vaex Documentation

Maarten A. Breddels

Aug 29, 2018
## Contents

1 Short version                                         3

2 Longer version                                        5

3 For developers                                        7

4 Vaex tutorial                                         9
   4.1 Dataset                                           9
   4.2 Statistics on N-d grids                          11
   4.3 Getting your data in                             13
   4.4 Plotting                                         14
   4.5 Propagation of uncertainties                     28
   4.6 Parallel computations                           30
   4.7 Interactive widgets                              31
   4.8 Joining                                          32
   4.9 Just-In-Time compilation                         33

5 API documentation for vaex library                    35
   5.1 Quick list for opening/reading in your data.     35
   5.2 Quick list for visualization.                    35
   5.3 Quick list for statistics.                       35
   5.4 vaex module                                      36
   5.5 Dataset class                                    39
   5.6 vaex.stat module                                 66
   5.7 Machine learning with vaex.ml                   67

6 Vaex-ml - Machine Learning                            69
   6.1 KMeans                                            70
   6.2 KMeans benchmark                                  71
   6.3 PCA Benchmark                                     73
   6.4 XGBoost                                           74
   6.5 One hot encoding                                 75

7 Datasets to download                                  77
   7.1 New york taxi dataset                            77
   7.2 SDSS - dereddened                                78
   7.3 Gaia                                             79
Note: For the impatient:

- If you want a standalone Python environment with vaex installed that does not interfere with your system Python, execute `curl http://vaex.astro.rug.nl/install_conda.sh | bash` on your terminal.
- To remove, execute `rm -rf ~/miniconda-vaex ~/.condarc ~/.conda ~/.continuum`.

Warning: It is recommended not to install directly into your operating system’s Python using sudo since it may break your system. Instead, you should install Anaconda, which is a Python distribution that makes installing Python packages much easier or use `virtualenv` or `venv`. 


• **Anaconda users**: `conda install -c conda-forge vaex`
• **Regular Python users using virtualenv**: `pip install --pre vaex`
• **Regular Python users (not recommended)**: `pip install --user --pre vaex`
• **System install (not recommended)**: `sudo pip install --pre vaex`
If you don’t want all packages installed, do not install the vaex package. The vaex package is a meta packages that depends on all other vaex packages so it will instal them all, but if you don’t need astronomy related parts (vaex-astro), or don’t care about distributed (vaex-distributed), you can leave out those packages. Copy paste the following lines and remove what you do not need:

- **Regular Python users:**
  ```
  pip install vaex-core vaex-viz vaex-server vaex-ui
  vaex-hdf5 vaex-astro vaex-distributed
  ```

- **Anaconda users:**
  ```
  conda install -c conda-forge vaex-core vaex-viz vaex-server
  vaex-ui vaex-hdf5 vaex-astro vaex-distributed
  ```

When installing *vaex-ui* it does not install PyQt4, PyQt5 or PySide, you have to choose yourself and installing may be tricky. If running pip install PyQt5 fails, you may want to try your favourite package manager (brew, macports) to install it instead. You can check if you have one of these packages by running:

- `python -c "import PyQt4"
- `python -c "import PyQt5"
- `python -c "import PySide"`
If you want to work on vaex for a Pull Request from the source, use the following recipe:

- git clone https://github.com/maartenbreddels/vaex
- cd vaex
- install using:
  - pip install -e . (again, use (ana)conda or virtualenv/venv)
- If you want to do a PR
  - git remote rename origin upstream
  - (now fork on github)
  - git remote add origin https://github.com/yourusername/vaex/
  - ... edit code ... (or do this after the next step)
  - git checkout -b feature_X
  - git commit -a -m "new: some feature X"
  - git push origin feature_X
  - git checkout master
- Get your code in sync with upstream
  - git checkout master
  - git fetch upstream
  - git merge upstream/master
4.1 Dataset

Central to vaex is the dataset (similar, but not to be confused with a pandas dataframe, hence a different name), and we often use the variables `ds` to represent it. A dataset is an efficient representation for large tabular data, and has:

- A bunch of columns, say x, y and z
- Backed by a numpy array, e.g. `ds.data.x` (but you shouldn’t work with this)
- Wrapped by an expression system, e.g. `ds.x, ds['x']` or `ds.col.x` is an expression
- Columns/expression can perform lazy computations, e.g. `ds.x * np.sin(ds.y)` does nothing, until the result is needed
- A set of virtual columns, columns that are backed by a (lazy) computation, e.g. `ds['r'] = ds.x/ds.y`
- A set of selection, that can be used to explore the dataset, e.g. `ds.select(ds.x < 0)`
- Filtered datasets, that does not copy the data, `ds_negative = ds[ds.x < 0]`

Let’s start with an example dataset, included in vaex

```python
In [1]: import vaex
ds = vaex.example()
ds
# begin the last statement it will print out the tabular data
Out[1]: <vaex.hdf5.dataset.Hdf5MemoryMapped at 0x7f21bc2f5e10>
```

4.1.1 Columns

The above preview shows this dataset contains > 300,000 rows, and columns named x,y,z (positions), vx, vy, vz (velocities), E (energy), L (angular momentum). Printing out a column, shows it is not a numpy array, but an expression

```python
In [2]: ds.x  # ds.col.x or ds['x'] are equivalent, but may be preferred because it is more tab comp.
```
The underlying data is often accessible using `ds.data.x`, but should not be used, since selections and filtering are not reflected in this. However sometimes it is useful to access the raw numpy array.

```python
In [3]: ds.data.x
```
```
Out[3]: array([-0.77747077,  3.77427316,  1.3757627 , ..., -1.14041007, -14.2985935 ,  10.5450506 ])
```

A better way, if you need a numpy array (for instance for plotting, or passing to a different library) it to use `evaluate`, which will also work with virtual columns, selections and filtered datasets (more on that below).

```python
In [4]: ds.evaluate(ds.x)
```
```
Out[4]: array([-0.77747077,  3.77427316,  1.3757627 , ..., -1.14041007, -14.2985935 ,  10.5450506 ])
```

Most numpy function (ufuncs) can be performed on expressions, and will not result in a direct result, but in a new expression.

```python
In [5]: import numpy as np
   ...: np.sqrt(ds.x**2 + ds.y**2 + ds.z**2)
```
```
Out[5]: <vaex.expression.Expression(expressions='sqrt((((x ** 2) + (y ** 2)) + (z ** 2)))')> instance at 0x7f21978f42e8 ... 0.882561312135 ... (total 330000 values) ... 7.45383176151, 15.3984124911, 8.86425027393, 17.601047186, 14.540181525
```

### 4.1.2 Virtual functions

Sometimes it is convenient to store an expression as a column, or virtual column, a column that does not take up memory, but will be computed on the fly. A virtual column can be treated as a normal column.

```python
In [6]: ds['r'] = np.sqrt(ds.x**2 + ds.y**2 + ds.z**2)
   ...: ds[['x', 'y', 'z', 'r']]
```

```html
Out[6]: <IPython.core.display.HTML object>
```

### 4.1.3 Selections and filtering

Vaex can be efficient when exploring subsets of the data, for instance to remove outlier or to inspect only a part of the data. Instead of making copies, internally vaex keeps track which rows is selected.

```python
In [7]: ds.select(ds.x < 0)
   ...: ds.evaluate(ds.x, selection=True)
```
```
Out[7]: array([-0.77747077,  -7.06737804,  -5.17174435, ..., -1.87310386, -1.14041007, -14.2985935 ])
```

Selections can be useful if you want to change what you select frequently, as in visualization, or when you want to compute statistics on several selections efficiently. Instead, you can also create a filtered dataset, and is similar in use to pandas, except that it does not copy the data.

```python
In [8]: ds_negative = ds[ds.x < 0]
   ...: ds_negative[['x', 'y', 'z', 'r']]
```

```html
Out[8]: <IPython.core.display.HTML object>
```
4.2 Statistics on N-d grids

A core feature of vaex, and used for visualization, is calculation of statistics on N dimensional grids.

```
In [9]: ds.count(), ds.mean(ds.x), ds.mean(ds.x, selection=True)
Out[9]: (330000.0, -0.067131491264005971, -5.2110379721119671)
```

Similar to SQL’s groupby, vaex uses the binby concept, which tells vaex that a statistic should be calculated on a regular grid (for performance reasons)

```
In [10]: xcounts = ds.count(binby=ds.x, limits=[-10, 10], shape=64)
xcounts
                2157., 2357., 2653., 2786., 3012., 3215., 3619., 3890.,
                3973., 4400., 4782., 5126., 5302., 5729., 6042., 6562.,
                6852., 7167., 7456., 7633., 7910., 8415., 8619., 8246.,
                8358., 8769., 8294., 7870., 7499., 7389., 7174., 6901.,
                6557., 6173., 5721., 5367., 4963., 4655., 4246., 4110.,
                3939., 3611., 3289., 3018., 2811., 2570., 2505., 2267.,
                2013., 1803., 1687., 1563., 1384., 1326., 1257., 1189.])
```

This results in a numpy array with the number counts in 64 bins distributed between x = -10, and x = 10. We can quickly visualize this using matplotlib.

```
In [12]: import matplotlib.pyplot as plt
   plt.plot(np.linspace(-10, 10, 64), xcounts)
   plt.show()
```

We can instead of doing 1d binning, do it in 2d as well (N-d actually), and visualize it using imshow.

```
In [13]: xycounts = ds.count(binby=[ds.x, ds.y], limits=[[-10, 10], [-10, 20]], shape=(64, 128))
   xycounts
Out[13]: array([[ 9.,  3.,  3., ...,  3.,  2.,  1.],
                [ 5.,  3.,  1., ...,  1.,  3.,  3.],
                [11.,  3.,  2., ...,  1.,  1.,  4.],
```

We can instead of doing 1d binning, do it in 2d as well (N-d actually), and visualize it using imshow.
...,
[12., 6., 8., ..., 0., 1., 0.],
[7., 6., 12., ..., 3., 0., 0.],
[11., 10., 7., ..., 1., 1., 1.])

In [14]: plt.imshow(xycounts.T, origin='lower', extent=[-10, 10, -10, 20])
plt.show()

In [15]: v = np.sqrt(ds.vx**2 + ds.vy**2 + ds.vz**2)
xy_mean_v = ds.mean(v, binby=[ds.x, ds.y], limits=[[-10, 10], [-10, 20]], shape=(64, 128))
xy_mean_v

Out[15]: array([[144.38495511, 183.45775869, 187.78325557, ..., 138.99392387,
               168.66141282, 142.5518784],
              [143.72427758, 152.14679337, 107.90949865, ..., 119.65318885,
               94.00098292, 104.35109636],
              [172.08240652, 137.47896886, 72.51331138, ..., 179.85933835,
               33.36968912, 111.81826254],
              ..., ...
              [186.56949934, 161.3747346, 174.27411865, ..., nan,
               105.96746091, nan],
              [179.55997022, 137.48979882, 113.82121826, ..., 104.90205692,
               nan, nan],
              [151.94323763, 135.44083212, 84.81787495, ..., 175.79289144,
               129.63799565, 108.19069385]])

In [16]: plt.imshow(xy_mean_v.T, origin='lower', extent=[-10, 10, -10, 20])
plt.show()
Other statistics can be computed, such as:

- `Dataset.count`
- `Dataset.mean`
- `Dataset.std`
- `Dataset.var`
- `Dataset.median_approx`
- `Dataset.percentile_approx`
- `Dataset.mode`
- `Dataset.min`
- `Dataset.max`
- `Dataset.minmax`
- `Dataset.mutual_information`
- `Dataset.correlation`

Or see the full list at the API docs.

### 4.3 Getting your data in

Before continuing, you may want to read in your own data. Ultimately, a vaex Dataset just wraps a set of numpy arrays. If you can access your data as a set of numpy arrays, you can therefore make dataset using `from_arrays`

```
In [17]: import vaex
   ...: import numpy as np
   ...: x = np.arange(5)
   ...: y = x**2
```
Other quick ways to get your data in are:

- `from_csv`: Comma separated files
- `from_ascii`: Space/tab separated files
- `from_pandas`: Converts a pandas DataFrame
- `from_astropy_table`: Converts a astropy table

## 4.4 Plotting

### 4.4.1 1d and 2d

Most visualization can be done in 1 and 2d, and vaex wraps matplotlib to provide most use cases.

```python
In [18]: import vaex
   ...: import numpy as np
   ...: ds = vaex.example()
   ...: %matplotlib inline
```

The simplest visualization is a 1d plot using Dataset.plot1d. When only given one arguments, it will show a histogram showing 99.8% of the data.

```python
In [19]: ds.plot1d(ds.x)
```

```python
Out[19]: [<matplotlib.lines.Line2D at 0x7f2151713a58>]
```
A slightly more complicated visualization is to not plot the counts, but a different statistic for that bin. In most cases, passing the `what='<statistic>(<expression>)'` argument will do, where `<statistic>` is any of the statistics mentioned in the list above, or in the API docs.

```python
In [20]: ds.plot1d(ds.x, what='mean(E)')
Out[20]: [<matplotlib.lines.Line2D at 0x7f215167e390>]
```

An equivalent method is to use the `vaex.stat.<statistic>` functions, e.g. `vaex.stat.mean`.

```python
In [21]: ds.plot1d(ds.x, what=vaex.stat.mean(ds.E))
Out[21]: [<matplotlib.lines.Line2D at 0x7f215056fba8>]
```
These objects are very similar to vaex’ expression, in that they represent an underlying calculation, while normal arithmetic and numpy functions can be applied to it. However, these object represent a statistics computation, and not a column.

```
In [22]: np.log(vaex.stat.mean(ds.x)/vaex.stat.std(ds.x))
Out[22]: log((mean(x) / std(x)))
```

These statistical objects can be passed to the what argument. The advantage being that the data will only have to be passed over once.

```
In [23]: ds.plot1d(ds.x, what=np.clip(np.log(-vaex.stat.mean(ds.E)), 11, 11.4))
Out[23]: [<matplotlib.lines.Line2D at 0x7f2150559908>]
```
A similar result can be obtained by calculating the statistic ourselves, and passing it to plot1d’s grid argument. Care has to be taken that the limits used for calculating the statistics and the plot are the same, otherwise the x axis may not correspond to the real data.

```
In [24]: limits = [-30, 30]
shape = 64
meanE = ds.mean(ds.E, binby=ds.x, limits=limits, shape=shape)
grid = np.clip(np.log(-meanE), 11, 11.4)
ds.plot1d(ds.x, grid=grid, limits=limits, ylabel='clipped E')
```

Out[24]: [<matplotlib.lines.Line2D at 0x7f21504c3a58>]

---

4.4. Plotting
The same applies for 2d plotting.

In [25]: ds.plot(ds.x, ds.y, what=vaex.stat.mean(ds.E)**2)

Out[25]: <matplotlib.image.AxesImage at 0x7f21515cec18>
4.4.2 Selections for plotting

While filtering is useful for narrowing down a selection (e.g. `ds_negative = ds[ds.x < 0]`) there are a few downsides to this. First, a practical issue is that when you filter 4 different ways, you will need to have 4 different objects, polluting your namespace. However, more importantly, when vaex executes a bunch of statistical computations, it will do that per dataset, meaning for 4 different datasets (although pointing to the same underlying data) it will do a total of 4 passes over the data. If instead, we have 4 (named) selections in our dataset, it can calculate statistics in one single pass over the data, which can speed up especially when you dataset is larger than your memory.

