class
redshift
Columns

A similar plot can be done with the `missingno` module. We select here a sub-sample of the data for the plot.

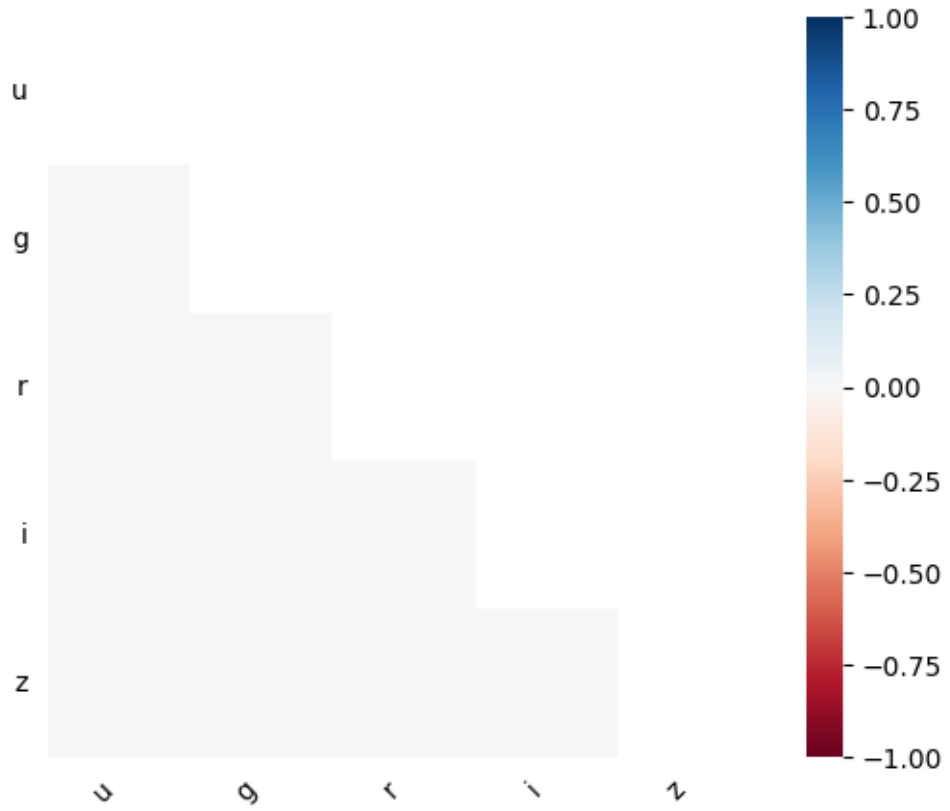
[109]:
```
ax= msno.matrix(df.sample(1000), figsize=(8,10), fontsize=10)
```



A correlation heatmap measures missing data correlation, e.g. how strongly the presence or absence of one variable affects the presence of another.

[112]:
```
msno.heatmap(df, figsize=(6,5), fontsize=10)
```

[112]: <Axes: >



4.2 Checking for duplicates

Let's find if the dataset has duplicated rows. We consider all columns for the search (e.g. `subset=None`).

[16]:
```
n_dupl= df.duplicated(keep='first', subset=None).sum()
print(f"Dataset has {n_dupl} duplicated rows")
```

```
Dataset has 2000 duplicated rows
```

We can remove duplicated rows from the current data frame, leaving only the first occurrence with the following method.

[17]:
```
df.drop_duplicates(keep='first',subset=None, inplace=True)
n_dupl= df.duplicated(keep='first', subset=None).sum()
```

```
print(f"[AFTER DUPLICATE REMOVAL] Dataset has {n_dupl} duplicated rows")
```

```
[AFTER DUPLICATE REMOVAL] Dataset has 0 duplicated rows
```

4.3 Checking for unique values

Let's inspect the value "diversity" within each column. - Some columns (e.g. `obj_ID`), suitable to be used as frame index, are expected to contain unique values, so the presence of non unique values is suspicious. - Low unique value counts in categorical columns (e.g. `class`) are usually fine - Low unique value counts in numerical quantitative columns (e.g. flux `u`, `g`, `r`) is likely as issue.

Let's print the fraction of unique values for each column.

[27]:
```
def dump_unique_counts(df, top_k=3):
  total_rows = len(df)

  print("--> Distinct values per column:")
  for col in df.columns:
    vc = df[col].value_counts(dropna=False)
    n_distinct = vc.size
    distinct_pct = n_distinct / total_rows * 100

    singleton_rows = vc[vc == 1].sum()
    singleton_rows_pct = singleton_rows / total_rows * 100

    duplicated_rows = vc[vc > 1].sum()
    duplicated_rows_pct = duplicated_rows / total_rows * 100

    top_counts = (vc / total_rows * 100).head(top_k)
    top_counts_str = ', '.join(
      f'{repr(key)}({value:.2f}%)' for key, value in top_counts.items()
    )

    print(
      f"{col}: {n_distinct} ({distinct_pct:.2f}%), "
      f"singleton/dupl rows={singleton_rows_pct:.2f}%/{duplicated_rows_pct:.
↪2f}% - "
      f"Top-{top_k}: {top_counts_str}"
    )

dump_unique_counts(df)
```

```
--> Distinct values per column:
obj_ID: 100000 (100.00%), singleton/dupl rows=100.00%/0.00% - Top-3:
1237663277926252765(0.00%), 1237668494707392644(0.00%),
1237666339188572196(0.00%)
u: 77950 (77.95%), singleton/dupl rows=61.67%/38.33% - Top-3: nan(2.28%),
19.31901(0.01%), 19.49681(0.01%)
```

```
g: 84082 (84.08%), singleton/dupl rows=71.73%/28.27% - Top-3: nan(1.80%),
18.02139(0.01%), 17.68875(0.01%)
r: 86485 (86.48%), singleton/dupl rows=75.42%/24.58% - Top-3: nan(1.11%),
9.823508969(0.10%), 17.6764(0.01%)
i: 87114 (87.11%), singleton/dupl rows=76.61%/23.39% - Top-3: nan(1.10%),
9.337089131(0.10%), 16.64155(0.01%)
z: 87497 (87.50%), singleton/dupl rows=77.77%/22.23% - Top-3: nan(1.79%),
16.5112(0.01%), 15.66088(0.01%)
class: 12 (0.01%), singleton/dupl rows=0.00%/100.00% - Top-3: 'GALAXY'(51.25%),
'STAR'(37.58%), 'QSO'(10.43%)
redshift: 93939 (93.94%), singleton/dupl rows=92.32%/7.68% - Top-3: 0.0(0.39%),
-5.5e-05(0.02%), -1.42e-05(0.01%)
```

4.4 Detecting misannotations

Let's inspect the dataset `class` column to count how many instances we have per each class, and find if we have potential issues with the class label annotation.

[28]:
```
n_instances_class= df['class'].value_counts(dropna=True)
print(n_instances_class)
```

```
class
GALAXY     51247
STAR       37583
QSO        10428
galaxy       130
 GALAXY      125
GALAXY       111
 STAR         99
star          98
STAR          84
QSO           36
 QSO          31
qso           28
Name: count, dtype: int64
```

As can be seen, there are a few misannotations present in the `class` column. These are fundamentally of two types:

- leading/trailing blank spaces
- lowercase vs uppercase

Let's fix that and check.

[29]:
```
df['class']= df['class'].str.strip().str.upper()

n_instances_class= df['class'].value_counts(dropna=True)
print(n_instances_class)
```

```
class
```

```
GALAXY    51613
STAR      37864
QSO       10523
Name: count, dtype: int64
```

5 Visual analysis

Let's visualize the distribution and correlation of some variables.

Below, we apply a log transform to the numerical columns. As we haven't yet handled the missing data, we will limit to plot the complete data, replacing missing values with a fixed value (-1).

[30]:
```
# - Log transform data
df_transf= df[['u','g','r','i','z','redshift','class']].copy()
df_mins= df_transf.dropna().min().to_dict()
print(df_mins)

for col in df_transf.columns:
  if col=='class':
    continue
  df_transf[col] = df_transf[col].apply(lambda x: np.log(x - df_mins[col] + 1)␣
 ↪if np.isfinite(x) else -1)

df_transf_mins= df_transf.dropna().min().to_dict()
print(df_transf_mins)
```

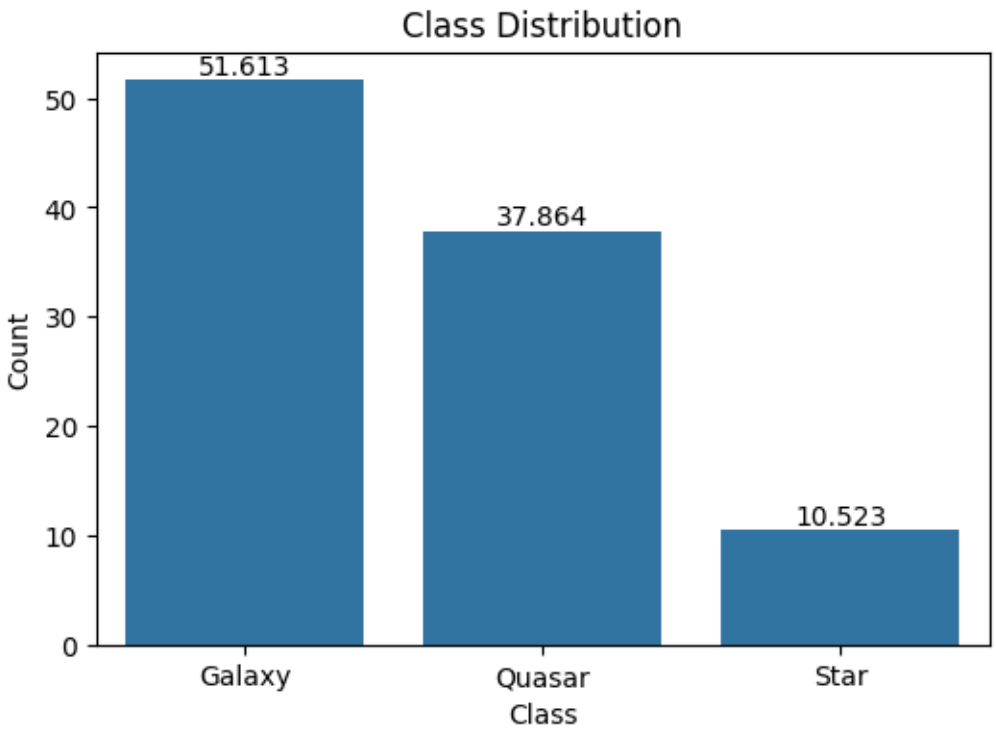
```
{'u': 10.61181, 'g': 9.668339, 'r': 9.823508969, 'i': 9.337089131, 'z':
8.947795, 'redshift': -0.004136078, 'class': 'GALAXY'}
{'u': -1.0, 'g': -1.0, 'r': -1.0, 'i': -1.0, 'z': -1.0, 'redshift': 0.0,
'class': 'GALAXY'}
```

5.1 Class distribution

Below, we plot the distribution of source classes (column `class`), after having fixed the annotations.

[33]:
```
def plot_class_counts(df, normalize=True):
  """ Plot the class count distribution """
  plt.figure(figsize=(6, 4))
  stat= "percent" if normalize else "count"
  ax = sns.countplot(x=df['class'], stat=stat)

  labels = ["Galaxy", "Quasar", "Star"]
  ax.xaxis.set_ticks([0,1,2])
  ax.set_xticklabels(labels)

  # Add labels on top of bars
  for container in ax.containers:
    ax.bar_label(container)
```

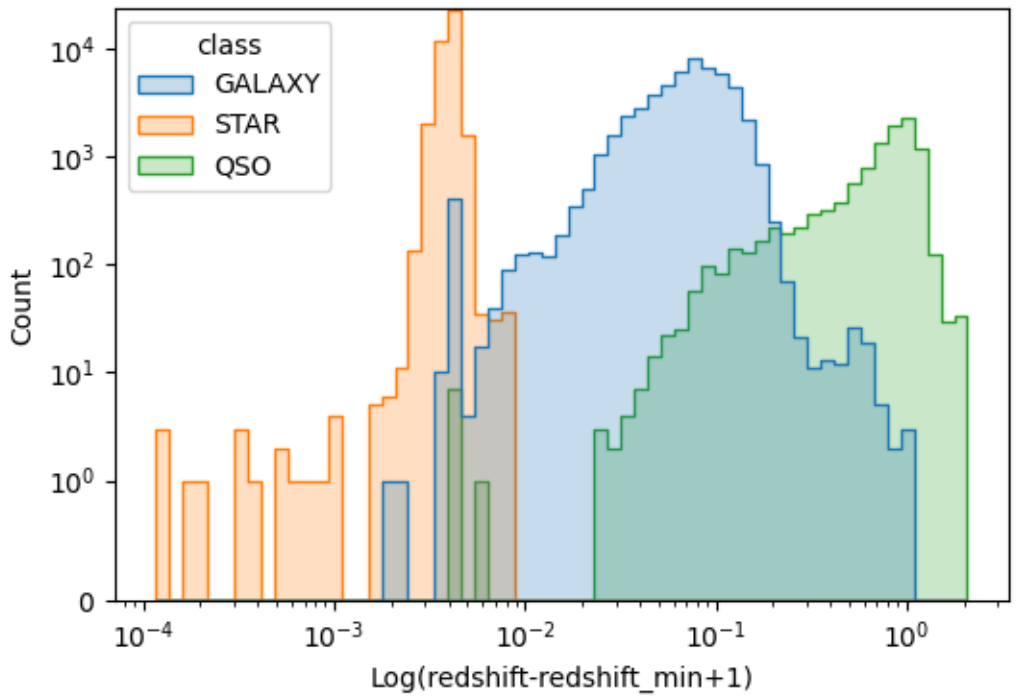
```
  plt.title("Class Distribution")
  plt.xlabel("Class")
  plt.ylabel("Count")
  plt.show()

plot_class_counts(df_transf, normalize=True)
```



5.2 Redshift distribution

Below, we plot the redshift distribution per each class.

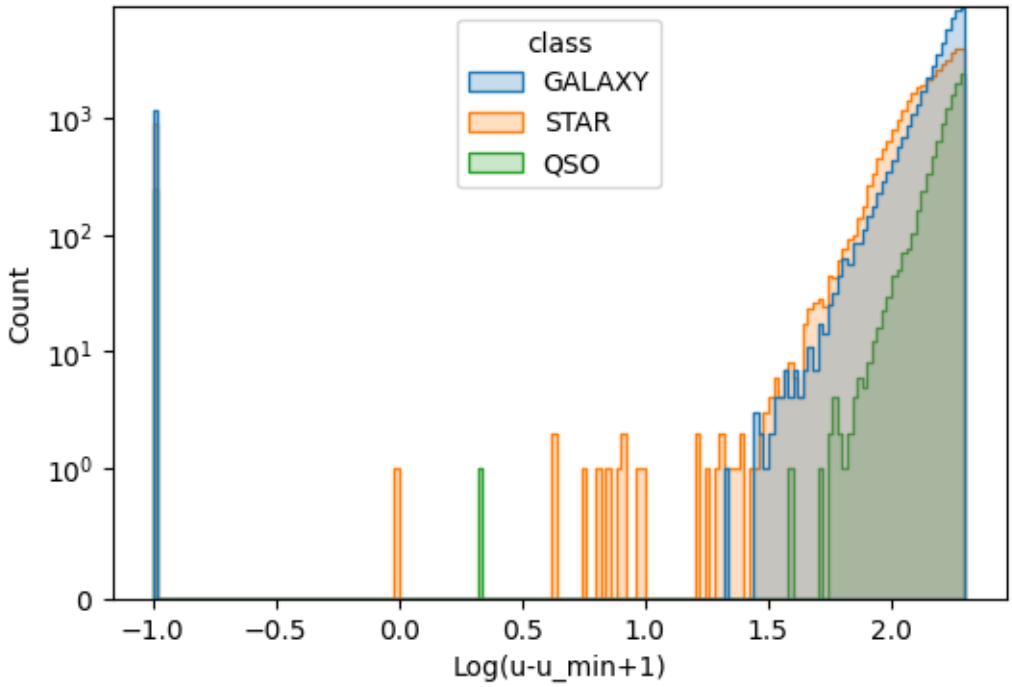
[34]:
```
# - Plot redshift
plt.figure(figsize=(6, 4))
ax= sns.histplot(data=df_transf, x='redshift', hue='class', element="step",
             stat='count', fill=True, binwidth=0.07, log_scale=(True,False))
plt.xlabel("Log(redshift-redshift_min+1)")
ax.set_yscale("symlog", linthresh=1) # 0..1 is linear, above is ~log
ax.set_ylim(0, None)
plt.show()
```

