Universität Zürich[UZH]

# Efficiency of Univariate Kernel Density Estimation with TensorFlow

Bachelor Thesis

Author: Marc Steiner

Supervisors: Jonas Eschle, Nicola Serra

University of Zurich

## Abstract

An implementation of a one-dimensional Kernel Density Estimation in TensorFlow is proposed. Starting from the basic algorithm, several optimizations from recent papers are introduced and combined to ameliorate the efficiency of the algorithm. By comparing its accuracy and efficiency to implementations in pure Python as well to density estimations by smoothed histograms it is shown as competitive and useful in real world applications.

# Contents



# 1 Introduction

## 1.1 Kernel Density Estimation

In many fields of science and in physics especially, scientists need to estimate the probability density function (PDF) from which a set of data is drawn without having a approximate model of the underlying mechanisms, since they are too complex to be fully understood analytically. So called parametric methods fail, because the knowledge of the system is to poor to design a model, which parameters can then be fitted by some goodness-of-fit criterion like log-likelihood or $\chi^2$. In the particle accelerator at CERN for instance, they record a whooping 25 gigabytes of data per second[1], resulting from the process of many physical interactions that occur almost simultaneously, making it impossible to anticipate features of the distribution one observes.

To combat this so called non-parametric methods like histograms are used. By summing the the data up in discrete bins we can approximate the underlying parent distribution, without needing any knowledge of the physical interactions. However there are also many more sophisticated non-parametric methods, one in particular is Kernel Density Estimation (KDE), which can be looked at as a sort of generalized histogram.[2]

Histograms tend to produce PDFs that are highly dependent on bin width and bin positioning, meaning the interpretion of the data changes by a lot by two arbitrary parameters. KDE circumvents this problem by replacing each data point with a so called kernel that specifies how much it influences its

neighbouring regions as well. The kernels themselves are centered at the data point directly, eliminating the need for arbitrary bin positioning[3]. Since KDE still depends on kernel width (instead of bin width), one might argue that this is not a major improvement. However, upon closer inspection, one finds that the underlying PDF does depend less strongly on the kernel width than histograms on bin width and it is much easier to specify rules for an approximately optimal kernel width than it is to do so for bin width[4]. In addition, by specifying a smooth kernel, one gets a smooth distribution as well, which is often desirable or even expected from theory.

Due to this increased robustness, KDE is particular useful in High-Energy Physics (HEP) where it has been used for confidence level calculations for the Higgs Searches at the Large Electron Positron Collider (LEP)[5]. However there is still room for improvement and certain more sophisticated approaches to Kernel Density Estimation have been proposed in dependence on specific areas of application[5].

### 1.2 zfit and TensorFlow

Currently the basic principle of KDE has been implemented in various programming languages and statistical modeling tools. The standard framework used in HEP, that includes KDE is the ROOT/RooFit toolkit written in C++. However, Python plays an increasingly large role in the natural sciences due to support by corporations involved in Big Data and its superior accessibility. To elevate research in HEP, zfit, a new alternative to RooFit, was proposed. It is implemented on top of TensorFlow, one of the leading Python frameworks to handle large data and high parallelization, allowing a transparent usage of CPUs and GPUs[6].

So far there exists no direct implementation of Kernel Density Estimation in zfit nor TensorFlow, but various implementations in Python. This implementations will be discussed in the next chapter (2) before I propose an implementation of Kernel Density Estimation in TensorFlow (3). Starting with a rather simple implementation, multiple improvements from recent papers are integrated to ameliorate its efficiency. Efficiency and accuracy of the implementation are then tested by comparing it to other implementations in pure Python and simple smoothed histograms (4).

In the last chapter (5) I compare its accurracy and efficiency to smoothed histograms and implementations in pure Python.

## 2 Current state of the art

…

For humongous data streams (like at CERN 1), Kernel Density Estimation itself needs to be approximated.

