Day 2.2 Hands on Introduction to Visualisation and Preprocessing with Python (updated, continuing)

December 19, 2018

Author:
Heru Praptono

In this part, we would like to demonstrate on how we use some related scientific Python libraries to do simple data visualisation. The data that we use in this occassion comes from **synthetic (toys) dataset**. This is generated from random number generator, following some specific distributions. In this tutorial we will use some popular distributions, including (but not limited to) uniform distribution, Gaussian distribution. Subsequently, this hands on demonstrates the simple data preprocessing.

We will discuse some general yet common graphics being used for visualisation. The goal is not only to help you get familiar on how we utilise some very common visualisation methods with Python, but also to get familiar with the relationship of **mathematical formulation/notation** vs **Python implementation.**

1 Visualisation: from Toys Dataset

1.1 Import the Common Required Modules

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

1.2 (Optional, if needed) Import warning and disable any warning

This is rather an optional, just to disable any warning, so that the output becomes nice

```
In [2]: import warnings
        warnings.filterwarnings('ignore')
```

1.3 Start Visualisation

```
In [3]: # (Optional) controlling the font style
        plt.rc('text', usetex=True)
        plt.rc('font', family='serif')
        plt.rc('font', size=12)
```

1.3.1 Simple Sinusoidal Graphic

An example figure below shows a simple sinusoidal graphic, as

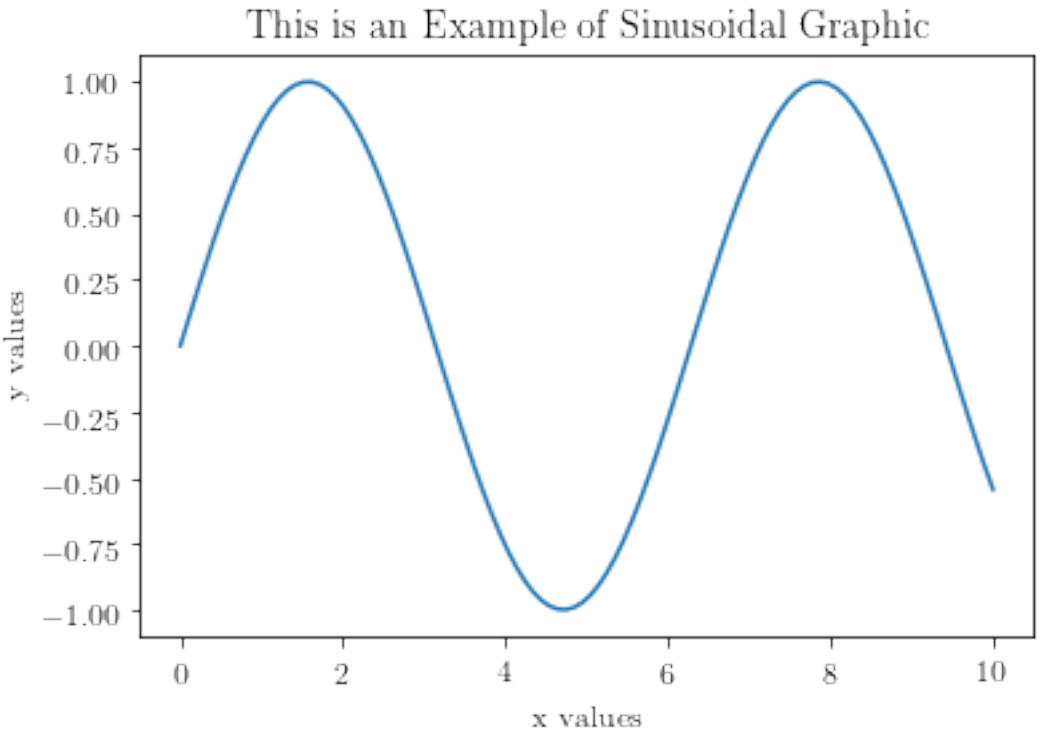
$$y = sin(x)$$

where $x \in \mathbb{R}$.

