dc1_check

March 26, 2021

1 RAIL Evaluation - Check results against DC1 paper

Contact: *Julia Gschwend* (julia@linea.gov.br), *Sam Schmidt, Alex Malz, Eric Charles*

The purpose of this notebook is to validate the new implementation of the DC1 metrics, previously available on Github repository PZDC1paper, now refactored to be part of RAIL Evaluation module. The metrics here were implemented in object-oriented Python 3, inheriting features from *qp*. In this notebook we use the same input dataset used in DC1 PZ paper (Schmidt et al. 2020), copied from cori (/global/cfs/cdirs/lsst/groups/PZ/PhotoZDC1/photoz_results/TESTDC1FLEXZ).

[1]:
```
from IPython.display import Markdown
from sample import Sample
from metrics import *
import utils
import os
import matplotlib.pyplot as plt
%matplotlib inline
%reload_ext autoreload
%autoreload 2
```

1.1 Sample

[2]:
```
my_path = "/Users/julia/TESTDC1FLEXZ"

pdfs_file =  os.path.join(my_path, "Mar5Flexzgold_pz.out")
ztrue_file =  os.path.join(my_path, "Mar5Flexzgold_idszmag.out")

#pdfs_file =  os.path.join(my_path, "1pct_Mar5Flexzgold_pz.out")
#ztrue_file =  os.path.join(my_path, "1pct_Mar5Flexzgold_idszmag.out")
```

[3]:
```
%%time
sample = Sample(pdfs_file, ztrue_file, code="FlexZBoost", name="DC1 paper data")
sample
```

```
CPU times: user 40.5 s, sys: 5.03 s, total: 45.5 s
Wall time: 52.1 s
```

[3]:
```
<sample.Sample at 0x7fc1aa8cbf90>
```

[4]:
```
print(sample)
```

```
----------------------
Sample: DC1 paper data
Algorithm: FlexZBoost
----------------------
399356 PDFs with 200 probabilities each
qp representation: interp
z grid: 200 z values from 0.016282 to 1.99986 inclusive
```

1.2 Metrics

[5]:
```
%%time
metrics = Metrics(sample)
```

```
CPU times: user 12min 7s, sys: 11.5 s, total: 12min 19s
Wall time: 13min 44s
```