…

## 3 Implementation

The following is an implementation of Kernel Density Estimation using TensorFlow.

```
import numpy as np
import tensorflow as tf

from tensorflow_probability.python.distributions import Categorical
from tensorflow_probability.python.distributions import Independent
from tensorflow_probability.python.distributions import MixtureSameFamily
from tensorflow_probability.python.distributions import Normal
from tensorflow_probability.python.distributions import distribution
from tensorflow_probability.python.distributions.normal import Normal
from tensorflow_probability.python.internal import assert_util
from tensorflow_probability.python.internal import dtype_util
from tensorflow_probability.python.internal import reparameterization
from tensorflow_probability.python.internal import tensor_util


class KernelDensityEstimation(MixtureSameFamily):
    """ Kernel density estimation based on data.

        Implements Linear Binning and Fast Fourier Transform to speed up the
 ↪  computation.
    """

    def __init__(self,
                 data,
                 bandwidth=0.01,
                 kernel=Normal,
                 use_grid=False,
                 use_fft=False,
                 num_grid_points=1024,
                 reparameterize=False,
                 validate_args=False,
                 allow_nan_stats=True,
                 name='KernelDensityEstimation'):

        components_distribution_generator = lambda loc, scale:
 ↪  Independent(kernel(loc=loc, scale=scale))

        with tf.name_scope(name) as name:
            self._use_fft=use_fft
```

```
            dtype = dtype_util.common_dtype([bandwidth, data], tf.float32)
            self._bandwidth = tensor_util.convert_nonref_to_tensor(
                bandwidth, name='bandwidth', dtype=dtype)
            self._data = tensor_util.convert_nonref_to_tensor(
                data, name='data', dtype=dtype)

            if(use_fft):
                self._grid = self._generate_grid(num_grid_points)
                self._grid_data = self._linear_binning()

                mixture_distribution=Categorical(probs=self._grid_data)
                compo-
    nents_distribution=components_distribution_generator(loc=self._grid,
    scale=self._bandwidth)

            elif(use_grid):
                self._grid = self._generate_grid(num_grid_points)
                self._grid_data = self._linear_binning()

                mixture_distribution=Categorical(probs=self._grid_data)
                compo-
    nents_distribution=components_distribution_generator(loc=self._grid,
    scale=self._bandwidth)

            else:
                self._grid = None
                self._grid_data = None
                n = self._data.shape[0]
                mixture_distribution=Categorical(probs=[1 / n] * n)
                compo-
    nents_distribution=components_distribution_generator(loc=self._data,
    scale=self._bandwidth)

            super(KernelDensityEstimation, self).__init__(
                mixture_distribution=mixture_distribution,
                components_distribution=components_distribution,
                reparameterize=reparameterize,
                validate_args=validate_args,
                allow_nan_stats=allow_nan_stats,
                name=name
            )

    def _generate_grid(self,
```

```
                        num_points):

        minimum = tf.math.reduce_min(self._data)
        maximum = tf.math.reduce_max(self._data)
        return tf.linspace(minimum, maximum, num=num_points)

    def _linear_binning(self,
                        weights=None):

        if weights is None:
            weights = np.ones_like(self._data)

        grid_min = tf.math.reduce_min(self._grid)
        grid_max = tf.math.reduce_max(self._grid)
       num_intervals = tf.math.subtract(tf.size(self._grid), tf.constant(1))
        dx = tf.math.divide(tf.math.subtract(grid_max, grid_min),
↪   tf.cast(num_intervals, tf.float32))

        transformed_data = tf.math.divide(tf.math.subtract(self._data,
↪   grid_min), dx)

        # Compute the integral and fractional part of the data
        # The integral part is used for lookups, the fractional part is used
        # to weight the data
        integral = tf.math.floor(transformed_data)
        fractional = tf.math.subtract(transformed_data, integral)

        # Compute the weights for left and right side of the linear binning
         ↪  routine
        frac_weights = tf.math.multiply(fractional, weights)
        neg_frac_weights = tf.math.subtract(weights, frac_weights)

       # If the data is not a subset of the grid, the integral values will be
        # outside of the grid. To solve the problem, we filter these values
         ↪  away
        #unique_integrals = np.unique(integral)
        #unique_integrals = unique_integrals[(unique_integrals >= 0) &
         ↪  (unique_integrals <= len(grid_points))]

        bincount_left = tf.roll(tf.concat(tf.math.bincount(tf.cast(integral,
↪   tf.int32), weights=frac_weights), tf.constant(0)), shift=1, axis=0)
        bincount_right = tf.math.bincount(tf.cast(integral, tf.int32),
↪   weights=neg_frac_weights)
```

```
        bincount = tf.add(bincount_left, bincount_right)

        return bincount

    def _generate_fft_grid(self,
                           num_points):

        grid_min = tf.math.reduce_min(self._data)
        grid_max = tf.math.reduce_max(self._data)
        num_intervals = tf.math.subtract(num_points, tf.constant(1))
        dx = tf.math.divide(tf.math.subtract(grid_max, grid_min),
↪   tf.cast(num_intervals, tf.float32))

        return tf.linspace(tf.math.multiply(tf.math.negative(dx),
         ↪  num_points), tf.math.multiply(dx, num_points),
         ↪  tf.math.add(tf.math.multiply(tf.cast(num_points, tf.int32),
         ↪  tf.constant(2)), tf.constant(1)))

    def _evaluate_fft(self, x):
        # Reshape in preparation to

        #! Not implemented yet!
        return super(KernelDensityEstimation, self)._log_prob(x)


        """num_points = tf.size(self._grid)
        x = tensor_util.convert_nonref_to_tensor(x, name='x',
↪   dtype=tf.float32)
        l = tf.linspace(tf.math.multiply(tf.math.negative(dx), num_points),
↪   tf.math.multiply(dx, num_points),
↪   tf.math.add(tf.math.multiply(tf.cast(num_points, tf.int32),
↪   tf.constant(2)), tf.constant(1)))

        components_distribution_generator = lambda loc, scale:
↪   Independent(kernel(loc=loc, scale=scale))

        n = x.shape[0]
        mixture_distribution=Categorical(probs=[1/n] * n)
        components_distribution=components_distribution_generator(loc=x,
↪   scale=self._bandwidth)

        super(KernelDensityEstimation, self).__init__(
```

```
                mixture_distribution=mixture_distribution,
                components_distribution=components_distribution)

        super(KernelDensityEstimation, self)._log_prob(x)

        kernel_weights = super(KernelDensityEstimation, self)._log_prob(x)
        zeros_count =
↪   tf.cast(tf.math.divide(tf.math.subtract(kernel_weights.shape[0],
↪   x.shape[0]), tf.constant(2)), tf.int32)
        paddings = paddings = [[zeros_count, zeros_count]]
        data = tf.pad(x, paddings, "CONSTANT")

        tf.print(kernel_weights.shape)
        tf.print(data.shape)

        tf.print(data)

        # Use FFT
        kernel_weights = tf.signal.rfft(kernel_weights)
        data = tf.signal.rfft(data)

        return tf.signal.irfft(tf.math.multiply(data,
↪   kernel_weights))[zeros_count:-zeros_count]"""

    def _log_prob(self, x):

        if(self._use_fft):
            return self._evaluate_fft(x)
        else:
            return super(KernelDensityEstimation, self)._log_prob(x)
```

