VU Data Science in Astrophysics (Summer 2024)
Chapter 2: Density Estimation

Oliver Hahn*

March 18, 2024

*oliver.hahn@univie.ac.at

1 Density Estimation

1.1 Histograms

In the last chapter, we saw that the empirical distribution $\hat{p}$ of empirical data $\mathcal{S} = \{X_i \in \Omega\}_{i=1\ldots N}$ (the sum of Dirac-δs)

$$\hat{p}_X(x) = \frac{1}{N}\sum_{i=1}^{N} \delta_D(x - X_i). \tag{1}$$

is not a particular good approximation to the underlying true distribution $p(x)$, since it is zero almost everywhere. At the same time, the empirical cumulative distribution

$$\hat{C}_X(x) = \frac{1}{N}\sum_{i=1}^{N} \theta_H(x - X_i). \tag{2}$$

proved to be more useful. The two are of course related by definition, $\hat{p} = \frac{\mathrm{d}\hat{C}}{\mathrm{d}x}$.

Fixed bin width histograms. Given a finite **bin width** h and an anchor point x_0, the **histogram** of the empirical data $\mathcal{S}$ is defined as the finite difference approximation

$$\hat{f}_h(n) = \frac{\hat{C}_X(x_0 + (n+\frac{1}{2})h) - \hat{C}_X(x_0 + (n-\frac{1}{2})h)}{h} \qquad n \in \mathbb{Z} \tag{3}$$

$$= \frac{\#\text{of samples } X_i \quad \text{with} \quad n - \frac{1}{2} < \frac{X_i - x_0}{h} < n + \frac{1}{2}}{Nh} \tag{4}$$

while the histogram of the true PDF is given by

$$f_h(n) = \frac{1}{h}\int_{x_0+(n-\frac{1}{2})h}^{x_0+(n+\frac{1}{2})h} p_X(x)\,\mathrm{d}x \tag{5}$$

where h is called the **bin width**. Unlike the eCDF (and ePDF), the histogram has two a priori undetermined parameters: the bin width, and the anchor point. The bin width choice leads to a trade-off between the bias (=systematic difference between $\hat{f}_h$ and f_h) and the variance of the histogram (=variance of the difference). The variance can be reduced by choosing large h, leading however to a large bias. In contrast, the bias can be reduced by choosing small h so that the bins are narrow; this however leads to a large variance. They can be trivially produced with `matplotlib`. Given a vector of univariate data `Xdata` and a vector of bin edges `Xbins`, one simply calls

```
1  import matplotlib.pyplot as plt
2  plt.hist( Xdata, Xbins, density=True )
```

If the parameter `density` is set to `False`, then `matplotlib` produces a histogram of raw counts, rather than a normalized empirical PDF.

In Figure 1, we show the effect of different bin widths on the histogram illustrating the competing effects of bias and variance. Another important limitation of histograms is that they are not translation invariant. This means that if we shift the data by an amount that is not an integer multiple of the bin size h, then the histogram will intrinsically change.

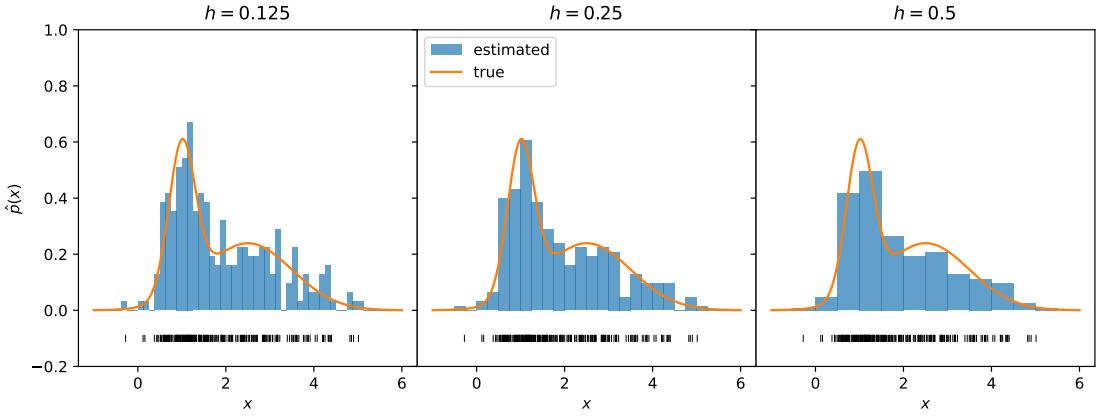

Figure 1: Histograms with fixed bin width h. Increasing the bin width (left to right) reduces the variance but increases the bias. Note that histograms are not translation invariant, i.e. adding a constant to the random variable leads to a different histogram that is not simply shifted by the corresponding amount.