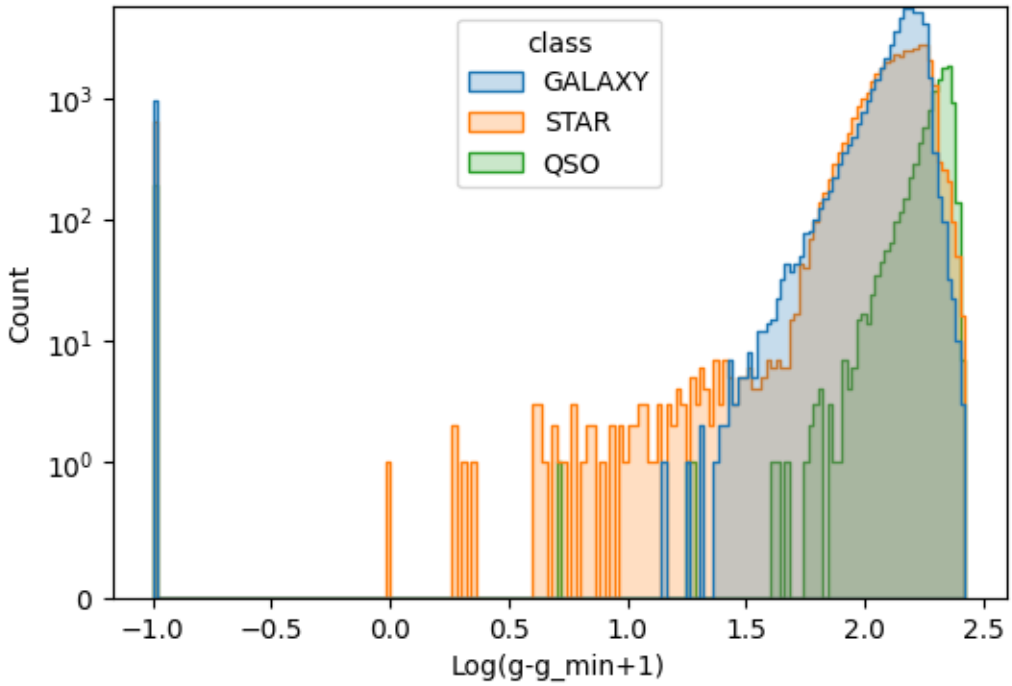


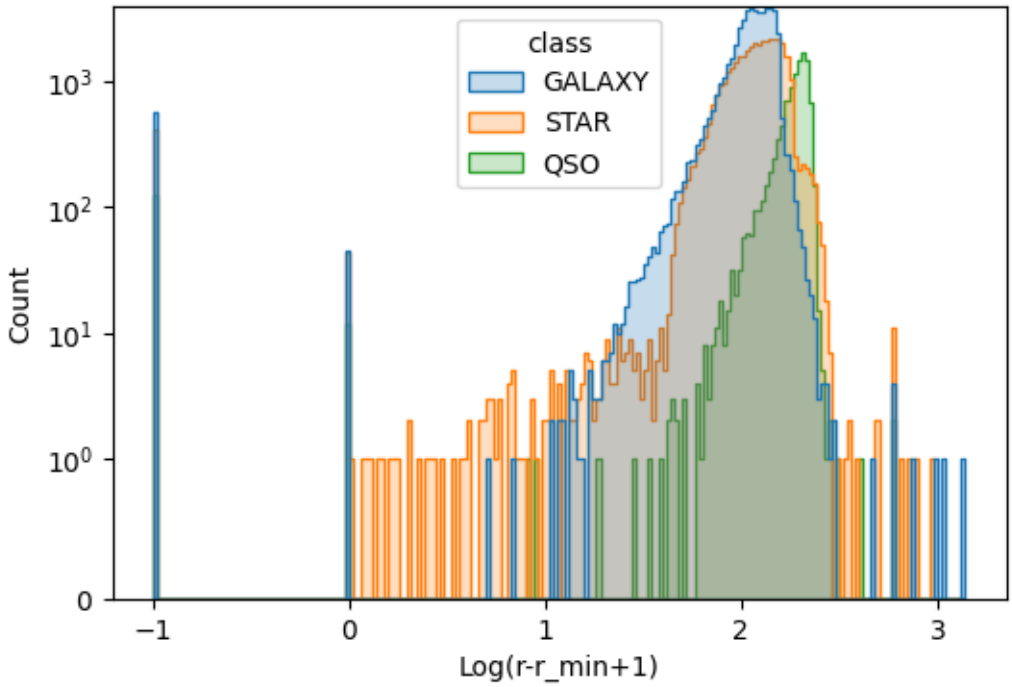
5.3 Band flux distribution

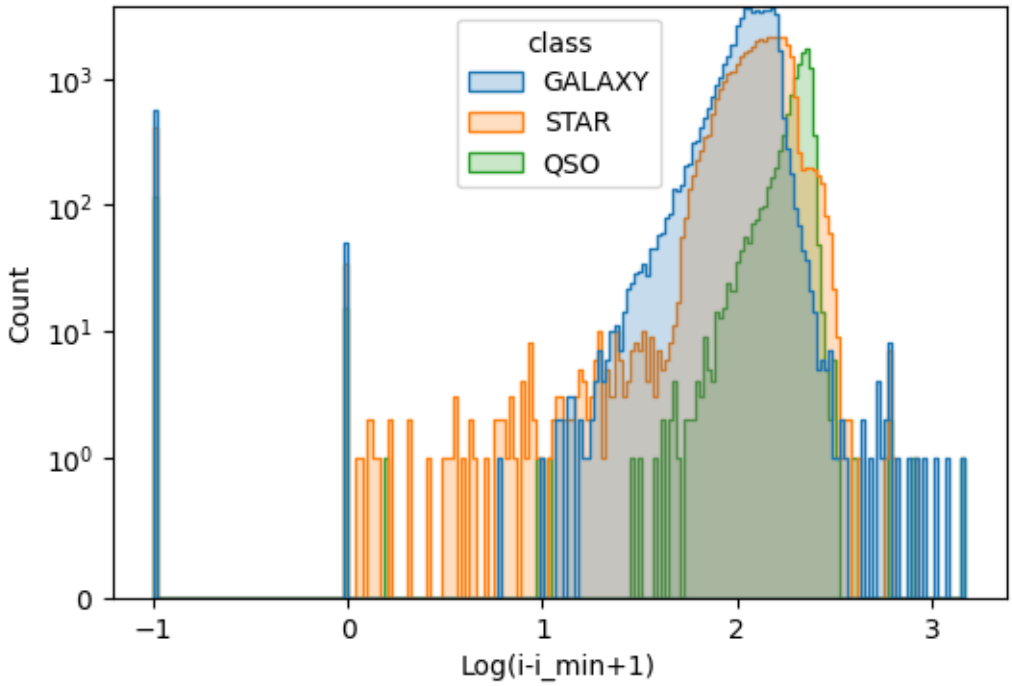
Below, we plot the band flux distribution per each class.

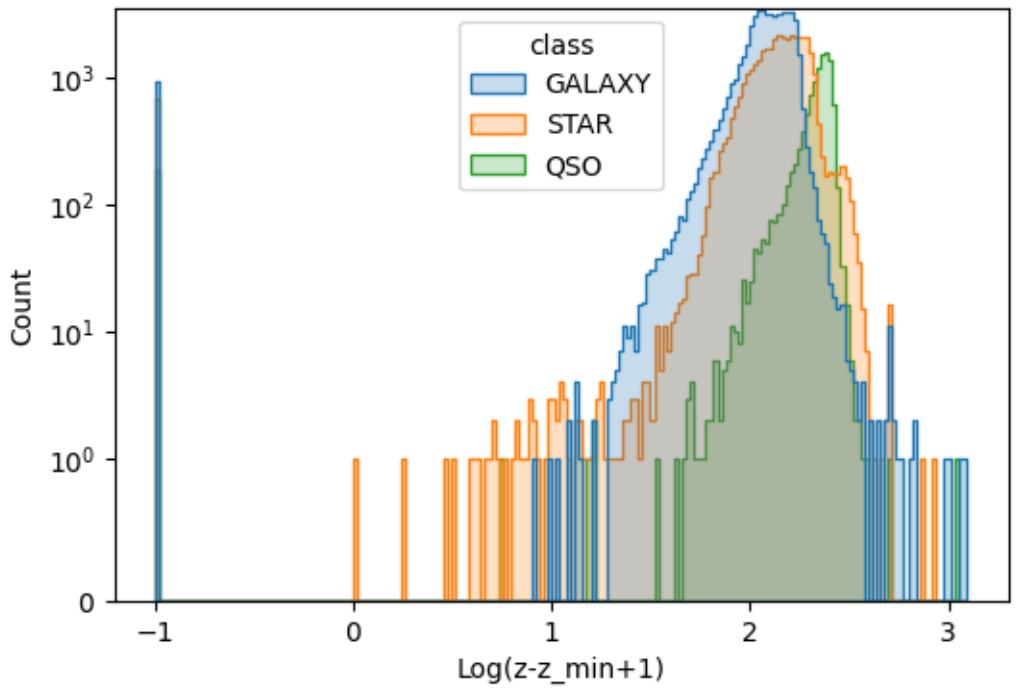
[35]:
```
# - Plot flux vars
cols_flux = ['u','g','r','i','z']
cols= ['u','g','r','i','z','redshift']
for col in cols_flux:
  plt.figure(figsize=(6, 4))
  ax= sns.histplot(data=df_transf, x=col, hue='class', element="step",
              stat='count', fill=True, binwidth=0.02, log_scale=(False,False))
  xlabel = "Log({var}-{var}_min+1)".format(var=col)
  plt.xlabel(xlabel)
  ax.set_yscale("symlog", linthresh=1) # 0..1 is linear, above is ~log
  ax.set_ylim(0, None)
  plt.show()
```


class
GALAXY
STAR
QSO
Count
10^3
10^2
10^1
10^0
0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
Log(u-u_min+1)


class
GALAXY
STAR
QSO
Count
10^3
10^2
10^1
10^0
0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
Log(g-g_min+1)


class
GALAXY
STAR
QSO
Count
10^3
10^2
10^1
10^0
0
−1
0
1
2
3
Log(r-r_min+1)


class
GALAXY
STAR
QSO
Count
10^3
10^2
10^1
10^0
0
−1
0
1
2
3
Log(i-i_min+1)



5.4 Variable scatter plots

Let's visualize the variable (band flux, redshift) correlations with paired scatter plots.

[31]:
```
df_transf.describe(include=[np.number])
```

[31]:

	u	g	r	i	\
count	100000.000000	100000.000000	100000.000000	100000.000000	
mean	2.122886	2.104195	2.040214	2.067568	
std	0.487531	0.436975	0.360665	0.363618	
min	-1.000000	-1.000000	-1.000000	-1.000000	
25%	2.145375	2.096366	1.994139	2.014916	
50%	2.223727	2.178120	2.085942	2.109596	
75%	2.267265	2.238114	2.169328	2.196909	
max	2.301399	2.425391	3.142711	3.169870	

	z	redshift
count	100000.000000	100000.000000
mean	2.074545	0.121282
std	0.442953	0.246173
min	-1.000000	0.000000
25%	2.031924	0.004131
50%	2.134048	0.049263
75%	2.226292	0.095149

```
max          3.094185       2.081362
```

First, we compute the **Pearson**, **Spearman** and **Kendall correlation coefficients** on the transformed observed data, using either Pandas or Scipy implementations.

[37]:
```
from scipy import stats
from itertools import combinations

def scipy_corr_matrix(df, method="pearson"):
  """ Helper method to compute correlation coeff & p-values using scipy on␣
 ↪input data frame"""
  cols = df.columns
  n = len(cols)

  corr = pd.DataFrame(np.eye(n), columns=cols, index=cols)
  pval = pd.DataFrame(np.zeros((n,n)), columns=cols, index=cols)

  for col1, col2 in combinations(cols, 2):
    x = df[col1].values
    y = df[col2].values
    mask = ~(np.isnan(x) | np.isnan(y))
    x = x[mask]
    y = y[mask]

    if method == "pearson":
      r, p = stats.pearsonr(x, y)
    elif method == "spearman":
      r, p = stats.spearmanr(x, y, nan_policy="omit")
    elif method == "kendall":
      r, p = stats.kendalltau(x, y, nan_policy="omit")
    else:
      raise ValueError("Method must be pearson, spearman, or kendall")

    corr.loc[col1, col2] = r
    corr.loc[col2, col1] = r
    pval.loc[col1, col2] = p
    pval.loc[col2, col1] = p

  return corr, pval

# - Set data
df_transf_nans= df_transf.copy()
df_transf_nans.replace(to_replace=-1, value=np.nan, inplace=True)

# - Compute Pearson coefficient
corr_p_pandas = df_transf_nans[cols].corr(method='pearson')
corr_p, pval_p = scipy_corr_matrix(df_transf_nans[cols], method="pearson")
```

```
print("--> Pearson corr coeff (Pandas)")
print(corr_p_pandas)
print("--> Pearson corr coeff (Scipy)")
print(corr_p)

# - Compute Spearman coefficient
corr_s_pandas = df_transf_nans[cols].corr(method='spearman')
corr_s, pval_s = scipy_corr_matrix(df_transf_nans[cols], method="spearman")
print("\n--> Spearman corr coeff (Pandas)")
print(corr_s_pandas)
print("--> Spearman corr coeff (Scipy)")
print(corr_s)

# - Compute Kendall coefficient
corr_k_pandas = df_transf_nans[cols].corr(method='kendall')
corr_k, pval_k = scipy_corr_matrix(df_transf_nans[cols], method="kendall")
print("\n--> Kendall corr coeff (Pandas)")
print(corr_k_pandas)
print("--> Kendall corr coeff (Scipy)")
print(corr_k)
```

```
--> Pearson corr coeff (Pandas)
                u         g         r         i         z  redshift
u        1.000000  0.857186  0.670545  0.602137  0.599309  0.194901
g        0.857186  1.000000  0.884671  0.844985  0.887230  0.411810
r        0.670545  0.884671  1.000000  0.825161  0.883791  0.393831
i        0.602137  0.844985  0.825161  1.000000  0.894637  0.387066
z        0.599309  0.887230  0.883791  0.894637  1.000000  0.417018
redshift 0.194901  0.411810  0.393831  0.387066  0.417018  1.000000
--> Pearson corr coeff (Scipy)
                u         g         r         i         z  redshift
u        1.000000  0.857186  0.670545  0.602137  0.599309  0.194901
g        0.857186  1.000000  0.884671  0.844985  0.887230  0.411810
r        0.670545  0.884671  1.000000  0.825161  0.883791  0.393831
i        0.602137  0.844985  0.825161  1.000000  0.894637  0.387066
z        0.599309  0.887230  0.883791  0.894637  1.000000  0.417018
redshift 0.194901  0.411810  0.393831  0.387066  0.417018  1.000000

--> Spearman corr coeff (Pandas)
                u         g         r         i         z  redshift
u        1.000000  0.779375  0.627709  0.556142  0.500445  0.359393
g        0.779375  1.000000  0.956374  0.915578  0.884069  0.364612
r        0.627709  0.956374  1.000000  0.979830  0.967353  0.245169
i        0.556142  0.915578  0.979830  1.000000  0.984708  0.166923
z        0.500445  0.884069  0.967353  0.984708  1.000000  0.112279
redshift 0.359393  0.364612  0.245169  0.166923  0.112279  1.000000
--> Spearman corr coeff (Scipy)
```

```
                u         g         r         i         z  redshift
u        1.000000  0.779375  0.627709  0.556142  0.500445  0.359393
g        0.779375  1.000000  0.956374  0.915578  0.884069  0.364612
r        0.627709  0.956374  1.000000  0.979830  0.967353  0.245169
i        0.556142  0.915578  0.979830  1.000000  0.984708  0.166923
z        0.500445  0.884069  0.967353  0.984708  1.000000  0.112279
redshift 0.359393  0.364612  0.245169  0.166923  0.112279  1.000000

--> Kendall corr coeff (Pandas)
                u         g         r         i         z  redshift
u        1.000000  0.595807  0.452650  0.393196  0.349850  0.246576
g        0.595807  1.000000  0.832127  0.761532  0.712933  0.262753
r        0.452650  0.832127  1.000000  0.910676  0.860005  0.174889
i        0.393196  0.761532  0.910676  1.000000  0.927646  0.118770
z        0.349850  0.712933  0.860005  0.927646  1.000000  0.079643
redshift 0.246576  0.262753  0.174889  0.118770  0.079643  1.000000
--> Kendall corr coeff (Scipy)
                u         g         r         i         z  redshift
u        1.000000  0.595807  0.452650  0.393196  0.349850  0.246576
g        0.595807  1.000000  0.832127  0.761532  0.712933  0.262753
r        0.452650  0.832127  1.000000  0.910676  0.860005  0.174889
i        0.393196  0.761532  0.910676  1.000000  0.927646  0.118770
z        0.349850  0.712933  0.860005  0.927646  1.000000  0.079643
redshift 0.246576  0.262753  0.174889  0.118770  0.079643  1.000000
```

Below, we display variable scatter plots of transformed observed data (NaNs removed). We also display the correlation coefficients computed above in the upper off-diagonal panels. Variable distributions are displayed in the diagonal panels.