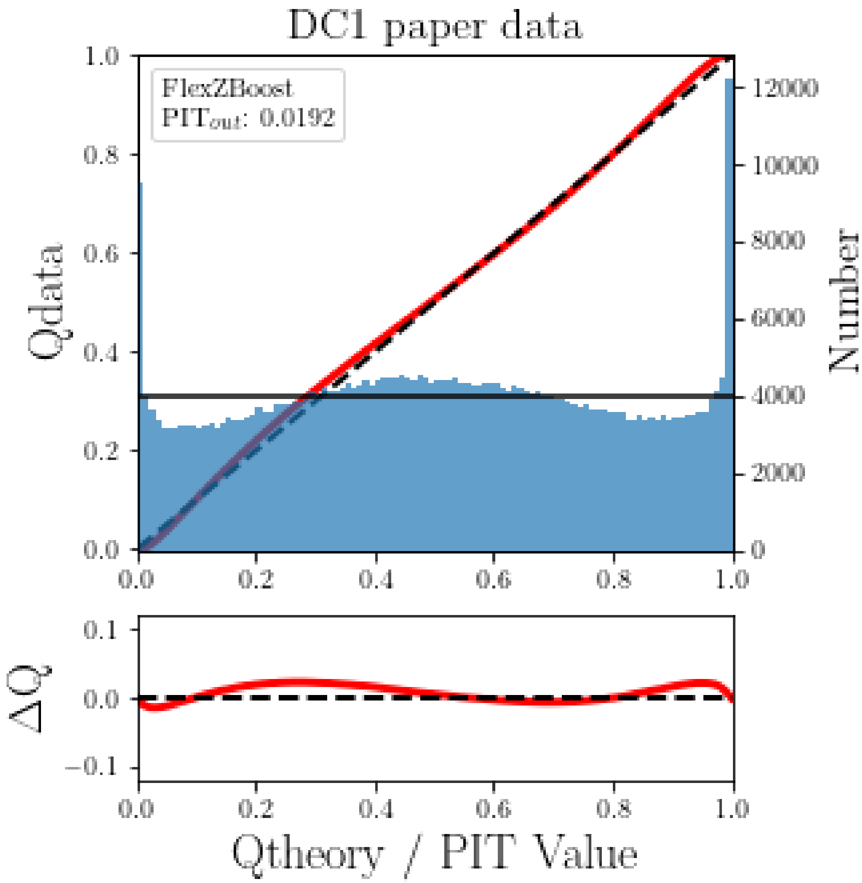
The metrics below are based on the PIT and the CDF(PIT), both computed via qp.Ensemble object method. The PIT array is computed as the qp.Ensemble CDF function for an object containing the photo-z PDFs, evaluated at the true z for each galaxy. The PIT distribution is implemented as the normalized histogram of PIT values. The uniform U(0,1) is implemented as a mock normalized distribution with the same number of bins of PIT distribution, where all values are equal to $1/N_{quant}$.
Then a new qp.Ensemble object is instantiated for each distribution, PITs and U(0,1), to use the CDF functionallity (an ensemble with only 1 PDF each).

```
class Metrics:
    """
       ***   Metrics class   ***
    Receives a Sample object as input.
    Computes PIT and QQ vectors on the initialization.
    It's the basis for the other metrics, such as KS, AD, and CvM.
    """
    def __init__(self, sample, n_quant=100, pit_min=0.0001, pit_max=0.9999, debug=False):
        """Class constructor
        Parameters
        ----------
        sample: `Sample`
            sample object defined in ./sample.py
        n_quant: `int`, (optional)
            number of quantiles for the QQ plot
        pit_min: `float`
            lower limit to define PIT outliers
            default is 0.0001
        pit_max:
            upper limit to define PIT outliers
            default is 0.9999
        """
```

```
        self._sample = sample
        self._n_quant = n_quant
        self._pit_min = pit_min
        self._pit_max = pit_max
        self._debug = debug
        n = len(self._sample)
        if debug:
            #n = 1000 # subset for quick tests
            print("DEBUG MODE")
            #ids = np.random.choice(n, 10000)
            self._pit = np.loadtxt(os.path.join(sample.path,"TESTPITVALS.out"), unpack=True, us
            self.new_pit = np.nan_to_num([self._sample._pdfs[i].cdf(self._sample._ztrue[i])[0]
        else:
            n = len(self._sample)
            self._pit = np.nan_to_num([self._sample._pdfs[i].cdf(self._sample._ztrue[i])[0][0]
        # Quantiles
        Qtheory = np.linspace(0., 1., self.n_quant)
        Qdata = np.quantile(self._pit, Qtheory)
        self._qq_vectors = (Qtheory, Qdata)
        # Normalized distribution of PIT values (PIT PDF)
        self._xvals = Qtheory
        self._pit_pdf, self._pit_bins_edges = np.histogram(self._pit, bins=n_quant, density=Tru
        #self._uniform_pdf = stats.uniform(self._xvals, scale=n_quant)
        self._uniform_pdf = np.full(n_quant, 1.0 / float(n_quant))
        # Define qp Ensemble to use CDF functionality (an ensemble with only 1 PDF)
        self._pit_ensemble = qp.Ensemble(qp.hist, data=dict(bins=self._pit_bins_edges,
                                                            pdfs=np.array([self._pit_pdf])))
        self._uniform_ensemble = qp.Ensemble(qp.interp, data=dict(xvals=self._xvals,
                                                                  yvals=np.array([self._uniforn
        self._pit_cdf = self._pit_ensemble.cdf(self._xvals)[0]
        self._uniform_cdf = self._uniform_ensemble.cdf(self._xvals)[0]
```

PIT-QQ plot

[6]:
```
metrics.plot_pit_qq() #savefig=True)
```



1.2.1 DC1 results

The DC1 results are stored in Metrics class object as a table and as a dictionary, inheriting from an independent class DC1 (in `utils.py` ancillary file), which exists only to provide the reference values.

[7]:
```
metrics.dc1.table
```

[7]:

Code	PIT out rate	KS	CvM	AD	CDE loss
ANNz2	0.0265	0.0174	60.3397	564.0189	-6.8800
BPZ	0.0192	0.0112	37.0919	358.0953	-7.8200
CMNN	0.0034	0.0050	2.9165	30.6465	-10.4300
Delight	0.0006	0.0240	105.6534	624.1780	-8.3300
EAZY	0.0154	0.0430	440.0701	2000.1168	-7.0700
FlexZBoost	0.0202	0.0129	19.7154	303.6520	-10.6000
GPz	0.0058	0.0145	61.6023	618.6360	-9.9300
LePhare	0.0486	0.0245	141.0847	1212.0725	-1.6600
METAPhoR	0.0229	0.0297	153.0529	1445.5312	-6.2800
SkyNet	0.0001	0.0491	961.5396	5689.3225	-7.8900