It is common to represent histograms as bar diagrams. In some cases it is more convenient to represent them as stem plots, or as step functions, as shown in Fig. 2. This is easily achieved via

```
# compute the histogram via numpy (not matplotlib as before)
px,bin_centers = np.histogram( Xdata, Xbins, density=True )

plt.stem( bin_centers, px ) # plot a stem plot

plt.step( bin_centers, px, where='mid') # plot a step plot
```

If we want to estimate errors on histograms, this can be conveniently achieved using the **jackknife** or the **bootstrap**, see chapter 1 for details. Here, we use a delete-d jackknife to estimate means and errors directly from the sample of random variables, by drawing N_{jack} random subsamples from our data, and empirically estimating the mean and (co)-variance over these subsamples. This is easily achieved with a few lines of code:

```
# number of subsamples
nsubsample = 10
# use 80 per cent of the data in each subsample
bootsize   = int(0.8 * len(Xdata))
```

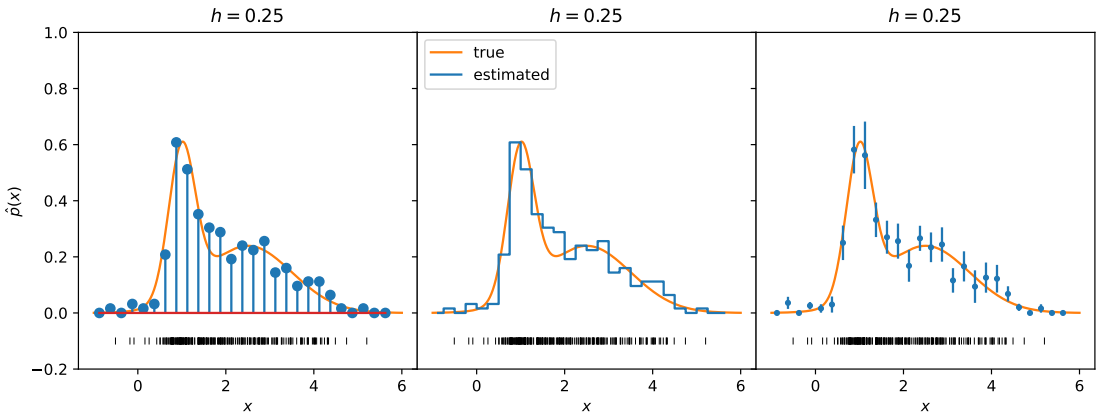

Figure 2: Histograms displayed in different styles as a stem plot (left), as a step plot (middle), or using errorbars obtained via statistical jackknife.

```
  hp_sum  = np.zeros( len(Xbins)-1 )
  hp_sum2 = np.zeros( len(Xbins)-1 )

  # accumulate for first and second empirical moments
  for _ in range(nsubsample):
    Xsubsample = np.random.choice( Xdata, bootsize, Replace=False )
    hp,hx = np.histogram( Xsubsample, Xbins, density=True )
    hp_sum += hp
    hp_sum2 += hp**2

  # compute the empirical mean over the samples
  hmean = hp_sum / nbootstrap
  # compute the empirical variance over the samples
  hvar  = hp_sum2 / nbootstrap - hmean*hmean

  plt.errorbar( xbin, hmean, np.sqrt(hvar), fmt='.' )
```

Adaptive bin width histograms. In adaptive bin width histograms, bins that have below a minimum number of counts are combined, or the bins are even chosen to maintain a constant number of data points falling into them. Keeping the number of samples per bin fixed can be achieved with quantile binning. In this case, we do not choose bins on the range of the value of the random variable, but on its quantile rank (i.e. the rank divided by the sample size). This is achieved as follows:

```
  Nbins = 10
  # create quantile bins
  pbins = np.linspace( 0, 1, Nbins, endpoint=True )
  # obtain the bin centers as the median of each bin
  pxbins = 0.5*(pbins[1:]+pbins[:-1])

  # get the bin boundaries from the quantile bins
  Xbins = np.quantile( d1, pbins )
```

```
38    # same for the bin centers
39    bin_centers = np.quantile( d1, pxbins )
40    # use the quantile binning to compute the histogram
41    hp,_ = np.histogram( d1, Xbins, density=True )
42
43    # plot
44    plt.stem( bin_centers, hp, label='estimated')
```

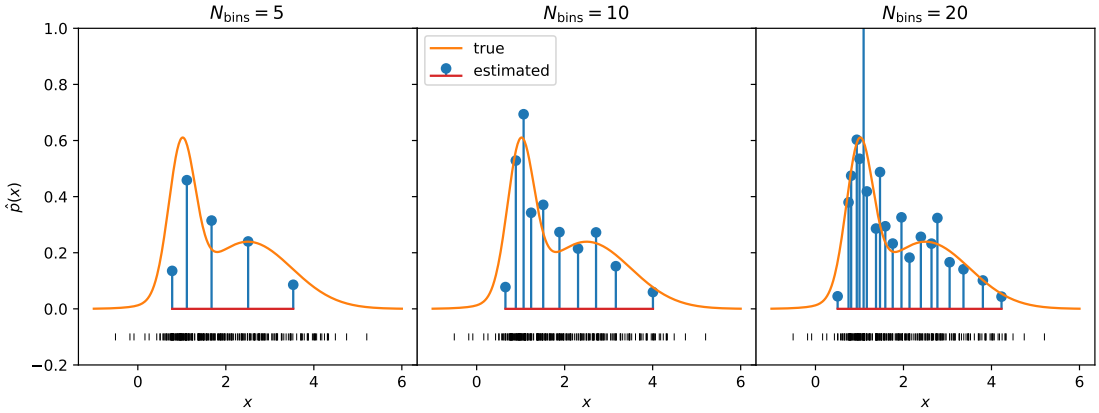

Figure 3: Histogram with adaptive quantile binning: the number of bins is set and the amount of data per bin is fixed.

Histograms are thus the simplest density estimators that we can use for empirical data, and you will encounter them all over the place.

Two-dimensional histograms. In principle histograms can be generalised to arbitrary dimensions, but due to the problem of visualising higher dimensional data, for plotting mostly the two-dimensional version is used.