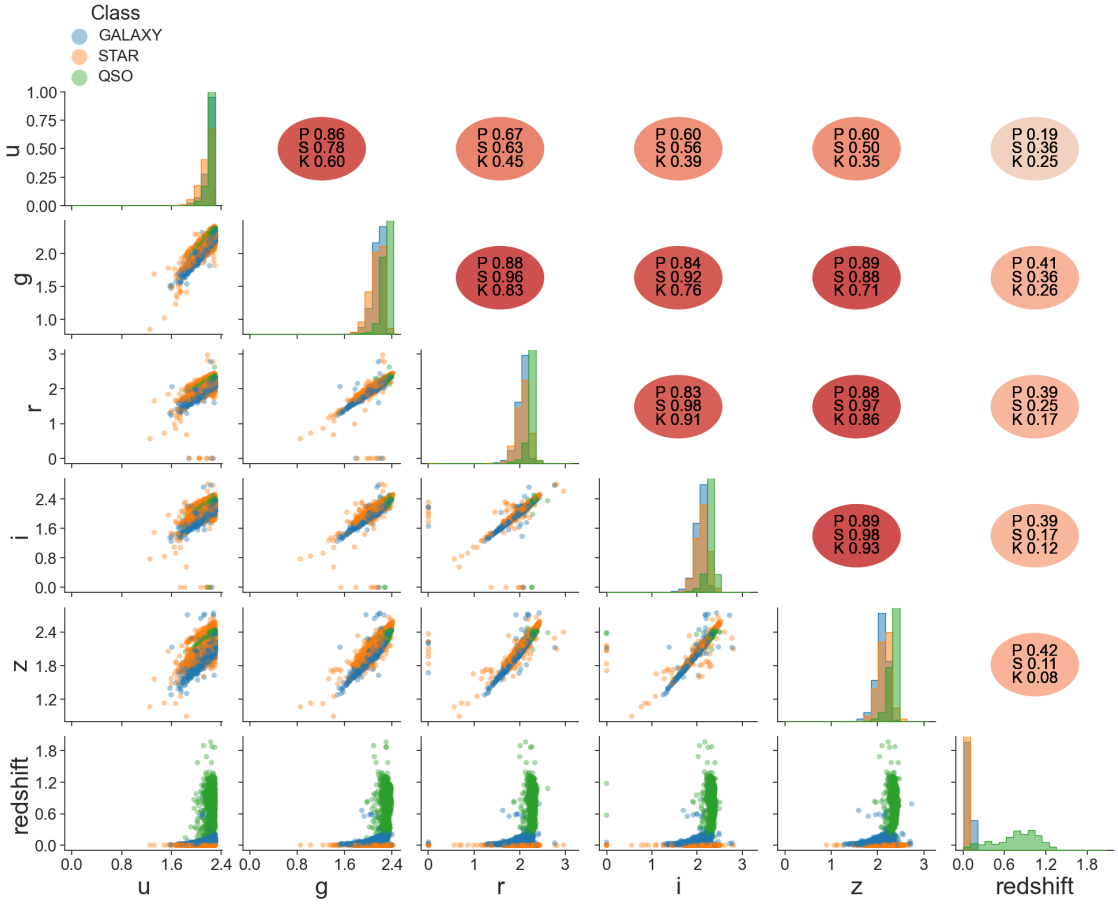
[68]:
```
from functools import partial
import matplotlib as mpl
import matplotlib.ticker as mticker
from matplotlib.patches import Circle
from matplotlib.patches import Ellipse
import matplotlib.lines as mlines
import matplotlib.patches as mpatches

def corrdot(
  x, y,
  corr_p=None, corr_s=None, corr_k=None,
  axis_aspect=None,
  **kwargs
):
  ax = plt.gca()
  ax.set_axis_off()
  ax.set_autoscale_on(False)

  col_x = x.name
```

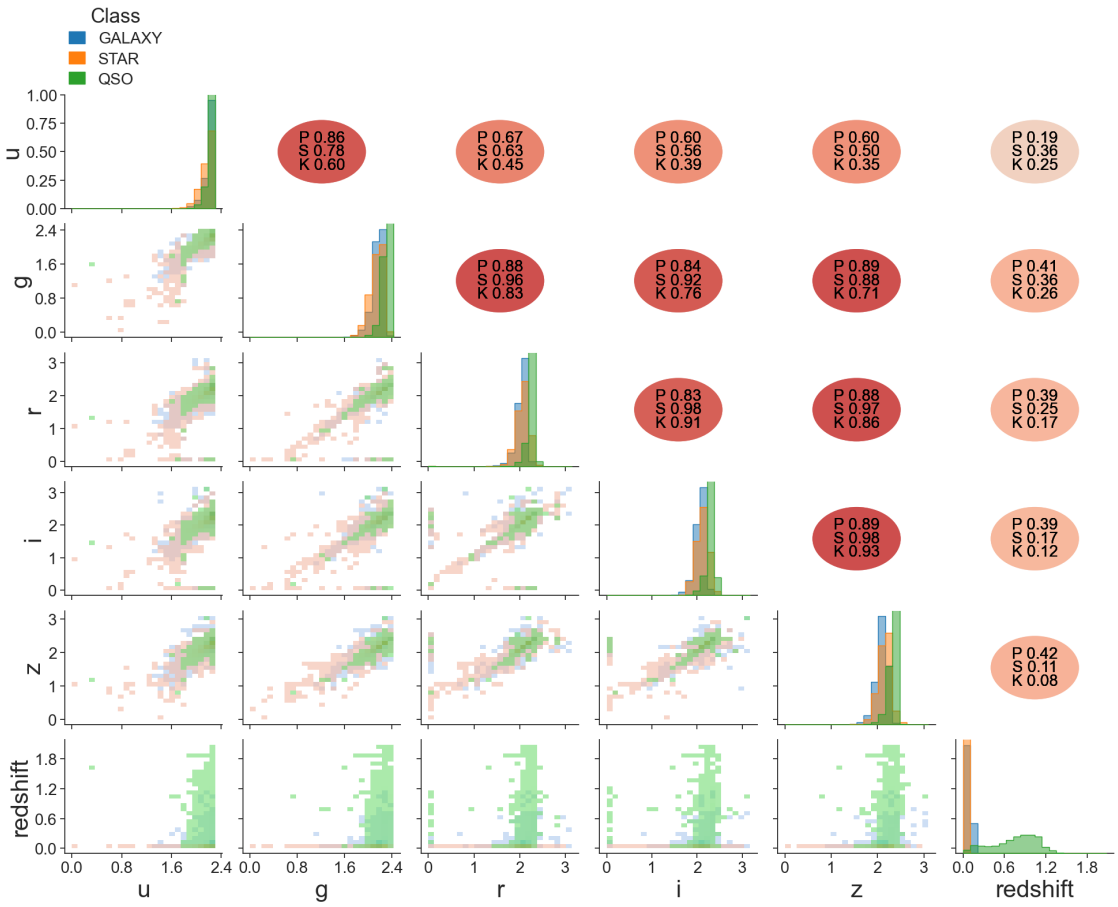
```
  col_y = y.name

  rp = corr_p.loc[col_x, col_y]
  rs = corr_s.loc[col_x, col_y]
  rk = corr_k.loc[col_x, col_y]

  cmap = mpl.colormaps["coolwarm"]
  norm = mpl.colors.Normalize(vmin=-1, vmax=1)
  color_p = cmap(norm(rp))
  color_s = cmap(norm(rs))
  color_k = cmap(norm(rk))

  # - Draw ellipse background
  radius = 0.28
  ellipse = Ellipse(
    (0.5, 0.5),
    width=2 * radius,
    height=2 * radius * axis_aspect,
    transform=ax.transAxes,
    facecolor=color_p,
    edgecolor="none",
    alpha=0.5,
    zorder=1
  )
  ax.add_patch(ellipse)

  # - Three vertically spaced values
  ax.text(0.5, 0.62, f"P {rp:.2f}",
        ha="center", va="center",
        transform=ax.transAxes, fontsize=20, zorder=2, color="black")

  ax.text(0.5, 0.5, f"S {rs:.2f}",
        ha="center", va="center",
        transform=ax.transAxes, fontsize=20, zorder=2, color="black")

  ax.text(0.5, 0.38, f"K {rk:.2f}",
        ha="center", va="center",
        transform=ax.transAxes, fontsize=20, zorder=2, color="black")


def draw_scatter_plot(df, offdiag_hist=False):

  with sns.axes_style("ticks"), sns.plotting_context("notebook", font_scale=1.
 ↪2):
    # - Create grid
    g = sns.PairGrid(
      df[cols + ["class"]],
```

```
    hue="class",
    aspect=1.3,
    diag_sharey=False
  )

  # - Add lower panels (scatter plots)
  if offdiag_hist:
    g.map_lower(
      sns.histplot,
      bins=25,
      alpha=0.6,
    )
  else:
    g.map_lower(
      sns.scatterplot,
      s=45,
      alpha=0.4,
      linewidth=0,
      edgecolor=None
    )

  # - Add diagonal panels (histograms)
  def diag_hist(x, color=None, **kwargs):
    ax = plt.gca()

    # Compute common bins from full variable range
    full_min = df_plot[x.name].min()
    full_max = df_plot[x.name].max()
    bins = np.linspace(full_min, full_max, 21)

    sns.histplot(
      x,
      bins=bins,
      stat="density",
      alpha=0.5,
      color=color,
      element="step",      # cleaner overlay
      fill=True,
      ax=ax
    )
    ax.set_ylim(bottom=0)

  g.map_diag(diag_hist)

  # - Add upper panels (correlation coeff)
  g.fig.canvas.draw()
  # Get aspect ratio from first upper axis
```

```
sample_ax = g.axes[0, 1]
bbox = sample_ax.get_position()
width = bbox.width
height = bbox.height
axis_aspect = height / width

corr_func = partial(
  corrdot,
  corr_p=corr_p,
  corr_s=corr_s,
  corr_k=corr_k,
  axis_aspect=axis_aspect
)
g.map_upper(corr_func)

# - Set axis label/ticks
for ax in g.axes.flatten():
  ax.tick_params(axis='both', labelsize=20)
  ax.set_xlabel(ax.get_xlabel(), fontsize=30)
  ax.set_ylabel(ax.get_ylabel(), fontsize=30)
  ax.xaxis.set_major_locator(mticker.MaxNLocator(nbins=4))
  ax.yaxis.set_major_locator(mticker.MaxNLocator(nbins=4))

# - Add legend
g.add_legend(title="Class", adjust_subtitles=True)
g._legend.get_title().set_fontsize(25)

# - Move legend to top
g._legend.set_bbox_to_anchor((0.1, 1.05))
g._legend.set_loc("upper center")
g._legend.set_frame_on(False)

# - Increase label size
for text in g._legend.texts:
  text.set_fontsize(20)

# - Increase marker size
for handle in g._legend.legend_handles:
  if isinstance(handle, mlines.Line2D): # scatter plot
    handle.set_markersize(20)
  elif isinstance(handle, mpatches.Patch): # hist2d, bar, etc
    handle.set_width(20)
    handle.set_height(15)

g.fig.subplots_adjust(top=0.93)

plt.show()
```

```
# - Set data for the plot
df_plot_sample = df_transf_nans.sample(10000, random_state=0).dropna() # Reduce␣
 ↪number of data to plot
df_plot = df_transf_nans.copy().dropna()

# - Draw scatter plot
draw_scatter_plot(df_plot_sample, offdiag_hist=False)
draw_scatter_plot(df_plot, offdiag_hist=True)
```



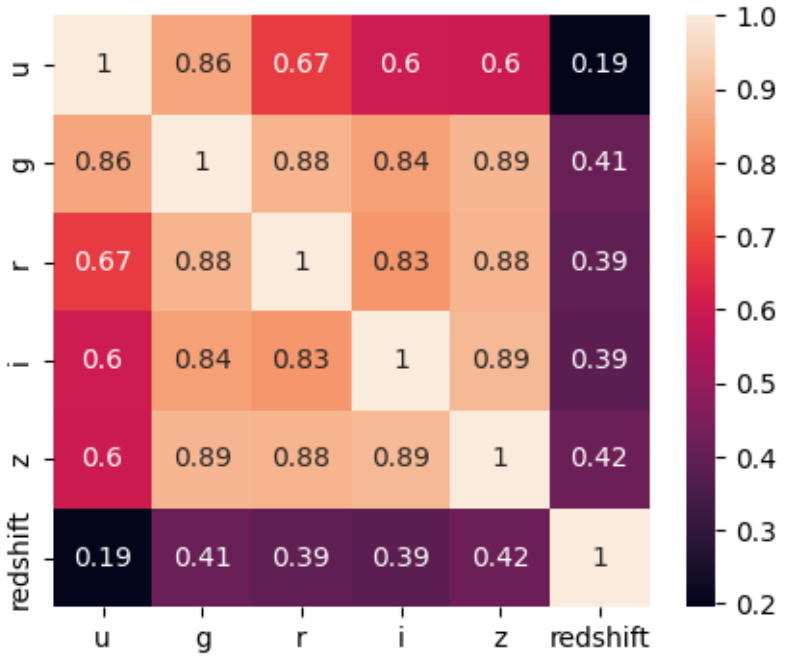


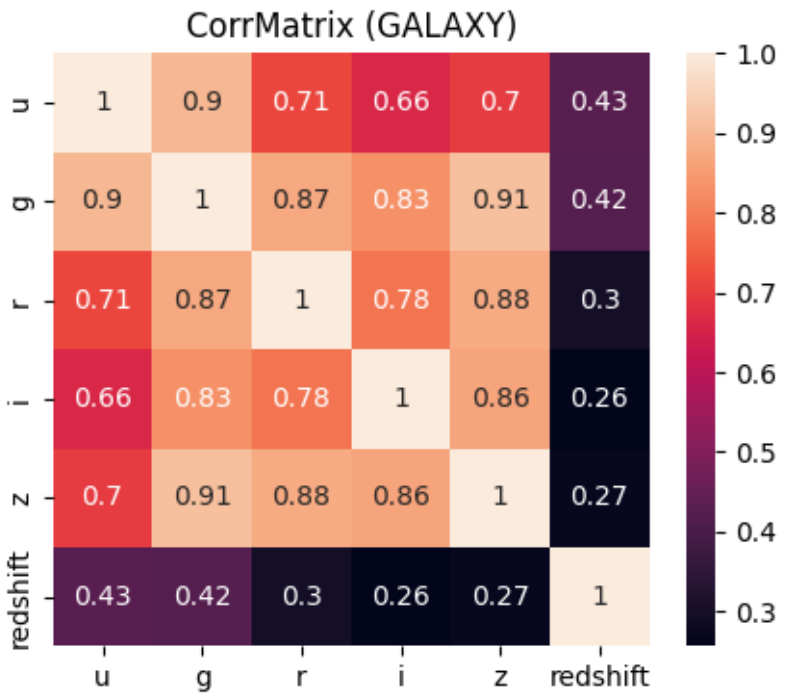
Below, we plot variable correlation matrix for each class, using the Pearson correlation coefficient as metric.

[39]:
```
# - Compute and plot correlation matrix (global)
cm = df_transf_nans.corr(method='pearson', numeric_only=True)

print("--> Correlation matrix")
print(cm)
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True)

# - Compute and plot correlation matrix per class
for cls, g in df_transf_nans.groupby("class"):
  cm_class = g.corr(method="pearson", numeric_only=True)
  print(f"\n--> Correlation matrix for class: {cls}")
  print(cm_class)
  plt.figure(figsize=(5, 4))
  sns.heatmap(cm_class, annot=True)
  title= "CorrMatrix ({c})".format(c=cls)
```

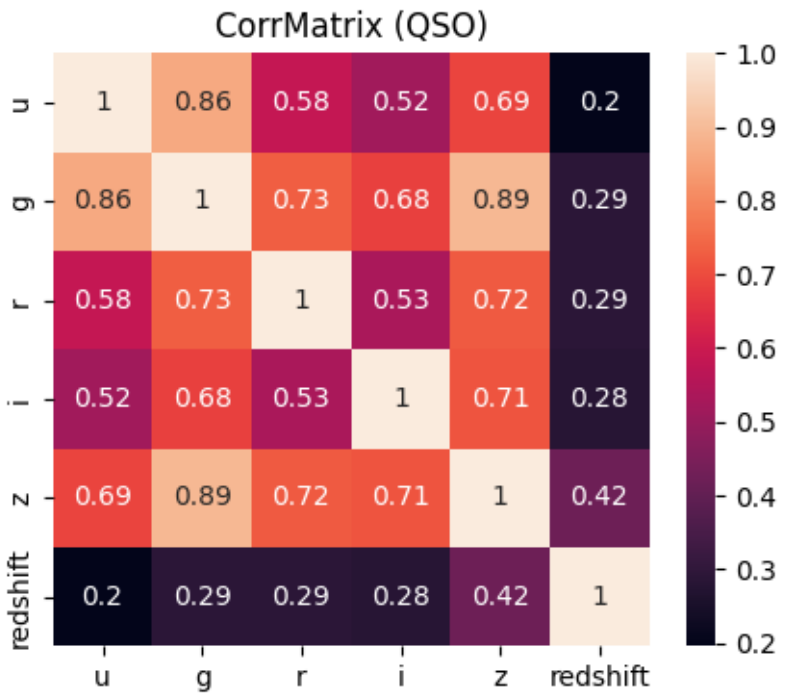
```
  plt.title(title)
  plt.show()
```

```
--> Correlation matrix
                u         g         r         i         z  redshift
u        1.000000  0.857186  0.670545  0.602137  0.599309  0.194901
g        0.857186  1.000000  0.884671  0.844985  0.887230  0.411810
r        0.670545  0.884671  1.000000  0.825161  0.883791  0.393831
i        0.602137  0.844985  0.825161  1.000000  0.894637  0.387066
z        0.599309  0.887230  0.883791  0.894637  1.000000  0.417018
redshift 0.194901  0.411810  0.393831  0.387066  0.417018  1.000000

--> Correlation matrix for class: GALAXY
                u         g         r         i         z  redshift
u        1.000000  0.899185  0.714118  0.661336  0.695319  0.428685
g        0.899185  1.000000  0.871016  0.831047  0.914315  0.420186
r        0.714118  0.871016  1.000000  0.784424  0.875274  0.296934
i        0.661336  0.831047  0.784424  1.000000  0.863817  0.256631
z        0.695319  0.914315  0.875274  0.863817  1.000000  0.268398
redshift 0.428685  0.420186  0.296934  0.256631  0.268398  1.000000
```