The `linspace` command having parameter(a,b,N), here, is to initialise the x values with the range of $[a, b]$ -- inclusive, where N is the number of points that we would like to generate. The a and b values represent starting number and ending number. Each step is equal to each other, that is on the other hand each x one to other has the same interval.

```
In [4]: x = np.linspace(0,10,100)
        y = np.sin(x)
        plt.xlabel('x values')
        plt.ylabel('y values')
        plt.title('This is an Example of Sinusoidal Graphic')
        plt.plot(x,y)
```

```
Out[4]: [<matplotlib.lines.Line2D at 0x7f10bcbdf780>]
```



1.3.2 Simple Linear Relationship Graphic

Suppose that we want to simulate a very simple linear (in parameter) regression. We know that in general, linear regression is of the form:

$$y = w\phi(x) + b + \epsilon$$

where $x \in \mathbb{R}$, w is th weight (this represents the slope), $\phi(x)$ is any basis function for x, before it is multiplied with the weight, and b is simply an intercept (sometimes we ignore this term). The ϵ represents the random noise that comes from, say, the observation. This random noise usualy assumed to be distributed normally, having centered in 0 ($\mu = 0$) with the variance 1 ($\sigma^2 = 1$), that is

$$\epsilon \sim \mathcal{N}(0, 1)$$

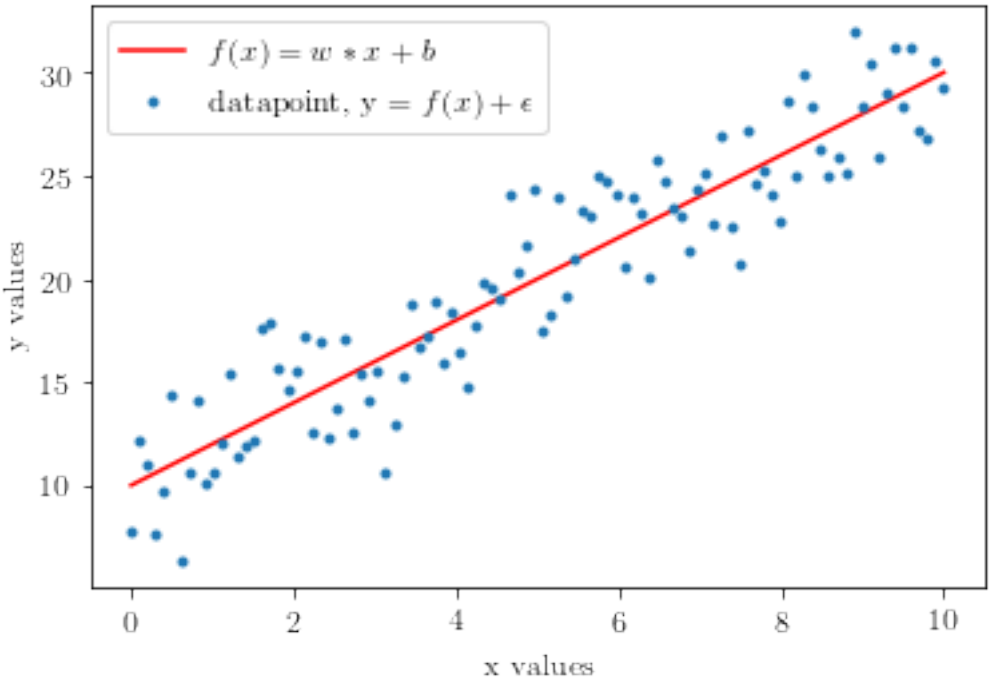
If the datapoint is rather a multivariate D dimensional variable (that is consists of D features), then the representative notation becomes $\mathbf{x}$ (bold x), where $\mathbf{x} = (x_1, x_2, ...x_D)$. Therefore, for multivariate datapoint we represent it as a D dimensional vector.

Now we would like to create a visualisation for one dimensional of x (x is univariate). The basis function maps x into its value, that is $\phi(x) = x$. Our linear equation becomes

$$y = wx + b + \epsilon$$

Amazing fact: Linear (in parameter) regression gives **the central concept/idea** for **so many Machine Learning algorithms**!