# 4 Comparison

To compare the different implementations I created a simple test distribution comprised of three gaussian, one uniform and one exponential distribution. The distribution is created by using the TensorFlow Probability package and its Mixture Model.

## 4.1 Generation of Test Distribution

Listing: Test Distribution generation

```
import numpy as np
import tensorflow as tf
import tensorflow_probability as tfp
import tensorflow_probability.python.distributions as tfd
from tf_kde.distribution import KernelDensityEstimation

r_seed = 1978239485
n_datapoints = 1000000
tfd = tfp.distributions

mix_3gauss_1exp_1uni = tfd.Mixture(

  cat=tfd.Categorical(probs=[0.1, 0.2, 0.1, 0.4, 0.2]),

  components=[
    tfd.Normal(loc=-1., scale=0.4),
    tfd.Normal(loc=+1., scale=0.5),
    tfd.Normal(loc=+1., scale=0.3),
    tfd.Exponential(rate=2),
    tfd.Uniform(low=-5, high=5)
])

data = mix_3gauss_1exp_1uni.sample(sample_shape=n_datapoints, seed=r_seed)
data = data.numpy()
```

### 4.2 Comparing different methods

```
import tensorflow as tf
import numpy as np
from KDEpy import FFTKDE
from tf_kde.distribution import KernelDensityEstimation,
 ↪  KernelDensityEstimationBasic
from zfit_benchmark.timer import Timer
import seaborn as sns
import pandas as pd

from tf_kde.tests.test_distribution import data


n_testpoints = 200
```

```
def kde_basic(data, x):

    fac = 1.0 / np.sqrt(2.0 * np.pi)
    exp_fac = -1.0/2.0
    h = 0.01
    y_fac = 1.0/(h*data.size)

    gauss_kernel = lambda x: fac * np.exp(exp_fac * x**2)

    y = np.zeros(x.size)

    for i, x_i in enumerate(x):
        y[i] = y_fac * np.sum(gauss_kernel((x_i-data)/h))

    return y

def kde_seaborn(data, x):
    sns.distplot(data, bins=1000, kde=True, rug=False)
    return np.NaN

def kde_kdepy_fft(data, x):
    x = np.array(x)
    return FFTKDE(kernel="gaussian", bw="silverman").fit(data).evaluate(x)

@tf.function(autograph=False)
def kde_basic_tf_internal(data, x, n_datapoints):

    h1 = 0.01

    fac = tf.constant(1.0 / np.sqrt(2.0 * np.pi), tf.float32)
    exp_fac = tf.constant(-1.0/2.0, tf.float32)
    y_fac = tf.constant(1.0/(h1 * n_datapoints), tf.float32)
    h = tf.constant(h1, tf.float32)

    gauss_kernel = lambda x: tf.math.multiply(fac,
    ↪ tf.math.exp(tf.math.multiply(exp_fac, tf.math.square(x))))
    calc_value = lambda x: tf.math.multiply(y_fac,
    ↪ tf.math.reduce_sum(gauss_kernel(tf.math.divide(tf.math.subtract(x,
    ↪ data), h))))

    return tf.map_fn(calc_value, x)

def kde_basic_tf(data, x):
```