In the plot below, we show three selection, which by default are blended together, requiring just one pass over the data.

In [26]: ds.plot(ds.x, ds.y, what=np.log(vaex.stat.count()+1),
   selection=[None, ds.x < ds.y, ds.x < -10])

Out[26]: <matplotlib.image.AxesImage at 0x7f21515c7cf8>

4.4.3 Advanced Plotting

Lets say we would like to see two plots next to eachother, we can pass a list of expression pairs.

In [27]: ds.plot(["x", "y"], ["x", "z"],
   title="Face on and edge on", figsize=(10,4));

/net/gaia/data/users/breddels/python/anaconda3/envs/dev/lib/python3.6/site-packages/matplotlib/cbook/deprecation.py:106: ... this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
warnings.warn(message, mplDeprecation, stacklevel=1)

/net/gaia/data/users/breddels/python/anaconda3/envs/dev/lib/python3.6/site-packages/matplotlib/cbook/deprecation.py:106: ... this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
warnings.warn(message, mplDeprecation, stacklevel=1)
By default, if you have multiple plots, they are shown as columns, multiple selections are overplotted, and multiple ‘whats’ (statistics) are shown as rows.

```python
In [28]: ds.plot(["x", "y"], ["x", "z"],
               what=[np.log(vaex.stat.count()+1), vaex.stat.mean(ds.E)],
               selection=[None, ds.x < ds.y],
               title="Face on and edge on", figsize=(10,10));
```
However, this behaviour can be changed using the `visual` argument.

```python
In [29]: ds.plot(["x", "y"], ["x", "z"],
   what=vaex.stat.mean(ds.E),
   selection=[None, ds.Lz < 0],
   visual=dict(column='selection'),
   title="Face on and edge on", figsize=(10,10));
```

(Note that the selection has no effect in the bottom rows)
4.4.4 Slices in a 3rd dimension

If a 3rd axis (z) is given, you can ‘slice’ through the data, displaying the z slices as rows. Note that here the rows are wrapped, which can be changed using the `wrap_columns` argument.

```python
In [30]: ds.plot("Lz", "E", z="FeH:-3,-1,10", show=True, visual=dict(row="z"), figsize=(12,8), f="log", wrap_columns=3);
```

warnings.warn(message, mplDeprecation, stacklevel=1)
4.4.5 Smaller datasets / scatter plot

Although vaex focusses on large datasets, sometimes you end up with a fraction of the data (due to a selection) and you want to make a scatter plot. You could try the following approach:

```python
In [31]: import vaex
ds = vaex.example()
%matplotlib inline

In [32]: import matplotlib.pyplot as plt
x = ds.evaluate("x", selection=ds.Lz < -2500)
y = ds.evaluate("y", selection=ds.Lz < -2500)
plt.scatter(x, y, c="red", alpha=0.5, s=4);
```
4.4.6 In control

While vaex provides a wrapper for matplotlib, there are situations where you want to use the Dataset.plot method, but want to be in control of the plot. Vaex simply uses the current figure and axes, so that is easy to do.

In [34]: import numpy as np
4.4.7 Healpix (Plotting)

Using healpix is made available by the vaex-healpix package using the healpy package. Vaex does not need special support for healpix, only for plotting, but some helper functions are introduced to make working with healpix easier. By diving the source_id by 34359738368 you get a healpix index level 12, and diving it further will take you to lower levels.

To understand healpix better, we will start from the beginning. If we want to make a density sky plot, we would like to pass healpy a 1d numpy array where each value represents the density at a location of the sphere, where the location is determined by the array size (the healpix level) and the offset (the location). Since the Gaia data includes the healpix index encoded in the source_id. By diving the source_id by 34359738368 you get a healpix index level 12, and diving it further will take you to lower levels.

In [36]: import vaex
   import healpy as hp
```
#matplotlib inline
tgas = vaex.datasets.tgas.fetch()
```

Downloading http://vaex.astro.rug.nl/data/tgas.hdf5 to /Users/users/breddels/.vaex/data/tgas.hdf5

Wget failed, using urlretrieve

We will start showing how you could manually do statistics on healpix bins using vaex.count. We will do a really
course healpix scheme (level 2).

In [37]: level = 2
   factor = 34359738368 * (4**(12-level))
   nmax = hp.nside2npix(2**level)
   epsilon = 1e-16
   counts = tgas.count(binby=tgas.source_id/factor, limits=[-epsilon, nmax-epsilon], shape=nmax)

Out[37]: array([ 4021.,  6171.,  5318.,  7114.,  5755., 13420., 12711.,
   10193.,  7782., 14187., 12578., 22038., 17313., 13064.,
   17298., 11887., 3859., 3488., 9036., 5533., 4007.,
   3899.,  4884.,  5664., 10741.,  7678., 12092., 10182.,
   6652.,  6793., 10117.,  9614.,  3727., 5849., 4028.,
   5505.,  8462., 10059.,  6581.,  8282., 4757., 5116.,
   4578.,  5452.,  6023.,  8340.,  6440.,  8623.,  7308.,
   6197., 21271.,  23176.,  12975.,  17138., 26783., 30575.,
  31931.,  29697.,  17986.,  16987., 19802., 15632., 14273.,
   10594.,  4807.,  4551.,  4028.,  4357.,  4067.,  4206.,
   3505.,  4137.,  3311.,  3582.,  3586.,  4218.,  4529.,
   4360.,  6767.,  7579., 14462.,  24291., 10638., 11250.,
  29619.,  9678., 23322., 18205.,  7625.,  9891.,  5423.,
   5808., 14438., 17251.,  7833., 15226.,  7123.,  3708.,
   6135.,  4110.,  3587.,  3222.,  3074.,  3941.,  3846.,
   3402.,  3564.,  3425.,  4125.,  4026.,  3689.,  4084.,
  16617., 13577.,  6911.,  4837., 13553., 10074.,  9534.,
  20824.,  4976.,  6707.,  5396.,  8366., 13494., 19766.,
  11012., 16130.,  8521.,  8245.,  6871.,  5977.,  8789.,
  10016.,  6517.,  8019.,  6122.,  5465.,  5414.,  4934.,
   5788.,  6139.,  4310.,  4144.,  11437., 30731., 13741.,
  27285.,  40227., 16320., 23039., 10812., 14686., 27690.,
  15155., 32701., 18780.,  5895., 23348.,  6081., 17050.,
  28498., 35232., 26223., 22341., 15867., 17688.,  8580.,
  24895., 13027., 11223.,  7880.,  8386., 6988.,  5815.,
   4717.,  9088.,  8283., 12059.,  9161.,  6952.,  4914.,
   6652.,  4666., 12014., 10703., 16518., 10270.,  6724.,
   4553.,  9282.,  4981.])

And using healpy’s mollview we can visualize this.

In [38]: hp.mollview(counts, nest=True)
To simplify life, vaex includes `Dataset.healpix_count` to take care of this.

```python
In [39]: counts = tgas.healpix_count(healpix_level=6)
   ...: hp.mollview(counts, nest=True)
```

Or even simpler, use `Dataset.healpix_plot`

```python
In [40]: tgas.healpix_plot(f="log1p", healpix_level=6, figsize=(10,8),
   ...:         healpix_output="ecliptic")
```
4.5 Propagation of uncertainties

In science we often deal with measurement uncertainties (sometimes referred to as measurement errors). When transformations are made with quantities that have uncertainties associated with them, the uncertainties on these transformed quantities can be calculated automatically by vaex. Note that propagation of uncertainties requires derivatives and matrix multiplications of lengthy equations, which is not complex, but tedious. Vaex can automatically calculate all dependencies, derivatives and compute the full covariance matrix.

```python
In [41]: import vaex
   : import pylab as plt
   : %matplotlib inline
   : tgas = vaex.datasets.tgas_1percent.fetch()
```

Even though the TGAS dataset already contains galactic sky coordinates (l and b), we add them again as virtual columns such that the transformation between RA. and Dec. and the galactic sky coordinates is known.

```python
In [42]: # convert parallaxes to distance
tgas.add_virtual_columns_distance_from_parallax(tgas.parallax)
# 'overwrite' the real columns 'l' and 'b' with virtual columns
tgas.add_virtual_columns_eq2gal('ra', 'dec', 'l', 'b')
# and combined with the galactic sky coordinates gives galactic cartesian coordinates of the
# j2000

tgas.add_virtual_columns_spherical_to_cartesian(tgas.l, tgas.b, tgas.distance, 'x', 'y', 'z')
```

Since RA. and Dec. are in degrees, while ra_error and dec_error is in milliarcseconds, so we put them on the same scale

```python
In [43]: tgas['ra_error'] = tgas.ra_error / 1000 / 3600
   : tgas['dec_error'] = tgas.dec_error / 1000 / 3600
```
We now let vaex sort out what the covariance matrix is for the cartesian coordinates x, y, and z. And take 50 samples from the datasets for visualization.

```python
In [44]: tgas.propagate_uncertainties([tgas.x, tgas.y, tgas.z])
   tgas_50 = tgas.sample(50, random_state=42)
```

For this small dataset we visualize the uncertainties, with and without the covariance.

```python
In [45]: tgas_50.scatter(tgas_50.x, tgas_50.y, xerr=tgas_50.x_uncertainty, yerr=tgas_50.y_uncertainty)
   plt.xlim(-10, 10)
   plt.ylim(-10, 10)
   plt.show()
   tgas_50.scatter(tgas_50.x, tgas_50.y, xerr=tgas_50.x_uncertainty, yerr=tgas_50.y_uncertainty, cov=tgas_50.y_x_covariance)
   plt.xlim(-10, 10)
   plt.ylim(-10, 10)
   plt.show()
```
From the second plot, we see that showing error ellipses (so narrow that they appear as lines) instead of error bars reveal that the distance information dominates the uncertainty in this case.

4.6 Parallel computations

As mentioned in the sections on selections, vaex can do computations on a dataset in parallel. Often, this is taken care of, when for instance passing multiple selections, or multiple arguments to one of the statistical functions. However, sometimes it is difficult or impossible to express a computation in one expression, and we need to resort to doing so called ‘delayed’ computationed, similar as in joblib and dask.

In [46]:
import vaex

ds = vaex.example()
limits = [-10, 10]
delayed_count = ds.count(ds.E, binby=ds.x, limits=limits,
shape=4, delay=True)
delayed_count

Out[46]: <vaex.promise.Promise at 0x7f214809d2e8>

Note that now the returned value is not a promise (TODO: a more Pythonic way would be to return a Future). This may be subject to change, and the best way to work with this is to use the delayed decorator. And call Dataset.execute when the result is needed.

In addition to the above delayed computation, we schedule another computation, such that both the count and mean are execute in parallel such that we only do a single pass over the data. We schedule the execution of two extra functions using the vaex.delayed decorator, and run the whole pipeline using ds.execute().

In [47]: delayed_sum = ds.sum(ds.E, binby=ds.x, limits=limits,
shape=4, delay=True)
@vaex.delayed
def calculate_mean(sums, counts):
    print('calculating mean')
return sums/counts

print('before calling mean')
# since calculate_mean is decorator with vaex.delayed
# this now also returns a 'delayed' object (a promise)
delayed_mean = calculate_mean(delayed_sum, delayed_count)

# if we'd like to perform operations on that, we can again
# use the same decorator
@vaex.delayed
def print_mean(means):
    print('means', means)
print_mean(delayed_mean)

print('before calling execute')
ds.execute()

# Using the .get on the promise will also return the result
# However, this will only work after execute, and may be
# subject to change
means = delayed_mean.get()
print('same means', means)

before calling mean
before calling execute
calculating mean
means [-94415.16581227 -118856.63989386 -118919.86423543 -95000.5998913 ]
same means [-94415.16581227 -118856.63989386 -118919.86423543 -95000.5998913 ]

4.7 Interactive widgets

Note: The interactive widgets require a running Python kernel, if you are viewing this documentation online you mean get a feeling for what the widgets can do, but computation will not be possible!

Using the `vaex-jupyter` package, we get access to interactive widgets.

In [48]:
import vaex
import vaex.jupyter
import numpy as np
import pylab as plt
%matplotlib inline
ds = vaex.example()

The simplest way to get a more interactive visualization (or even print out statistics) is to the the `vaex.jupyter.interactive_selection` decorator, which will execute the decorated function each time the selection is changed.

In [49]:
ds.select(ds.x > 0)
@vaex.jupyter.interactive_selection(ds)
def plot():
    print("Mean x for the selection is:", ds.mean(ds.x, selection=True))
ds.plot(ds.x, ds.y, what=np.log(vaex.stat.count()+1), selection=[None, True])
plt.show()
After changing the selection programmatically, the visualization will update, as well as the print output.

```python
In [50]: ds.select(ds.x > ds.y)
```

However, to get truly interactive visualization, we need to use widgets, such as the `bqplot` library. Again, if we make a selection here, the above visualization will also update, so let's select a square region. One issue is that if you have installed ipywidget, bqplot, ipyvolume etc, it may not be enabled if you installed them from pip (from conda-forge will enabled it automatically). To enable it, run the next cell, and refresh the notebook if there were not enabled already.