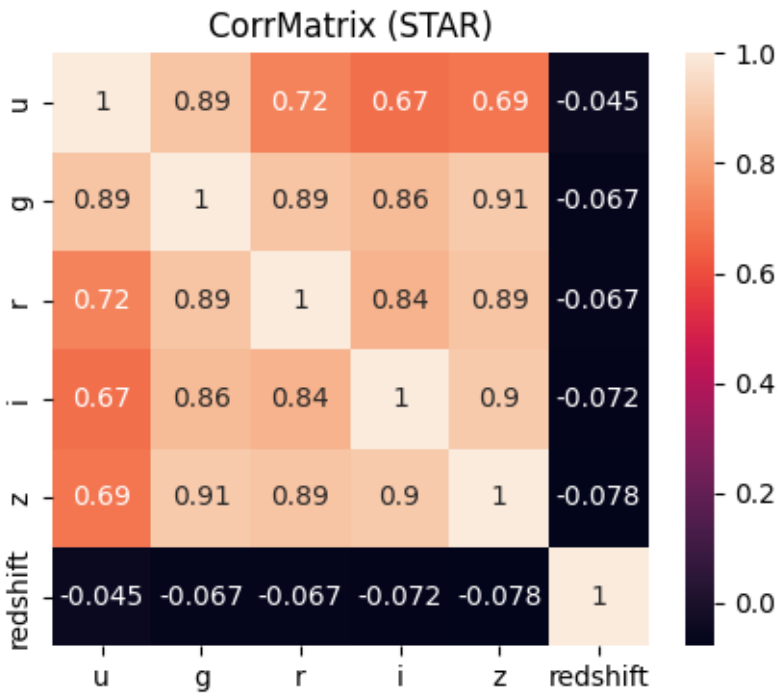





```
--> Correlation matrix for class: QSO
                  u         g         r         i         z  redshift
u          1.000000  0.864266  0.583919  0.518567  0.687772  0.195566
g          0.864266  1.000000  0.729276  0.681283  0.890106  0.286960
r          0.583919  0.729276  1.000000  0.532616  0.720802  0.286814
i          0.518567  0.681283  0.532616  1.000000  0.713660  0.279529
z          0.687772  0.890106  0.720802  0.713660  1.000000  0.420579
redshift   0.195566  0.286960  0.286814  0.279529  0.420579  1.000000
```




```
--> Correlation matrix for class: STAR
                  u         g         r         i         z  redshift
u          1.000000  0.888260  0.719990  0.674406  0.690263 -0.045032
g          0.888260  1.000000  0.887888  0.863716  0.905843 -0.067214
r          0.719990  0.887888  1.000000  0.840691  0.888609 -0.066817
i          0.674406  0.863716  0.840691  1.000000  0.903735 -0.071580
z          0.690263  0.905843  0.888609  0.903735  1.000000 -0.078271
redshift -0.045032 -0.067214 -0.066817 -0.071580 -0.078271  1.000000
```



5.5 Dimensionality reduction

Let's project the numerical variables (5 band flux, redshift) in a 2D space, using PCA, UMAP and tSNE dimensionality reduction tools.

First, we set the input data for all tools.

[40]:
```
# - Set UMAP data & labels using data without NaNs rows
df_nonans= df.dropna()
X_nonans= df_nonans[cols].to_numpy() # this does not contains nans
y_nonans= df_nonans['class']

# - Set UMAP data & labels using transformed data without NaNs rows
df_transf_nonans= df_transf.copy()
df_transf_nonans.replace(to_replace=-1, value=np.nan, inplace=True)
df_transf_nonans.dropna(inplace=True)
X_transf_nonans= df_transf_nonans[cols].to_numpy() # here NANs are =-1
y_transf_nonans= df_transf_nonans['class']

# - Set UMAP data & labels using transformed data (nans set to -1)
X_transf= df_transf[cols].to_numpy() # here NANs are =-1
y_transf= df_transf['class']
has_nan_values= (
  (df_transf[cols_flux].eq(-1))
     .any(axis=1)
```

```
    .to_numpy()
    .astype(int)
)
print("#no nans")
print(np.count_nonzero(has_nan_values == 0))
print("#nans")
print(np.count_nonzero(has_nan_values == 1))
```

```
#no nans
92182
#nans
7818
```

Below, we define an helper method to draw the 2D embeddings for all methods.

[41]:
```
def plot_embeddings(
  embeddings, # embeddings
  y, # class
  mask=None,
  title="UMAP projection",
  s_normal=0.1,
  s_masked=0.1,
  alpha_normal=0.7,
  alpha_masked=0.9
):
  """ Draw 2D embeddings """
  plt.figure(figsize=(8, 6))
  classes = np.unique(y)
  cmap = plt.get_cmap('tab10')
  class_to_idx = {cls: i for i, cls in enumerate(classes)}

  for cls in np.unique(y):
    idx_cls = (y == cls)
    base_color = cmap(class_to_idx[cls])

    if mask is None:
      idx_normal = idx_cls
    else:
      idx_normal = idx_cls & (mask == 0)

    plt.scatter(
      #embeddings[idx, 0], embeddings[idx, 1],
      embeddings[idx_normal, 0], embeddings[idx_normal, 1],
      s=s_normal,
      marker="o",
      color=base_color,
      label=cls,
      alpha=alpha_normal
```

```
    )

    if mask is not None:
      # ---- masked points ----
      idx_masked = idx_cls & (mask == 1)

      # slightly darker version of class color
      darker = tuple(np.clip(np.array(base_color[:3]) * 0.6, 0, 1))

      plt.scatter(
        embeddings[idx_masked, 0],
        embeddings[idx_masked, 1],
        s=s_masked,
        color=darker,
        marker="x",
        alpha=alpha_masked
      )

  plt.title(title)
  plt.xlabel("Var1")
  plt.ylabel("Var2")
  plt.legend(markerscale=5, fontsize=8)
  plt.tight_layout()
  plt.show()
```

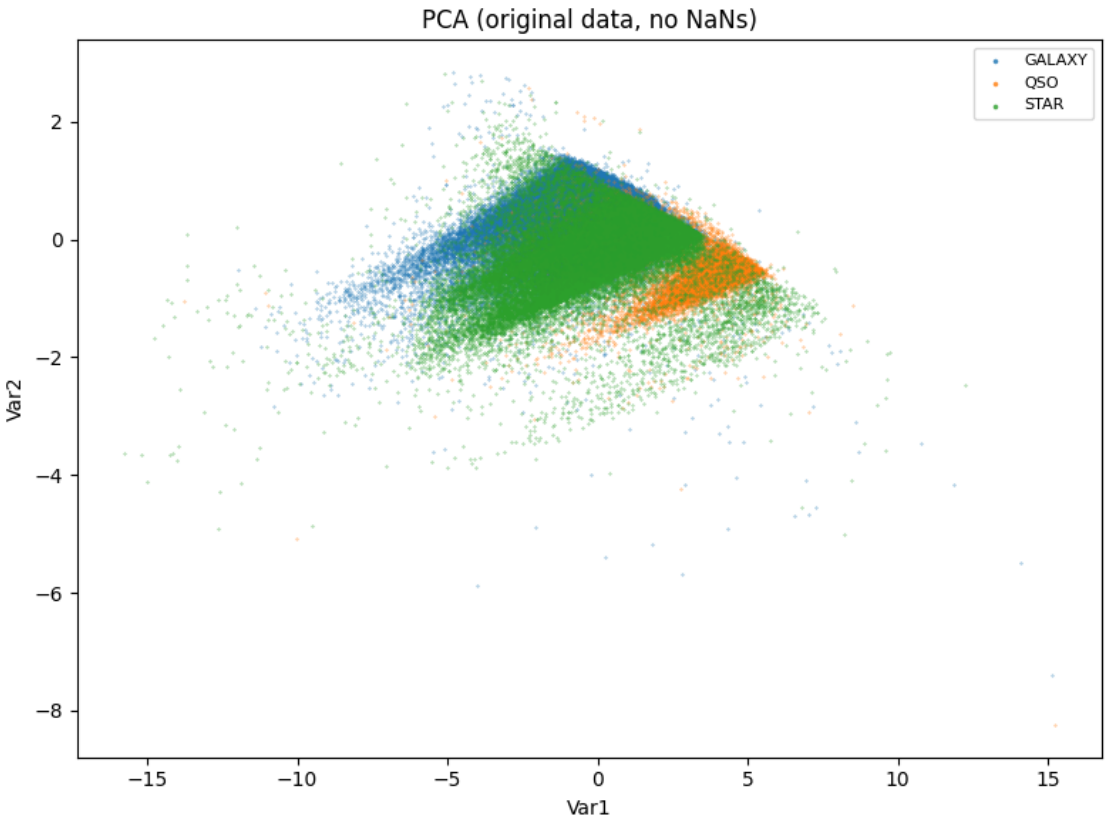
5.5.1 PCA

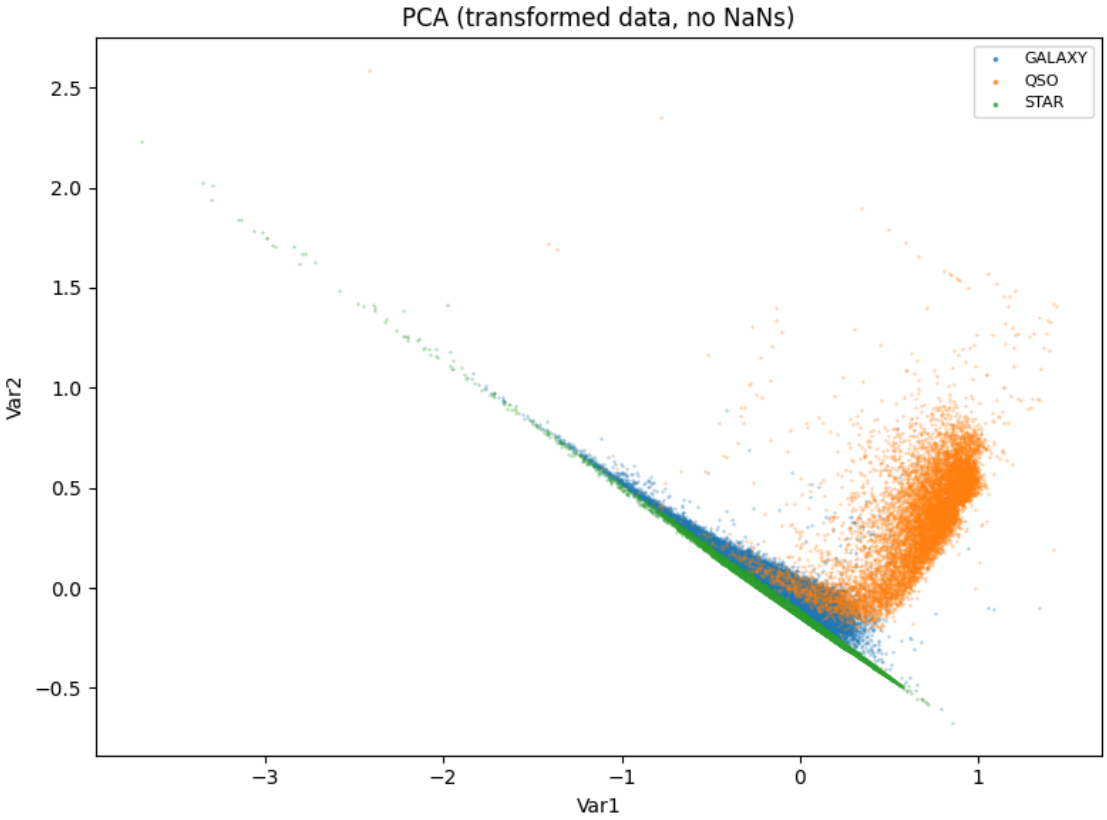
Let's run PCA on transformed data.

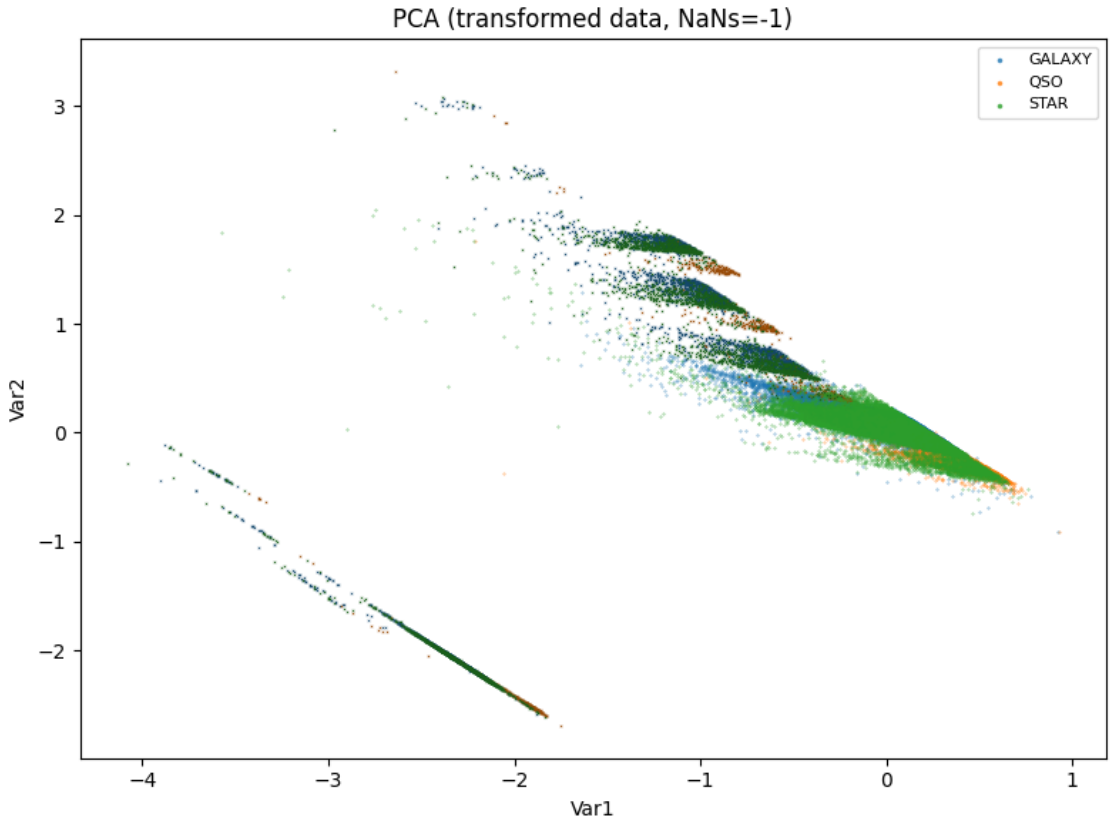
[54]:
```
def run_pca(X):
  # - Init PCA
  pca= PCA(n_components=2)    # this set 2 principal components at maximum
  #pca= PCA(n_components=0.9) # this set a number of components retaining >=90%␣
 ↪variance

  # - Run PCA
  X_embed= pca.fit_transform(X)

  # - Retrieve outputs
  n_sel_components= pca.n_components_
  explained_var_ratio= pca.explained_variance_ratio_
  explained_var_ratio_cum = np.cumsum(explained_var_ratio)
  print(f"PCA components left: {n_sel_components}")
  print(f"Explained variance perc: {explained_var_ratio}")
  print(f"Explained variance perc (cum): {explained_var_ratio_cum}")

  return X_embed
```

[55]:
```
# - Run PCA on original & transformed data
print("\n--> Run PCA on original data (no nans) ...")
X_nonans_embed= run_pca(X_nonans)

print("\n--> Run PCA on transformed data (no nans) ...")
X_transf_nonans_embed= run_pca(X_transf_nonans)

print("\n--> Run PCA on transformed data (nans=-1) ...")
X_transf_embed= run_pca(X_transf)

# - Plot PCA embeddings
plot_embeddings(X_nonans_embed, y_nonans, mask=None, title="PCA (original data,␣
 ↪no NaNs)")
plot_embeddings(X_transf_nonans_embed, y_transf_nonans, mask=None, title="PCA␣
 ↪(transformed data, no NaNs)")
plot_embeddings(X_transf_embed, y_transf, mask=has_nan_values, title="PCA␣
 ↪(transformed data, NaNs=-1)")
```

```
--> Run PCA on original data (no nans) …
PCA components left: 2
Explained variance perc: [0.87257049 0.07746431]
Explained variance perc (cum): [0.87257049 0.95003481]

--> Run PCA on transformed data (no nans) …
PCA components left: 2
Explained variance perc: [0.66233098 0.25374815]
Explained variance perc (cum): [0.66233098 0.91607912]

--> Run PCA on transformed data (nans=-1) …
PCA components left: 2
Explained variance perc: [0.26914763 0.22827788]
Explained variance perc (cum): [0.26914763 0.49742552]
```


PCA (original data, no NaNs)
GALAXY
QSO
STAR
Var2
Var1
2
0
−2
−4
−6
−8
−15
−10
−5
0
5
10
15


PCA (transformed data, no NaNs)
GALAXY
QSO
STAR
Var2
Var1
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−3
−2
−1
0
1



5.5.2 UMAP

Let's run UMAP on transformed data.