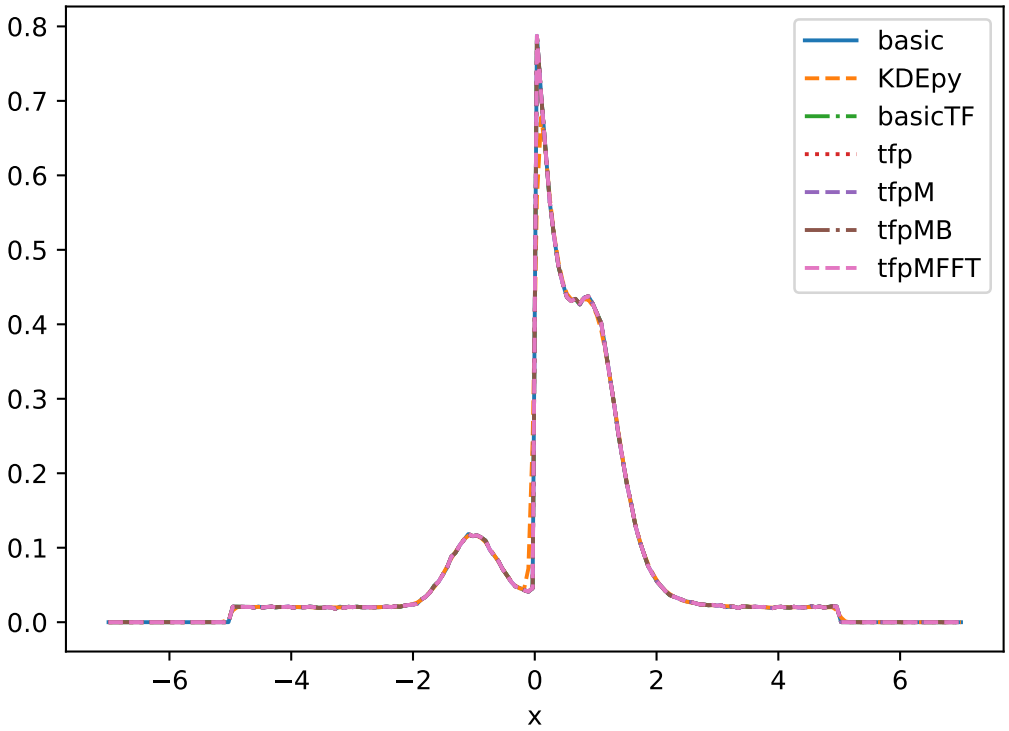
```
    n_datapoints = data.size
    return kde_basic_tf_internal(data, x, n_datapoints).numpy()

def kde_tfp(data, x):
    dist = KernelDensityEstimationBasic(bandwidth=0.01, data=data)
    return dist.prob(x).numpy()

def kde_tfp_mixture(data, x):
    dist = KernelDensityEstimation(bandwidth=0.01, data=data)
    return dist.prob(x).numpy()

def kde_tfp_mixture_with_binned_data(data, x):
    dist = KernelDensityEstimation(bandwidth=0.01, data=data, use_grid=True)
    return dist.prob(x).numpy()

def kde_tfp_mixture_with_fft(data, x):
    dist = KernelDensityEstimation(bandwidth=0.01, data=data, use_grid=True,
 ↪  use_fft=True)
    return dist.prob(x).numpy()

methods = pd.DataFrame({
    'identifier': [
        'basic',
        'seaborn',
        'KDEpy',
        'basicTF',
        'tfp',
        'tfpM',
        'tfpMB',
        'tfpMFFT'
    ],
    'label': [
        'Basic KDE with Python',
        'KDE using seaborn.distplot',
        'KDE using KDEpy.FFTKDE',
        'Basic KDE in TensorFlow',
        'KDE implemented as TensorFlow Probability Distribution Subclass',
        'KDE implemented as TensorFlow Probability MixtureSameFamily
         ↪  Subclass',
        'KDE implemented as TensorFlow Probability MixtureSameFamily
         ↪  Subclass with Binned Data',
        'KDE implemented as TensorFlow Probability MixtureSameFamily
         ↪  Subclass with Fast Fourier Transform'
```

```
    ],
    'function':[
        kde_basic,
        kde_seaborn,
        kde_kdepy_fft,
        kde_basic_tf,
        kde_tfp,
        kde_tfp_mixture,
        kde_tfp_mixture_with_binned_data,
        kde_tfp_mixture_with_fft
    ]
})
methods.set_index('identifier', drop=False, inplace=True)

estimations = pd.DataFrame()
estimations['x'] = np.linspace(-7.0, 7.0, num=n_testpoints,
 ↪  dtype=np.float32)

methods['runtime'] = np.NaN
for index, method in methods.iterrows():
    with Timer('Benchmarking') as timer:
        estimations[method['identifier']] = method['function'](data,
 ↪  estimations['x'])
        timer.stop()
    methods.at[method['identifier'], 'runtime'] = timer.elapsed

methods.drop('function', axis=1, inplace=True)
```