(Note that these commands will execute in the environment where the notebook is running, not where the kernel is running)

```python
In [50]: import sys
!jupyter nbextension enable --sys-prefix --py widgetsnbextension
!jupyter nbextension enable --sys-prefix --py bqplot
!jupyter nbextension enable --sys-prefix --py ipyvolume
!jupyter nbextension enable --sys-prefix --py ipympl
!jupyter nbextension enable --sys-prefix --py ipyleaflet
```

Enabling notebook extension jupyter-js-widgets/extension...
  - Validating: OK
Enabling notebook extension bqplot/extension...
  - Validating: OK
Enabling notebook extension ipyvolume/extension...
  - Validating: OK
Enabling notebook extension jupyter-matplotlib/extension...
  - Validating: OK
Enabling notebook extension jupyter-leaflet/extension...
  - Validating: OK

```python
In [51]: # the default backend is bqplot, but we pass it here explicitly
ds.plot_widget(ds.x, ds.y, f='log1p', backend='bqplot')
```

4.8 Joining

Joining in `vaex` is similar to pandas, except the data will no be copied. Internally an index array is kept for each row on the left dataset, pointing to the right dataset, requiring about 8GB for a billion row $10^9$ dataset. Let's start with 2 small dataset, `ds1` and `ds2`:

```python
In [52]: a = np.array(['a', 'b', 'c'])
x = np.arange(1,4)
ds1 = vaex.from_arrays(a=a, x=x)
ds1

Out[52]: <vaex.dataset.DatasetArrays at 0x7f214850ab00>
```

```python
In [53]: b = np.array(['a', 'b', 'd'])
y = x**2

ds2 = vaex.from_arrays(b=b, y=y)
ds2

Out[53]: <vaex.dataset.DatasetArrays at 0x7f20b386e6a0>
```

The default join, is a 'left' join, where all rows for the left dataset (ds1) are kept, and matching rows of the right dataset (ds2) are added. We see for the columns b and y, some values are missing, as expected.
In [54]: ds1.join(ds2, left_on='a', right_on='b')
<IPython.core.display.HTML object>
Out[54]: <vaex.dataset.DatasetArrays at 0x7f20b3869ef0>

A ‘right’ join, is basically the same, but now the roles of the left and right label swapped, so now we have some values from columns x and a missing.

In [55]: ds1.join(ds2, left_on='a', right_on='b', how='right')
<IPython.core.display.HTML object>
Out[55]: <vaex.dataset.DatasetArrays at 0x7f20b386e6a0>

Other joins (inner and outer) aren’t supported, feel free open an issue on github for this.

### 4.9 Just-In-Time compilation

Lets start with a function that converts from two angles, to an angular distance. The function assumes as input, 2 pairs on angular coordinates, in radians.

In [56]:
import vaex
import numpy as np

# From http://pythonhosted.org/pythran/MANUAL.html
def arc_distance(theta_1, phi_1, theta_2, phi_2):
    
    Calculates the pairwise arc distance
    between all points in vector a and b.
    
    temp = (np.sin((theta_2-2-theta_1)/2)**2
    + np.cos(theta_1)*np.cos(theta_2) * np.sin((phi_2-phi_1)/2)**2)
distance_matrix = 2 * np.arctan2(np.sqrt(temp), np.sqrt(1-temp))
return distance_matrix

Let us use the New York Taxi dataset of 2015, as can be downloaded in hdf5 format

In [58]: nytaxi = vaex.open("/Users/maartenbreddels/datasets/nytaxi/nyc_taxi2015.hdf5")
    # lets use just 20% of the data, since we want to make sure it fits
    # into memory (so we don’t measure just hdd/ssd speed)
    nytaxi.set_active_fraction(0.2)

Although the function above expected numpy arrays, vaex can pass in columns or expression, which will delay execution till needed, and add the resulting expression as a virtual column.

In [59]: nytaxi['arc_distance'] = arc_distance(nytaxi.pickup_longitude * np.pi/180,
    nytaxi.dropoff_longitude * np.pi/180,
    nytaxi.pickup_latitude * np.pi/180,
    nytaxi.dropoff_latitude * np.pi/180)

When we calculate the mean angular distance of a taxi trip, we encounter some invalid data, that will give warnings, which we can savely ignore for this demonstration.

In [60]: %%time
    nytaxi.mean(nytaxi.arc_distance)
CPU times: user 7.64 s, sys: 2.8 s, total: 10.4 s
Wall time: 2.63 s

Out[60]: 1.9999877196037037
This computation uses quite some heavy mathematical operation, and since it's (internally) using numpy arrays, also uses quite some temporary arrays. We can optimize this calculation by doing a Just-In-Time compilation, based on numba or pythran. Choose whichever gives the best performance or is easiest to install.

```
In [61]: nytaxi['arc_distance_jit'] = nytaxi.arc_distance.jit_numba()
    # nytaxi['arc_distance_jit'] = nytaxi.arc_distance.jit_pythran()
```

```
In [62]: %%time
    nytaxi.mean(nytaxi.arc_distance_jit)
```

CPU times: user 3.22 s, sys: 23.3 ms, total: 3.24 s
Wall time: 511 ms

```
Out[62]: 1.9999877196036921
```

We can that we can get a significant speedup ($\approx 4x$) in this case.
CHAPTER 5

API documentation for vaex library

5.1 Quick list for opening/reading in your data.

```
vaex.open(path[, convert, shuffle, copy_index])  # Open a dataset from file given by path
vaex.from_arrays(**arrays)                    # Create an in memory dataset from numpy arrays
vaex.from_csv(filename_or_buffer[, copy_index])  # Shortcut to read a csv file using pandas and convert to a dataset directly
vaex.from_ascii(path[, seperator, names,...])  # Create an in memory dataset from an ascii file (whitespace seperated by default).
vaex.from_pandas(df[, name, copy_index,...])   # Create an in memory dataset from a pandas data frame
vaex.from_astropy_table(table)
```

5.2 Quick list for visualization.

```
vaex.dataset.Dataset.plot(**kwargs)
vaex.dataset.Dataset.plot1d(**kwargs)
vaex.dataset.Dataset.scatter(**kwargs)
vaex.dataset.Dataset.plot_widget(x, y[, z,...])
vaex.dataset.Dataset.healpix_plot(...)

param healpix_expression
  {healpix_max_level}
```

5.3 Quick list for statistics.
5.4 vaex module

Vaex is a library for dealing with big tabular data. The most important class (datastructure) in vaex is the `Dataset`. A dataset is obtained by either, opening the example dataset:

```python
>>> import vaex
>>> t = vaex.example()
```

Or using `open()` or `from_csv()`, to open a file:

```python
>>> t1 = vaex.open("somedata.hdf5")
>>> t2 = vaex.open("somedata.fits")
>>> t3 = vaex.from_csv("somedata.csv")
```

Or connecting to a remove server:

```python
>>> tbig = vaex.open("http://bla.com/bigtable")
```

The main purpose of vaex is to provide statistics, such as mean, count, sum, standard deviation, per columns, possibly with a selection, and on a regular grid. To count the number of rows:

```python
```
```python
>>> t = vaex.example()
>>> t.count()
330000.0

Or the number of valid values, which for this dataset is the same:

```python
>>> t.count("x")
330000.0
```

Count them on a regular grid:

```python
>>> t.count("x", binby=["x", "y"], shape=(4,4))
array([[ 902.,  5893.,  5780.,  1193.],
       [ 4097.,  71445.,  75916.,  4560.],
       [ 4743.,  71131.,  65560.,  4108.],
       [ 1115.,  6578.,  4382.,  821.]])
```

Visualise it using matplotlib:

```python
>>> t.plot("x", "y", show=True)
<matplotlib.image.AxesImage at 0x1165a5090>
```

```python
vaex.open(path, convert=False, shuffle=False, copy_index=True, *args, **kwargs)
```

Open a dataset from file given by path

Example:

```python
>>> ds = vaex.open('sometable.hdf5')
>>> ds = vaex.open('somedata*.csv', convert='bigdata.hdf5')
```

### Parameters

- **path** (str) – local or absolute path to file, or glob string
- **convert** – convert files to an hdf5 file for optimization, can also be a path
- **shuffle** (bool) – shuffle converted dataset or not
- **args** – extra arguments for file readers that need it
- **kwargs** – extra keyword arguments
- **copy_index** (bool) – copy index when source is read via pandas

### Returns

return dataset if file is supported, otherwise None

### Return type

**Dataset**

**Example**

```python
>>> import vaex as vx
>>> vx.open('myfile.hdf5')
<vaex.dataset.Hdf5MemoryMapped at 0x1136ee3d0>
```

```python
vaex.from_arrays(**arrays)
```

Create an in memory dataset from numpy arrays

### Param

**arrays**: keyword arguments with arrays
vaex Documentation

Example

```python
>>> x = np.arange(10)
>>> y = x ** 2
>>> dataset = vx.from_arrays(x=x, y=y)
```

vaex.**from_csv**(filename_or_buffer, copy_index=True, **kwargs)
Shortcut to read a csv file using pandas and convert to a dataset directly

vaex.**from_ascii**(path, seperator=None, names=True, skip_lines=0, skip_after=0, **kwargs)
Create an in memory dataset from an ascii file (whitespace seperated by default).

```python
>>> ds = vx.from_ascii("table.asc")
>>> ds = vx.from_ascii("table.csv", seperator="", names=\["x", "y", "z"])
```

Parameters

- **path** – file path
- **seperator** – value seperator, by default whitespace, use “,” for comma seperated values.
- **names** – If True, the first line is used for the column names, otherwise provide a list of strings with names
- **skip_lines** – skip lines at the start of the file
- **skip_after** – skip lines at the end of the file
- **kwargs** –

Returns

vaex.**from_pandas**(df, name='pandas', copy_index=True, index_name='index')
Create an in-memory dataset from a pandas dataframe

Param pandas.DataFrame df: Pandas dataframe

Param name: unique for the dataset

```python
>>> import pandas as pd
>>> df = pd.from_csv("test.csv")
>>> ds = vx.from_pandas(df, name="test")
```

vaex.**from_astropy_table**(table)
vaex.**from_samp**(username=None, password=None)
Connect to a SAMP Hub and wait for a single table load event, disconnect, download the table and return the dataset

Useful if you want to send a single table from say TOPCAT to vaex in a python console or notebook

vaex.**open_many**(filenames)
Open a list of filenames, and return a dataset with all datasets concatenated

Parameters filenames(list[str]) – list of filenames/paths

Return type Dataset

vaex.**server**(url, **kwargs)
Connect to hostname supporting the vaex web api

Parameters hostname(str) – hostname or ip address of server
Return `vaex.dataset.ServerRest` returns a server object, note that it does not connect to the server yet, so this will always succeed

Return type `ServerRest`

`vaex.example (download=True)`
Returns an example dataset which comes with vaex for testing/learning purposes

Return type `vaex.dataset.Dataset`

`vaex.app(*args, **kwargs)`
Create a vaex app, the QApplication mainloop must be started.

In ipython notebook/jupyter do the following: import vaex.ui.main # this causes the qt api level to be set properly
import vaex as xs Next cell: %gui qt Next cell app = vx.app()
From now on, you can run the app along with jupyter

`vaex.zeldovich(dim=2, N=256, n=-2.5, t=None, scale=1, seed=None)`
Creates a zeldovich dataset

`vaex.set_log_level_debug()`
set log level to debug

`vaex.set_log_level_info()`
set log level to info

`vaex.set_log_level_warning()`
set log level to warning

`vaex.set_log_level_exception()`
set log level to exception

`vaex.set_log_level_off()`
Disabled logging

`vaex.delayed(f)`

### 5.5 Dataset class

```python
class vaex.dataset.Dataset (name, column_names, executor=None)
```
All datasets are encapsulated in this class, local or remote datasets

Each dataset has a number of columns, and a number of rows, the length of the dataset.

The most common operations are: `Dataset.plot` >>> >>>

All Datasets have one 'selection', and all calculations by Subspace are done on the whole dataset (default) or for the selection. The following example shows how to use the selection.

```python
>>> some_dataset.select("x < 0")
>>> subspace_xy = some_dataset("x", "y")
>>> subspace_xy_selected = subspace_xy.selected()
```

TODO: active fraction, length and shuffled

```python
__call__(*expressions, **kwargs)
```
Alias/shortcut for `Dataset.subspace()`

### 5.5. Dataset class

39
__getitem__(item)
Convenient way to get expressions, (shallow) copies of a few columns, or to apply filtering

Examples:
>> ds[‘Lz’] # the expression ‘Lz’
>> ds[‘Lz/2’] # the expression ‘Lz/2’
>> ds[[‘Lz’, ‘E’]] # a shallow copy with just two columns
>> ds[ds.Lz < 0] # a shallow copy with the filter Lz < 0 applied

__init__(name, column_names, executor=None)
x.__init__(...) initializes x; see help(type(x)) for signature

__iter__()
Iterator over the column names

__len__()
Returns the number of rows in the dataset (filtering applied)

__setitem__(name, value)
Convenient way to add a virtual column / expression to this dataset

Examples:

```python
>>> ds['r'] = np.sqrt(ds.x**2 + ds.y**2 + ds.z**2)
```

__weakref__
list of weak references to the object (if defined)

add_column(name, f_or_array)
Add an in memory array as a column

add_column_healpix(name='healpix', longitude='ra', latitude='dec', degrees=True, healpix_order=12, nest=True)
Add a healpix (in memory) column based on a longitude and latitude

Parameters:
- **name** – Name of column
- **longitude** – longitude expression
- **latitude** – latitude expression (astronomical convention latitude=90 is north pole)
- **degrees** – If lon/lat are in degrees (default) or radians.
- **healpix_order** – healpix order, >= 0
- **nest** – Nested healpix (default) or ring.

add_variable(name, expression, overwrite=True)
Add a variable column to the dataset

Parameters:
- **str name**: name of virtual variable
- **expression**: expression for the variable

Variable may refer to other variables, and virtual columns and expression may refer to variables

Example:

```python
>>> dataset.add_variable("center")
>>> dataset.add_virtual_column("x_prime", "x-center")
>>> dataset.select("x_prime < 0")
```

add_virtual_column(name, expression, unique=False)
Add a virtual column to the dataset

Example:

```python
>>> dataset.add_virtual_column("r", "sqrt(x**2 + y**2 + z**2)")
>>> dataset.select("r < 10")
```
**Param** str name: name of virtual column

**Param** expression: expression for the column

**Parameters** unique (**str**) – if name is already used, make it unique by adding a postfix, e.g. \_1, or \_2

**add_virtual_columns_aioffe** (alpha, delta, x, y, radians=True)


**Parameters**

- alpha – azimuth angle
- delta – polar angle
- x – output name for x coordinate
- y – output name for y coordinate
- radians – input and output in radians (True), or degrees (False)

**Returns**

**add_virtual_columns_cartesian_to_polar** (x='x', y='y', radius_out='r_polar', azimuth_out='phi_polar', propagate_uncertainties=False, radians=False)

Convert cartesian to polar coordinates

**Parameters**

- x – expression for x
- y – expression for y
- radius_out – name for the virtual column for the radius
- azimuth_out – name for the virtual column for the azimuth angle
- propagate_uncertainties – {propagate_uncertainties}
- radians – if True, azimuth is in radians, defaults to degrees

**Returns**

**add_virtual_columns_cartesian_to_spherical** (x='x', y='y', z='z', alpha='l', delta='b', distance='distance', radians=False, center=None, center_name='solar_position')

Convert cartesian to spherical coordinates.

**Parameters**

- x –
- y –
- z –
- alpha –
- delta – name for polar angle, ranges from -90 to 90 (or -pi to pi when radians is True).
- distance –
- radians –
- center –
• center_name

Returns

add_virtual_columns_cartesian_velocities_to_polar
\( x='x', \ y='y', \ vx='vx', \ radius_polar=None, \ vy='vy', \ vr_out='vr_polar', \ vazimuth_out='vphi_polar', \ propagate_uncertainties=False \)

Convert cartesian to polar velocities.

Parameters

• \( x \) –
• \( y \) –
• \( vx \) –
• **radius_polar** – Optional expression for the radius, may lead to a better performance when given.
• \( vy \) –
• \( vr_out \) –
• \( vazimuth_out \) –
• **propagate_uncertainties** – {propagate_uncertainties}

Returns

add_virtual_columns_cartesian_velocities_to_spherical
\( x='x', \ y='y', \ z='z', \ vx='vx', \ vy='vy', \ vz='vz', \ vr='vr', \ vlong='vlong', \ vlat='vlat', \ distance=\)None\)

Concert velocities from a cartesian to a spherical coordinate system

TODO: errors

Parameters

• \( x \) – name of \( x \) column (input)
• \( y \) – \( y \)
• \( z \) – \( z \)
• \( vx \) – \( vx \)
• \( vy \) – \( vy \)
• \( vz \) – \( vz \)
• \( vr \) – name of the column for the radial velocity in the \( r \) direction (output)
• \( vlong \) – name of the column for the velocity component in the longitude direction (output)
• \( vlat \) – name of the column for the velocity component in the latitude direction, positive points to the north pole (output)
• **distance** – Expression for distance, if not given defaults to \( \sqrt{x^2+y^2+z^2} \), but if this column already exists, passing this expression may lead to a better performance
Returns

```python
def add_virtual_columns_matrix3d(x, y, z, xnew, ynew, znew, matrix, matrix_name='deprecated',
                                 matrix_is_expression=False, translation=[0, 0, 0], propagate_uncertainties=False):
```

Parameters

- **x** *(str)* – name of x column
- **y** *(str)* –
- **z** *(str)* –
- **xnew** *(str)* – name of transformed x column
- **ynew** *(str)* –
- **znew** *(str)* –
- **matrix** *(list[list]*) – 2d array or list, with [row,column] order
- **matrix_name** *(str)* –

Returns

```python
def add_virtual_columns_polar_velocities_to_cartesian(x='x', y='y', azimuth=None,
                                                      vr='vr_polar', vazimuth='vphi_polar',
                                                      vx_out='vx', vy_out='vy',
                                                      propagate_uncertainties=False):
```

Convert cylindrical polar velocities to Cartesian.

Parameters

- **x** –
- **y** –
- **azimuth** – Optional expression for the azimuth in degrees, may lead to a better performance when given.
- **vr** –
- **vazimuth** –
- **vx_out** –
- **vy_out** –
- **propagate_uncertainties** – {propagate_uncertainties}

```python
def add_virtual_columns_rotation(x, y, xnew, ynew, angle_degrees, propagate_uncertainties=False):
```

Rotation in 2d

Parameters

- **x** *(str)* – Name/expression of x column
- **y** *(str)* – idem for y
- **xnew** *(str)* – name of transformed x column
- **ynew** *(str)* –
- **angle_degrees** *(float)* – rotation in degrees, anti clockwise
Returns

```
add_virtual_columns_spherical_to_cartesian(alpha, delta, distance, xname='x',
yname='y', zname='z', propagate_uncertainties=False, center=[0, 0, 0], center_name='solar_position', radians=False)
```

Convert spherical to cartesian coordinates.

Parameters

- **alpha** –
- **delta** – polar angle, ranging from the -90 (south pole) to 90 (north pole)
- **distance** – radial distance, determines the units of x, y and z
- **xname** –
- **yname** –
- **zname** –
- **propagate_uncertainties** – If true, will propagate errors for the new virtual columns, see :py:`Dataset.propagate_uncertainties` for details
- **center** –
- **center_name** –
- **radians** –

Returns

```
byte_size(selection=False)
```

Return the size in bytes the whole dataset requires (or the selection), respecting the active_fraction

```
classmethod can_open(path, *args, **kwargs)
```

Tests if this class can open the file given by path

```
close_files()
```

Close any possible open file handles, the dataset will not be in a usable state afterwards

```
col
```

Gives direct access to the data as numpy-like arrays.

Convenient when working with ipython in combination with small datasets, since this gives tab-completion

Columns can be accessed by their names, which are attributes. The attributes are currently strings, so you cannot do computations with them

Example

```
>>> ds = vx.example()
>>> ds.plot(ds.col.x, ds.col.y)
```

```
column_count()
```

Returns the number of columns, not counting virtual ones

```
combinations(expressions_list=None, dimension=2, exclude=None, **kwargs)
```

Generate a list of combinations for the possible expressions for the given dimension

Parameters

- **expressions_list** – list of list of expressions, where the inner list defines the subspace
• **dimensions** – if given, generates a subspace with all possible combinations for that dimension

• **exclude** – list of

**correlation**(x, y=None, binby=[], limits=None, shape=128, sort=False, sort_key=<ufunc 'absolute'>, selection=False, delay=False, progress=None)
Calculate the correlation coefficient \( \frac{\text{cov}[x,y]}{\text{std}[x] \times \text{std}[y]} \) between and x and y, possible on a grid defined by binby

Examples:

```python
>>> ds.correlation("\(x^{2}+y^{2}+z^{2}\)", "-\log(\(-E+1\))")
array(0.6366637382215669)
```

```python
>>> ds.correlation("\(x^{2}+y^{2}+z^{2}\)", "-\log(\(-E+1\))", binby="Lz", shape=4)
array([0.40594394, 0.69868851, 0.61394099, 0.65266318])
```

**Parameters**

• **x** – expression or list of expressions, e.g. ‘x’, or [‘x’, ‘y’]

• **y** – expression or list of expressions, e.g. ‘x’, or [‘x’, ‘y’]

• **binby** – List of expressions for constructing a binned grid

• **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]

• **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]

• **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections

• **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

• **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

**Returns** Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

**count**(expression=None, binby=[], limits=None, shape=128, selection=False, delay=False, edges=False, progress=None)
Count the number of non-NaN values (or all, if expression is None or “*”)

Examples:

```python
>>> ds.count()
330000.0
```

```python
>>> ds.count("*")
330000.0
```

```python
>>> ds.count("*", binby="x", shape=4)
array([10925., 155427., 152007., 10748.])
```

**Parameters**

• **expression** – Expression or column for which to count non-missing values, or None or ‘*’ for counting the rows

• **binby** – List of expressions for constructing a binned grid
• **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]

• **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]

• **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections

• **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

• **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

• **edges** – Currently for internal use only (it includes nan’s and values outside the limits at borders, nan and 0, smaller than at 1, and larger at -1

**Returns** Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

\[
\text{cov}(x, y=\text{None}, \text{binby}=[], \text{limits}=\text{None}, \text{shape}=128, \text{selection}=\text{False}, \text{delay}=\text{False}, \text{progress}=\text{None})
\]

Calculate the covariance matrix for x and y or more expressions, possible on a grid defined by binby

Either x and y are expressions, e.g:

```python
>>> ds.cov("x", "y")
```

Or only the x argument is given with a list of expressions, e.g.:

```python
>> ds.cov(["x", "y", "z"])
```

Examples:

```python
>>> ds.cov("x", "y")
array([[ 53.54521742, -3.8123135],
      [-3.8123135, 60.62257881]])

>>> ds.cov(["x", "y", "z"])
array([[ 53.54521742, -3.8123135, -0.98260511],
      [-3.8123135, 60.62257881, 1.21381057],
      [-0.98260511, 1.21381057, 25.55517638]])
```

```python
>>> ds.cov("x", "y", binby="E", shape=2)
array([[ 9.74852878e+00, -3.02004780e-02],
      [ 8.43996546e+01, -6.51984181e+00],
      [ 6.22578811e+01,  1.21381057e+02],
      [ 3.54521742e+01, -1.98260511e+01],
      [ 5.55517638e+01,  2.55551764e+02]])
```

```python
>> ds.cov("x", "y", binby="E", shape=2)
array([[ 3.02004780e-02,  9.99288215e+00],
      [ 3.0204780e-02,  9.99288215e+00],
      [ 3.0204780e-02,  9.99288215e+00],
      [ 9.74852878e+00,  9.99288215e+00],
      [ 3.02004780e-02,  9.99288215e+00]])
```

**param x** expression or list of expressions, e.g. ‘x’, or ['x', 'y']

**param y** if previous argument is not a list, this argument should be given

**param binby** List of expressions for constructing a binned grid

**param limits** description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]

**param shape** shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]

**param selection** Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections
param delay  Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

return  Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic, the last dimensions are of shape (2,2)

covar  (x, y, binby=[], limits=None, shape=128, selection=False, delay=False, progress=None)
Calculate the covariance $\text{cov}[x,y]$ between and x and y, possible on a grid defined by binby

Examples:

```python
>>> ds.covar("x**2+y**2+z**2", "-\log(-E+1)")
array(52.69461456005138)
>>> ds.covar("x**2+y**2+z**2", "-\log(-E+1)")/(ds.std("x**2+y**2+z**2") * ds.
˓std("-\log(-E+1)"))
0.63666373822156686
>>> ds.covar("x**2+y**2+z**2", "-\log(-E+1)", binby="Lz", shape=4)
array([ 10.17387143, 51.94954078, 51.24902796, 20.2163929 ])
```

Parameters

- **x** – expression or list of expressions, e.g. ‘x’, or ['x', 'y']
- **y** – expression or list of expressions, e.g. ‘x’, or ['x', 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns  Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

delete_variable (name)
Deletes a variable from a dataset

delete_virtual_column (name)
Deletes a virtual column from a dataset

dropna  (drop_nan=True, drop_masked=True, column_names=None)
Create a shallow copy dataset, with filtering set using select_non_missing

Parameters

- **drop_nan** – drop rows when there is a NaN in any of the columns (will only affect float values)
- **drop_masked** – drop rows when there is a masked value in any of the columns
- **column_names** – The columns to consider, default: all (real, non-virtual) columns

Returns  Dataset
**evaluate** *(expression, i1=None, i2=None, out=None, selection=None)*

Evaluate an expression, and return a numpy array with the results for the full column or a part of it.

Note that this is not how vaex should be used, since it means a copy of the data needs to fit in memory.

To get partial results, use i1 and i2.

**Parameters**

- **expression** *(str)* – Name/expression to evaluate
- **i1** *(int)* – Start row index, default is the start (0)
- **i2** *(int)* – End row index, default is the length of the dataset
- **out** *(ndarray)* – Output array, to which the result may be written (may be used to reuse an array, or write to a memory mapped array)

**evaluate_variable** *(name)*

Evaluates the variable given by name

**execute** ()

Execute all delayed jobs

**extract** ()

Return a dataset containing only the filtered rows.

Note that no copy of the underlying data is made, only a view/reference is make.

The resulting dataset may be more efficient to work with when the original dataset is heavily filtered (contains just a small number of rows).

If no filtering is applied, it returns a trimmed view. For returned datasets, len(ds) == ds.length_original() == ds.length_unfiltered()

**fillna** *(value, fill_nan=True, fill_masked=True, column_names=None, prefix='__original__', inplace=False)*

Return a dataset, where missing values/NaN are filled with 'value'

Note that no copy of the underlying data is made, only a view/reference is make.

Note that filtering will be ignored (since they may change), you may want to consider running :py:`Dataset.extract` first.

**Example:**

```python
def main():
    a = np.array(['a', 'b', 'c'])
    x = np.arange(1,4)
    ds = vaex.from_arrays(a=a, x=x)
    ds.sort('(x-1.8)**2', ascending=False)  # b, c, a will be the order

if __name__ == '__main__':
    main()
```

**Parameters**

- **or expression by** *(str)* – expression to sort by
- **ascending** *(bool)* – ascending (default, True) or descending (False)
- **kind** *(str)* – kind of algorithm to use (passed to numpy.argsort)
- **inplace** – Make modifications to self or return a new dataset
get_active_fraction()
Value in the range (0, 1], to work only with a subset of rows

get_column_names(virtual=False, hidden=False, strings=False)
Return a list of column names

Parameters

• virtual – If True, also return virtual columns
• hidden – If True, also return hidden columns

Return type list of str

get_current_row()
Individual rows can be ‘picked’, this is the index (integer) of the current row, or None there is nothing
picked

get_private_dir(create=False)
Each datasets has a directory where files are stored for metadata etc

Example

```python
>>> import vaex as vx
>>> ds = vx.example()
>>> ds.get_private_dir()
'/Users/users/breddels/.vaex/datasets/_Users_users_breddels_vaex-testing_data_
˓→helmi-dezeeuw-2000-10p.hdf5'
```

Parameters create (bool) – is True, it will create the directory if it does not exist

get_selection(name='default')
Get the current selection object (mostly for internal use atm)

get_variable(name)
Returns the variable given by name, it will not evaluate it.

For evaluation, see Dataset.evaluate_variable(), see also Dataset.set_variable()

has_current_row()
Returns True/False is there currently is a picked row

has_selection(name='default')
Returns True of there is a selection

healpix_count(expression=None, healpix_expression=None, healpix_max_level=12,
healpix_level=8, binby=None, limits=None, shape=128, delay=False,
progress=None, selection=None)
Count non missing value for expression on an array which represents healpix data.

Parameters

• expression – Expression or column for which to count non-missing values, or None
  or '*' for counting the rows
• healpix_expression – (healpix_max_level)
• healpix_max_level – (healpix_max_level)
• healpix_level – (healpix_level)
• binby – (binby), these dimension follow the first healpix dimension.
• limits – (limits)
vaex Documentation

- **shape** – {shape}
- **selection** – {selection}
- **delay** – {delay}
- **progress** – {progress}

**Returns**

`healpix_plot(healpix_expression='source_id/34359738368', healpix_max_level=12, healpix_level=8, what='count(*)', selection=None, grid=None, healpix_input='equatorial', healpix_output='galactic', f=None, colormap='afmhot', grid_limits=None, image_size=800, nest=True, figsize=None, interactive=False, title=None, smooth=None, show=False, colorbar=True, rotation=(0, 0, 0), **kwargs)`

**Parameters**

- **healpix_expression** – {healpix_max_level}
- **healpix_max_level** – {healpix_max_level}
- **healpix_level** – {healpix_level}
- **what** – {what}
- **selection** – {selection}
- **grid** – {grid}
- **healpix_input** – Specificy if the healpix index is in “equatorial”, “galactic” or “ecliptic”.
- **healpix_output** – Plot in “equatorial”, “galactic” or “ecliptic”.
- **f** – function to apply to the data
- **colormap** – matplotlib colormap
- **grid_limits** – Optional sequence [minvalue, maxvalue] that determine the min and max value that map to the colormap (values below and above these are clipped to the min/max). (default is [min(f(grid)), max(f(grid))])
- **image_size** – size for the image that healpy uses for rendering
- **nest** – If the healpix data is in nested (True) or ring (False)
- **figsize** – If given, modify the matplotlib figure size. Example (14,9)
- **interactive** – (Experimental, uses healpy.mollzoom is True)
- **title** – Title of figure
- **smooth** – apply gaussian smoothing, in degrees
- **show** – Call matplotlib’s show (True) or not (False, default)
- **rotation** – Rotate the plot, in format (lon, lat, psi) such that (lon, lat) is the center, and rotate on the screen by angle psi. All angles are degrees.

**Returns**

`is_local()`

Returns True if the dataset is a local dataset, False when a remote dataset

`iscategory(column)`

Returns true if column is a category
length_original()
the full length of the dataset, independent what active_fraction is, or filtering. This is the real length of the
underlying ndarrays

length_unfiltered()
The length of the arrays that should be considered (respecting active range), but without filtering

limits(expression, value=None, square=False, selection=None, delay=False, shape=None)
Calculate the [min, max] range for expression, as described by value, which is ‘99.7%’ by default.
If value is a list of the form [minvalue, maxvalue], it is simply returned, this is for convenience when using
mixed forms.

Example:

```python
ds.limits("x")
array([-28.86381927, 28.9261226 ])
ds.limits(["x", "y"])
(array([-28.86381927, 28.9261226 ]), array([-28.60476934, 28.96535249]))
ds.limits(["x", "y", "minmax"])
(array([-128.293991, 271.365997]), array([ -71.5523682, 146.465836 ]))
ds.limits(["x", "y", ["minmax", "90%"]]
(array([-128.293991, 271.365997]), array([-13.37438402, 13.4224423 ]))
ds.limits(["x", "y", ["minmax", [0, 10]]]
(array([-128.293991, 271.365997]), [0, 10])
```

Parameters

- **expression** – expression or list of expressions, e.g. ‘x’, or [‘x’, ‘y’]
- **value** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]
- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

Returns List in the form [[xmin, xmax], [ymin, ymax], . . . , [zmin, zmax]] or [xmin, xmax]
when expression is not a list

limits_percentage(expression, percentage=99.73, square=False, delay=False)
Calculate the [min, max] range for expression, containing approximately a percentage of the data as defined
by percentage.
The range is symmetric around the median, i.e., for a percentage of 90, this gives the same results as:

```python
ds.limits_percentage("x", 90)
array([-12.35081376, 12.14858052])
ds.percentile_approx("x", 5), ds.percentile_approx("x", 95)
(array([-12.36813152]), array([ 12.13275818]))
```

NOTE: this value is approximated by calculating the cumulative distribution on a grid. NOTE 2: The
values above are not exactly the same, since percentile and limits_percentage do not share the same code

Parameters

- **expression** – expression or list of expressions, e.g. ‘x’, or [‘x’, ‘y’]
- **percentage** (float) – Value between 0 and 100

5.5. Dataset class 51
• **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

**Returns** List in the form \([\text{xmin}, \text{xmax}], [\text{ymin}, \text{ymax}], \ldots, [\text{zmin}, \text{zmax}]\) or \([\text{xmin}, \text{xmax}]\) when expression is not a list

**materialize** (*virtual_column*, *inplace=False*)
Returns a new dataset where the virtual column is turned into an in memory numpy array

**Example:**

```python
>>> x = np.arange(1, 4)
>>> y = np.arange(2, 5)
>>> ds = vaex.from_arrays(x=x, y=y)
>>> ds['r'] = (ds.x**2 + ds.y**2)**0.5  # 'r' is a virtual column
→ (computed on the fly)
>>> ds = ds.materialize('r')  # now 'r' is a 'real' column (i.e. a numpy array)
```

**Parameters**

* inplace – {inplace}

**max** (*expression*, *binby=[], limits=None, shape=128, selection=False, delay=False, progress=None*)
Calculate the maximum for given expressions, possible on a grid defined by binby

**Example:**

```python
>>> ds.max("x")
array(271.365997)
>>> ds.max(['x', 'y'])
array([271.365997, 146.465836])
>>> ds.max("x", binby="x", shape=5, limits=[-10, 10])
array([-6.00010443, -2.00002384, 1.99998057, 5.99983597, 9.99984646])
```

**Parameters**

• **expression** – expression or list of expressions, e.g. ‘x’, or [‘x’, ‘y’]

• **binby** – List of expressions for constructing a binned grid

• **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]

• **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]

• **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections

• **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

• **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

**Returns** Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic, the last dimension is of shape (2)

**mean** (*expression*, *binby=[], limits=None, shape=128, selection=False, delay=False, progress=None*)
Calculate the mean for expression, possibly on a grid defined by binby.

**Examples:**

---

52 Chapter 5. API documentation for vaex library
>>> ds.mean("x")
-0.067131491264005971
>>> ds.mean("(x**2+y**2)**0.5", binby="E", shape=4)
array([ 2.43483742, 4.41840721, 8.26742458, 15.53846476])

Parameters

- **expression** – expression or list of expressions, e.g. ‘x’, or ['x', 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

median_approx(expression, percentage=50.0, binby=[], limits=None, shape=128, percentile_shape=256, percentile_limits='minmax', selection=False, delay=False)

Calculate the median, possible on a grid defined by binby

NOTE: this value is approximated by calculating the cumulative distribution on a grid defined by percentile_shape and percentile_limits

Parameters

- **expression** – expression or list of expressions, e.g. ‘x’, or ['x', 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **percentile_limits** – description for the min and max values to use for the cumulative histogram, should currently only be ‘minmax’
- **percentile_shape** – shape for the array where the cumulative histogram is calculated on, integer type
- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic
**min** *(expression, binby=[], limits=None, shape=128, selection=False, delay=False, progress=None)*

Calculate the minimum for given expressions, possible on a grid defined by binby

Example:

```python
>>> ds.min("x")
array(-128.293991)
>>> ds.min(['x', "y"])
array([-128.293991, -71.5523682])
>>> ds.min("x", binby="x", shape=5, limits=[-10, 10])
array([-9.99919128, -5.99972439, -1.99991322, 2.0000093, 6.0004878])
```

Parameters

- **expression** – expression or list of expressions, e.g. ‘x’, or ['x', 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

**Returns** Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic, the last dimension is of shape (2)

**minmax** *(expression, binby=[], limits=None, shape=128, selection=False, delay=False, progress=None)*

Calculate the minimum and maximum for expressions, possible on a grid defined by binby

Example:

```python
>>> ds.minmax("x")
array([-128.293991, 271.365997])
>>> ds.minmax(['x', "y"])
array([[-128.293991, 271.365997], [-71.5523682, 146.465836]])
>>> ds.minmax("x", binby="x", shape=5, limits=[-10, 10])
array([[-9.99919128, -6.00010443], [-5.99972439, -2.00002384], [-1.99991322, 1.99998057], [2.0000093, 5.99983597], [6.0004878, 9.99984646]])
```

Parameters

- **expression** – expression or list of expressions, e.g. ‘x’, or ['x', 'y']
- **binby** – List of expressions for constructing a binned grid
• **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]

• **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]

• **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections

• **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

• **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

**Returns** Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic, the last dimension is of shape (2)

```python
mutual_information(x, y=None, mi_limits=None, mi_shape=256, binby=[], limits=None, shape=128, sort=False, selection=False, delay=False)
```

Estimate the mutual information between and x and y on a grid with shape mi_shape and mi_limits, possible on a grid defined by binby

If sort is True, the mutual information is returned in sorted (descending) order and the list of expressions is returned in the same order

Examples:

```python
>>> ds.mutual_information("x", "y")
array(0.1511814526380327)
>>> ds.mutual_information(["x", "y"], ["x", "z"], ["E", "Lz"])
array([ 0.15118145, 0.18439181, 1.07067379])
>>> ds.mutual_information(["x", "y"], ["x", "z"], ["E", "Lz"], sort=True)
(array([ 1.07067379, 0.18439181, 0.15118145]),
[['E', 'Lz'], ['x', 'z'], ['x', 'y']])
```

**Parameters**

• **x** – expression or list of expressions, e.g. ‘x’, or [‘x’, ‘y’]

• **y** – expression or list of expressions, e.g. ‘x’, or [‘x’, ‘y’]

• **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]

• **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]

• **binby** – List of expressions for constructing a binned grid

• **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]

• **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]

• **sort** – return mutual information in sorted (descending) order, and also return the correspond list of expressions when sorted is True

• **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections
• **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

**Returns**  Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic,

```
percentile_approx(expression, percentage=50.0, binby=[], limits=None, shape=128, percentile_shape=1024, percentile_limits='minmax', selection=False, delay=False)
```

Calculate the percentile given by percentage, possible on a grid defined by binby

NOTE: this value is approximated by calculating the cumulative distribution on a grid defined by `percentile_shape` and `percentile_limits`

```
>>> ds.percentile_approx("x", 10), ds.percentile_approx("x", 90)
(array([-8.3220355]), array([ 7.92080358]))
>>> ds.percentile_approx("x", 50, binby="x", shape=5, limits=[-10, 10])
array([-7.56462982, -3.61036641, -0.01296306,  3.56697863,  7.45838367])
```

0:1:0.1 1:1:0.2 2:1:0.3 3:1:0.4 4:1:0.5
5:1:0.6 6:1:0.7 7:1:0.8 8:1:0.9 9:1:1.0

**Parameters**

- **expression** – expression or list of expressions, e.g. ‘x’, or [‘x’, ‘y’]
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. `shape=128, shape=[128, 256]`
- **percentile_limits** – description for the min and max values to use for the cumulative histogram, should currently only be ‘minmax’
- **percentile_shape** – shape for the array where the cumulative histogram is calculated on, integer type
- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

**Returns**  Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

```
plot3d(x, y, z, vx=None, vy=None, vz=None, vwhat=None, limits=None, grid=None, what='count(*)', shape=128, selection=[None, True], f=None, vcount_limits=None, smooth_pre=None, smooth_post=None, grid_limits=None, normalize='normalize', figure_key=None, fig=None, lighting=True, level=[0.1, 0.5, 0.9], opacity=[0.01, 0.05, 0.1], level_width=0.1, show=True, **kwargs)
```

Use at own risk, requires ipyvolume
propagate_uncertainties(columns, depending_variables=None, cov_matrix='auto',
covariance_format='{}{}'.format, uncertainty_format='{}{}'.format)

Propagates uncertainties (full covariance matrix) for a set of virtual columns.

Covariance matrix of the depending variables is guessed by finding columns prefixed by: ‘e’ or ‘e_’ or postfixed by ‘_error’, ‘_uncertainty’, ‘e’ and ‘_e’. Off diagonals (covariance or correlation) by postfixes with ‘_correlation’ or ‘_corr’ for correlation or ‘_covariance’ or ‘_cov’ for covariances. (Note that x_y_cov = x_e * y_e * x_y_correlation)

Example:

```python
def propagate_uncertainties(ds):
    from vaex import propagate_uncertainties
    propagate_uncertainties(ds, columns=ds.columns)
```

Parameters

- **columns** – list of columns for which to calculate the covariance matrix.
- **depending_variables** – If not given, it is found out automatically, otherwise a list of columns which have uncertainties.
- **cov_matrix** – List of list with expressions giving the covariance matrix, in the same order as depending_variables. If ‘full’ or ‘auto’, the covariance matrix for the depending_variables will be guessed, where ‘full’ gives an error if an entry was not found.

remove_virtual_meta()

Removes the file with the virtual column etc, it does not change the current virtual columns etc

rename_column(name, new_name, unique=False, store_in_state=True)

Renames a column, not this is only the in memory name, this will not be reflected on disk

sample(n=None, frac=None, replace=False, weights=None, random_state=None)

Returns a dataset with a random set of rows

Note that no copy of the underlying data is made, only a view/reference is make.

Provide either n or frac.

Parameters

- **n** (*int*) – number of samples to take (default 1 if frac is None)
- **frac** (*float*) – fractional number of takes to take
- **replace** (*bool*) – If true, a row may be drawn multiple times
- **or expression weights** (*str*) – (unnormalized) probability that a row can be drawn
- **or RandomState** (*int*) – seed or RandomState for reproducability, when None a random seed it chosen

Example:

```python
>>> a = np.array(['a', 'b', 'c'])
>>> x = np.arange(1,4)
>>> ds = vaex.from_arrays(a=a, x=x)
```
select (boolean_expression, mode='replace', name='default', executor=None)
Perform a selection, defined by the boolean expression, and combined with the previous selection using the given mode

Selections are recorded in a history tree, per name, undo/redo can be done for them separately

Parameters
- **boolean_expression** (str) – Any valid column expression, with comparison operators
- **mode** (str) – Possible boolean operator: replace/and/or/xor/subtract
- **name** (str) – history tree or selection ‘slot’ to use
- **executor** –

Returns

select_box (spaces, limits, mode='replace', name='default')
Select a n-dimensional rectangular box bounded by limits

The following examples are equivalent:

```python
>>> ds.select_box(['x', 'y'], [(0, 10), (0, 1)])
>>> ds.select_rectangle('x', 'y', [(0, 10), (0, 1)])
```

:param spaces: list of expressions
:param limits: sequence of shape [(x1, x2), (y1, y2)]
:param mode: :
:param name: :
:return:

select_circle (x, y, xc, yc, r, mode='replace', name='default', inclusive=True)
Select a circular region centred on xc, yc, with a radius of r.

Parameters
- **x** – expression for the x space
- **y** – expression for the y space
- **xc** – location of the centre of the circle in x
- **yc** – location of the centre of the circle in y
- **r** – the radius of the circle
- **name** – name of the selection
- **mode** –

Returns

Example: >>> ds.select_circle('x','y',2,3,1)

select_ellipse (x, y, xc, yc, width, height, angle=0, mode='replace', name='default', radius=False, inclusive=True)
Select an elliptical region centred on xc, yc, with a certain width, height and angle.

Parameters
- **x** – expression for the x space
- **y** – expression for the y space
- **xc** – location of the centre of the ellipse in x
• **yc** – location of the centre of the ellipse in y
• **width** – the width of the ellipse (diameter)
• **height** – the width of the ellipse (diameter)
• **angle** – (degrees) orientation of the ellipse, counter-clockwise measured from the y axis
• **name** – name of the selection
• **mode** –

**Returns**

Example: >>> ds.select_ellipse('x', 'y', 2, -1, 5, 1, 30, name='my_ellipse')

**select_inverse** (name='default', executor=None)

Invert the selection, i.e. what is selected will not be, and vice versa

**Parameters**

• **name** *(str)*
• **executor** –

**Returns**

**select_lasso** (expression_x, expression_y, xsequence, ysequence, mode='replace', name='default', executor=None)

For performance reasons, a lasso selection is handled differently.

**Parameters**

• **expression_x** *(str)* – Name/expression for the x coordinate
• **expression_y** *(str)* – Name/expression for the y coordinate
• **xsequence** – list of x numbers defining the lasso, together with y
• **ysequence** –
• **mode** *(str)* – Possible boolean operator: replace/and/or/xor/subtract
• **name** *(str)*
• **executor** –

**Returns**

**select_non_missing** (drop_nan=True, drop_masked=True, column_names=None, mode='replace', name='default')

Create a selection that selects rows having non missing values for all columns in column_names

The name reflect Panda’s, no rows are really dropped, but a mask is kept to keep track of the selection

**Parameters**

• **drop_nan** – drop rows when there is a NaN in any of the columns (will only affect float values)
• **drop_masked** – drop rows when there is a masked value in any of the columns
• **column_names** – The columns to consider, default: all (real, non-virtual) columns
• **mode** *(str)* – Possible boolean operator: replace/and/or/xor/subtract
• **name** *(str)* – history tree or selection ‘slot’ to use

**Returns**
select_nothing (name='default')
Select nothing

select_rectangle (x, y, limits, mode='replace', name='default')
Select a 2d rectangular box in the space given by x and y, bounds by limits
Example: >>> ds.select_box('x', 'y', [(0, 10), (0, 1)])

Parameters
• x – expression for the x space
• y – expression fo the y space
• limits – sequence of shape [(x1, x2), (y1, y2)]
• mode –

Returns

selected_length ()
Returns the number of rows that are selected

selection_can_redo (name='default')
Can selection name be redone?

selection_can_undo (name='default')
Can selection name be undone?

selection_redo (name='default', executor=None)
Redo selection, for the name

selection_undo (name='default', executor=None)
Undo selection, for the name

set_active_fraction (value)
Sets the active_fraction, set picked row to None, and remove selection
TODO: we may be able to keep the selection, if we keep the expression, and also the picked row

set_active_range (i1, i2)
Sets the active_fraction, set picked row to None, and remove selection
TODO: we may be able to keep the selection, if we keep the expression, and also the picked row

set_current_row (value)
Set the current row, and emit the signal signal_pick

set_selection (selection, name='default', executor=None)
Sets the selection object

Parameters
• selection – Selection object
• name – selection ‘slot’
• executor –

Returns

set_variable (name, expression_or_value, write=True)
Set the variable to an expression or value defined by expression_or_value

Example
```python
>>> ds.set_variable("a", 2.)
>>> ds.set_variable("b", "a**2")
>>> ds.get_variable("b")
'a**2'
>>> ds.evaluate_variable("b")
4.0
```

### Parameters

- **name** – Name of the variable
- **write** – write variable to meta file
- **expression** – value or expression

### sort (by, ascending=True, kind='quicksort')

Return a sorted dataset, sorted by the expression ‘by’

Note that no copy of the underlying data is made, only a view/reference is make.

Note that filtering will be ignored (since they may change), you may want to consider running

```python
ds = Dataset.extract
```

#### Example:

```python
>>> a = np.array(['a', 'b', 'c'])
>>> x = np.arange(1,4)
>>> ds = vaex.from_arrays(a=a, x=x)
>>> ds.sort('(x-1.8)**2', ascending=False)  # b, c, a will be the order of a
```

### Parameters

- **or expression by (str)** – expression to sort by
- **ascending (bool)** – ascending (default, True) or descending (False)
- **kind (str)** – kind of algorithm to use (passed to numpy.argsort)

### std(expression, binby=[], limits=None, shape=128, selection=False, delay=False, progress=None)

Calculate the standard deviation for the given expression, possible on a grid defined by binby

```python
>>> ds.std("vz")
110.31773397535071
>>> ds.std("vz", binby=['(x**2+y**2)**0.5'], shape=4)
array([ 123.57954851, 85.35190177, 61.14345748, 38.0740619 ])
```

### Parameters

- **expression** – expression or list of expressions, e.g. ‘x’, or [‘x’, ‘y’]
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections
• **delay** – Do not return the result, but a proxy for delayronous calculations (currently only for internal use)

• **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

**Returns** Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

```python
subspace(*expressions, **kwargs)
```

Return a Subspace for this dataset with the given expressions:

**Example:**

```python
>>> subspace_xy = some_dataset("x", "y")
```

**Return type** Subspace

**Parameters**

• **expressions** (`list[str]`) – list of expressions

• **kwargs** –

**Returns**

```python
subspaces(expressions_list=None, dimensions=None, exclude=None, **kwargs)
```

Generate a Subspaces object, based on a custom list of expressions or all possible combinations based on dimension

**Parameters**

• **expressions_list** – list of list of expressions, where the inner list defines the subspace

• **dimensions** – if given, generates a subspace with all possible combinations for that dimension

• **exclude** – list of

** Examples:**

```python
>>> ds.sum("L")
304054882.49378014
>>> ds.sum("L", binby="E", shape=4)
array([[ 8.83517994e+06, 5.92217598e+07, 9.55218726e+07,
        1.40008776e+08]])
```

**Parameters**

• **expression** – expression or list of expressions, e.g. ‘x’, or [‘x’, ‘y’]

• **binby** – List of expressions for constructing a binned grid

• **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]

• **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
• **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections

• **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

• **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

**Returns** Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

**take** *(indices)*

Returns a dataset containing only rows indexed by indices

Note that no copy of the underlying data is made, only a view/reference is make.

**Example:**

```python
>>> a = np.array(['a', 'b', 'c'])
>>> x = np.arange(1,4)
>>> ds = vaex.from_arrays(a=a, x=x)
>>> ds.take([0,2])
```

**to_astropy_table** *(column_names=None, selection=None, strings=True, virtual=False, index=None)*

Returns a astropy table object containing the ndarrays corresponding to the evaluated data

**Parameters**

- **column_names** – list of column names, to export, when None Dataset.get_column_names(strings=strings, virtual=virtual) is used

- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections

- **strings** – argument passed to Dataset.get_column_names when column_names is None

- **virtual** – argument passed to Dataset.get_column_names when column_names is None

- **index** – if this column is given it is used for the index of the DataFrame

**Returns** astropy.table.Table object

**to_copy** *(column_names=None, selection=None, strings=True, virtual=False, selections=True)*

Return a copy of the Dataset, if selection is None, it does not copy the data, it just has a reference

**Parameters**

- **column_names** – list of column names, to copy, when None Dataset.get_column_names(strings=strings, virtual=virtual) is used

- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections

- **strings** – argument passed to Dataset.get_column_names when column_names is None

- **virtual** – argument passed to Dataset.get_column_names when column_names is None

- **selections** – copy selections to new dataset

**Returns** dict

**to_dict** *(column_names=None, selection=None, strings=True, virtual=False)*

Return a dict containing the ndarray corresponding to the evaluated data
Parameters

- **column_names** – list of column names, to export, when `None`
  `Dataset.get_column_names(strings=strings, virtual=virtual)` is used

- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is `None` or `False`), or a list of selections

- **strings** – argument passed to `Dataset.get_column_names` when `column_names` is `None`

- **virtual** – argument passed to `Dataset.get_column_names` when `column_names` is `None`

Returns  dict
to_items (`column_names=None, selection=None, strings=True, virtual=False`)  
Return a list of [(column_name, ndarray), …] pairs where the ndarray corresponds to the evaluated data

Parameters

- **column_names** – list of column names, to export, when `None`
  `Dataset.get_column_names(strings=strings, virtual=virtual)` is used

- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is `None` or `False`), or a list of selections

- **strings** – argument passed to `Dataset.get_column_names` when `column_names` is `None`

- **virtual** – argument passed to `Dataset.get_column_names` when `column_names` is `None`

Returns  list of (name, ndarray) pairs
to_pandas_df (`column_names=None, selection=None, strings=True, virtual=False, index_name=None`)  
Return a pandas DataFrame containing the ndarray corresponding to the evaluated data

If index is given, that column is used for the index of the dataframe.

Example

```python
>>> df = ds.to_pandas_df(['x', 'y', 'z'])
>>> ds_copy = vx.from_pandas(df)
```

Parameters

- **column_names** – list of column names, to export, when `None`
  `Dataset.get_column_names(strings=strings, virtual=virtual)` is used

- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is `None` or `False`), or a list of selections

- **strings** – argument passed to `Dataset.get_column_names` when `column_names` is `None`

- **virtual** – argument passed to `Dataset.get_column_names` when `column_names` is `None`

- **index_column** – if this column is given it is used for the index of the DataFrame

Returns  pandas.DataFrame object

trim (`inplace=False`)  
Return a dataset, where all columns are ‘trimmed’ by the active range.

For returned datasets, `ds.get_active_range()` returns (0, `ds.length_original()`).

Note that no copy of the underlying data is made, only a view/reference is made.

Parameters  inplace – Make modifications to self or return a new dataset
**ucd_find** (*ucds, exclude=[]*)  
Find a set of columns (names) which have the ucd, or part of the ucd  
Prefix with a ^, it will only match the first part of the ucd  

**Example**
```python
>>> dataset.ucd_find('pos.eq.ra', 'pos.eq.dec')
['RA', 'DEC']
>>> dataset.ucd_find('pos.eq.ra', 'doesnotexist')
>>> dataset.ucds[dataset.ucd_find('pos.eq.ra')]  
'pos.eq raj meta.main'
>>> dataset.ucd_find('meta.main')
'dec'
>>> dataset.ucd_find('^meta.main')
```

**unit** (*expression, default=None*)  
Returns the unit (an astropy.unit.Units object) for the expression  

**Example**
```python
>>> import vaex as vx
>>> ds = vx.example()
>>> ds.unit("x")
Unit("kpc")
>>> ds.unit("x*L")
Unit("km kpc2 / s")
```

**Parameters**
- **expression** – Expression, which can be a column name  
- **default** – if no unit is known, it will return this

**Returns** The resulting unit of the expression  

**Return type** astropy.units.Unit

**update_meta** ()  
Will read back the ucd, descriptions, units etc, written by `Dataset.write_meta()`. This will be done when opening a dataset.  

**update_virtual_meta** ()  
Will read back the virtual column etc, written by `Dataset.write_virtual_meta()`. This will be done when opening a dataset.

**validate_expression** (*expression*)  
Validate an expression (may throw Exceptions)

**var** (*expression, binby=[], limits=None, shape=128, selection=False, delay=False, progress=None*)  
Calculate the sample variance for the given expression, possible on a grid defined by binby

**Examples:**
```python
>>> ds.var("vz")
12170.002429456246
>>> ds.var("vz", binby="(x**2+y**2)**0.5", shape=4)  
array([15271.90481083, 7284.94713504, 3738.52239232, 1449.63418988])
>>> ds.var("vz", binby="(x**2+y**2)**0.5", shape=4)**0.5
```

(continues on next page)
array([[ 123.57954851,  85.35190177,  61.14345748,  38.0740619 ]])

>>> ds.std("vz", binby=["(x**2+y**2)**0.5"], shape=4)
array([[ 123.57954851,  85.35190177,  61.14345748,  38.0740619 ]])

Parameters

- **expression** – expression or list of expressions, e.g. ‘x’, or ['x', 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. ‘minmax’, ‘99.7%’, [0, 10], or a list of, e.g. [[0, 10], [0, 20], ‘minmax’]
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the ‘default’), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns

Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

write_meta()

Writes all meta data, ucd,description and units

The default implementation is to write this to a file called meta.yaml in the directory defined by Dataset.get_private_dir(). Other implementation may store this in the dataset file itself. (For instance the vaex hdf5 implementation does this)

This method is called after virtual columns or variables are added. Upon opening a file, Dataset.update_meta() is called, so that the information is not lost between sessions.

Note: opening a dataset twice may result in corruption of this file.

write_virtual_meta()

Writes virtual columns, variables and their ucd,description and units

The default implementation is to write this to a file called virtual_meta.yaml in the directory defined by Dataset.get_private_dir(). Other implementation may store this in the dataset file itself.

This method is called after virtual columns or variables are added. Upon opening a file, Dataset.update_virtual_meta() is called, so that the information is not lost between sessions.

Note: opening a dataset twice may result in corruption of this file.

5.6 vaex.stat module

class vaex.stat.Expression

Describes an expression for a statistic

calculate (ds, binby=[], shape=256, limits=None, selection=None)

Calculate the statistic for a Dataset
vaex.stat.correlation(x, y)
    Creates a standard deviation statistic
vaex.stat.count(expression='*')
    Creates a count statistic
vaex.stat.covar(x, y)
    Creates a standard deviation statistic
vaex.stat.mean(expression)
    Creates a mean statistic
vaex.stat.std(expression)
    Creates a standard deviation statistic
vaex.stat.sum(expression)
    Creates a sum statistic

5.7 Machine learning with vaex.ml

Note that vaex.ml does not fall under the MIT, but the CC BY-CC-ND LICENSE, which means it’s ok for personal or academic use. You can install vaex-ml using `pip install vaex-ml`.
The `vaex.ml` package brings some machine learning algorithms to `vaex`. Install it by running `pip install vaex-ml`.

`Vaex.ml` stays close to the authoritative ML package: scikit-learn. We will first show two examples, KMeans and PCA, to see how they compare and differ, and what the gain is in performance.

```
In [1]: import vaex.ml.cluster
       import numpy as np
       %matplotlib inline

We use the well known iris flower dataset, a classical for machine learning.

```

```
In [2]: ds = vaex.ml.iris()
       ds.scatter(ds.petal_width, ds.petal_length, c_expr=ds.class_)

Out[2]: <matplotlib.collections.PathCollection at 0x1154b6b70>
```
### 6.1 KMeans

We use two features to do a KMeans, and roughly put the two features on the same scale by a simple division. We then construct a KMeans object, quite similar to what you would do in sklearn, and fit it.

```python
In [4]: features = ['petal_width/2', 'petal_length/5']
    #: init = [[0, 1/5], [1.2/2, 4/5], [2.5/2, 6/5]]
    #: kmeans = vaex.ml.cluster.KMeans(features=features, init=init, verbose=True)
    #: kmeans.fit(ds)

Iteration 0, inertia 6.2609999999999975
Iteration 1, inertia 2.5062184444444435
Iteration 2, inertia 2.443455900151798
Iteration 3, inertia 2.418136327962199
Iteration 4, inertia 2.4161501474358995
Iteration 5, inertia 2.4161501474358995
```

We now transform the original dataset, similar to sklearn. However, we now end up with a new dataset, which contains an extra column (prediction_kmeans).

```python
In [5]: ds_predict = kmeans.transform(ds)
    #: ds_predict
<IPython.core.display.HTML object>

Out[5]: <vaex.dataset.DatasetArrays at 0x11547ac18>
```

Although this column is special, it is actually a virtual column, it does not use up any memory and will be computed on the fly when needed, saving us precious ram. Note that the other columns reference the original data as well, so this
new dataset (ds_predict) almost takes up no memory at all, which is ideal for very large datasets, and quite different from what sklearn will do.

In [6]: ds_predict.virtual_columns['prediction_kmeans']
Out[6]: <vaex.expression.Expression(expressions='kmean_predict_function(petal_width/2, petal_length/5)')> instance at 0x1154c8da0 [1, 1, 1, 2, 0 ... (total 150 values) ... 0, 0, 1, 0, 1]

By making a simple scatter plot we can see the KMeans does a pretty good job.

In [7]: import matplotlib.pyplot as plt
   fig, ax = plt.subplots(1, 2, figsize=(12,5))
   plt.sca(ax[0])
   plt.title('original classes')
   ds.scatter(ds.petal_width, ds.petal_length, c_expr=ds.class_)
   plt.sca(ax[1])
   plt.title('predicted classes')
   ds_predict.scatter(ds_predict.petal_width, ds_predict.petal_length, c_expr=ds_predict.prediction_kmeans)

Out[7]: <matplotlib.collections.PathCollection at 0x1169e9e10>

6.2 KMeans benchmark

To demonstrate the performance and scaling of vaex, we continue with a special version of the iris dataset that has \(\sim 10^7\) rows, by repeating the rows many times.

In [8]: ds = vaex.ml.iris_1e7()

We now use random initial conditions, and execute 10 runs in parallel (n_init), for a maximum of 5 iterations and benchmark it.

In [9]: features = ['petal_width/2', 'petal_length/5']
   kmeans = vaex.ml.cluster.KMeans(features=features, n_clusters=3, init='random', random_state=1, max_iter=5, verbose=True, n_init=10)

In [10]: %timeit -n1 -r1 -o kmeans.fit(ds)
We now do the same using sklearn.

```python
In [12]: from sklearn.cluster import KMeans

kmeans_sk = KMeans(n_clusters=3, init='random', max_iter=5, verbose=True, algorithm='full', n_jobs=-1,
                  precompute_distances=False, n_init=10)

# Doing an unfortunate memory copy
X = np.array(ds[features])

In [13]: %timeit -n1 -r1 -o
kmeans_sk.fit(X)
```

```
Iteration 0, inertia 1784973.799998645 | 1548329.7999990159 | 354711.39999875583 | 434173.3999988521 | 1005871.0000026901 | 1312114.6000003854 | 1989377.3999927903 | 577104.4999989534 | 2747388.6000027955 | 628486.799997179
Iteration 1, inertia 481645.0225601919 | 233311.807648651 | 214794.26525253724 | 175205.9965848818 | 490218.54137152765 | 816598.0811733825 | 285786.25668654573 | 456305.06015295343 | 1205488.9851008556 | 262443.28449456714
Iteration 2, inertia 458443.87392026593 | 162015.13397359703 | 173081.69460305249 | 162580.0667193532 | 488402.9744732218 | 436698.8939923954 | 162626.54988994548 | 394680.5108569789 | 850103.6561417002 | 198213.0961053151
Iteration 3, inertia 394680.5108569789 | 161882.05987810466 | 162580.0667193532 | 161882.05987810466 | 487435.98983613244 | 214098.28159484005 | 161882.05987810466 | 275282.3731570135 | 594451.8937940609 | 169525.1971933692
Iteration 4, inertia 275282.37315701344 | 161882.05987810466 | 161882.05987810466 | 161882.05987810466 | 486000.8312405078 | 169097.27135654766 | 161882.05987810466 | 201144.2611065195 | 512055.18086238694 | 162023.37977993552
```

```
8.63 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Out[10]: <timeitResult : 8.63 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)>
```

```
In [11]: time_vaex = _
```

We now do the same using sklearn.

```
In [12]: from sklearn.cluster import KMeans

kmeans_sk = KMeans(n_clusters=3, init='random', max_iter=5, verbose=True, algorithm='full', n_jobs=-1,
                  precompute_distances=False, n_init=10)

# Doing an unfortunate memory copy
X = np.array(ds[features])

In [13]: %timeit -n1 -r1 -o
kmeans_sk.fit(X)
```
Iteration 0, inertia 951708.200
Iteration 3, inertia 188528.277
Iteration 1, inertia 491323.101
Iteration 4, inertia 167916.636
Initialization complete
Iteration 3, inertia 490998.649
Iteration 4, inertia 489551.146
Initialization complete
Iteration 0, inertia 1220612.700
Iteration 0, inertia 1469993.400
Iteration 1, inertia 514343.770
Iteration 1, inertia 498914.693
Iteration 2, inertia 214309.979
Iteration 2, inertia 211156.559
Iteration 3, inertia 167916.636
Iteration 3, inertia 171177.750
Iteration 4, inertia 163711.545
Iteration 4, inertia 162580.067
47.7 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

Out[13]: <TimeitResult : 47.7 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)>
In [14]: time_sklearn = _

We see that vaex is quite fast:

In [15]: print('vaex is approx', time_sklearn.best / time_vaex.best, 'times faster for KMeans')
vaex is approx 5.523496296321969 times faster for KMeans

But also, sklearn will need to copy the data, while vaex will be very careful not to do unnecessary copies, and minimal amounts of passes of the data (Out-of-core). Therefore vaex will happily scale to massive datasets, while with sklearn you will be limited to the size of the RAM.

6.3 PCA Benchmark

We now continue with benchmarking a PCA on 4 features:

In [16]: features = [k.expression for k in [ds.col.petal_width, ds.col.petal_length, ds.col.sepal_width, ds.col.sepal_length]]
pca = ds.ml.pca(features=features)

In [17]: %%timeit -n1 -r3 -o pca = ds.ml.pca(features=features)
478 ms ± 13.9 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)

Out[17]: <TimeitResult : 478 ms ± 13.9 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)>
In [18]: time_vaex = _

Since sklearn takes too much memory with this dataset, we only use 10% for sklearn, and correct later.

In [19]: # on my laptop this takes too much memory with sklearn, use only a subset
factor = 0.1
ds.set_active_fraction(factor)
len(ds)

Out[19]: 1005000
In [20]: from sklearn.decomposition import PCA
pca_sk = PCA(n_components=2, random_state=33, svd_solver='full', whiten=False)
X = np.array(ds.trim()[features])

6.3. PCA Benchmark
In [21]: %timeit -n1 -r3 -o pca_sk.fit(X)
232 ms ± 37.4 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
Out[21]: <TimeitResult : 232 ms ± 37.4 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)>
In [22]: time_sklearn = _
In [23]: print('vaex is approx', time_sklearn.best / time_vaex.best / factor, 'times faster for a PCA')
vaex is approx 4.449269957142027 times faster for a PCA

Again we see that vaex not only will outperform sklearn, but more importantly it will scale to much larger datasets.

In [24]: ds_big = vaex.ml.iris_1e8()
In [25]: %timeit -n1 -r2 -o pca = ds_big.ml.pca(features=features)
10.4 s ± 4.94 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
Out[25]: <TimeitResult : 10.4 s ± 4.94 s per loop (mean ± std. dev. of 2 runs, 1 loop each)>

Note the although this dataset is 10× larger, it takes more than 10× to execute. This is because this dataset did not fit into memory this time, and is limited to the harddrive speeds. But note that it possible to actually run it, instead of giving a MemoryError!

## 6.4 XGBoost

This example shows integration with xgboost, this is work in progress.

In [26]: import vaex.ml.xgboost

In [27]: ds = vaex.ml.iris()
In [28]: features = [k.expression for k in [ds.col.petal_width, ds.col.petal_length, ds.col.sepal_width, ds.col.sepal_length]]
In [29]: ds_train, ds_test = ds.ml.train_test_split()
In [30]: param = {
    'max_depth': 3,  # the maximum depth of each tree
    'eta': 0.3,  # the training step for each iteration
    'silent': 1,  # logging mode - quiet
    'objective': 'multi:softmax',  # error evaluation for multiclass training
    'num_class': 3}  # the number of classes that exist in this datset
xgmodel = vaex.ml.xgboost.XGBoostModel(features=features, num_round=10, param=param)
In [31]: xgmodel.fit(ds_train, ds_train.class_, copy=True)
In [32]: ds_predict = xgmodel.transform(ds_test)
ds_predict
<IPython.core.display.HTML object>
Out[32]: <vaex.dataset.DatasetArrays at 0x14b6d1358>
In [33]: import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 2, figsize=(12,5))
plt.sca(ax[0])
plt.title('original classes')
ds_predict.scatter(ds_predict.petal_width, ds_predict.petal_length, c_expr=ds_predict.class_)
plt.sca(ax[1])
plt.title('predicted classes')
ds_predict.scatter(ds_predict.petal_width, ds_predict.petal_length, c_expr=ds_predict.predict_class)
6.5 One hot encoding

Shortly showing one hot encoding

In [34]: ds.ml_one_hot_encoding(ds.col.class_.expression)

In [35]: ds

<IPython.core.display.HTML object>

Out[35]: <vaex.hdf5.dataset.Hdf5MemoryMapped at 0x14b170518>
Datasets to download

Here we list a few datasets, that might be interesting to explore with vaex

### 7.1 New york taxi dataset

See for instance Analyzing 1.1 Billion NYC Taxi and Uber Trips, with a Vengeance for some ideas.

- Year: 2015 - 146 million rows - 23GB
- Year 2009-2015 - 1 billion rows - 135GB

```python
In [2]: import vaex
In [12]: ds = vaex.open("/Users/users/breddels/.vaex/data/nyc_taxi/nyc_taxi2015.hdf5")
ds.plot(ds.col.pickup_longitude, ds.col.pickup_latitude, f="log1p", show=True, limits="96%")
```
7.2 SDSS - dereddened

Only: ra, dec, g, r, g_r (derededned using Schlegel maps).

The original query at SDSS archive was (although split in small parts):

```
SELECT ra, dec, g, r
FROM PhotoObjAll
WHERE type = 6 AND clean = 1 AND r >= 10.0 AND r <= 23.5;
```

- 162 million rows - 10GB

```
In [22]: sdfs = vaex.open("/Users/maartenbreddels/vaex/data/sdss/sdss_dereddened.hdf5")
   sdfs.healpix_plot(sdfs.col.healpix, show=True, f="log", healpix_max_level=9, healpix_level=9,
   healpix_input='galactic', healpix_output='galactic', rotation=(0,45)
   )
```

/Users/maartenbreddels/vaex/src/vaex/vaex/dataset.py:2071: RuntimeWarning: divide by zero encountered in log
fgrid = f(grid)
/Users/maartenbreddels/anaconda3/lib/python3.5/site-packages/numpy/core/numeric.py:190: VisibleDeprecationWarning: VisibleDeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
a = empty(shape, dtype, order)
7.3 Gaia

See the Gaia Science Homepage for details, and you may want to try the Gaia Archive for ADQL (SQL like) queries.

- Gaia data release 2 (DR2)
  - Full Gaia DR2 - 1.7 billion rows 1.2TB
  - Split in two sets of columns:
    - All astrometry and error (without covariances), radial velocity and basic photometry - 253 GB
    - Everything not contained in the above - 1 TB
    - Only with radial velocities - 7 million - 5.2GB

- Gaia data release 1 (DR1)
  - Full Gaia DR1 - 1 billion row - 351GB
  - A few columns of Gaia DR1 - 1 billion row - 88GB
  - 10% of Gaia DR1 - 1 billion row - 35GB
  - TGAS (subset of DR1 with proper motions) - 662MB

```python
In [3]: gaia = vaex.open("/data/users/gaia/gaia-dr2/gaia-dr2-sort-by-source_id.hdf5")
gaia.plot("ra", "dec", f="log", limits=[[360, 0], [-90, 90]], show=True);
```
7.4 Helmi & de Zeeuw 2000

Result of an N-body simulation of the accretion of 33 satellite galaxies into a Milky Way dark matter halo * 3 million rows - 252MB

In [26]: hdz = vaex.datasets.helmi_de_zeeuw.fetch() # this will download it on the fly
dz.plot([["x", "y"], ["Lz", "E"]], f="log", figsize=(12,5), show=True);
CHAPTER 8

What is Vaex?

Vaex is a python library for lazy Out-of-Core DataFrames (similar to Pandas), to visualize and explore big tabular datasets. It can calculate statistics such as mean, sum, count, standard deviation etc, on an N-dimensional grid up to a billion \(10^9\) objects/rows per second. Visualization is done using histograms, density plots and 3d volume rendering, allowing interactive exploration of big data. Vaex uses memory mapping, zero memory copy policy and lazy computations for best performance (no memory wasted).

8.1 Why vaex

- **Performance**: Works with huge tabular data, process \(10^9\) rows/second
- **Lazy / Virtual columns**: compute on the fly, without wasting ram
- **Memory efficient** no memory copies when doing filtering/selections/subsets.
- **Visualization**: directly supported, a one-liner is often enough.
- **User friendly API**: You will only need to deal with a Dataset object, and tab completion + docstring will help you out: `ds.mean<tab>`, feels very similar to Pandas.
- **Lean**: separated into multiple packages
  - **vaex-core**: Dataset and core algorithms, takes numpy arrays as input columns.
  - **vaex-hdf5**: Provides memory mapped numpy arrays to a Dataset.
  - **vaex-viz**: Visualization based on matplotlib.
  - **vaex-jupyter**: Interactive visualization based on Jupyter widgets / ipywidgets, bqplot, ipyvolume and ipyleaflet.
  - **vaex-astro**: Astronomy related transformations and FITS file support.
  - **vaex-server**: Provides a server to access a dataset remotely.
  - **vaex-distributed**: (Proof of concept) combined multiple servers / cluster into a single dataset for distributed computations.
- **vaex-qt**: Program written using Qt GUI.
- **vaex**: meta package that installs all of the above.
- **vaex-ml**: *Machine learning*

- **Jupyter integration**: vaex-jupyter will give you interactive visualization and selection in the Jupyter notebook and Jupyter lab.
CHAPTER 9

Installation

Using conda:
  
  • conda install -c conda-forge vaex

Using pip:
  
  • pip install --upgrade vaex

Or read the detailed instructions

9.1 Getting started

We assuming you have installed vaex, and are running a Jupyter notebook server. We start by importing vaex and ask it to give us sample example dataset.

In [36]: import vaex
ds = vaex.example()  # open the example dataset provided with vaex

Instead, you can download some larger datasets, or read in your csv file.

In [49]: ds  # will pretty print a table
<IPython.core.display.HTML object>

Out[49]: <vaex.hdf5.dataset.Hdf5MemoryMapped at 0x1194d0240>

Using 'square brackets[] <api.rst#vaex.dataset.Dataset.__getitem__>'__, we can easily filter or get different views on the dataset.

In [20]: ds_negative = ds[ds.x < 0]  # easily filter your dataset, without making a copy
ds_negativel[5][['x', 'y']]  # take the first five rows, and only the 'x' and 'y' column (no memory copy!)
<IPython.core.display.HTML object>

Out[20]: <vaex.dataset.DatasetArrays at 0x1194c9fd0>

When dealing with huge datasets, say a billion rows (10^9), computations with the data can waste memory, up to 8 GB for a new column. Instead, vaex uses lazy computation, only a representation of the computation is stored, and
computations done on the fly when needed. Even though, you can just many of the numpy functions, as if it was a normal array.

In [21]: import numpy as np
   # creates an expression (nothing is computed)
   r = np.sqrt(ds.x**2 + ds.y**2 + ds.z**2)
   r  # for convinience, we print out some values

Out[21]: <vaex.expression.Expression(expressions='sqrt((((x) ** (2)) + ((y) ** (2))) + ((z) ** (2)))')>

These expressions can be added to the dataset, creating what we call a virtual column. These virtual columns are simular to normal columns, except they do not waste memory.

In [22]: ds['r'] = r  # add a (virtual) column that will be computed on the fly
    ds.mean(ds.x), ds.mean(ds.r)
   # calculate statistics on normal and virtual columns

Out[22]: (-0.067131491264005971, 9.407082338299773)

One of the core features of vaex is its ability to calculate statistics on a regular (N-dimensional) grid. The dimensions of the grid are specified by the binby argument (analogous to SQL's groupby), and the shape and limits.

In [15]: ds.mean(ds.r, binby=ds.x, shape=32, limits=[-10, 10])  # create statistics on a regular grid


In [23]: array([[ 22., 33., 37., ..., 58., 38., 45.],
[ 37., 36., 47., ..., 52., 36., 53.],
[ 42., 47., ..., 59., 44., 56.],
[ 73., 84., ..., 41., 40., 37.],
[ 53., 58., 63., ..., 34., 35., 28.],
[ 51., 32., 46., ..., 47., 33., 36.]])

These one and two dimensional grids can be visualized using any plotting library, such as matplotlib, but the setup can be tedious. For convenience we can use plot1d, plot, or see the list of plotting commands

In [17]: ds.plot(ds.x, ds.y, show=True);  # make a plot quickly
Out[17]: <matplotlib.image.AxesImage at 0x1193c8dd8>
CHAPTER 10

Continue the tutorial here
Python Module Index

v

vaex, 36
vaex.stat, 66
Symbols

__call__() (vaex.dataset.Dataset method), 39
__getitem__() (vaex.dataset.Dataset method), 39
__init__() (vaex.dataset.Dataset method), 40
__iter__() (vaex.dataset.Dataset method), 40
__len__() (vaex.dataset.Dataset method), 40
__setitem__() (vaex.dataset.Dataset method), 40
__weakref__ (vaex.dataset.Dataset attribute), 40

A

add_column() (vaex.dataset.Dataset method), 40
add_column_healpix() (vaex.dataset.Dataset method), 40
add_variable() (vaex.dataset.Dataset method), 40
add_virtual_column() (vaex.dataset.Dataset method), 40
add_virtual_columns_aitoff() (vaex.dataset.Dataset method), 41
add_virtual_columns_cartesian_to_polar() (vaex.dataset.Dataset method), 41
add_virtual_columns_cartesian_to_spherical() (vaex.dataset.Dataset method), 41
add_virtual_columns_cartesian_velocities_to_polar() (vaex.dataset.Dataset method), 42
add_virtual_columns_cartesian_velocities_to_spherical() (vaex.dataset.Dataset method), 42
add_virtual_columns_matrix3d() (vaex.dataset.Dataset method), 43
add_virtual_columns_polar_velocities_to_cartesian() (vaex.dataset.Dataset method), 43
add_virtual_columns_rotation() (vaex.dataset.Dataset method), 43
add_virtual_columns_spherical_to_cartesian() (vaex.dataset.Dataset method), 43
app() (in module vaex), 39

B

byte_size() (vaex.dataset.Dataset method), 44

C

calculate() (vaex.stat.Expression method), 66
can_open() (vaex.dataset.Dataset class method), 44
close_files() (vaex.dataset.Dataset method), 44
col (vaex.dataset.Dataset attribute), 44
column_count() (vaex.dataset.Dataset method), 44
combinations() (vaex.dataset.Dataset method), 44
correlation() (in module vaex.stat), 66
correlation() (vaex.dataset.Dataset method), 45
count() (in module vaex.stat), 67
count() (vaex.dataset.Dataset method), 45
cov() (vaex.dataset.Dataset method), 46
covar() (in module vaex.stat), 67
covar() (vaex.dataset.Dataset method), 47

dropna() (vaex.dataset.Dataset method), 47
delayed() (in module vaex), 39
delete_variable() (vaex.dataset.Dataset method), 47
delete_virtual_column() (vaex.dataset.Dataset method), 47
dataset (class in vaex.dataset), 39
dropna() (vaex.dataset.Dataset method), 47
evaluate() (vaex.dataset.Dataset method), 47
evaluate_variable() (vaex.dataset.Dataset method), 48
event() (in module vaex), 39
evaluate() (vaex.dataset.Dataset method), 47
Expression (class in vaex.stat), 66
extract() (vaex.dataset.Dataset method), 48

F

fillna() (vaex.dataset.Dataset method), 48
from_arrays() (in module vaex), 37
from_ascii() (in module vaex), 38
from_astropy_table() (in module vaex), 38
from_csv() (in module vaex), 38
from_pandas() (in module vaex), 38
from_samp() (in module vaex), 38

G

generate() (vaex.dataset.Dataset method), 48
get_active_fraction() (vaex.dataset.Dataset method), 48
get_column_names() (vaex.dataset.Dataset method), 49
get_current_row() (vaex.dataset.Dataset method), 49
get_private_dir() (vaex.dataset.Dataset method), 49
get_selection() (vaex.dataset.Dataset method), 49
get_variable() (vaex.dataset.Dataset method), 49

has_current_row() (vaex.dataset.Dataset method), 49
has_selection() (vaex.dataset.Dataset method), 49
healpix_count() (vaex.dataset.Dataset method), 49
healpix_plot() (vaex.dataset.Dataset method), 50

is_local() (vaex.dataset.Dataset method), 50
iscategory() (vaex.dataset.Dataset method), 50

length_original() (vaex.dataset.Dataset method), 50
length_unfiltered() (vaex.dataset.Dataset method), 51
limits() (vaex.dataset.Dataset method), 51
limits_percentage() (vaex.dataset.Dataset method), 51

materialize() (vaex.dataset.Dataset method), 52
max() (vaex.dataset.Dataset method), 52
mean() (in module vaex.stat), 67
mean() (vaex.dataset.Dataset method), 52
median_approx() (vaex.dataset.Dataset method), 53
min() (vaex.dataset.Dataset method), 53
minmax() (vaex.dataset.Dataset method), 54
mutual_information() (vaex.dataset.Dataset method), 55

open() (in module vaex), 37
open_many() (in module vaex), 38

percentile_approx() (vaex.dataset.Dataset method), 56
plot3d() (vaex.dataset.Dataset method), 56
propagate_uncertainties() (vaex.dataset.Dataset method), 56

remove_virtual_meta() (vaex.dataset.Dataset method), 57
rename_column() (vaex.dataset.Dataset method), 57

sample() (vaex.dataset.Dataset method), 57
select() (vaex.dataset.Dataset method), 58
select_box() (vaex.dataset.Dataset method), 58
select_circle() (vaex.dataset.Dataset method), 58
select_ellipse() (vaex.dataset.Dataset method), 58
select_inverse() (vaex.dataset.Dataset method), 59

select_lasso() (vaex.dataset.Dataset method), 59
select_non_missing() (vaex.dataset.Dataset method), 59
select_nothing() (vaex.dataset.Dataset method), 59
select_rectangle() (vaex.dataset.Dataset method), 60
selected_length() (vaex.dataset.Dataset method), 60
selection_can_redo() (vaex.dataset.Dataset method), 60
selection_can_undo() (vaex.dataset.Dataset method), 60
selection_redo() (vaex.dataset.Dataset method), 60
selection_undo() (vaex.dataset.Dataset method), 60

server() (in module vaex), 38
set_active_fraction() (vaex.dataset.Dataset method), 60
set_active_range() (vaex.dataset.Dataset method), 60
set_current_row() (vaex.dataset.Dataset method), 60
set_log_level_debug() (in module vaex), 39
set_log_level_exception() (in module vaex), 39
set_log_level_info() (in module vaex), 39
set_log_level_off() (in module vaex), 39
set_log_level_warning() (in module vaex), 39
set_selection() (vaex.dataset.Dataset method), 60
set_variable() (vaex.dataset.Dataset method), 60
sort() (vaex.dataset.Dataset method), 61
std() (in module vaex.stat), 67
std() (vaex.dataset.Dataset method), 61
subspace() (vaex.dataset.Dataset method), 62
subspaces() (vaex.dataset.Dataset method), 62
sum() (in module vaex.stat), 67
sum() (vaex.dataset.Dataset method), 62

take() (vaex.dataset.Dataset method), 63
to_astropy_table() (vaex.dataset.Dataset method), 63
to_copy() (vaex.dataset.Dataset method), 63
to_dict() (vaex.dataset.Dataset method), 63
to_items() (vaex.dataset.Dataset method), 64
to_pandas_df() (vaex.dataset.Dataset method), 64
to_pandas_diff() (vaex.dataset.Dataset method), 64
trim() (vaex.dataset.Dataset method), 64

write_meta() (vaex.dataset.Dataset method), 66
write_virtual_meta() (vaex.dataset.Dataset method), 66
Z
zeldovich() (in module vaex), 39