Two-dimensional histograms are readily implemented in `matplotlib`. The simplest version uses rectangular boxes (i.e. the simple tensor product of two one-dimensional bin vectors), it can be invoked as follows, assuming our data `Xdata` is a NUMPY array of size $N_{\text{sample}} \times 2$

```
45    x1bins = np.linspace(x1min,x1max,n1bins)
46    x2bins = np.linspace(x1min,x2max,n2bins)
47    _ = plt.hist2d( Xdata[...,0], Xdata[...,1], (x1bins,x2bins), density=True)
```

Evidently any other tessellation of the plane can be used to define the binning. Another common choice is hexagonal binnings (yielding a honeycomb tessellation of the plane), which is also directly implemented in `matplotlib`

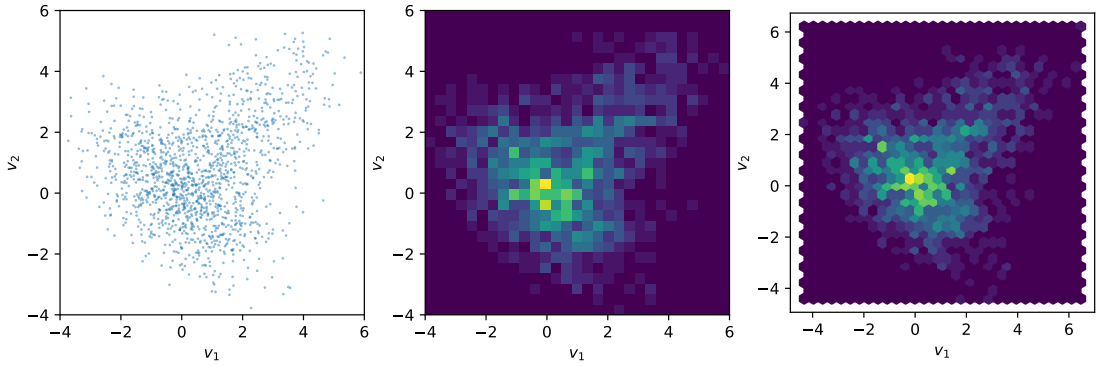

Figure 4: Histograms for bivariate data: left shows a simple scatter plot. In the middle panel we see a histogram with square bins, while on the right we see a histogram with hexagonal bins.

```
48    plt.hexbin( Xdata[...,0], Xdata[...,1], gridsize=nbins)
```

Unfortunately it has no `density` parameter, i.e. it can only represent the raw counts rather than the normalised PDF.

1.2 Kernel density estimators

As we have seen, histograms come in a large range of forms and a routine tool of data analysis. Their limitation is that they require to define a binning, i.e. a space tessellation, which breaks translation invariance. To overcome this limitation is by 'attaching the bin width directly to the data points' in what is then called **kernel density estimation (KDE)**. The idea is to replace the Dirac-δs of the empirical PDF

$$\hat{p}_X(\boldsymbol{x}) = \frac{1}{N}\sum_{i=1}^{n} \delta_D(\boldsymbol{x} - \boldsymbol{X}_i). \tag{6}$$

with a kernel whose support extends beyond the data point itself, i.e. in the d-variate case to use the estimator

$$\hat{f}_h(\boldsymbol{x}) = \frac{1}{Nh^d}\sum_{i=1}^{n} K(d(\boldsymbol{x}, \boldsymbol{X}_i)/h). \tag{7}$$

where we call h the bandwidth, and $K(r)$ is a normalised kernel

$$\int_{\mathbb{R}} \mathrm{d}r K(r) = 1. \tag{8}$$

The optimal bandwidth can be determined using cross-validation methods, but very often it is also tuned by hand.

Distances. Kernel density estimation requires to define a metric $d_{ij} = d(\boldsymbol{x}_i, \boldsymbol{x}_j)$ on the sample space. Note that for metrics we require that they are symmetric and positive $d_{ij} = d_{ji} \geq 0$. Some common distances are

$$\text{Euclidean distance} \qquad \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2 \tag{9a}$$

$$\text{squared Euclidean distance} \qquad \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2 \tag{9b}$$

$$\text{Manhattan distance} \qquad \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_1 \tag{9c}$$

$$\text{Maximum distance} \qquad \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_\infty \tag{9d}$$

Note that it is further common to non-dimensionalise the data by first **standardizing** it (i.e. subtract mean and divide by standard deviation for each variable).

Kernels. Very common is the Gaussian kernel

$$K_{\text{Gauss}}(r) = (2\pi)^{-d/2} \exp(-r^2/2), \tag{10}$$

the top hat kernel

$$K_{\text{TH}}(r) = \frac{1}{V_d} \begin{cases} 1 & \text{if } |r| \leq 1 \\ 0 & \text{otherwise} \end{cases}, \tag{11}$$

where V_d is the volume of the d-dimensional unit ball, or the exponential kernel

$$K_{\exp}(r) = \frac{\exp(-|r|)}{d!\, V_d} \tag{12}$$

KDE is implemented in the `scikit-learn` framework

```
49  from sklearn.neighbors import KernelDensity
50
51  # create a kernel density estimator object
52  kde = KernelDensity(kernel='gaussian', bandwidth=0.2)
53  # create a model by 'fitting' the KDE to our data (no actual fitting)
54  model = kde.fit( Xdata[:,np.newaxis] )
55  # evaluate the KDE at test locations Xtest
56  ptest = np.exp(model.score_samples(Xtest[:,np.newaxis]) )
57  # could plot now Xtest vs. ptest to show density estimate
```

The somewhat peculiar statement `Xdata[:,np.newaxis]` is required to promote a simple 1D data vector of length `N` to a `N`×`1` array. The result for different bandwidths is shown in Fig. 5. Note that KDE is quite costly if performed using the explicit summary formula, as (number of data points)×(number of test points) evaluations are necessary. Since the bandwidth is fixed, KDEs can however be speed up dramatically if the evaluation is desired on a regular grid, in which case the FFT can be used to evaluate the convolution sum. The implementation for bivariate KDEs is analogous, but some care needs to be taken if the result is supposed be shown as an image.