Code	PIT out rate	KS	CvM	AD	CDE loss
TPZ	0.0130	0.0095	24.3082	282.3698	-9.5500

[8]:
```
print(metrics.dc1.codes)
```

```
('ANNz2', 'BPZ', 'CMNN', 'Delight', 'EAZY', 'FlexZBoost', 'GPz', 'LePhare',
'METAPhoR', 'SkyNet', 'TPZ')
```

[9]:
```
print(metrics.dc1.metrics)
```

```
('PIT out rate', 'CDE loss', 'KS', 'CvM', 'AD')
```

[10]:
```
metrics.dc1.results['PIT out rate']['FlexZBoost']
```

[10]:
```
0.0202
```

1.3 Results

Summary table with all metrics containing DC1 paper results for comparison

[11]:
```
metrics.markdown_metrics_table(show_dc1=True)
```

[11]:

Metric	Value	DC1 reference value
PIT out rate	0.0192	0.0202
CDE loss	-10.62	-10.60
KS	0.0233	0.0129
CvM	0.0133	19.7154
AD	30.6594	303.6520

In the first attempt, the results do not match, except for the PIT outliers rate. The CDE loss is close to the reference values.

[12]:
```
delta = abs(metrics.cde_loss - metrics.dc1.results['CDE loss']['FlexZBoost'])
perc = abs(delta/metrics.dc1.results['CDE loss']['FlexZBoost'])*100.
print(f"CDE loss differs from DC1 value by {delta:.3f} ({perc:.1f}%).")
```

```
CDE loss differs from DC1 value by 0.020 (0.2%).
```

Such small difference could be explained by differences in the binning used for the numerical integration.

However, the KS, CvM, and AD tests still need to be fixed. Let's investigate these numbers by comparing the results with what we would get if using the scipy built-in statistical tests (implemented as alternative methods for each metric).

1.3.1 Kolmogorov-Smirnov

$$\mathrm{KS} \equiv \max_{PIT} \left(\left| \ \mathrm{CDF}[\hat{f}, z] - \mathrm{CDF}[\tilde{f}, z] \ \right| \right)$$

```
    def __init__(self, metrics, scipy=False):
        self._metrics = metrics
        if scipy:
            self._stat, self._pvalue = stats.kstest(metrics._pit, "uniform")
        else:
            self._stat, self._pvalue = np.max(np.abs(metrics._pit_cdf - metrics._uniform_cdf))
        # update Metrics object
        metrics._ks_stat = self._stat
```

[13]:
```
ks_dc1 = metrics.dc1.results['KS']['FlexZBoost']
ks_dc1
```

[13]: 0.01294894

[14]:
```
ks = KS(metrics)
ks.stat
```

[14]: 0.02331976704716826

[15]:
```
ks_sci = KS(metrics, scipy=True)
ks_sci.stat
```

[15]: 0.0234037782702739

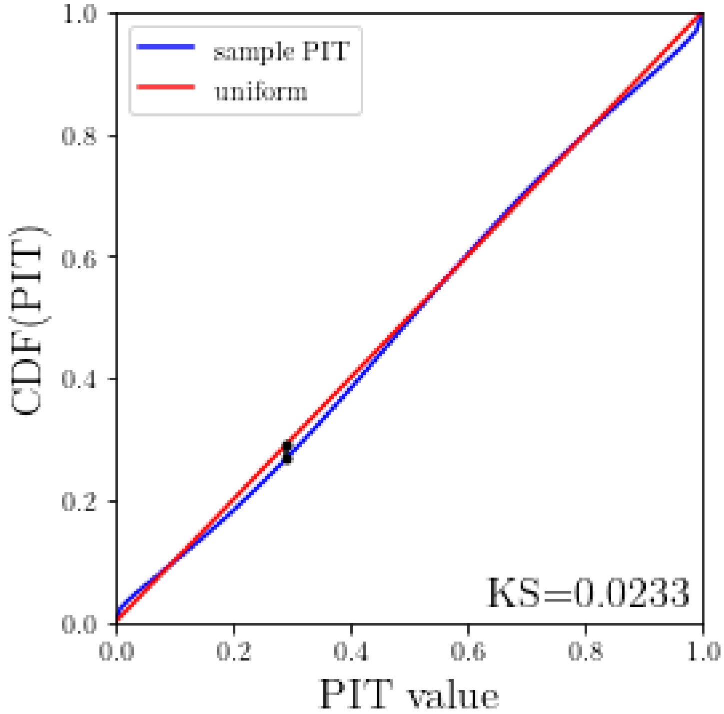
For the Komolgorof-Smirnov test, the values with and without using scipy.stats.ks_test function are compatible with each other and both disagree with the DC1 result significantly.