Running this, leads to the following comparison:

**Table 1:** Runtime comparison

| | identifier | label |
|---|---|---|
| basic | basic | Basic KDE with Python |
| seaborn | seaborn | KDE using seaborn.distplot |
| KDEpy | KDEpy | KDE using KDEpy.FFTKDE |
| basicTF | basicTF | Basic KDE in TensorFlow |
| tfp | tfp | KDE implemented as TensorFlow Probability Distribution Subclass |
| tfpM | tfpM | KDE implemented as TensorFlow Probability MixtureSameFamily Subclass |
| tfpMB | tfpMB | KDE implemented as TensorFlow Probability MixtureSameFamily Subclass with Binned Data |
| tfpMFFT | tfpMFFT | KDE implemented as TensorFlow Probability MixtureSameFamily Subclass with Fast Fourier |




Plotted for reference:

## 5 Summary

In summary we can conclude that…

**Table 2:** Accuracy comparison

| x | basic | seaborn | KDEpy | basicTF | tfp | tfpM | tfpMB | tfpMFFT |
|---|---|---|---|---|---|---|---|---|
| -7.0000000 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.9296484 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.8592963 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.7889447 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.7185931 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.6482410 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.5778894 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.5075378 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.4371858 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.3668342 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.2964826 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.2261305 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.1557789 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.0854273 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -6.0150752 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.9447236 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.8743720 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.8040199 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.7336683 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.6633167 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.5929646 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.5226130 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.4522614 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.3819094 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.3115578 | 0.0000000 | NaN | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.2412062 | 0.0000000 | NaN | 0.0000004 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.1708541 | 0.0000000 | NaN | 0.0000442 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.1005025 | 0.0000000 | NaN | 0.0009742 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 | 0.0000000 |
| -5.0301509 | 0.0000293 | NaN | 0.0062010 | 0.0000293 | 0.0000293 | 0.0000293 | 0.0000548 | 0.0000548 |
| -4.9597988 | 0.0208125 | NaN | 0.0158229 | 0.0208125 | 0.0208125 | 0.0208125 | 0.0207540 | 0.0207540 |
| -4.8894472 | 0.0208162 | NaN | 0.0200963 | 0.0208162 | 0.0208162 | 0.0208162 | 0.0208610 | 0.0208610 |
| -4.8190956 | 0.0204375 | NaN | 0.0206161 | 0.0204375 | 0.0204375 | 0.0204375 | 0.0205128 | 0.0205128 |
| -4.7487435 | 0.0204646 | NaN | 0.0205411 | 0.0204646 | 0.0204646 | 0.0204646 | 0.0204957 | 0.0204957 |
| -4.6783919 | 0.0205341 | NaN | 0.0204012 | 0.0205341 | 0.0205341 | 0.0205341 | 0.0204630 | 0.0204630 |
| -4.6080403 | 0.0195643 | NaN | 0.0202287 | 0.0195643 | 0.0195643 | 0.0195643 | 0.0196219 | 0.0196219 |
| -4.5376883 | 0.0210927 | NaN | 0.0201746 | 0.0210927 | 0.0210927 | 0.0210927 | 0.0209914 | 0.0209914 |

## References

[1]CERN (n.d.).

[2] M. Rosenblatt, (1956).

[3] T. Duong, An Introduction to Kernel Density Estimation (2001).

[4] M. Lerner, Histograms and Kernel Density Estimation | Biophysics and Beer (2013).

[5] K.S. Cranmer, (2000).

[6] J. Eschle, A.P. Navarro, R.S. Coutinho, and N. Serra, (2019).

## List of Tables



## List of Figures

## Listings