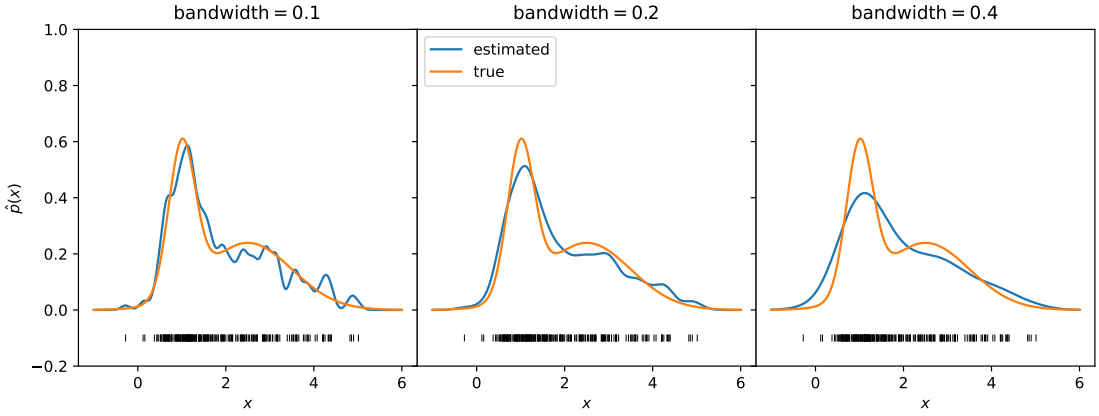

Figure 5: Kernel density estimates (KDEs) using a Gaussian kernel with band width h. Increasing the band width (left to right) reduces the variance but increases the bias as for the histogram. In contrast to the histogram, the KDE is translation invariant.

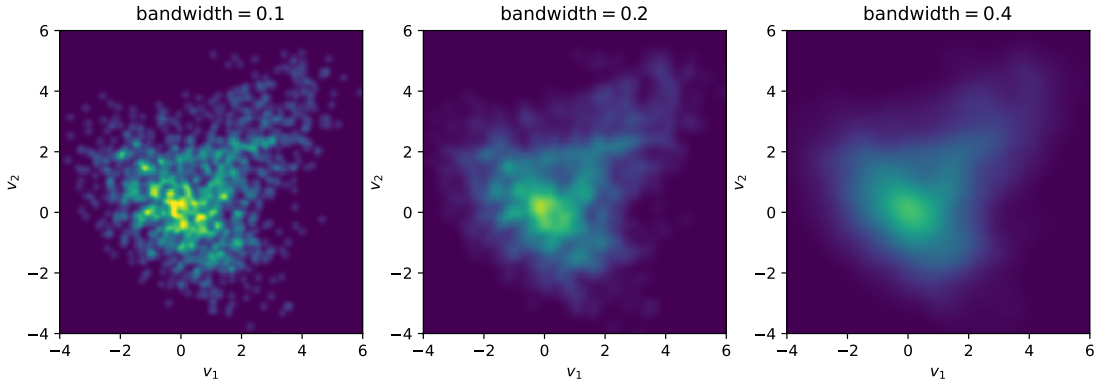

Figure 6: Kernel density estimates (KDEs) in two dimensions.

A sample implementation to evaluate a bivariate KDE on a regular set of grid points can be achieved as follows, leading to images like those in Fig. 6

```
58    # create regular space vectors covering the data range
59    v1g = np.linspace(v1min,v1max,n1)
60    v2g = np.linspace(v2min,v2max,n2)
61    # get the mesh points from v1g and v2g
62    xg,yg = np.meshgrid( v1g, v2g )
63    # flatten to get a (Ntest,2) sized array of test points
64    Xgrid = np.vstack((xg.flatten(),yg.flatten())).T
65
66    # evaluate the KDE at the test points
67    kde_dens_at_x = np.exp(model.score_samples(Xgrid) )
68    # reshape from (Ntest,2) to n1xn2 image
69    ptest = kde_dens_at_x.reshape((n1,n2))
70    # plot
```

```
71    plt.pcolor( v1g, v2g, ptest )
```

Shortcomings of KDE. A shortcoming of the KDE (as for the fixed bin histogram) is that the bandwidth is fixed. This can be overcome by allowing the kernel bandwidth to vary between the data points. This leads to a procedure that is called adaptive KDE.