[56]:
```
# - Init UMAP: play with n_neighbors & min_dist parameters
umap_dimred= UMAP(
  n_components=2,
  n_neighbors=30,
  min_dist=0.2,
  random_state=42
)
```

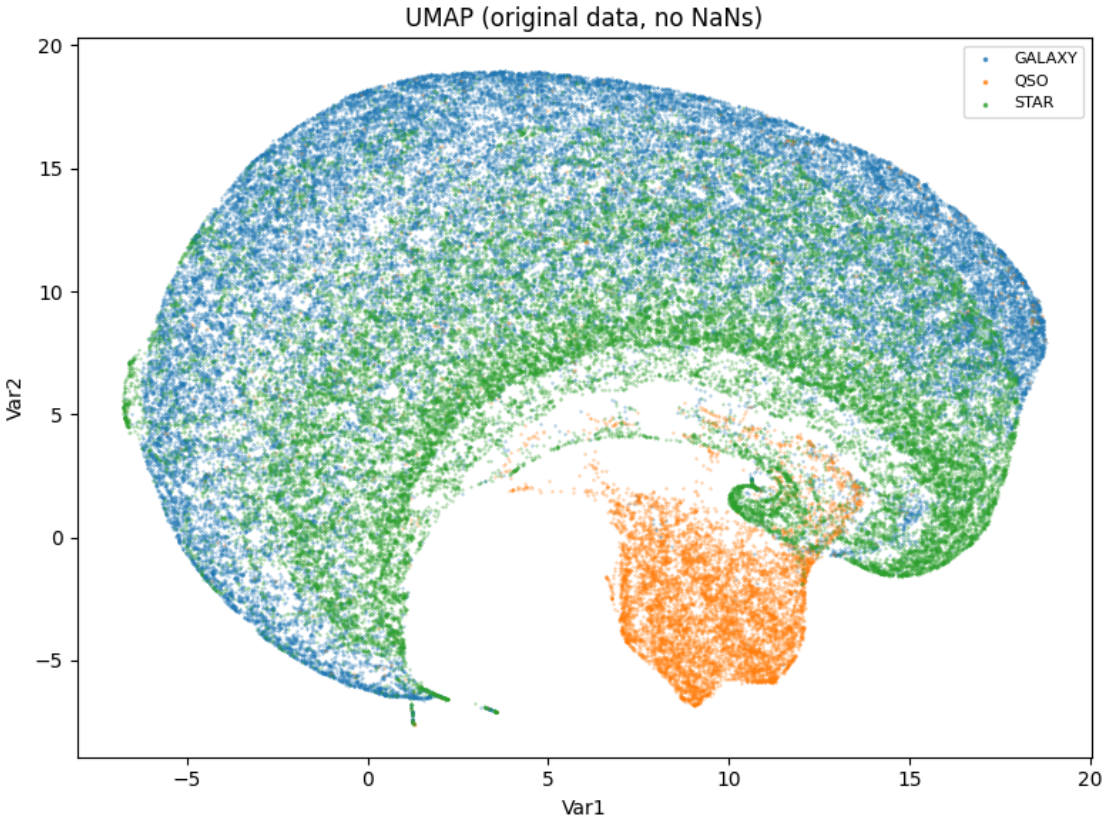
[57]:
```
# - Run UMAP on original & transformed data
print("Run UMAP on original data (no nans) ...")
X_nonans_embed= umap_dimred.fit_transform(X_nonans)

print("Run UMAP on transformed data (no nans) ...")
X_transf_nonans_embed= umap_dimred.fit_transform(X_transf_nonans)

print("Run UMAP on transformed data (nans=-1) ...")
X_transf_embed= umap_dimred.fit_transform(X_transf)
```

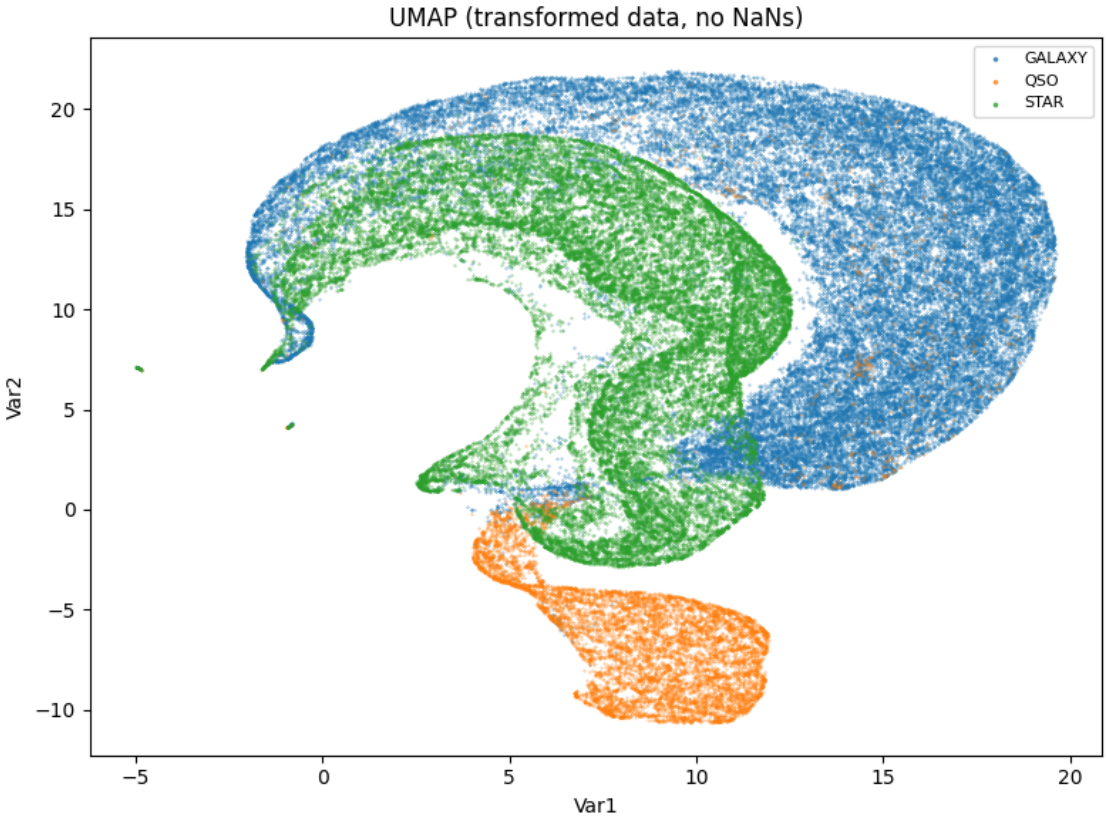
```
# - Plot UMAP embeddings on original & transformed data
plot_embeddings(X_nonans_embed, y_nonans, mask=None, title="UMAP (original␣
 ↪data, no NaNs)")
plot_embeddings(X_transf_nonans_embed, y_transf_nonans, mask=None, title="UMAP␣
 ↪(transformed data, no NaNs)")
plot_embeddings(X_transf_embed, y_transf, mask=has_nan_values, title="UMAP␣
 ↪(transformed data, NaNs=-1)")
```

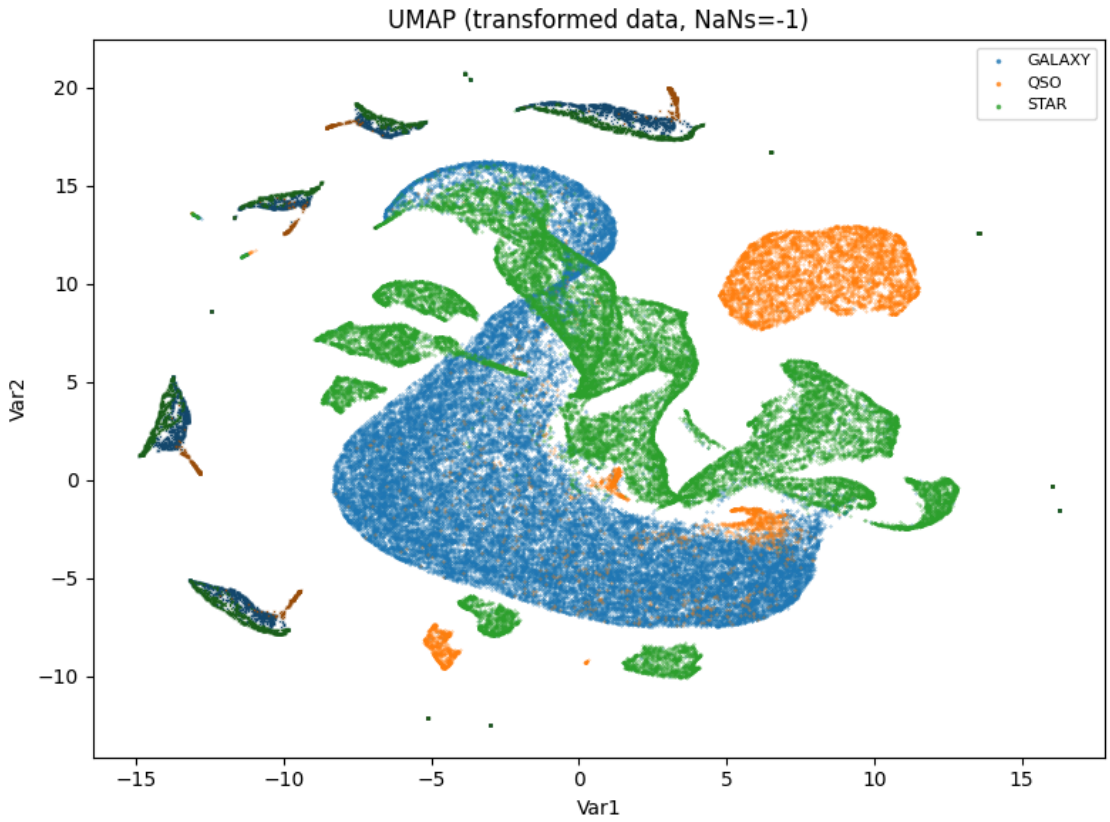
```
Run UMAP on original data (no nans) …
```

```
/home/riggi/Software/venvs/usc-c-ai-school-2026/lib/python3.10/site-
packages/umap/umap_.py:1952: UserWarning: n_jobs value 1 overridden to 1 by
setting random_state. Use no seed for parallelism.
  warn(
```

```
Run UMAP on transformed data (no nans) …
Run UMAP on transformed data (nans=-1) …
```




UMAP (transformed data, no NaNs)
GALAXY
QSO
STAR
Var2
Var1
20
15
10
5
0
−5
−10
−5
0
5
10
15
20



5.5.3 tSNE

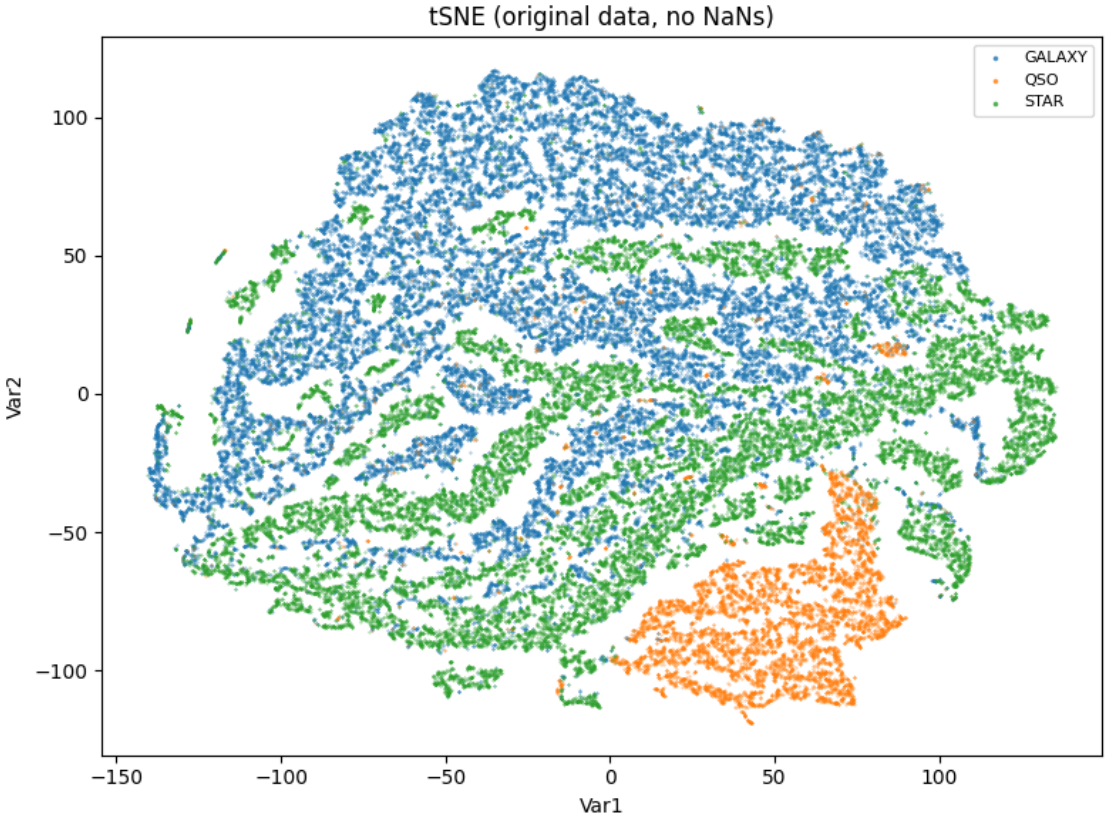
Let's run tSNE on transformed data.

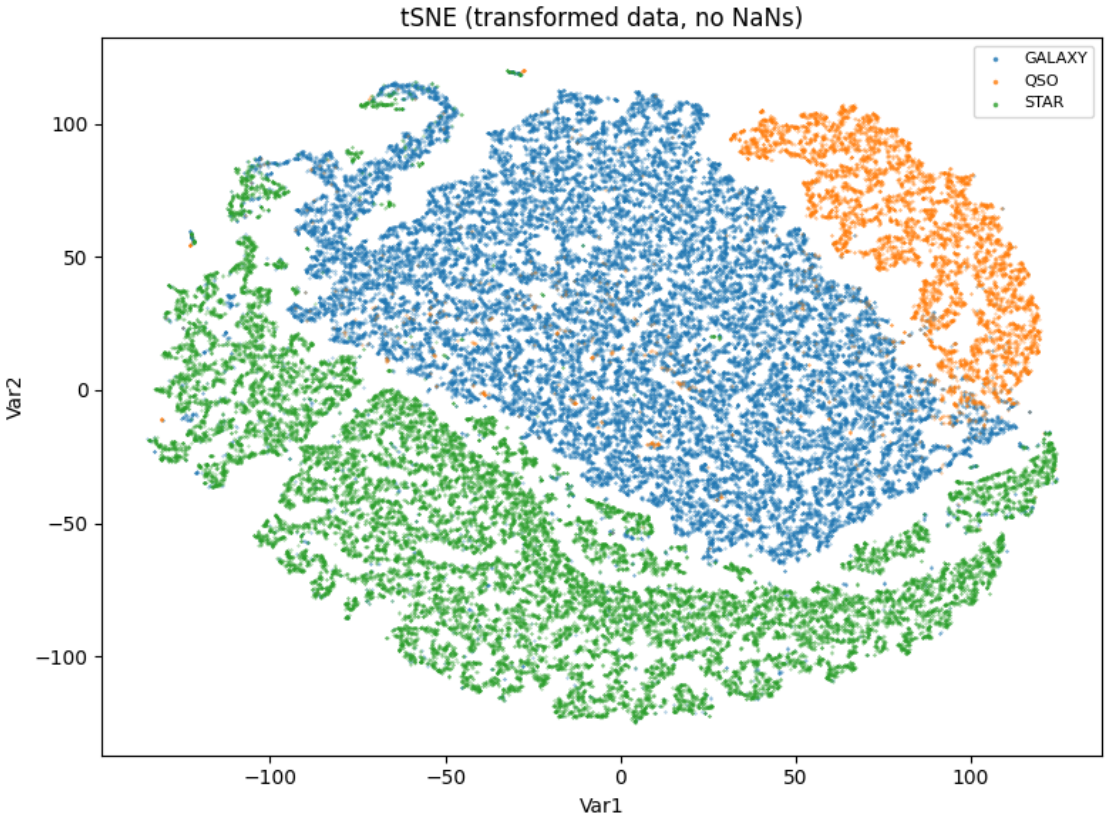
[58]:
```
# - Init tSNE: play with n_neighbors & min_dist parameters
tsne_dimred= TSNE(
        n_components=2,
        perplexity=30,
        random_state=42,
        init='pca', # {'pca','random'}
        method='barnes_hut',  # {'barnes_hut','exact'}
        max_iter=1000, # n_iter in old API
        learning_rate='auto'
)
```