[16]:
```
delta = abs(ks_sci.stat - metrics.dc1.results['KS']['FlexZBoost'])
perc = abs(delta/metrics.dc1.results['KS']['FlexZBoost'])*100.
print(f"KS differs from DC1 value by {delta:.3f} ({perc:.1f}%).")
```

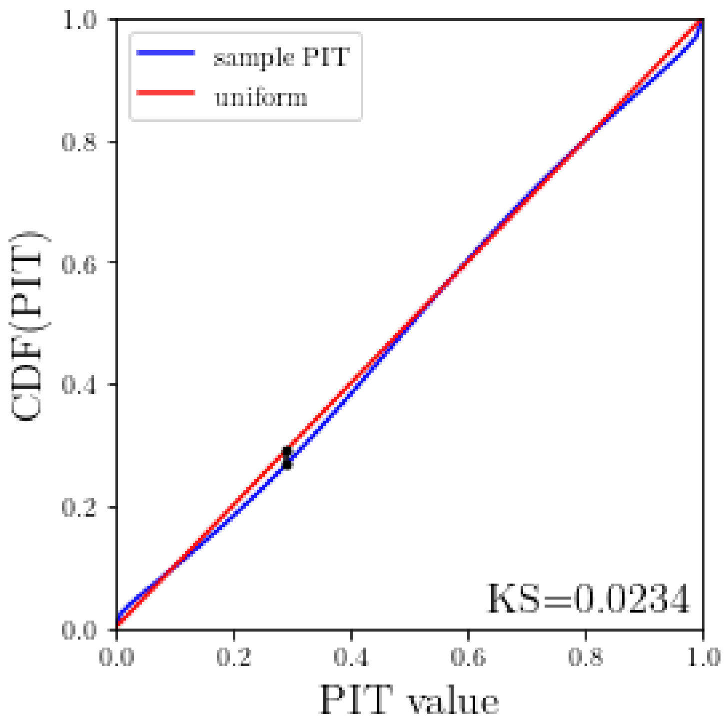
```
KS differs from DC1 value by 0.010 (80.7%).
```

Visual interpretation of KS test

[17]:
```
ks.plot()
```



[18]:
```
ks_sci.plot()
```



SOLUTION STILL PENDING!!!

1.3.2 Cramer-von Mises

Let's fepeat the same excercise with the CvM test.

$$\mathrm{CvM}^2 \equiv \int_{-\infty}^{\infty} \Big(\mathrm{CDF}[\hat{f}, z] \;-\; \mathrm{CDF}[\tilde{f}, z]\Big)^2 \mathrm{dCDF}(\tilde{f}, z)$$

```
    def __init__(self, metrics, scipy=False):
        if scipy:
            cvm_result = stats.cramervonmises(metrics._pit_dist, "uniform")
            self._stat, self._pvalue = cvm_result.statistic, cvm_result.pvalue
        else:
            self._stat, self._pvalue = np.sqrt(np.trapz((metrics._pit_cdf - metrics._uniform_cd
        # update Metrics object
        metrics._cvm_stat = self._stat
```

[19]:
```
cvm_dc1 = metrics.dc1.results['CvM']['FlexZBoost']
cvm_dc1
```

[19]: 19.71544373

[20]:
```
cvm = CvM(metrics)
cvm.stat
```

[20]: 0.013274572987502615

[21]:
```
cvm_sci = CvM(metrics, scipy=True)
cvm_sci.stat
```

[21]: 71.14392872038036

This time, all numbers disagree. I have checked the code fr CvM test in `skgof` library, and it doesn't look like the equation for the definition of CvM shown in the paper.

From https://github.com/wrwrwr/scikit-gof/blob/master/skgof/ecdfgof.py:

```
def cvm_stat(data):
    """
    Calculates the Cramer-von Mises statistic for sorted values from U(0, 1).
    """
    samples2 = 2 * len(data)
    minuends = arange(1, samples2, 2) / samples2
    return 1 / (6 * samples2) + ((minuends - data) ** 2).sum()
```

```
(...)

cvm_test = partial(simple_test, stat=cvm_stat, pdist=cvm_unif)
```

SOLUTION STILL PENDING!!!