1.3 Nearest neighbour estimators

The lack of adaptivity of KDE can be overcome by a completely adaptive method, the **k-th Nearest Neighbor** density estimate. The idea is fairly simple. To measure 'nearest' neighbors, we again need to define a metric on our space, so assume we once again have a metric $d(\boldsymbol{X}_i, \boldsymbol{X}_j) \geq 0$. The distance $d_k(\boldsymbol{x})$ between $\boldsymbol{x}$ to the k-th nearest point in the set of sample points $\{\boldsymbol{X}_j\}$ is directly correlated to the local density, so that in D dimensions

$$\hat{f}_k(\boldsymbol{x}) = \frac{k}{V_D \, N_{\text{sample}} \, d_k(\boldsymbol{x})^D} \propto \frac{1}{d_k(\boldsymbol{x})^D} \tag{13}$$

Note however that this estimator is biased, particularly so in the tails of the distribution. It is also implemented in the `scikit-learn` framework and can be invoked as follows

```
72    from sklearn.neighbors import NearestNeighbors
73
74    kNN  = 16 # number of nearest neighbours to use
75
76    # create nearest neighbor object
77    nbrs = NearestNeighbors(n_neighbors=kNN, algorithm='ball_tree')
78
79    # 'fit' the model to our data
80    model = nbrs.fit(Xdata[:,np.newaxis])
81
82    # evaluate at test locations
83    distances, indices = model.kneighbors(Xtest[:,np.newaxis])
84
85    # turn the distances into a density estimate
86    ptest = 1/distances[...,-1]/len(Xdata)*kNN/2
```

The result of a nearest neighbor estimate for different numbers of neighbors is shown in Figure 7. Clearly this estimator is very noisy for small neighbor numbers and has large bias for large neighbour numbers.

Hybrid kNN + KDE estimators. A good choice for adaptive KDE estimators is to estimate the bandwidth from a kNN estimator first, and then select the bandwidth in the KDE based on this estimate.

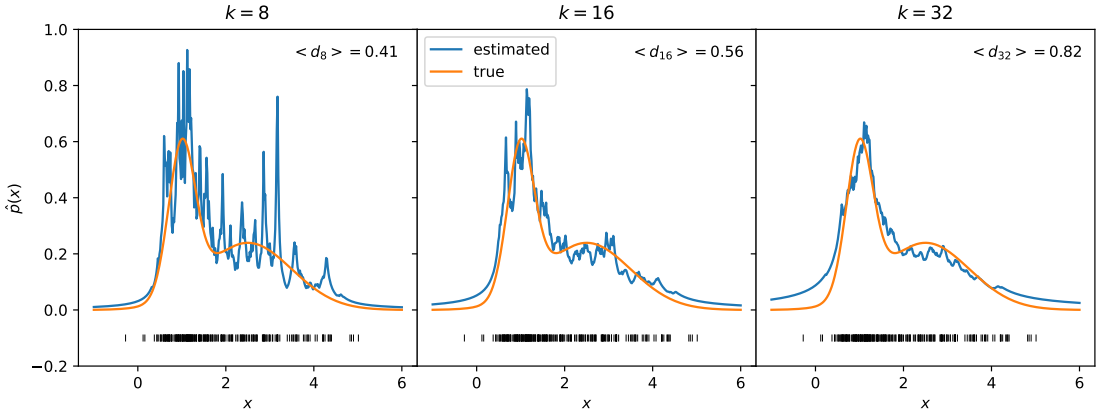

Figure 7: Density estimates using k-nearest neighbour distances. Increasing k effectively increases the band width. k-NN density estimates are adaptive, but can have significant bias in the tails of the distribution.

kNN for multivariate data. the kNN estimator is readily extended for multivariate data, just like the KDE estimator.

1.4 Voronoi based density estimators

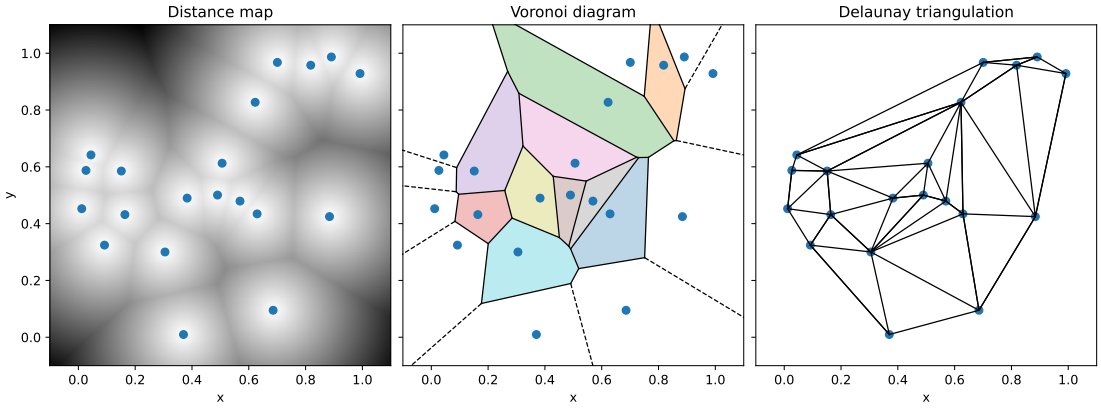

Figure 8: Left: Distance map, each pixel is shaded by the distance to the nearest data point (in blue). Middle: Voronoi tessellation of the data points. The Voronoi cells are given by those points that are closer to the data point than to any other data point. Right: Delaunay triangulation of the data points.

Assume we are given n data points $\mathcal{S} = \{\boldsymbol{X}_i\}_{i=1\ldots n}$ in d-dimensional space. The **Voronoi tessellation** of the data points is the partition of the space into n convex cells, such that each cell contains all points that are closer to the i-th data point than to any other data point. These cells are separated by $d-1$ dimensional simplices (straight line segments

in 2D, triangles in 3D, tetrahedra in 4D etc.).