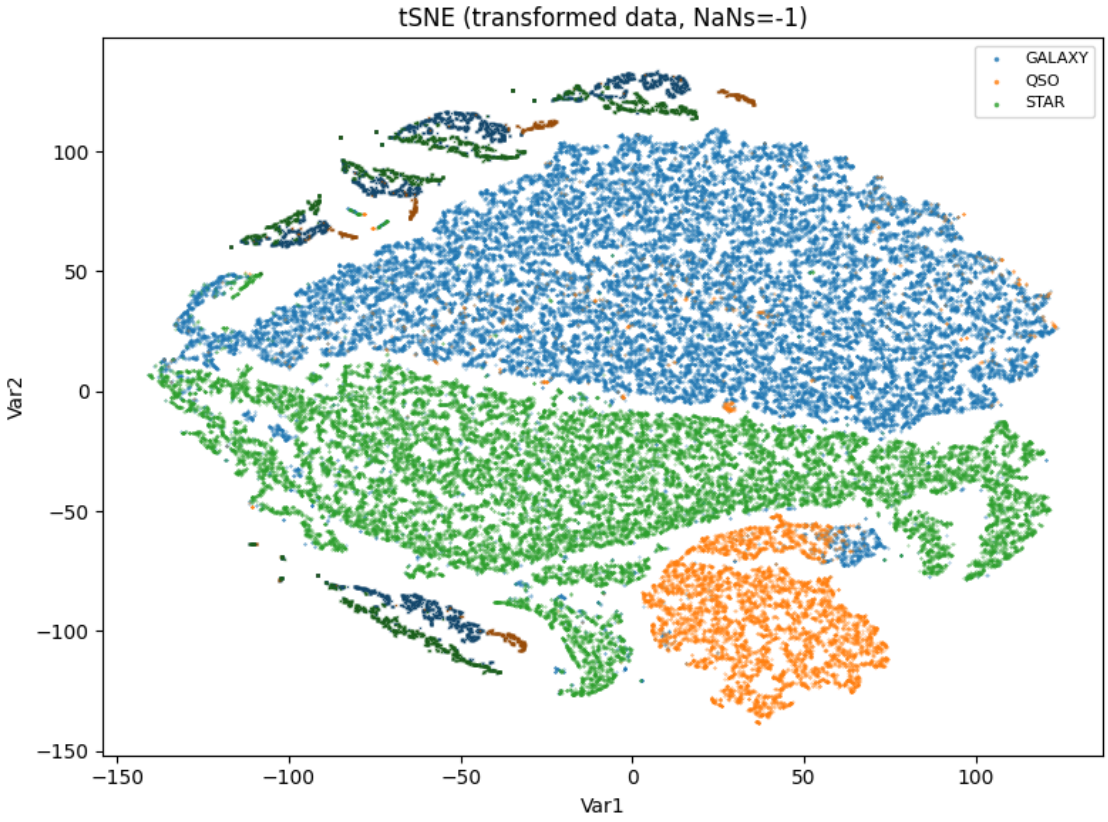
[59]:
```
# - Run tSNE on original & transformed data
print("Run tSNE on original data (no nans) ...")
X_nonans_embed= tsne_dimred.fit_transform(X_nonans)

print("Run tSNE on transformed data (no nans) ...")
X_transf_nonans_embed= tsne_dimred.fit_transform(X_transf_nonans)
```

```
print("Run tSNE on transformed data (nans=-1) ...")
X_transf_embed= tsne_dimred.fit_transform(X_transf)

# - Plot tSNE embeddings on original & transformed data
plot_embeddings(X_nonans_embed, y_nonans, mask=None, title="tSNE (original␣
 ↪data, no NaNs)")
plot_embeddings(X_transf_nonans_embed, y_transf_nonans, mask=None, title="tSNE␣
 ↪(transformed data, no NaNs)")
plot_embeddings(X_transf_embed, y_transf, mask=has_nan_values, title="tSNE␣
 ↪(transformed data, NaNs=-1)")
```

```
Run tSNE on original data (no nans) …
Run tSNE on transformed data (no nans) …
Run tSNE on transformed data (nans=-1) …
```




tSNE (transformed data, no NaNs)
GALAXY
QSO
STAR
100
50
0
−50
−100
Var2
−100
−50
0
50
100
Var1



6 Data cleaning

We are now going to demonstrate a possible data cleaning pipeline applied on raw data:

- Step 1: Extreme outlier identification with simple method (IQR)
- Step 2: Missing data imputation with `IterativeImputer` method

- Step 3: Refined outlier identification with more advanced methods (LOF/IF)
- Step 4: Refined missing data imputation with `IterativeImputer` method

6.1 Step 1: Extreme outlier identification

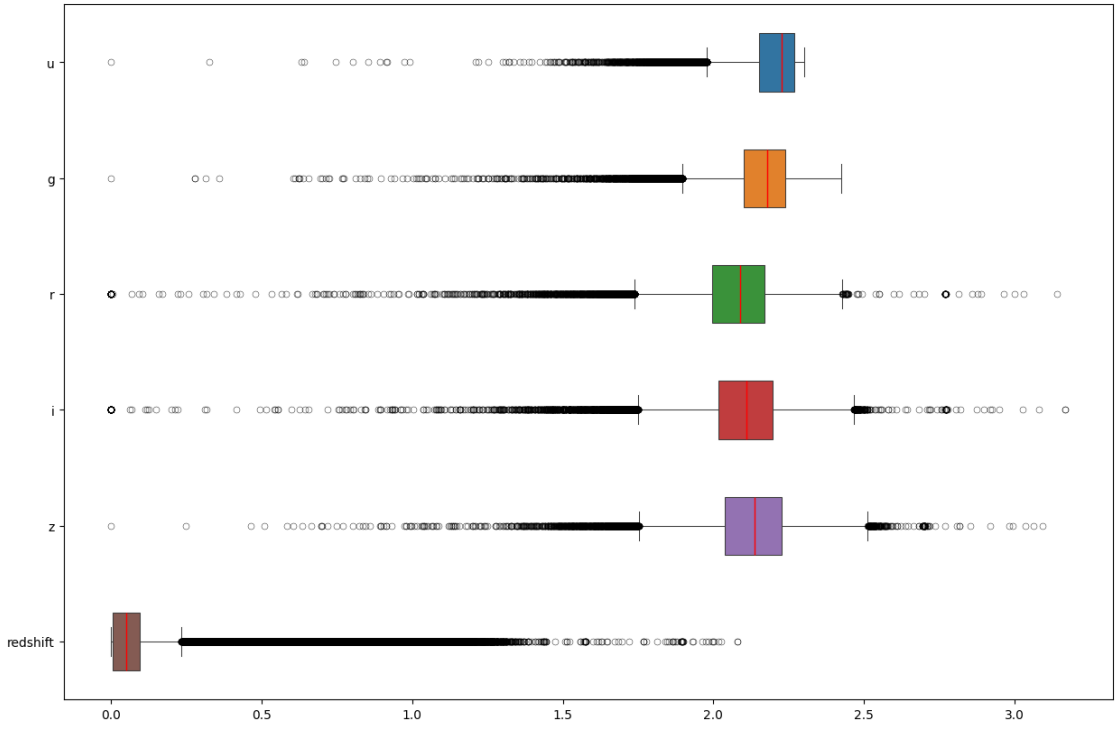
Let's search for possible outliers in each columns. We will first use simple methods (IQR-based) to identify the most extreme outliers, and later on (Step 3) we will report to more advanced methods (e.g. LOF, IsolationForest) to refine the outlier search.

First, let's first make a box plot of each numeric data variable.

[43]:
```
df_transf_copy= df_transf.copy()
df_transf_copy.replace(to_replace=-1, value=np.nan, inplace=True)

# - Define outlier properties
```

```
flierprops = dict(marker='o', markerfacecolor='None', markersize=5,␣
 ↪markeredgecolor='black',
                  markeredgewidth=0.3)
medianprops={"color": "r", "linewidth": 1}

# - Box plot
plt.figure(figsize=(15, 10))
sns.boxplot(data=df_transf_copy[cols],
            linewidth=.75, width=.5,
            orient="h", showmeans=False, medianprops=medianprops,
            flierprops=flierprops, whis=1.5)
```

[43]: <Axes: >



6.1.1 Interquartile Range method

This is perhaps the most simple and robust method for finding outliers, suitable for non-normal data distributions. The method identifies data points falling 1.5 times the IQR below the first quartile (Q1) or above the third quartile (Q3). It thus only requires sorting data to find quartiles (Q1, Q3), and calculating IQR= Q3-Q1. We will use a less stringent outlier range, e.g. setting the bounds to:

lower= Q1 - 3.0 x IQR
upper= Q3 + 3.0 x IQR

Let's therefore compute the Q1 & Q3 quartiles of the data. We are using the log-transformed data.

[44]:
```
Q1 = df_transf_copy[cols].quantile(0.25)
Q3 = df_transf_copy[cols].quantile(0.75)
IQR = Q3 - Q1
lower = Q1 - 3.0 * IQR
upper = Q3 + 3.0 * IQR

print("Q1")
print(Q1)
print("Q3")
print(Q3)
print("IQR")
print(IQR)
print("lower_bound")
print(lower)
print("upper_bound")
print(upper)
```

```
Q1
u           2.152774
g           2.102400
r           1.997758
i           2.018685
z           2.037990
redshift    0.004131
Name: 0.25, dtype: float64
Q3
u           2.268140
g           2.239237
r           2.170243
i           2.197854
z           2.228120
redshift    0.095149
Name: 0.75, dtype: float64
IQR
u           0.115365
g           0.136837
r           0.172484
i           0.179169
z           0.190129
redshift    0.091018
dtype: float64
lower_bound
u           1.806679
g           1.691890
r           1.480305
i           1.481178
```

```
z           1.467603
redshift   -0.268924
dtype: float64
upper_bound
u           2.614235
g           2.649748
r           2.687696
i           2.735362
z           2.798507
redshift    0.368204
dtype: float64
```

We can either remove the outliers (entire row) from the data and create a new data frame with outliers removed **OR** we can set them to NaN values, keep them and impute them in the following pipeline steps.

The following code completely removes the outliers from the dataset.

[45]:
```
#mask = df_transf_copy[cols].ge(lower).all(axis=1) & df_transf_copy[cols].
 ↪le(upper).all(axis=1)
#df_clean_step1 = df_transf_copy.loc[mask].copy()

in_range = (
  df_transf_copy[cols].ge(lower) &
  df_transf_copy[cols].le(upper)
)

# Treat NaNs as valid (ignore them)
mask = (in_range | df_transf_copy[cols].isna()).all(axis=1)
df_clean_step1 = df_transf_copy.loc[mask].copy()

print(df_clean_step1)

n_before = len(df_transf_copy)
n_after = mask.sum()

print(f"Rows before: {n_before}")
print(f"Rows after: {n_after}")
print(f"Removed rows: {n_before - n_after}")

# - Counts removed rows (ignoring NaNs)
in_range = df_transf_copy[cols].ge(lower) & df_transf_copy[cols].le(upper)
mask_keep = in_range | df_transf_copy[cols].isna()
mask_keep = mask_keep.all(axis=1)
removed_rows = (~mask_keep).sum()
print("Removed rows (ignoring NaNs):", removed_rows)

# - Count how many columns have more than 1 outlier
```

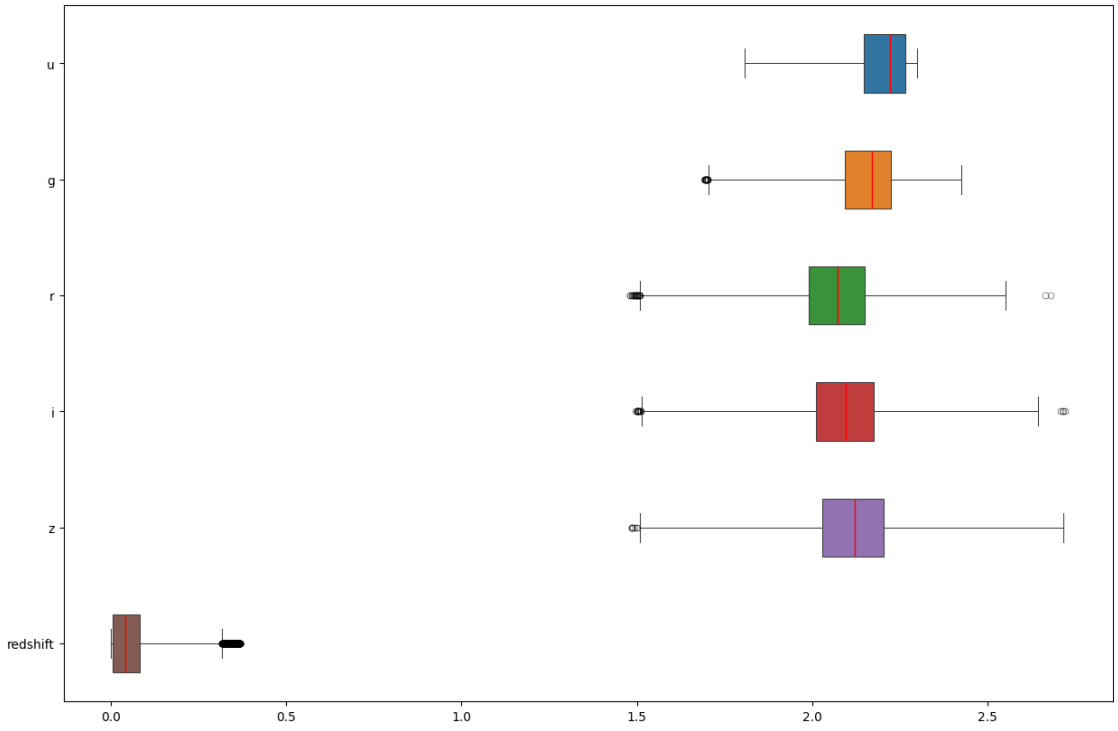
```
outlier_cells = df_transf_copy[cols].lt(lower) | df_transf_copy[cols].gt(upper)
rows_outlier_counts = outlier_cells.sum(axis=1)
n_2plus = (rows_outlier_counts >= 2).sum()
print("Rows with >=2 outlier columns:", n_2plus)

print("Outlier counts")
outlier_counts = outlier_cells.sum(axis=1)
print(outlier_counts.value_counts().sort_index())

for X in range(1, 6):
  n = (outlier_counts > X).sum()
  print(f"Rows with > {X} outliers: {n}")
```

```
              u         g         r         i         z  redshift   class
0           NaN  2.171097  2.084649  2.095145  2.121302  0.144278  GALAXY
1      2.191312  2.066912  1.913392  1.922731  1.930075  0.099123  GALAXY
3      2.202160  2.176309  2.109996  2.147579  2.184472  0.004008    STAR
4      2.127816  2.099761  2.030986  2.076653  2.117667  0.004397    STAR
5      2.256286  2.253554  2.201341  2.237581  2.273587  0.003639    STAR
…           …         …         …         …         …         …       …
101994 2.175660  2.179642  2.124757  2.168944  2.206190  0.005064    STAR
101995 2.272589  2.229333  2.150194  2.183171  2.210361  0.004336    STAR
101996 2.281834  2.248874  2.179048  2.189386  2.209752  0.072824  GALAXY
101998      NaN  1.759983  1.534919  1.506574  1.502123  0.028155  GALAXY
101999 2.221428  2.216486  2.158917  2.195798  2.229542  0.003502    STAR

[90288 rows x 7 columns]
Rows before: 100000
Rows after: 90288
Removed rows: 9712
Removed rows (ignoring NaNs): 9712
Rows with >=2 outlier columns: 459
Outlier counts
0    90288
1     9253
2      132
3       79
4      122
5      122
6        4
Name: count, dtype: int64
Rows with > 1 outliers: 459
Rows with > 2 outliers: 327
Rows with > 3 outliers: 248
Rows with > 4 outliers: 126
Rows with > 5 outliers: 4
```

Below, we make a new box plot on updated data (outliers removed with IQR method).

[46]:
```
# - Make box plot with cleaned data
#   NB: setting whisker attribute to 3.0, default=1.5) to check we effectively␣
 ↪removed the outliers
plt.figure(figsize=(15, 10))
sns.boxplot(data=df_clean_step1[cols],
            linewidth=.75, width=.5,
            orient="h", showmeans=False, medianprops=medianprops,
            flierprops=flierprops, whis=3.0)
```

[46]: <Axes: >



The following code replaces the outliers found the dataset with NaN values.

[47]:
```
df_clean_step1 = df_transf_copy.copy()

# - Create boolean mask of outliers (cell-wise)
outlier_mask = (
  df_clean_step1[cols].lt(lower) |
  df_clean_step1[cols].gt(upper)
)

# - Replace outliers with NaN
df_clean_step1.loc[:, cols] = df_clean_step1[cols].mask(outlier_mask, np.nan)
n_outliers = outlier_mask.sum().sum()
```

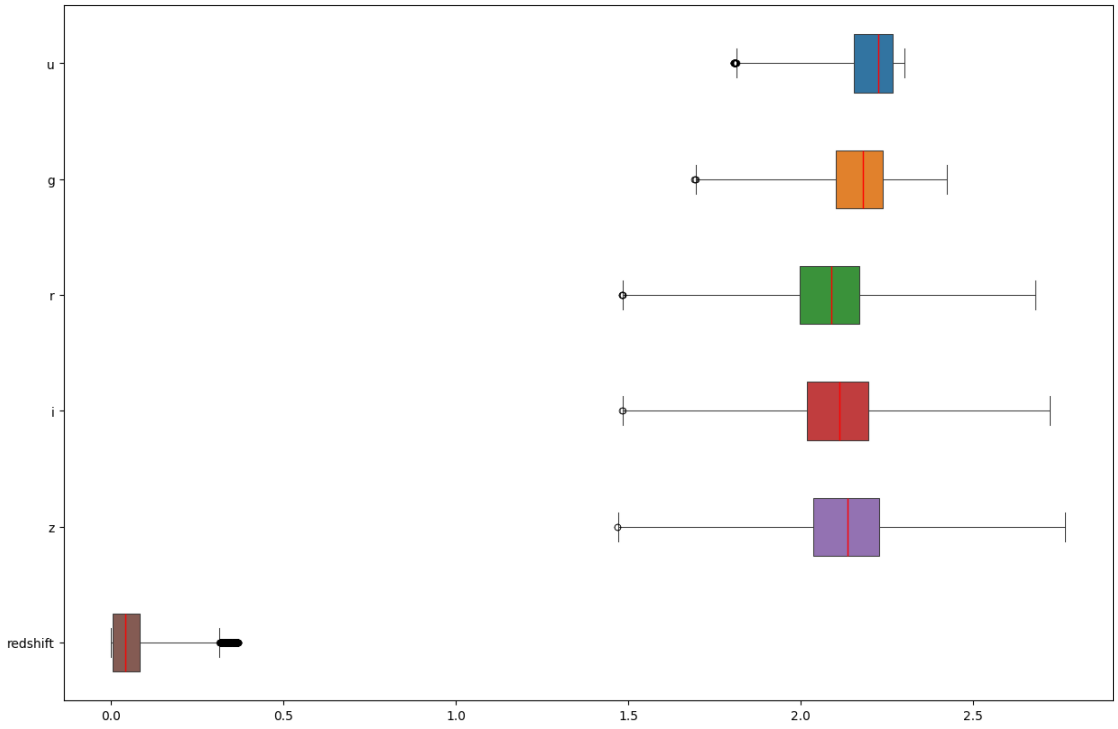
```
print(f"Replaced {n_outliers} values with NaN")
```

```
Replaced 10876 values with NaN
```

Let's make a box plot of the cleaned data.

