Topic for the class-Histograms, binnings and density

Unit _2 : Title-Digital data – an Imprint

Date & Time : 19.8.24 11.00 AM – 11.50 AM

Dr. Bhramaramba Ravi
Professor
Department of Computer Science and Engineering
GITAM School of Technology (GST)
Visakhapatnam – 530045
Email: **bravi@gitam.edu**

Unit2-syllabus

- **UNIT 2 Digital Data-An Imprint 9 hours, P - 2 hours** Type of data analytics (Descriptive, diagnostic, perspective, predictive, Prescriptive.) Exploratory Data Analysis (EDA), EDA-Quantitative Technique, EDA - Graphical Technique. Data Types for Plotting, Data Types and Plotting, Simple Line Plots, Simple Scatter Plots, Visualizing Errors, Density and Contour Plots, Histograms, Binnings, and Density, Customizing Plot Legends, Customizing Color bars, Multiple Subplots, Text and Annotation, Customizing Ticks.
- https://www.coursera.org/learn/data-visualization-r

Histograms, binnings and density

- A simple histogram can be a great first step in understanding a dataset.
- In[1]: %matplotlib inline
- **import numpy as np**
- **import matplotlib.pyplot as plt**
- plt.style.use('seaborn-white')
- data = np.random.randn(1000)
- In[2]: plt.hist(data);

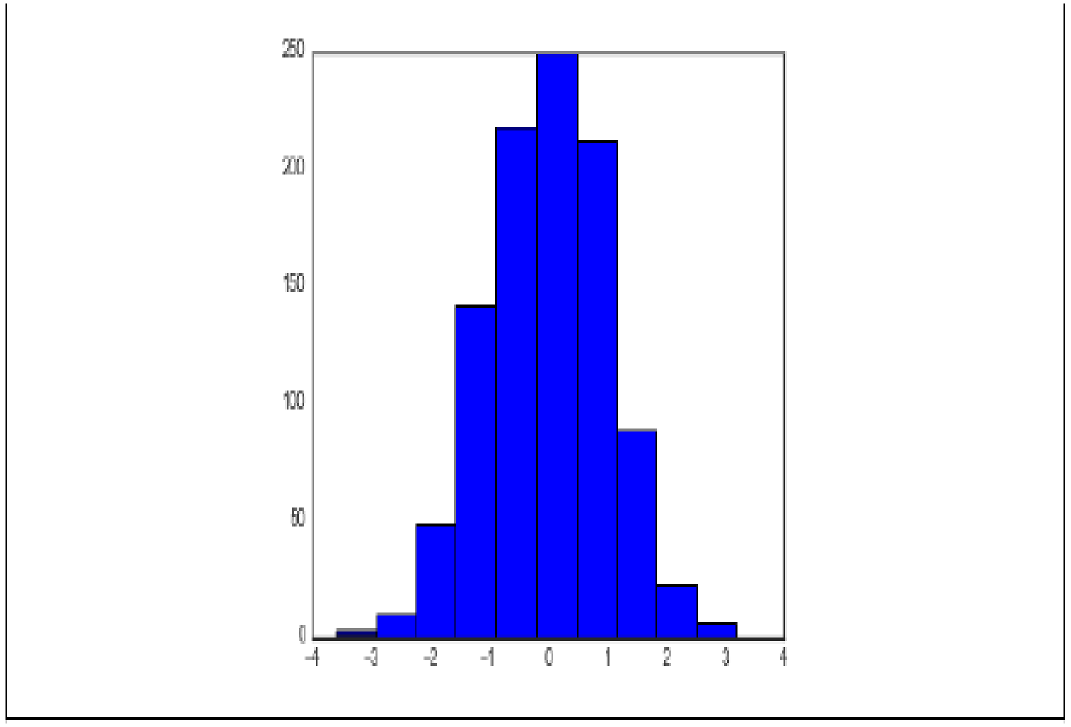

Figure 4-35. A simple histogram

Histograms

- The hist() function has many options to tune both the calculation and the display;
- here’s an example of a more customized histogram (Figure 4-36)
- In[3]: plt.hist(data, bins=30, normed=True, alpha=0.5,
- histtype='stepfilled', color='steelblue',
- edgecolor='none');