How these cells arise can most easily be seen by plotting the **distance map** $\mathcal{D}$, i.e. the distance of each pixel to the nearest data point, as shown in Fig. 8, left panel, and defined as k subsets $R_k \subset \Omega$ such that

$$\mathcal{D}(\boldsymbol{x}) = \min_i d(\boldsymbol{x}, \boldsymbol{X}_i), \tag{14}$$

where the standard Euclidean distance is most commonly used. The Voronoi tessellation is then simply defined as

$$R_k = \{\boldsymbol{x} \in \Omega \mid d(\boldsymbol{x}, \boldsymbol{X}_k) < d(\boldsymbol{x}, \boldsymbol{X}_j) \; \forall \; j \neq k\} \; . \tag{15}$$

An example for a distance map and a Voronoi tessellation is shown in Fig. 8. One sees the direct correpsondence between the distance map and the Voronoi tessellation. In the Voronoi diagram, we have colored all cells of finite volume in color. Some cells are unbounded and have infinite volume, these are shown in white. Voronoi diagrams in arbitrary dimensions can be computed using the `scipy` package

```
from scipy.spatial import Voronoi

# create a Voronoi object (computes already the Voronoi tessellation)
vor = Voronoi( Xdata )
```

The volume (or area in 2D) of the Voronoi cell can be used as a measure of the local density of the data points, let us call it V_i, then a measure of the local density is given by

$$\hat{f}_V(\boldsymbol{x}) = \frac{1}{n\, V_k} \qquad \text{if } \boldsymbol{x} \in R_k, \tag{16}$$

and where $V_k = \mathrm{vol}(R_k)$. This quantity can be readily computed as follows

```
from scipy.spatial import ConvexHull

def voronoi_volumes( v : Voronoi ):
  # returns the volume of the Voronoi cells associated with each data point
  vol = np.zeros(v.npoints)
  for i, reg_num in enumerate(v.point_region):
      indices = v.regions[reg_num]
      if -1 in indices or len(indices)==0: # extends to infinity
          vol[i] = np.inf
      else:
          vol[i] = ConvexHull(v.vertices[indices]).volume
  return vol
```

The result of such a density estimator can be seen in Fig. 9. The Voronoi tessellation is useful in some cases, but it can be quite sensitive to the exact location of the data points, i.e. it is not very robust. Also, points that are at the surface of the data will

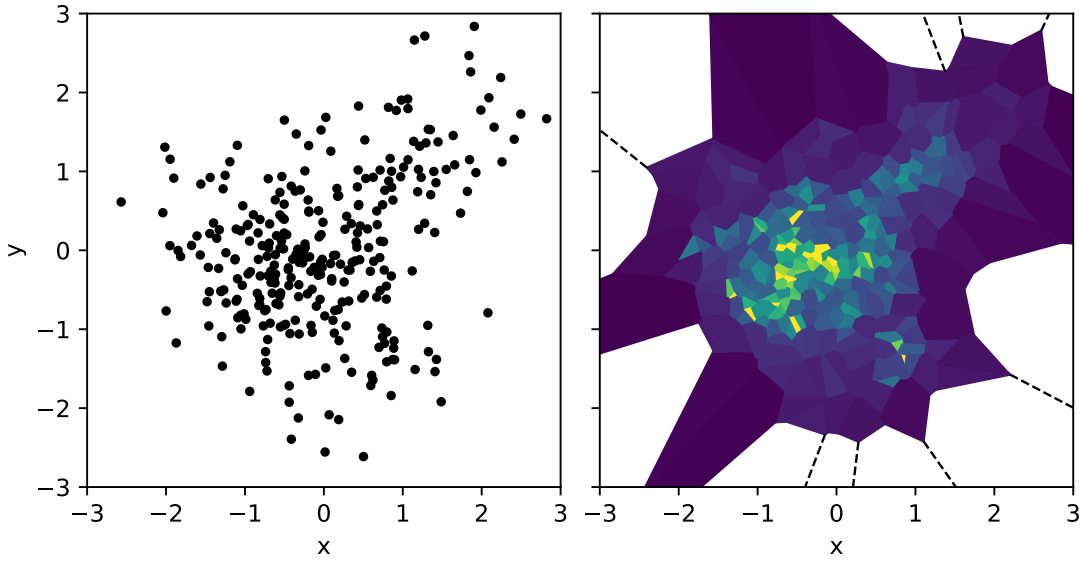

Figure 9: Left: a collection of data points sampled from a normal distribution. Right: the Voronoi tessellation of the data points, with the density $\hat{f}$ of each cell shaded in color. White regions are unbounded and thus have zero density.

have unbounded volume. With increasing dimensions, the surface to volume ratio of the Voronoi cells increases, and the Voronoi tessellation becomes less useful. This is related to the curse of dimensionality.

The Voronoi tessellation is dual to the **Delaunay triangulation**, i.e. the vertices of the Delaunay triangulation are the centers of the Voronoi cells, and the edges of the Delaunay triangulation are the edges of the Voronoi cells. The **Delaunay triangulation** of the data points is the tessellation of the space into simplices (triangles in 2D, tetrahedra in 3D, etc.) such that no data point is inside the circumcircle of any simplex. The Delaunay triangulation is shown in Fig. 8, right panel.

1.5 Gaussian mixture models

The idea of Gaussian mixture models is to fit a collection of (possibly multivariate) Gaussian distributions to the data (unlike in Bayesian methods, this is done without a prior). The number of 'clusters' N_{cluster} is the only free parameter of such Gaussian mixture models. All other parameters are determined by optimization. Let us consider the n-variate case to be general. The PDF of a p-component n-variate Gaussian mixture is given by a normalised sum of N_{cluster} Gaussians with relative weights α

$$p_{\text{GMM}}\left(\boldsymbol{x} \mid \{(\alpha_j, \boldsymbol{\mu}_j, \Sigma_j)\}\right) = \sum_{j=1}^{N_{\text{cluster}}} \alpha_j \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_j, \Sigma_j) \qquad \text{where} \qquad \sum_{j=1}^{N_{\text{cluster}}} \alpha_j = 1. \quad (17)$$

And fixing N_{cluster} the free parameters $\{(\alpha_j,\, \boldsymbol{\mu}_j,\, \Sigma_j)\}_{j=1\ldots N_{\text{cluster}}}$ can be fit via the **expectation maximization (EM) algorithm** to the data. The first step, the so-called 'E'-step, first computes the posterior probability of membership of a point $\boldsymbol{X}$ to be from the j-th distribution, i.e.

$$\tau_p(\boldsymbol{X}) := \mathbb{P}[\boldsymbol{X} \sim \mathcal{N}_j] = \frac{\alpha_j \mathcal{N}(\boldsymbol{X};\, \boldsymbol{\mu}_j, \Sigma_j)}{p_{\text{GMM}}\left(\boldsymbol{X} \mid \{(\alpha_j,\, \boldsymbol{\mu}_j,\, \Sigma_j)\}\right)} \tag{18}$$

The idea of the EM algorithm is now to consider an iterative scheme, in which we adjust the free parameters in steps, i.e. we have e.g. $\alpha_j^{(k)}$ as the value of α_j in the k-th step, then

$$\text{E-step:} \qquad \tau_p^{(k)}(\boldsymbol{X}) := \frac{\alpha_j \mathcal{N}(\boldsymbol{X};\, \boldsymbol{\mu}_j^{(k)}, \Sigma_j^{(k)})}{p_{\text{GMM}}\left(\boldsymbol{X} \mid \{(\alpha_j^{(k)},\, \boldsymbol{\mu}_j^{(k)},\, \Sigma_j^{(k)})\}\right)} \tag{19a}$$

The latent parameters are then updated in the M-step by computing the first three empirical moments of the data with respect to the posterior probability distribution of membership, i.e.

$$\text{M-step:} \qquad \alpha_j^{(k+1)} = \frac{1}{N_{\text{samples}}} \sum_{i=1}^{N_{\text{samples}}} \tau_j^{(k)}(\boldsymbol{X}_i) \tag{19b}$$

$$\boldsymbol{\mu}_j^{(k+1)} = \frac{\sum_{i=1}^{N_{\text{samples}}} \boldsymbol{X}_i\, \tau_j^{(k)}(\boldsymbol{X}_i)}{\sum_{i=1}^{N_{\text{samples}}} \tau_j^{(k)}(\boldsymbol{X}_i)} \tag{19c}$$

$$\Sigma_j^{(k+1)} = \frac{\sum_{i=1}^{N_{\text{samples}}} (\boldsymbol{X}_i - \boldsymbol{\mu}_j^{(k+1)}) \otimes (\boldsymbol{X}_i - \boldsymbol{\mu}_j^{(k+1)})\, \tau_j^{(k)}(\boldsymbol{X}_i)}{\sum_{i=1}^{N_{\text{samples}}} \tau_j^{(k)}(\boldsymbol{X}_i)} \tag{19d}$$

The E-M steps are alternated until convergence, i.e. until the parameters do not change by more than a pre-specified amount any longer.

In PYTHON, we can easily use the GMM EM algorithm from within the SCIKIT-LEARN framework. Assume `X` is the vector holding our independent samples of the random variable $\{\hat{X}_i\}_{i=1\ldots N}$, then a GMM can be trained as follows

```
from sklearn.mixture import GaussianMixture

# create the GMM fixing some hyperparameters
GM = GaussianMixture( N_cluster )
# fit the GMM to the data set X
GMmodel = GM.fit( Xdata[:,np.newaxis] )
# evaluate the model posterior at trial points xtest
GMM_dens_at_x = np.exp( GMmodel.score_samples( Xtest[:,np.newaxis] ) )
```

The code is trivially modified to take higher dimensional data into account.

The quality of the fit can be assessed using information theoretic criteria. One problem is that we are **overfitting** the data if we are using to many clusters. In this case,

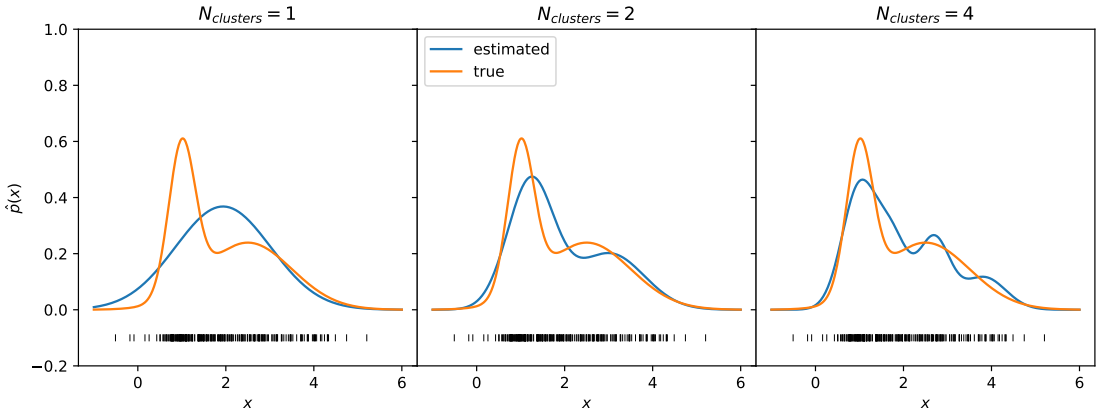

Figure 10: Density estimates based on a Gaussian mixture model of N_{clusters} Gaussians for our univariate distribution.

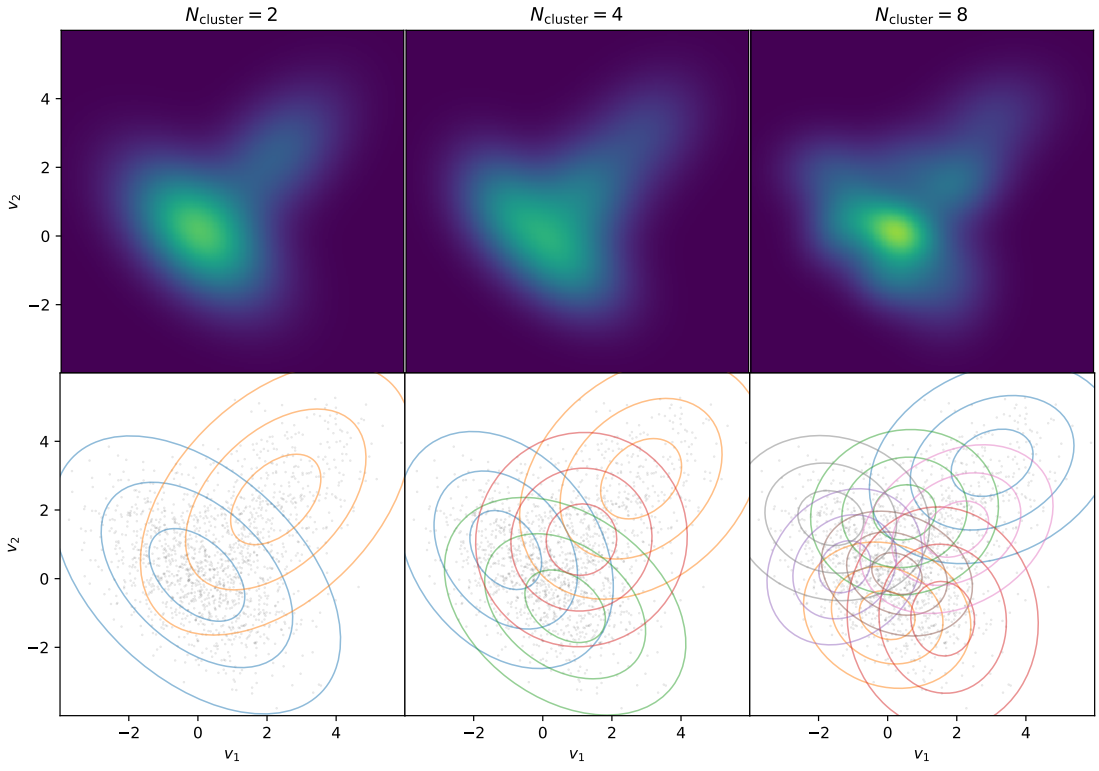

Figure 11: Gaussian mixture model in two dimensions indicating also the components of the mixture distributions in the bottom panels.

we will clearly improve the agreement, until in the extreme case we essentially fit one Gaussian per data point. In machine learning lingo, we would say that the model will not generalise well, as it represents more the specific data, than the underlying distribution. In essence, one requires a criterion that indicates the trade-off between a good fit (where the likelihood is maximal) while at the same time penalising having too many degrees of freedom given the amount of data. Well suited for this task is the **Bayes Information Criterion (BIC)** (which we will not derive here), which measures

$$\mathrm{BIC} = K \log N_{\mathrm{samples}} - 2 \log \widehat{L}, \tag{20}$$

where K is the number of parameters, i.e. $K = N_{\mathrm{cluster}}(1 + n + n^2/2)$ in the n-variate GMM case, $\widehat{L}$ is the likelihood function given the best fit latent parameters, i.e.

$$\log \widehat{L} = \sum_{i=1}^{N_{\mathrm{samples}}} \log p_{\mathrm{GMM}} \left(\boldsymbol{X}_i \mid \{(\alpha_j, \boldsymbol{\mu}_j, \Sigma_j)\} \right) \tag{21}$$

This is implemented in `scikit-learn` and can be easily evaluated as

```
111    bic = model.bic( X[:,np.newaxis] )
```

For a GMM to our double Gaussian test sample, we obtain a BIC score as a function of the hyperparameter N_{cluster}. The result is shown in Figure 12. Clearly the value 2 is favoured here.

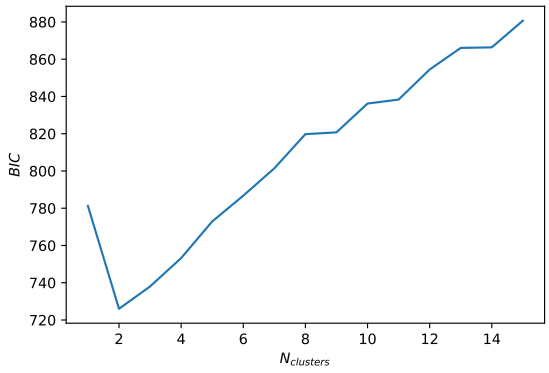

Figure 12: BIC score for the univariate GMM density estimates shown in Fig. 10 (up to 16 clusters). The optimal mixture model for the data indeed corresponds to a two-component mixture. The plot for the two-dimensional data looks qualitatively identical.