[48]:
```
# - Make box plot with cleaned data
#   NB: setting whisker attribute to 3.0, default=1.5) to check we effectively␣
 ↪removed the outliers
plt.figure(figsize=(15, 10))
sns.boxplot(data=df_clean_step1[cols],
            linewidth=.75, width=.5,
            orient="h", showmeans=False, medianprops=medianprops,
            flierprops=flierprops, whis=3.0)
```

[48]: <Axes: >



6.2 Step 2: Handling of missing data

In this stage, we will perform missing data imputation on the cleaned data obtained in the previous stage (no extreme outliers).

We will use the multivariate imputation method implemented in `sklearn IterativeImputer`. It supports multiple imputation with different estimators (e.g. regression, kNN, random forest, etc). Below, we define an helper method to run it, eventually performing multiple rounds of imputations.

[49]:
```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.linear_model import BayesianRidge

# - Define an helper function to run imputation
def get_imputed_data(X, stochastic=False, seed=0):
  """ Run iterative imputation and returns data with imputed values """
  # - Define imputer class
  #   An iteration is a single imputation of each feature with missing values
  imputer= IterativeImputer(
    estimator=BayesianRidge(),    # Default, but you can use other methods here␣
 ↪(e.g. RandomForest, etc)
    n_nearest_features=None,      # All features used
    imputation_order='ascending', # Impute from features with fewest to most␣
 ↪missing values
    initial_strategy='median',    # Initialization imputation method
    max_iter=10,                  # Max number of iterations
    sample_posterior=stochastic,  # False to disable stochastic imputation
    random_state=seed
  )

  # - Fit imputer and get the dataset with imputed data
  X_imp= imputer.fit_transform(X)
  return X_imp
```

For this tutorial, we will run the imputation method only once with random drawn disabled (single imputation).

[50]:
```
# - Set data to be used for imputation (the numeric columns)
sel_cols= df_clean_step1.select_dtypes(include=[np.number]).columns
X = df_clean_step1[sel_cols].copy()
y = df_clean_step1["class"].copy()
print(X)

# - Impute data
X_imputed= get_imputed_data(X, seed=42, stochastic=False)

# - Set final output cleaned data
df_clean_step2 = pd.DataFrame(
  X_imputed,
  columns=sel_cols,
  index=X.index
)
df_clean_step2["class"] = y
print(df_clean_step2)
```

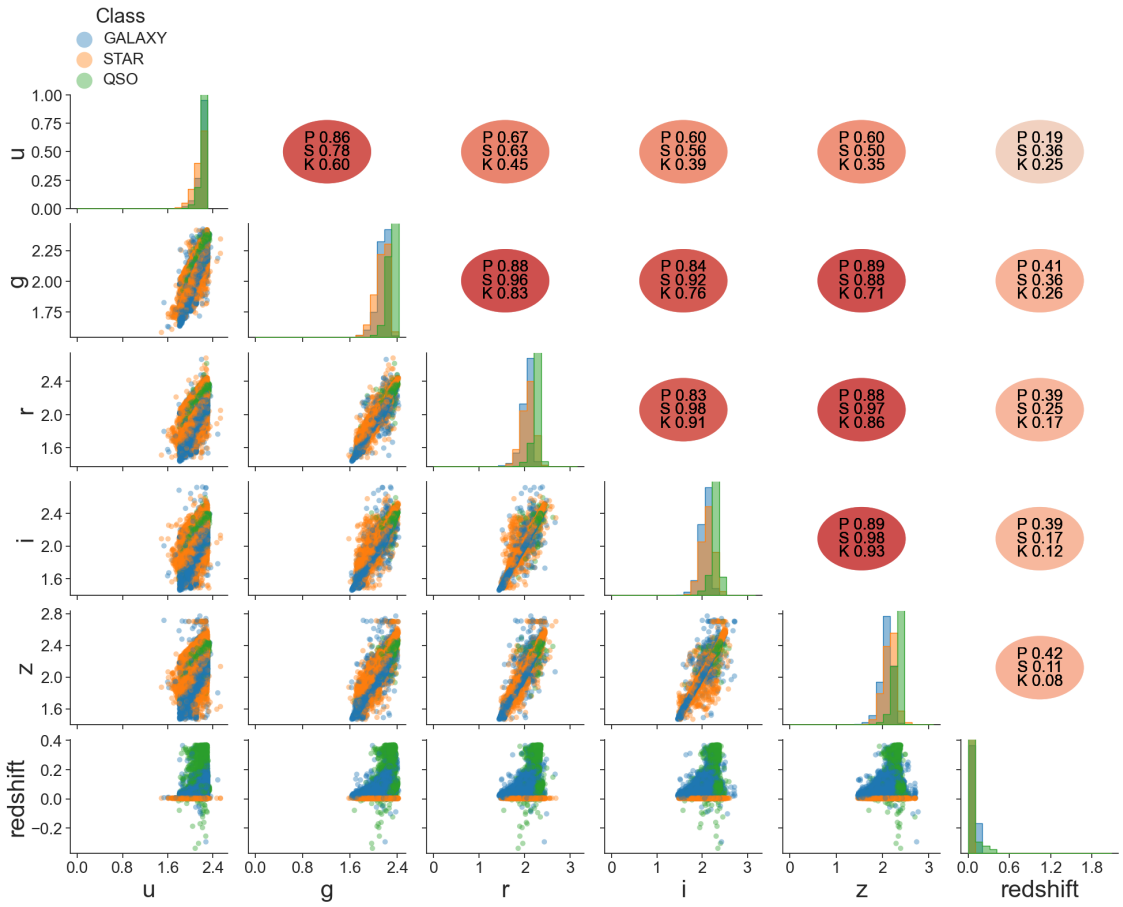
```
              u         g         r         i         z  redshift
0           NaN  2.171097  2.084649  2.095145  2.121302  0.144278
```

```
1       2.191312  2.066912  1.913392  1.922731  1.930075  0.099123
2            NaN  1.929527  1.972835  2.081477  2.164165  0.004158
3       2.202160  2.176309  2.109996  2.147579  2.184472  0.004008
4       2.127816  2.099761  2.030986  2.076653  2.117667  0.004397
…            …         …         …         …         …         …
101995  2.272589  2.229333  2.150194  2.183171  2.210361  0.004336
101996  2.281834  2.248874  2.179048  2.189386  2.209752  0.072824
101997  2.220891  2.277972  2.275484  2.306781  2.350632       NaN
101998       NaN  1.759983  1.534919  1.506574  1.502123  0.028155
101999  2.221428  2.216486  2.158917  2.195798  2.229542  0.003502

[100000 rows x 6 columns]
                u         g         r         i         z  redshift   class
0       2.188547  2.171097  2.084649  2.095145  2.121302  0.144278  GALAXY
1       2.191312  2.066912  1.913392  1.922731  1.930075  0.099123  GALAXY
2       1.836542  1.929527  1.972835  2.081477  2.164165  0.004158    STAR
3       2.202160  2.176309  2.109996  2.147579  2.184472  0.004008    STAR
4       2.127816  2.099761  2.030986  2.076653  2.117667  0.004397    STAR
…            …         …         …         …         …         …       …
101995  2.272589  2.229333  2.150194  2.183171  2.210361  0.004336    STAR
101996  2.281834  2.248874  2.179048  2.189386  2.209752  0.072824  GALAXY
101997  2.220891  2.277972  2.275484  2.306781  2.350632  0.048307     QSO
101998  2.003364  1.759983  1.534919  1.506574  1.502123  0.028155  GALAXY
101999  2.221428  2.216486  2.158917  2.195798  2.229542  0.003502    STAR

[100000 rows x 7 columns]
```

```
/home/riggi/Software/venvs/usc-c-ai-school-2026/lib/python3.10/site-
packages/sklearn/impute/_iterative.py:895: ConvergenceWarning:
[IterativeImputer] Early stopping criterion not reached.
  warnings.warn(
```

Let's view a scatter plot of the imputed data.

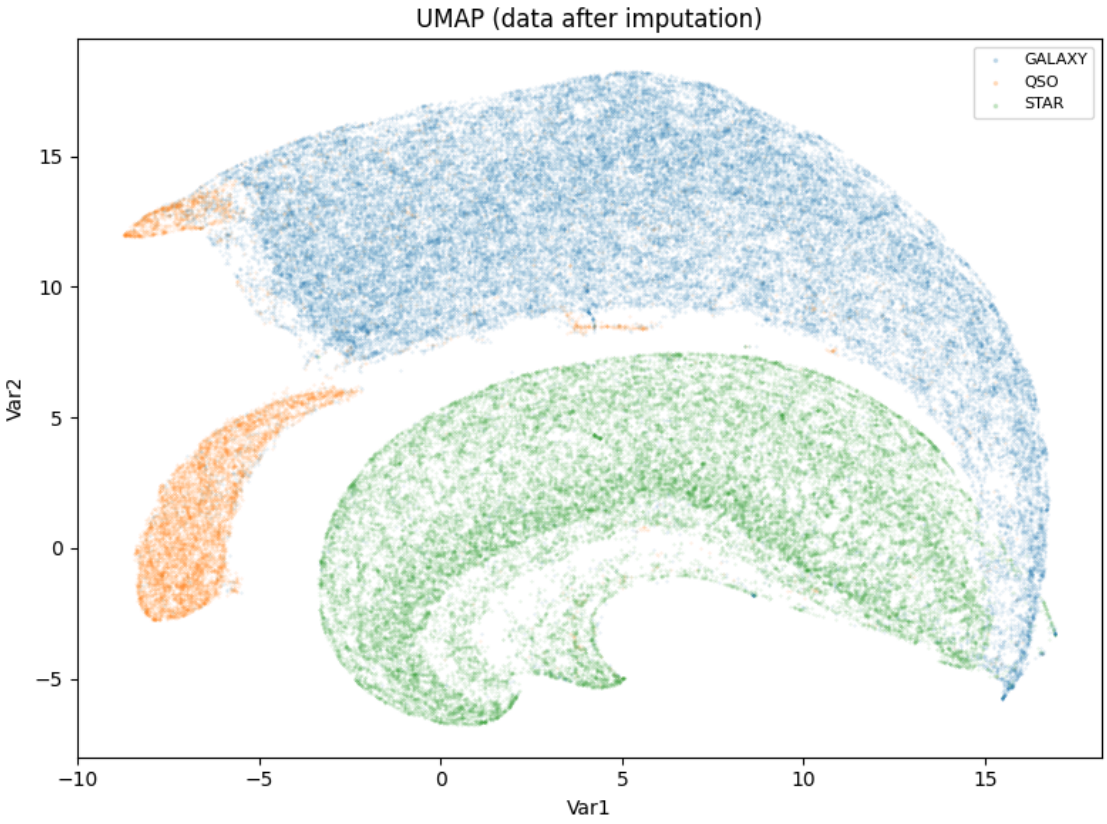
[52]:
```
# - Draw scatter plot
draw_scatter_plot(df_clean_step2, offdiag_hist=False)
```



Let's make a UMAP 2D plot of the final cleaned data.

[60]:
```
# - Set data for UMAP projection
X_clean_step2= df_clean_step2[cols]
y_clean_step2= df_clean_step2["class"]

# - Scale data before UMAP
scaler = RobustScaler()
X_scaled = scaler.fit_transform(X_clean_step2)

# - Run UMAP & plot embeddings
X_clean_step2_embed= umap_dimred.fit_transform(X_scaled)

plot_embeddings(
  X_clean_step2_embed,
  y_clean_step2,
  mask=None,
  title="UMAP (data after imputation)",
  s_normal=0.1,
```

```
  s_masked=0.1,
  alpha_normal=0.2,
  alpha_masked=0.2
)
```



6.3 Step 3: Refined search of outliers

So far, we have removed outliers from the data using the simple IQR range method. This allowed us to obtain a preliminary cleaned dataset, that still had missing values. We imputed missing values using different methods, obtaining a preliminary cleaned dataset.

We now revisit outlier detection, applying advanced outlier detection algorithms, like LOF or IsolationForest. They both require a fully observed dataset (e.g. no NaNs), so we run them on the preliminary cleaned dataset.

6.3.1 Local Outlier Factor method

Local Outlier Factor (LOF) is an unsupervised anomaly detection method, considering outliers as the observations having a substantially lower density than their neighbors. It compares the local density of a point to local density of its k-nearest neighbors and gives a score as final output.

The algorithm is based on the following steps:

1) Compute the local density deviation of a given data point with respect to its neighbors, e.g. find k-nearest-neighbors N_k for each data point x;
2) Compute the local density for a data point using the local reachability density (LRD) (inverse of the average reachability distance (RD) of its neighbors);
3) Compute the LOF score by comparing the LRD of a point with the LRD's of its k-neighbors.

The returned LOF score captures the "degree of abnormality", as the average of the ratio of the local reachability density of a sample and those of its k-nearest neighbors. **Inliers tend to have a LOF score close to 1, while outliers tend to have a larger LOF score.**

The main algorithm parameters to be set by the user are:

- Number of neighbors `NN`, typically greater than the minimum number of observations a cluster has to contain, and smaller than the maximum number of close by observations that can potentially be local outliers.
- Contamination `C`, i.e. the proportion of outliers in the data set

Let's apply the method on our log-transformed data.
NB: sklearn returned LOF score is the opposite of original LOF score algorithm

[74]:
```
from sklearn.preprocessing import RobustScaler
from sklearn.neighbors import LocalOutlierFactor

# - Set data to be used for outlier search (the numeric columns)
X = df_clean_step2[sel_cols].copy()
y = df_clean_step2["class"].copy()
print(X)

# - Scale features using previous scaler
X_scaled = scaler.transform(X)
print(X_scaled.shape)

# - Init LOF algorithm
#   NB: Experiment with parameters: n_neighbors & contamination
lof = LocalOutlierFactor(
  n_neighbors=20,
  contamination=0.005,
  novelty=False
)

# - Search outliers
y_pred = lof.fit_predict(X_scaled) # Returns -1 for outliers and +1 for inliers
inlier_mask = (y_pred == 1)
outlier_mask = (y_pred == -1)
threshold= lof.offset_ # Returns the threshold used to mark outliers␣
 ↪(<threshold=outliers)

# - Obtain scores (more negative => more abnormal)
scores = lof.negative_outlier_factor_
```

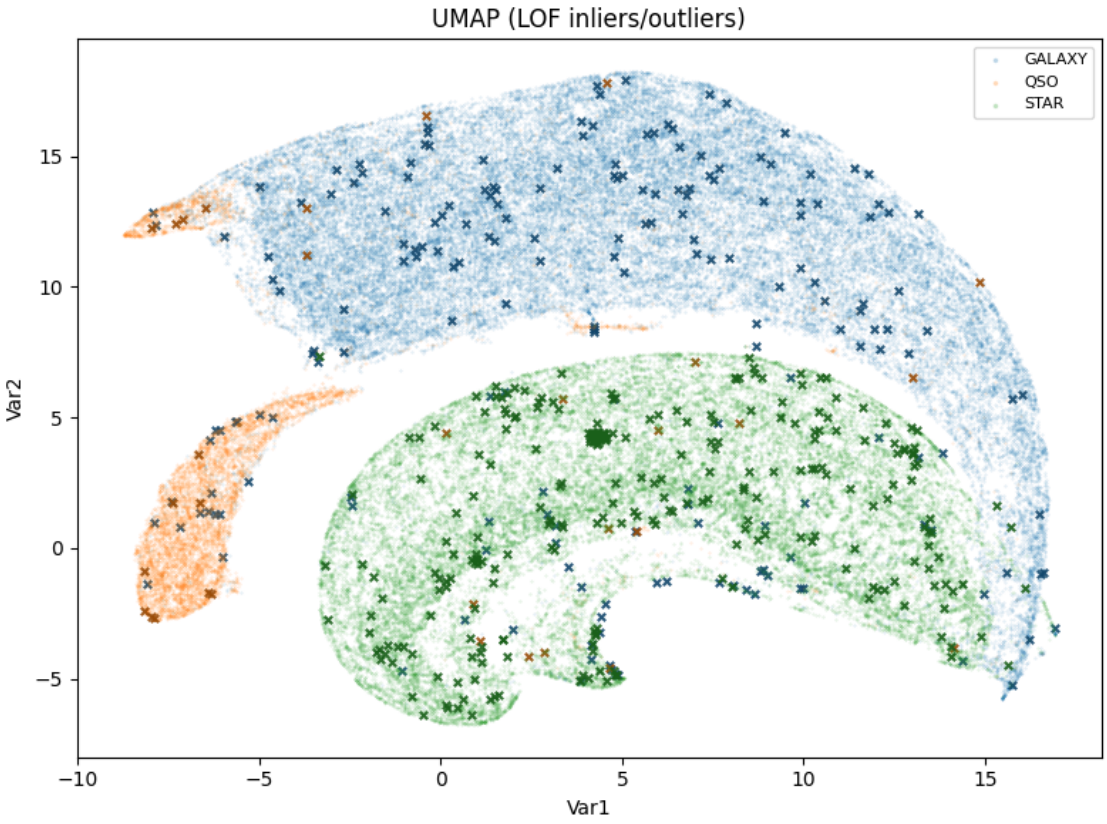
```
# - Create new data frame with flags/scores (useful for plotting & debugging)
df_lof = df_clean_step2.copy()
df_lof["lof_is_outlier"] = outlier_mask.astype(int)
df_lof["lof_score"] = scores

# - Filtered dataframe (Step 3 result)
df_clean_step3_lof = df_lof.loc[inlier_mask].copy()

print("Rows before LOF:", len(df_clean_step2))
print("Rows after LOF:", len(df_clean_step3_lof))
print("Removed by LOF:", (~inlier_mask).sum())
```

```
                u         g         r         i         z  redshift
0        2.188547  2.171097  2.084649  2.095145  2.121302  0.144278
1        2.191312  2.066912  1.913392  1.922731  1.930075  0.099123
2        1.836542  1.929527  1.972835  2.081477  2.164165  0.004158
3        2.202160  2.176309  2.109996  2.147579  2.184472  0.004008
4        2.127816  2.099761  2.030986  2.076653  2.117667  0.004397
…             …         …         …         …         …         …
101995   2.272589  2.229333  2.150194  2.183171  2.210361  0.004336
101996   2.281834  2.248874  2.179048  2.189386  2.209752  0.072824
101997   2.220891  2.277972  2.275484  2.306781  2.350632  0.048307
101998   2.003364  1.759983  1.534919  1.506574  1.502123  0.028155
101999   2.221428  2.216486  2.158917  2.195798  2.229542  0.003502

[100000 rows x 6 columns]
(100000, 6)
Rows before LOF: 100000
Rows after LOF: 99500
Removed by LOF: 500
```

Let's make a UMAP 2D plot to view inliers & outliers.