1.3.3 Anderson-Darling

The last matric is the AD test, which is the onluy metric that allows the removal of extreme outliers before the calculation:

$$\mathrm{AD}^2 \equiv N_{tot} \int_{-\infty}^{\infty} \frac{\left(\mathrm{CDF}[\hat{f}, z] \; - \; \mathrm{CDF}[\tilde{f}, z]\right)^2}{\mathrm{CDF}[\hat{f}, z](1 \; - \; \mathrm{CDF}[\tilde{f}, z])} \mathrm{dCDF}(\tilde{f}, z)$$

```
    def __init__(self, metrics, ad_pit_min=0.0, ad_pit_max=1.0):

        mask_pit = (metrics._pit >= ad_pit_min) & (metrics._pit  <= ad_pit_max)
        if (ad_pit_min != 0.0) or (ad_pit_max != 1.0):
            n_out = len(metrics._pit) - len(metrics._pit[mask_pit])
            perc_out = (float(n_out)/float(len(metrics._pit)))*100.
            print(f"{n_out} outliers (PIT<{ad_pit_min} or PIT>{ad_pit_max}) removed from the ca

        ad_xvals = np.linspace(ad_pit_min, ad_pit_max, metrics.n_quant)
        ad_yscale_uniform = (ad_pit_max-ad_pit_min)/float(metrics._n_quant)
        ad_pit_dist, ad_pit_bins_edges = np.histogram(metrics.pit[mask_pit], bins=metrics.n_qua
        ad_uniform_dist = np.full(metrics.n_quant, ad_yscale_uniform)
        # Redo CDFs to account for outliers mask
        ad_pit_ensemble = qp.Ensemble(qp.hist, data=dict(bins=ad_pit_bins_edges, pdfs=np.array
        ad_pit_cdf = ad_pit_ensemble.cdf(ad_xvals)[0]
        ad_uniform_ensemble = qp.Ensemble(qp.hist,
                                          data=dict(bins=ad_pit_bins_edges, pdfs=np.array([ad_u
        ad_uniform_cdf = ad_uniform_ensemble.cdf(ad_xvals)[0]
        numerator = ((ad_pit_cdf - ad_uniform_cdf)**2)
        denominator = (ad_uniform_cdf*(1.-ad_uniform_cdf))
        with np.errstate(divide='ignore', invalid='ignore'):
            self._stat = np.sqrt(float(len(metrics._sample)) * np.trapz(np.nan_to_num(numerato
        # update Metrics object
        metrics._ad_stat = self._stat
```

For the Anderson-Darling test, the comparison to a uniform distribution is not available in scipy.stats.anderson method, so using it does not make sense.

[22]:
```
ad_dc1 = metrics.dc1.results['AD']['FlexZBoost']
ad_dc1
```

[22]: 303.65198293

[23]:
```
ad = AD(metrics).stat
ad
```

[23]: 30.659390963941615

Let's remove the catastrophic autliers (as done in the paper), to see the impact.

[24]:
```
ad_clean = AD(metrics, ad_pit_min=0.01, ad_pit_max=0.99).stat
ad_clean
```

```
21764 outliers (PIT<0.01 or PIT>0.99) removed from the calculation (5.4%)
```

[24]: 24.66810022296736

Once more, the results disagree.

SOLUTION STILL PENDING!!!

2 Debugging

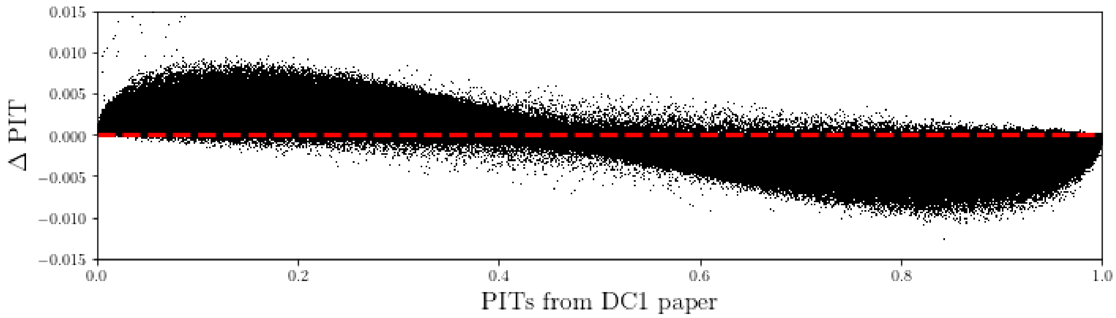
Following Sam's suggestion, I also computed the metrics reading the PIT values from the partial results of DC1 paper, instead of calculating them in advance. The "debug" mode of `metrics` class uses DC1's PIT values. This mode will probably be removed of the code after solving all bugs.