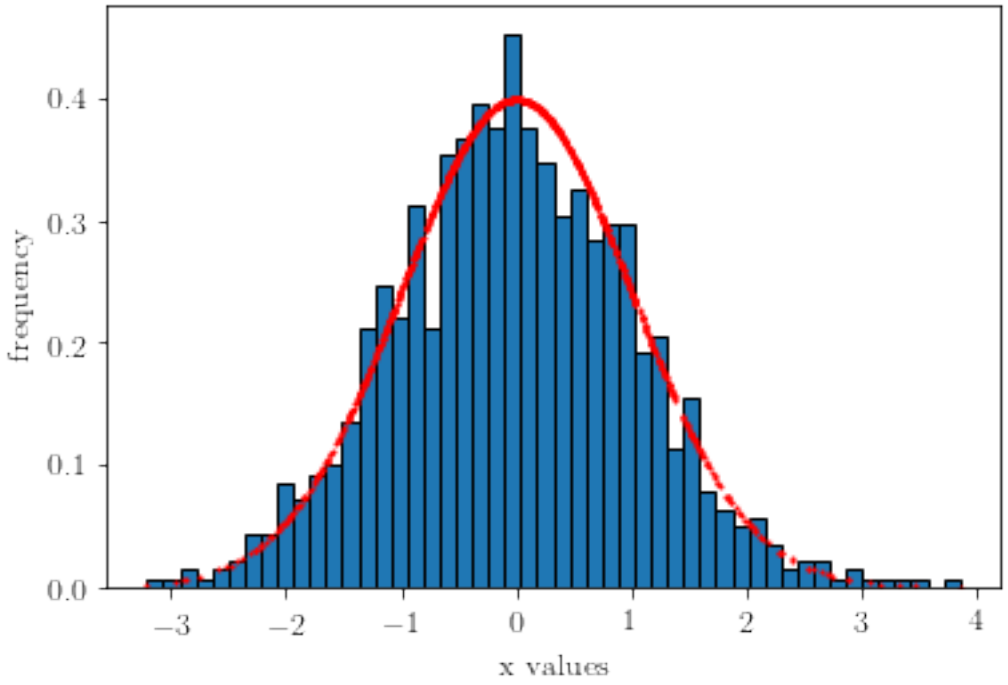
```
In [5]: N = 100
        x = np.linspace(0,10,N)
        w = 2
        b = 10

        np.random.seed(123)
        eps = np.random.normal(0,2,np.shape(x))

        y = w * x + b + eps

        plt.plot(x,w*x+b,label = r"$f(x) = w * x + b$",color='r')
        plt.plot(x,y,'.',label='datapoint, y = $f(x) + \epsilon$')
        plt.legend()
        plt.xlabel('x values')
        plt.ylabel('y values')

Out[5]: Text(0,0.5,'y values')
```



1.3.3 Histogram

A histogram is also already provided by patplotlib. This is a very useful tool not only for visualisation, but also as a simple density estimator. Histogram is usualy (but not limited to) used in the stage of exploratory data analysis.

```
In [6]: #import matplotlib.mlab as mlab
        from scipy.stats import norm

        N = 1000
        np.random.seed(100)
        x = np.random.normal(0,1,N)
        y = norm.pdf(x,0,1)
        plt.plot(x,y,'.',markersize=2, color='red')
        plt.hist(x, 50, density=True, color=None,edgecolor='black')

        plt.xlabel('x values')
        plt.ylabel('frequency')
        plt.show()
```



1.3.4 Scatter Plot

A scatter plot can be seen as the way to visualise multivariate variables (more specifically, bivariate variables). It is indeed a two or three dimensional cartesian coordinates. This plot is very useful in visualising the relationship of two (or three) variables, and usualy is also used in the stage of exploratory data analysis.

Suppose that we would like to visualise a dataset that contains of two groups. Both are normally distributed, but are having different central. Assume that those two distribution has the unit variances. For the group 1,

$$\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbb{I}_2)$$

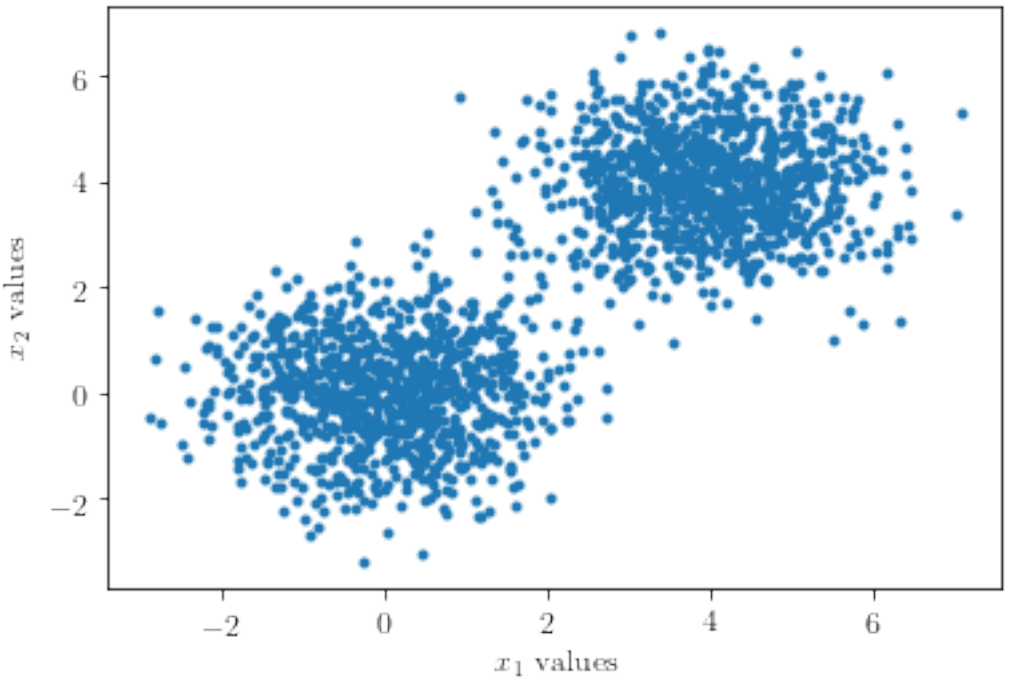
where $\mathbf{x} = (x_1, x_2)^T$ (here T represents matrix transpose), and $\mathbb{I}_2$ is a 2x2 identity matrix (identity matrix: a diagonal matrix, with value 1 on its diagonal). On the other hand, for the group 2,