[75]:
```
X_clean_step3= df_lof[cols]
X_scaled = scaler.transform(X_clean_step3)
y_clean_step3= df_lof["class"]
outlier_mask = df_lof["lof_is_outlier"].values
#X_clean_step3_embed= umap_dimred.fit_transform(X_scaled)
X_clean_step3_embed= umap_dimred.transform(X_scaled)
plot_embeddings(
  X_clean_step3_embed,
  y_clean_step3,
  mask=outlier_mask,
  title="UMAP (LOF inliers/outliers)",
  s_normal=0.1,
  s_masked=15,
  alpha_normal=0.2,
```

```
  alpha_masked=0.9
)
```



6.3.2 Isolation Forest method

Isolation Forest (IF) is an unsupervised anomaly detection method that isolates data points through a process of random partitioning, creating a forest of decision trees. Outliers, being different and fewer in number, tend to be isolated with fewer splits compared to normal data points.

The method returns a score for each observation, capturing the "degree of abnormality". **Inliers have a score <<0.5, while outliers have a score ~1**.

The main algorithm parameters to be set by the user are:

- `n_estimators`: number of trees in the ensemble
- `max_samples`: number of samples to draw from the dataset to train each tree
- `max_features`: number of features to draw to train each tree
- `contamination`: proportion of outliers in the data set

Let's apply the method on our log-transformed data.
NB: sklearn returned IF score is the opposite of original IF score algorithm

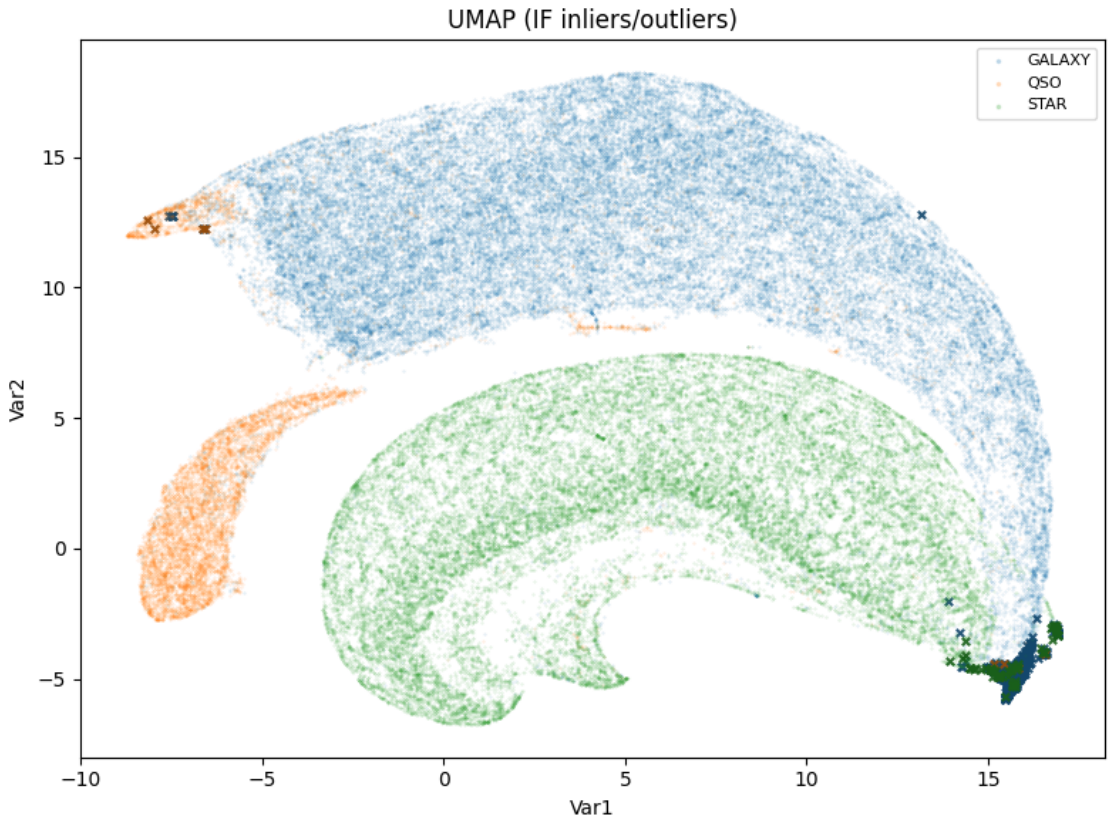
[76]:
```
from sklearn.preprocessing import RobustScaler
from sklearn.ensemble import IsolationForest

# - Set data to be used for outlier search (the numeric columns)
X = df_clean_step2[sel_cols].copy()
y = df_clean_step2["class"].copy()
print(X)

# - Scale features using previous scaler
X_scaled = scaler.transform(X)
print(X_scaled.shape)

# - Init LOF algorithm
#   NB: Experiment with parameters: n_neighbors & contamination
IF= IsolationForest(
  n_estimators=100,   # number of trees
  contamination=0.005,
  max_features=1.0,   # 1.0 = all features
  max_samples="auto", # auto: min(256, n_samples)
  random_state=0
)

# - Search outliers
y_pred= IF.fit_predict(X_scaled) # Returns -1 for outliers and +1 for inliers
inlier_mask = (y_pred == 1)
outlier_mask = (y_pred == -1)

# - Obtain outlier scores
#   NB: score_samples returns the opposite of the anomaly score
#       defined in the original paper. The lower, the more abnormal.
#       Outliers (negative scores), Inliers (positive scores)
scores = IF.score_samples(X_scaled)

# - Create new data frame with flags/scores (useful for plotting & debugging)
df_if = df_clean_step2.copy()
df_if["if_is_outlier"] = outlier_mask.astype(int)
df_if["if_score"] = scores

# - Filtered dataframe (Step 3 result)
df_clean_step3_IF = df_if.loc[inlier_mask].copy()

print("Rows before IF:", len(df_clean_step2))
print("Rows after IF:", len(df_clean_step3_IF))
print("Removed by IF:", (~inlier_mask).sum())
```

```
             u         g         r         i         z  redshift
0     2.188547  2.171097  2.084649  2.095145  2.121302  0.144278
```

```
1       2.191312  2.066912  1.913392  1.922731  1.930075  0.099123
2       1.836542  1.929527  1.972835  2.081477  2.164165  0.004158
3       2.202160  2.176309  2.109996  2.147579  2.184472  0.004008
4       2.127816  2.099761  2.030986  2.076653  2.117667  0.004397
…            …         …         …         …         …         …
101995  2.272589  2.229333  2.150194  2.183171  2.210361  0.004336
101996  2.281834  2.248874  2.179048  2.189386  2.209752  0.072824
101997  2.220891  2.277972  2.275484  2.306781  2.350632  0.048307
101998  2.003364  1.759983  1.534919  1.506574  1.502123  0.028155
101999  2.221428  2.216486  2.158917  2.195798  2.229542  0.003502

[100000 rows x 6 columns]
(100000, 6)
Rows before IF: 100000
Rows after IF: 99500
Removed by IF: 500
```

Let's make a UMAP 2D plot to view inliers & outliers.

[77]:
```
X_clean_step3= df_if[cols]
X_scaled = scaler.transform(X_clean_step3)
y_clean_step3= df_if["class"]
outlier_mask = df_if["if_is_outlier"].values
#X_clean_step3_embed= umap_dimred.fit_transform(X_scaled)
X_clean_step3_embed= umap_dimred.transform(X_scaled)
plot_embeddings(
  X_clean_step3_embed,
  y_clean_step3,
  mask=outlier_mask,
  title="UMAP (IF inliers/outliers)",
  s_normal=0.1,
  s_masked=15,
  alpha_normal=0.2,
  alpha_masked=0.9
)
```



6.4 Step 4: Refined missing data imputation

In this final stage, we revisit missing data imputation, refining it after we removed the outliers in Step 3.

We need to restart from the cleaned data output produced in Step 1, and remove the outliers of Step 3.

[78]:
```
# - Get the index of inliers and restore dataset of step 1 (with NaNs) but␣
  ↪without outliers found in step 3
#   Select which outliers to remove: LOF, IF
#inlier_index = df_lof.index[df_lof["lof_is_outlier"] == 0]   # LOF outliers
inlier_index = df_if.index[df_if["if_is_outlier"] == 0]  # IF outliers

df_step4 = df_clean_step1.loc[inlier_index].copy()
print(df_step4)
```

	u	g	r	i	z	redshift	class
0	NaN	2.171097	2.084649	2.095145	2.121302	0.144278	GALAXY
1	2.191312	2.066912	1.913392	1.922731	1.930075	0.099123	GALAXY
2	NaN	1.929527	1.972835	2.081477	2.164165	0.004158	STAR
3	2.202160	2.176309	2.109996	2.147579	2.184472	0.004008	STAR

```
4       2.127816  2.099761  2.030986  2.076653  2.117667  0.004397    STAR
…          …         …         …         …         …         …      …
101994  2.175660  2.179642  2.124757  2.168944  2.206190  0.005064    STAR
101995  2.272589  2.229333  2.150194  2.183171  2.210361  0.004336    STAR
101996  2.281834  2.248874  2.179048  2.189386  2.209752  0.072824  GALAXY
101997  2.220891  2.277972  2.275484  2.306781  2.350632       NaN     QSO
101999  2.221428  2.216486  2.158917  2.195798  2.229542  0.003502    STAR

[99500 rows x 7 columns]
```

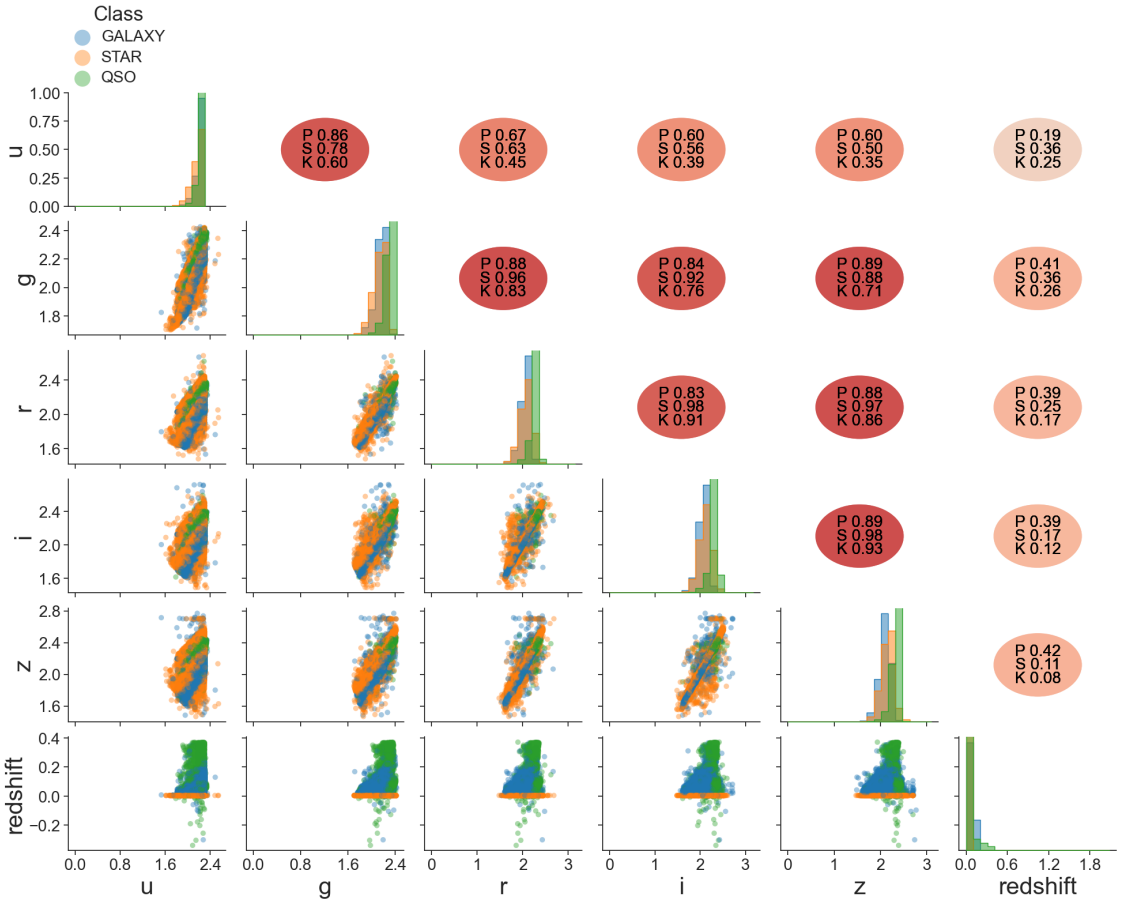
We now perform imputation.

[79]:
```
# - Set input data for the imputation
X = df_step4[sel_cols].copy()
y = df_step4["class"].copy()

# - Impute data
X_imputed= get_imputed_data(X, seed=42, stochastic=False)

# - Set final output cleaned data
df_clean_step4 = pd.DataFrame(
  X_imputed,
  columns=sel_cols,
  index=X.index
)
df_clean_step4["class"] = y
print(df_clean_step4)
```

```
                u         g         r         i         z  redshift   class
0        2.188531  2.171097  2.084649  2.095145  2.121302  0.144278  GALAXY
1        2.191312  2.066912  1.913392  1.922731  1.930075  0.099123  GALAXY
2        1.836899  1.929527  1.972835  2.081477  2.164165  0.004158    STAR
3        2.202160  2.176309  2.109996  2.147579  2.184472  0.004008    STAR
4        2.127816  2.099761  2.030986  2.076653  2.117667  0.004397    STAR
…           …         …         …         …         …         …      …
101994   2.175660  2.179642  2.124757  2.168944  2.206190  0.005064    STAR
101995   2.272589  2.229333  2.150194  2.183171  2.210361  0.004336    STAR
101996   2.281834  2.248874  2.179048  2.189386  2.209752  0.072824  GALAXY
101997   2.220891  2.277972  2.275484  2.306781  2.350632  0.047928     QSO
101999   2.221428  2.216486  2.158917  2.195798  2.229542  0.003502    STAR

[99500 rows x 7 columns]

/home/riggi/Software/venvs/usc-c-ai-school-2026/lib/python3.10/site-
packages/sklearn/impute/_iterative.py:895: ConvergenceWarning:
[IterativeImputer] Early stopping criterion not reached.
  warnings.warn(
```

Let's view a scatter plot of the final cleaned data.

[80]:
```
# - Draw scatter plot
draw_scatter_plot(df_clean_step4, offdiag_hist=False)
```



Let's make a UMAP 2D plot of the final cleaned data.

[81]:
```
# - Set data for UMAP
X_clean_step4= df_clean_step4[cols]
y_clean_step4= df_clean_step4["class"]

# - Scale data before UMAP (new scaler as we removed outlier data)
scaler = RobustScaler()
X_scaled = scaler.fit_transform(X_clean_step4)

# - Run UMAP & plot embeddings
X_clean_step4_embed= umap_dimred.fit_transform(X_scaled)

plot_embeddings(
  X_clean_step4_embed,
  y_clean_step4,
```

```
  mask=None,
  title="UMAP (cleaned data)",
  s_normal=0.1,
  s_masked=0.1,
  alpha_normal=0.2,
  alpha_masked=0.2
)
```