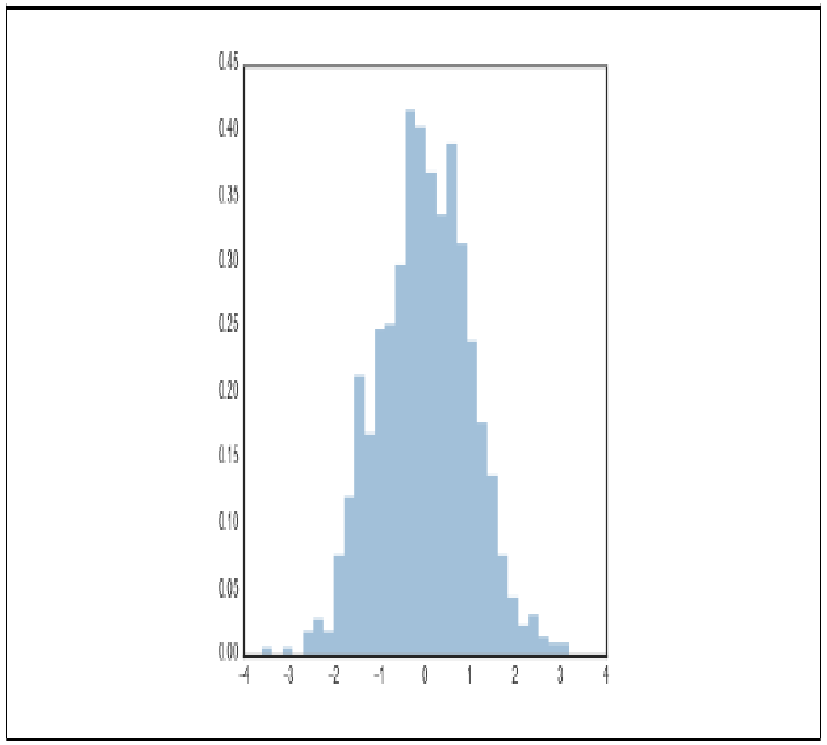

Figure 4-36. A customized histogram

Histograms

- The plt.hist docstring has more information on other customization options available.
- I find this combination of histtype='stepfilled' along with some transparency
- alpha to be very useful when comparing histograms of several distributions
- (Figure 4-37):
- In[4]: x1 = np.random.normal(0, 0.8, 1000)
- x2 = np.random.normal(-2, 1, 1000)
- x3 = np.random.normal(3, 2, 1000)
- kwargs = dict(histtype='stepfilled', alpha=0.3, normed=True, bins=40)
- plt.hist(x1, **kwargs)
- plt.hist(x2, **kwargs)
- plt.hist(x3, **kwargs);

Over-plotting multiple histograms

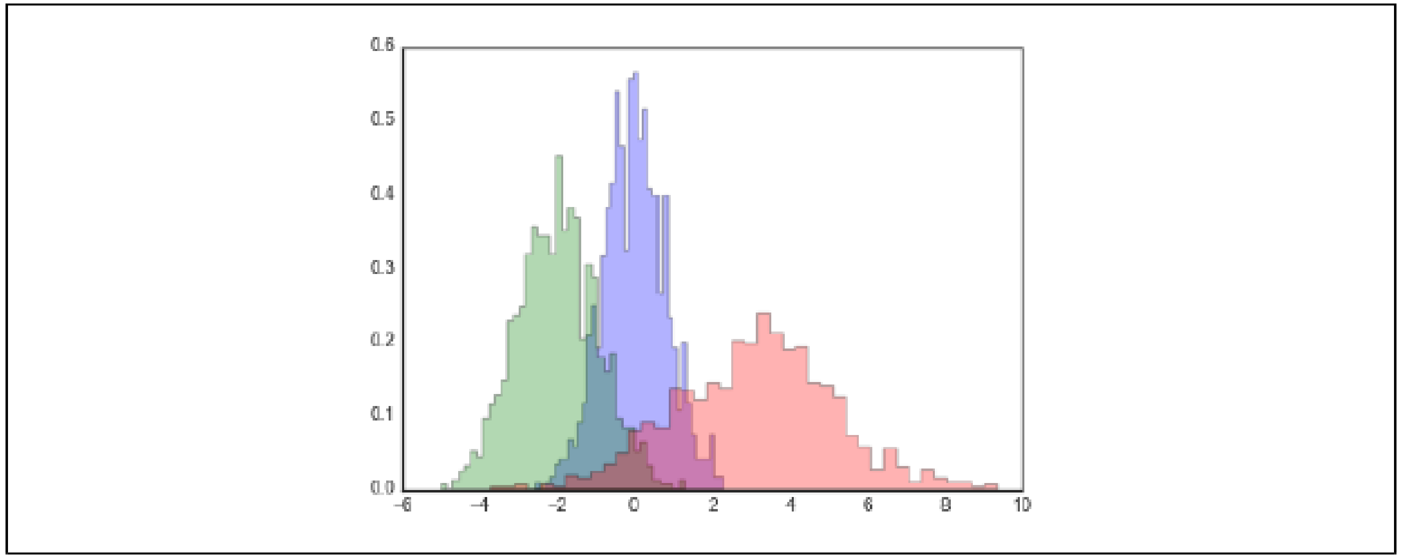

Figure 4-37. Over-plotting multiple histograms

Histograms

- If you would like to simply compute the histogram (that is, count the number of points in a given bin) and not display it, the np.histogram() function is available:
- In[5]: counts, bin_edges = np.histogram(data, bins=5)
- **print**(counts)
- [12 190 468 301 29]