[26]:
```
%%time
metrics_debug = Metrics(sample, debug=True)
```

```
DEBUG MODE
CPU times: user 14min 37s, sys: 19.8 s, total: 14min 57s
Wall time: 16min 13s
```

In the comment section of RAIL's pull request #54, Sam pointed out the small disagreement found between the PIT values of DC1 sample computed now (using current `qp` version), and those computed at the time of the paper writing. There is a trend or new values of PIT to be slightly larger than the old for PIT < 0.5 and slightly smaller for PIT > 0.5.

[27]:
```
plt.figure(figsize=[10,3])
plt.plot(metrics_debug.pit, metrics.pit - metrics_debug.pit, 'k,')
plt.plot([0,1], [0,0], 'r--', lw=3)
plt.xlim(0, 1)
plt.ylim(-0.015, 0.015)
plt.xlabel("PITs from DC1 paper")
plt.ylabel("$\Delta$ PIT")
plt.tight_layout()
```



Results using DC1's PIT values

[28]:
```
metrics_debug.markdown_metrics_table(show_dc1=True)
```

[28]:

Metric	Value	DC1 reference value
PIT out rate	0.0202	0.0202
CDE loss	-10.62	-10.60
KS	0.0239	0.0129
CvM	0.0130	19.7154
AD	32.1472	303.6520

Let's see the `scipy=True` version of the metrics:

[29]:
```
ks_debug_sci = KS(metrics_debug, scipy=True)
ks_debug_sci.stat
```

[29]: 0.02403350544376448

[30]:
```
ks_dc1
```

[30]: 0.01294894

[31]:
```
cvm_debug_sci = CvM(metrics_debug, scipy=True)
cvm_debug_sci.stat
```

[31]: 68.82623704066525

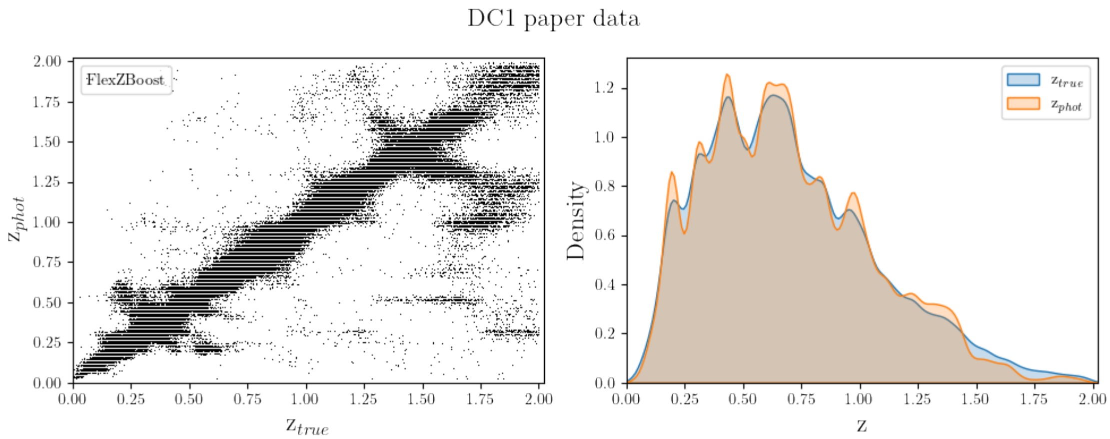
[32]:
```
cvm_dc1
```

[32]: 19.71544373

SOLUTION STILL PENDING!!!

2.0.1 Point estimates metrics

[33]:
```
old_metrics_table = sample.plot_old_valid()
```



[35]:
```
utils.old_metrics_table(sample, show_dc1=True)
```

[35]:

Metric	FlexZBoost DC1 paper data	DC1 paper
scatter	0.0155	0.0154
bias	-0.00027	-0.00027
outlier rate	0.020	0.020

At least the point metrics agree, so the PDFs are being read correctely.

2.1 Conclusion

I still need help to understand the disagreement in the results.

[ ]:

[ ]: