Redistributor Documentation

Contents

Module `redistributor`

:warning: | Still under development :—: | :—

This repository introduces three main classes, namely **Redistributor**, **LearnedDistribution**, and **KernelDensity**.

Redistributor is a tool for automatic transformation of empirical data distributions. It is implemented as a **Scikit-learn transformer**. It allows users to transform their data from arbitrary distribution into another arbitrary distribution. The source and target distributions, if known beforehand, can be specified exactly (e.g. as a Continuous Scipy distribution[1] or any other class which has cdf and pdf methods implemented), or can be inferred from data using LearnedDistribution or KernelDensityclasses. Transformation is **piece-wise-linear, monotonic, invertible**, and can be **saved for later use** on different data assuming the same source distribution.

LearnedDistribution is a subclass of Scipy.stats.rv_continous[2] class. It is a continuous random variable obtained by estimating the empirical distribution of a user provided array of numeric data x. It can be used to sample new random points from the estimated distribution.

[1] https://docs.scipy.org/doc/scipy/reference/tutorial/stats/continuous.html#continuous-distributions-in-scipy-stats

[2] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.rv_continuous.html#scipy-stats-rv-continuous

KernelDensity is a wrapper of sklearn.neighbors.KernelDensity[3] class. It only supports Gaussian kernel and is suitable for fitting distributions base on small number of data points. It implements cdf function based on the formula for Gaussian-mixture cdf. It also provides a sped-up version of the cdf which is its approximation (use `method='fast'`). It also implements ppf function, but since there is no explicit formula for the ppf of a Gaussian mixture, it is an approximation of the real ppf. As LearnedDistribution, also KernelDensity is a continuous random variable obtained by estimating the empirical distribution of a user provided array of numeric data x. It also can be used to sample new random points from the estimated distribution.

Installation

:warning: | Not yet published on PyPi. Coming soon. :—: | :—

The code is hosted in this GitLab repository[4]. To install the released version from Pypi use:

```
pip install redistributor
```

Or install the bleeding edge directly from git:

```
pip install git+https://gitlab.com/paloha/redistributor
```

For development, install the package in editable mode with extra dependencies for documentation and testing:

```
# Clone the repository
git clone git@gitlab.com:paloha/redistributor.git
cd redistributor

 # Use virtual environment [optional]
python3 -m virtualenv .venv
source .venv/bin/activate

# Install with pip in editable mode
pip install -e .[dev]
```

Compatibility

…

Dependencies

Required packages for Redistributor are specified in the install_requires list in the setup.py file.

Extra dependencies for running the tests, compiling the documentation, or running the examples are specified in the extras_require dictionary in the same file.

The full version-locked list of dependencies and subdependencies is frozen in requirements.txt. Installing with `pip install -r requirements.txt` in a virtual environment should always lead to a fully functional project.

Mathematical description

Assume we are given data $x \sim S$ distributed according to some source distribution S on $\mathbb{R}$ and our goal is to find a transformation R such that $R(x) \sim T$ for some target distribution T on $\mathbb{R}$.

One can mathematically show that a suitable $R\colon \mathbb{R} \to \mathbb{R}$ is given by

$$R := F_T^{-1} \circ F_S,$$

where F_S and F_T are the cumulative distribution functions of S and T, respectively.

If S and T is unknown, one can use approximations $\tilde{F}_S$ and $\tilde{F}_T$ of the corresponding cumulative distribution functions given by interpolating (partially) sorted data

$$(x_i)_{i=1}^N \text{ with } x_i \sim S$$

[3] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KernelDensity.html

[4] https://gitlab.com/paloha/redistributor

$$(y_i)_{i=1}^M \text{ with } y_i \sim T.$$

Defining

$$\tilde{R} := \tilde{F}_T^{-1} \circ \tilde{F}_S,$$

one can, under suitable conditions, show that

$$\tilde{R} \xrightarrow[N,M\to\infty]{} R.$$

How to cite

...

License

Acknowledgement

This work was supported by the *International Mobility of Researchers* (program call no.: CZ.02.2.69/0.0/0.0/16027/0008371[5]).

EUROPEAN UNION
European Structural and Investment Funds
Operational Programme Research,
Development and Education

MINISTRY OF EDUCATION,
YOUTH AND SPORTS

Functions

Function `load_redistributor`

```
def load_redistributor(
    path
)
```

Loads the Redistributor object from a file.

Function `make_unique`

```
def make_unique(
    array,
    dist='max',
    mode='raise',
    assume_sorted=True,
    inplace=False,
    random_state=None
)
```

UTILITY FUNCTION TO FORCE LATTICE VALUES TO HAVE NON-REPEATING ELEMENTS

Finds duplicate values in array and shifts them at most by dist to get an array of all unique values. Shifts are sampled randomly from uniform distribution.

If dist is not smaller or equal to half the smallest distance between two non-duplicates, a duplicate point + noise could “jump behind” the next non-duplicate. E.g. for array [0, 1, 1, 2, 3] and dist = 1.5 the result could be np.sort([0, 1, 2.5, 2, 3]), i.e. the second occurrence of number 1 was augmented by noise of 1.5 magnitude and in result it

[5] https://opvvv.msmt.cz/vyzva/vyzva-c-02-16-027-mezinarodni-mobilita-vyzkumnych-pracovniku.htm

jumped to position 2.5 which is larger than 2, which was one of the original non-duplicate values. (This is an extreme example)

NOTICE: there is no good way to implement this function as it changes the provided data to fullfill the assumption on non-repeating values. Whether it is a good idea to do it this way or some other way highly depends on use case. So make sure you know what you are doing.

Parameters

`array` **: 1D numpy array** Array with potential of having duplicate elements.
`dist` **: float or `'max'`, default `'max'`** Max allowed shift of a duplicate point. If 'max' is used the max_dist = 1/2 min distance between two non-duplicates.
`mode` **: one of `{'raise', 'clip', 'ignore', 'warn'}`, default `'raise'`** Behavior when specified dist is larger than max_dist. 'raise' - raises a ValueError 'clip' - clips the dist to max_dist 'ignore' - will use dist no matter the consequences, use with caution 'warn' - same as ignore, just a warning is issued
`assume_sorted` **: bool, default True** If not, we sort at the beginning.
`inplace` **: bool, default False** If True, adjust array inplace, otherwise make a copy.
`random_state` **: RandomState, int, or None, default None** Seed or generator for noise generation.

Returns

`array` **: sorted 1D numpy array with no duplicates** If `inplace=True`, returns None

Function `plot_cdf_ppf_pdf`

```
def plot_cdf_ppf_pdf(
    dist,
    a=None,
    b=None,
    bins=None,
    v=None,
    w=None,
    rows=1,
    cols=3,
    figsize=(16, 5)
)
```

Just a convinience function for visualizing the dist cdf, ppf and pdf functions.

Parameters

`a` **: float** Start of the cdf support
`b` **: float** End of the cdf support
`v` **: float** Start of the ppf support
`w` **: float** End of the ppf support
`rows` **: int,** Number of rows in the figure
`cols` **: int,** Number of cols in the figure
`figsize` **: None or tuple** If None, no new figure is created.

Function `save_redistributor`

```
def save_redistributor(
    d,
    path
)
```

Saves the Redistributor object to a file.

Classes

Class `KernelDensity`

```
class KernelDensity(
    x,
    ravel_x=True,
    grid_density=5000,
    cdf_method='fast',
    name='KDE',
    **kwargs
)
```

Wrapper around KernelDensity for ease of use as a source or target distribution of Redistributor. It extends the KDE by providing cdf and ppf functions.

Only supports 1D input because Redistributor also works only in 1D. Only supports gaussian kernel. CDF supports two methods, precise and fast. CDF precise is computed using a formula. CDF fast is a linear interpolation of the CDF precise on a grid of specified density. There is no explicit formula for PPF of gaussian mixutre, so here it is approximated using linear interpolation of the CDF precise on a grid of specified density.

Parameters

`x` : numeric or 1D numpy array 1D vector of which the distribution will be estimated.
`ravel_x` : bool, default True KDE requires 1D arrays. So the x is by default flattened to 1D using np.ravel().
`grid_density` : int, default 5e3 User specified number of grid points on which the CDF is computed precisely in order to build the interpolants for fast CDF and PPF. The same grid is used for CDF fast and PPF. The user specified value of grid_density is not it's final value. It is updated during initialization of this object on call of self._get_ppf().
`cdf_method` : str, one of `{'precise', 'fast'}` Specifies the default method to be used when self.cdf() is called. 'precise' computes cdf using a formula, 'fast' uses a precomputed interpolant to get a fast approximation.
`name` : str, default `'LearnedDistribution'` The name of the instance.

kwargs : keyword arguments accepted by sklearn.neighbors.KernelDensity.

Methods

pdf : Probability Density Function of a Gaussian Mixture cdf : Cumulative Distribution Function of a Gaussian Mixture ppf : Approximation of a Percent Point Function of a Gaussian Mixture rvs : Random sample generator

Methods

Method `cdf`

```
    def cdf(
        self,
        q,
        method=None
    )
```

Cumulative distribution function of the estimated distribution.

Parameters

`q` : array_like quantile

Returns

`p` : 1D numpy array of floats Cumulative distribution function evaluated at q. I.e. lower tail probability corresponding to the quantile q.

Method `pdf`

```
    def pdf(
        self,
        x
    )
```

Probability density function of the estimated distribution.

Parameters

`q` **: array_like** quantile

Returns

`d` **: 1D numpy array of floats** Probability density function evaluated at q. I.e. probability density corresponding to the quantile q.

Method `ppf`

```
def ppf(
    self,
    p
)
```

Percent point function of the estimated distribution. This method approximates the ppf based on linear interpolation of the cdf on self.grid_density many points. There is no formula for precise computation of gaussian mixture ppf. Therefore, if we wanted a precise function, we would need to bisect the cdf. Bisecting is very slow in comparison to just computing the cdf on a grid and using the interpolant to approximate the ppf.

Parameters

`p` **: array_like** lower tail probability

Returns

`q` **: 1D numpy array of floats** Percent point function evaluated at p. I.e. quantile corresponding to the lower tail probability p.

Method `rvs`

```
def rvs(
    self,
    size=1,
    random_state=None
)
```

Random sample from the estimated distribution.

Class `LearnedDistribution`

```
class LearnedDistribution(
    x,
    a=None,
    b=None,
    bins=None,
    keep_x_unchanged=True,
    subsample_x=None,
    ravel_x=True,
    assume_sorted=False,
    fill_value='auto',
    bounds_error='warn',
    resolve_duplicates=('max', 'raise'),
    seed=None,
    name='LearnedDistribution',
    **kwargs
)
```

A continuous random variable obtained by estimating the empirical distribution of a user provided 1D array of numeric data x. It can be used to sample new random points from the learned distribution.

It approximates the Cumulative Distribution Function (cdf) and Percent Point Function (ppf) of the underlying distribution of x using linear interpolation on a lattice.

An approximation of the Probability Density Function (pdf) is computed as an interpolation of the numerical derivative of the cdf function. Please note it can oscilate a lot if bins is high.

The distribution is defined on a closed finite interval [a, b] or [xmin, xmax] or combination thereof, depending on which bound/s were specified by the user.

WARNING: It can not be used to learn discrete distributions.

Parameters

`x` **: 1D numpy array** Values from which the distribution will be estimated. The size of the array should be rather large, in case you have too small sample, consider using KDE class instead. Large magnitude of the array values in combination with small amount of samples, e.g.

`a` **: numeric or None** Left boundary of the distribution support if known. If specified, must be smaller than x.min().

`b` **: numeric or None** Right boundary of the distribution support if known. If specified, must be bigger than x.max().

`bins` **: int or None** User specified value of bins. Min is 3, max is x.size. If None or 0, bins are set automatically. Upper bound is set to 5000 to prevent unnecessary computation. Used to specify the density of the lattice. More bins means higher precision but also more computation.

`keep_x_unchanged` **: bool, default True** If True, the x array will be copied before partial sorting. This will result in increased memory usage. But it will not reorder the user provided array.

If False, there will not be any additional memory consumption. But the user provided array x might change its order. This might be very useful if x is a large array and there is not enough available memory.

`subsample_x` **: int, default None** Sacrifice precision for speed by first subsampling array x with a defined integer step. Not doing random.choice() but rather simple slice(None, None, subsample_x) because it is faster and we assume the array is randomly ordered. Can lead to significant speedups. If you need different approach to subsampling, do it in advance, provide already subsampled x and set this to None.

`ravel_x` **: bool, default True** LearnedDistribution requires 1D arrays. So the x is by default flattened to 1D using np.ravel().

`assume_sorted` **: bool, default False** If the user knows that x is sorted, setting this to True will save computation by ommiting partial sorting the array. Especially useful if the array x is big. E.g. 1GB of data takes approx. 10s to partial sort on 5000 positions. If False and x is almost sorted, it will still be faster than if x is randomly ordered.

`fill_value` **:** `None, float, 2-tuple, 'auto'`**, default=**`'auto'` Specifies where to map the values out of the cdf support. See the docstring of scipy.interpolate.interp1d to learn more about the possible options. Additionally, this class enables the user to use the default auto option, which sets reasonable fill_value automatically.

WARNING: Not all choices of fill_value that are possible are also valid. E.g. fill_value should not be manually set to value smaller than 0 or larger than one. Also, fill_value should not be set such that it would make the output function decreasing. This also rules out the usage of ‘extrapolate’ option. All of these choices would not lead to a meaningful output in terms of a Cumulative Distribution Function.

`bounds_error` **: bool or** `'warn'`**, default** `'warn'` If True, raises an error when values out of cdf support are encountered. If False or ‘warn’, the invalid values are mapped to fill_value. For more details see the docstring of class interp1d_with_warning.

`resolve_duplicates` **:** `2-tuple` **(dist, mode) or None,** default (‘max’, ‘raise’) If not None, makes a call to make_unique() with specified dist and mode to make sure all lattice_values are unique. Read more in the docstring of make_unique() function.

WARNING: If None, the array is kept with duplicates which means the `p != cdf(ppf(p))`. In case there is mulitple duplicates of xmin or xmax values, cdf(xmin) will fail to map to Δ and cdf(xmax) will fail to map to 1 - Δ as it should.

`name` : **str, default `'LearnedDistribution'`** Name of the instance. Useful for locating source of warnings, etc.

`seed` : **{None, int,numpy.random.Generator,** numpy.random.RandomState}, default None See the docstring of scipy.stats.rv_continuous. Used in make_unique() and rvs().

kwargs : all other keyword arguments accepted by rv_continous.

Ancestors (in MRO)

- scipy.stats._distn_infrastructure.rv_continuous
- scipy.stats._distn_infrastructure.rv_generic

Methods

Method `cdf`

```
def cdf(
    self,
    q
)
```

Interpolates the lattice on lattice_vals to get the piecewise linear approximation to the emprical cumulative distribution function of the learned distribution.

Parameters

`q` : **array_like** quantile

Returns

`p` : **1D numpy array of floats** Cumulative distribution function evaluated at q. I.e. lower tail probability corresponding to the quantile q.

Method `ppf`

```
def ppf(
    self,
    p
)
```

Interpolates the lattice_vals on lattice to get the piecewise linear approximation to the inverse of the emprical cumulative distribution function of the learned distribution. I.e. a Percent point function of the learned distribution.

Parameters

`p` : **array_like** lower tail probability

Returns

`q` : **1D numpy array of floats** Percent point function evaluated at p. I.e. quantile corresponding to the lower tail probability p.

Method `rvs`

```
def rvs(
    self,
    size,
    random_state=None
)
```

Random sample from the learned distribution.

Class `Redistributor`

```
class Redistributor(
    source,
    target
)
```

An algorithm for automatic transformation of data from arbitrary distribution into arbitrary distribution. Source and target distributions can be known beforehandand or learned from the data using LearnedDistribution class. Transformation is piecewise linear, monotonic and invertible.

Implemented as a Scikit-learn transformer. Can be fitted on 1D vector and saved to be used later for transforming other data assuming the same source distribution.

Uses source's and target's cdf() and ppf() to infer the transform and inverse transform functions.

`transform_function = target_ppf(source_cdf(x))` `inverse_transform = source_ppf(target_cdf(x))`

Ancestors (in MRO)

- sklearn.base.TransformerMixin

Methods

Method `fit`

```
def fit(
    x=None,
    y=None
)
```

Redistributor does not need to be fitted.

Method `inverse_transform`

```
def inverse_transform(
    self,
    x
)
```

Inverse transform the data from target to source distribution.

Method `kstest`

```
def kstest(
    self,
    n=20
)
```

Performs the (one-sample or two-sample) Kolmogorov-Smirnov test.

Method `plot_transform_function`

```
def plot_transform_function(
    self,
    bins=1000,
    newfig=True,
    figsize=(16, 5)
)
```

Plotting the learned transformation from source to target.

Method `transform`

```
def transform(
    self,
    x
)
```

Transform the data from source to target distribution.

Class `interp1d_with_warning`

```
class interp1d_with_warning(
    *args,
    **kwargs
)
```

By default behaves exactly as scipy.interpolate.interp1d but allows the user to specify `bounds_error = 'warn'` which overrides the behaviour of _check_bunds to warn instead of raising an error.

Parameters

Accepts all the args and kwargs as scipy.interpolate.interp1d.

Initialize a 1-D linear interpolation class.

Ancestors (in MRO)

- scipy.interpolate.interpolate.interp1d
- scipy.interpolate.polyint._Interpolator1D

Generated by *pdoc* 0.9.2 (https://pdoc3.github.io).