Two dimensional histograms and binning

- Just as we create histograms in one dimension by dividing the number line into bins, we can also create histograms in two dimensions by dividing points among two dimensional bins.
- We'll start by defining some data—an x and y array drawn from a multivariate Gaussian distribution:
- In[6]: mean = [0, 0]
- cov = [[1, 1], [1, 2]]
- x, y = np.random.multivariate_normal(mean, cov, 10000).T

Two dimensional histograms and binning

- **plt.hist2d: Two-dimensional histogram**
- One straightforward way to plot a two-dimensional histogram is to use Matplotlib's plt.hist2d function (Figure 4-38):
- In[12]: plt.hist2d(x, y, bins=30, cmap='Blues')
- cb = plt.colorbar()
- cb.set_label('counts in bin')
- plt.hexbin has a number of interesting options, including the ability to specify weights for each point, and to change the output in each bin to any NumPy aggregate (mean of weights, standard deviation of weights, etc.).

Two dimensional histograms and binning

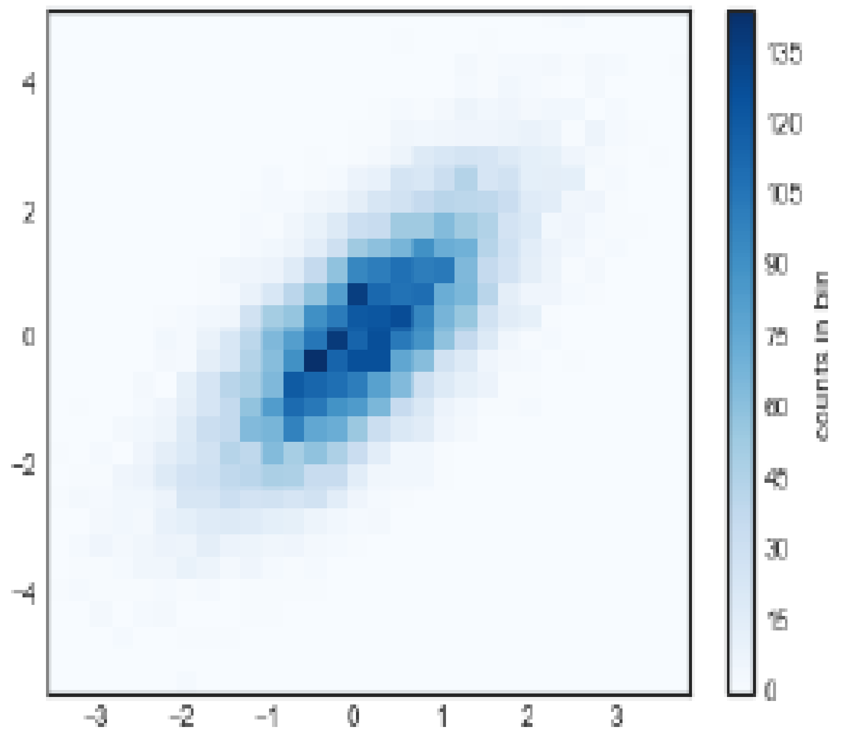

Figure 4-38. A two-dimensional histogram with plt.hist2d

Kernel density estimation

- Another common method of evaluating densities in multiple dimensions is *kernel density estimation* (KDE).
- KDE can be thought of as a way to “smear out” the points in space and add up the result to obtain a smooth function.
- One extremely quick and simple KDE implementation exists in the scipy.stats package.
- Here is a quick example of using the KDE on this data (Figure 4-40):
- In[10]: **from scipy.stats import** gaussian_kde
- *# fit an array of size [Ndim, Nsamples]*
- data = np.vstack([x, y])
- kde = gaussian_kde(data)
- *# evaluate on a regular grid*
- xgrid = np.linspace(-3.5, 3.5, 40)
- ygrid = np.linspace(-6, 6, 40)
- Xgrid, Ygrid = np.meshgrid(xgrid, ygrid)
- Z = kde.evaluate(np.vstack([Xgrid.ravel(), Ygrid.ravel()]))

Kernel density estimation

- *# Plot the result as an image*
- plt.imshow(Z.reshape(Xgrid.shape),
- origin='lower', aspect='auto',
- extent=[-3.5, 3.5, -6, 6],
- cmap='Blues')
- cb = plt.colorbar()
- cb.set_label("density")

Kernel density estimation

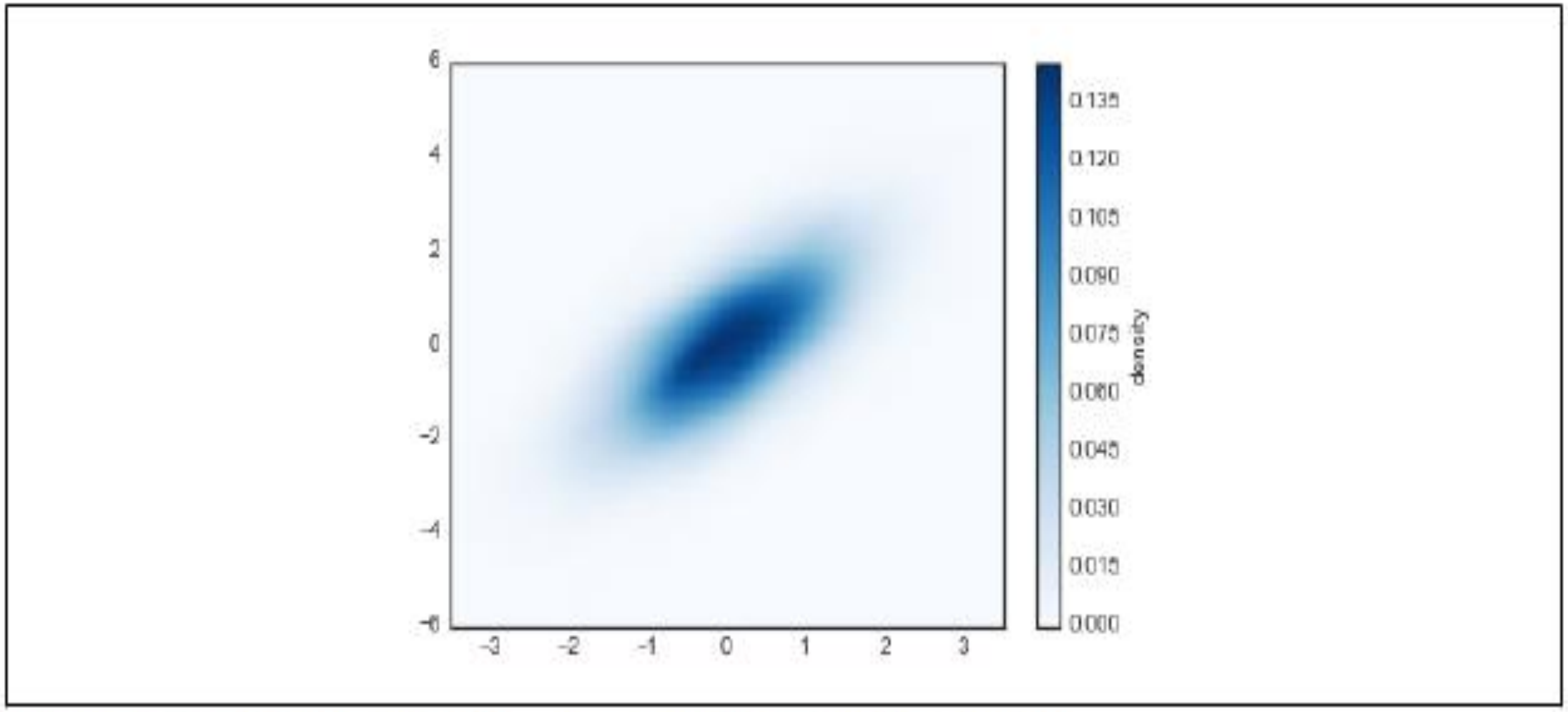

Figure 4-40. A kernel density representation of a distribution

Kernel density estimation

- KDE has a smoothing length that effectively slides the knob between detail and smoothness (one example of the ubiquitous bias–variance trade-off).
- The literature on choosing an appropriate smoothing length is vast: gaussian_kde uses a rule of thumb to attempt to find a nearly optimal smoothing length for the input data.
- Other KDE implementations are available within the SciPy ecosystem, each with its own various strengths and weaknesses; for example, sklearn.neighbors.Kernel Density and statsmodels.nonparametric.kernel_density.KDEMultivariate. For
- visualizations based on KDE, using Matplotlib tends to be overly verbose.
- The Seaborn library provides a much more terse API for creating KDE-based visualizations.

THANK YOU