$$\mathbf{x} \sim \mathcal{N}(\mathbf{0} + (4,4)^T, \mathbb{I}_2)$$

It is a simple translation of (4,4) from the group 1. The visualisation therefore can be seen as the following:

```
In [7]: N = 1000
        mean = np.asarray([0,0])
        cov = np.eye(np.size(mean))

        np.random.seed(124)
        x1_1, x1_2 = np.random.multivariate_normal(mean, cov, N).T
```

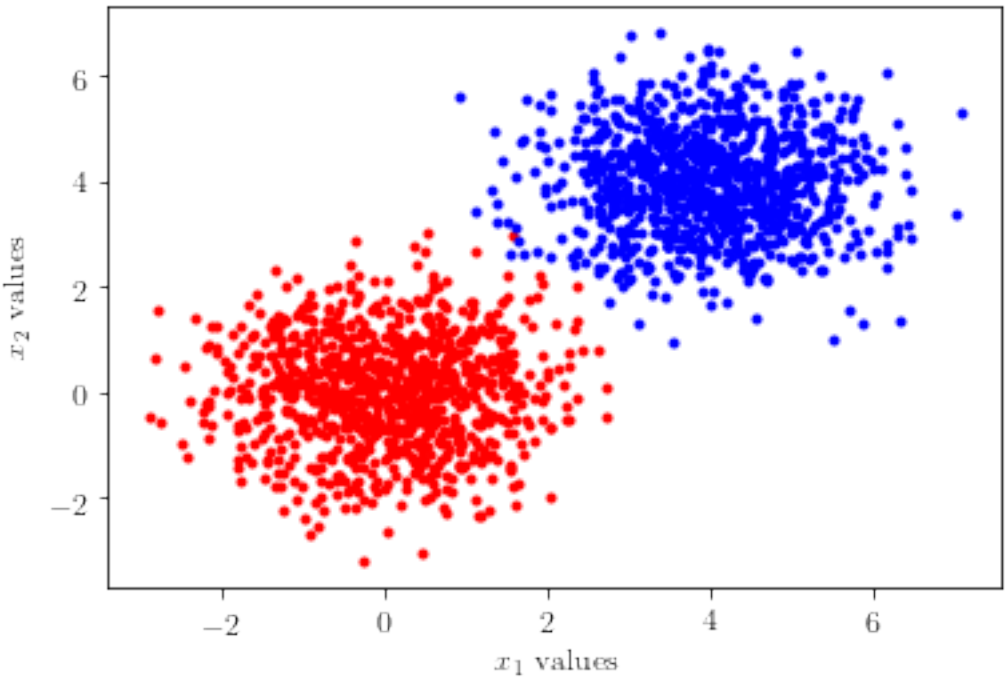
```
x2_1, x2_2 = np.random.multivariate_normal(mean+4, cov, N).T


xx1 = np.hstack((x1_1,x2_1))
xx2 = np.hstack((x1_2,x2_2))

#plot all data with the same color
plt.plot(xx1,xx2,'.')

plt.xlabel(r'$x_1$ values')
plt.ylabel(r'$x_2$ values')
plt.show()
```



Alternatively, if we want to show different colour for different group, we can use the colour property in the `plot()` function as given by pyplot. For example, in our case, we set red colour for the data points of group 1, and blue colour for the data points of group 2. The graphic thus becomes:

```
In [8]: N = 1000
        mean = np.asarray([0,0])
        cov = np.eye(np.size(mean))

        np.random.seed(124)
        x1_1, x1_2 = np.random.multivariate_normal(mean, cov, N).T
        x2_1, x2_2 = np.random.multivariate_normal(mean+4, cov, N).T
```

```
#plot all data with the same color
plt.plot(x1_1,x1_2,'.', color='red')
plt.plot(x2_1,x2_2,'.', color='blue')


plt.xlabel(r'$x_1$ values')
plt.ylabel(r'$x_2$ values')
plt.show()
```

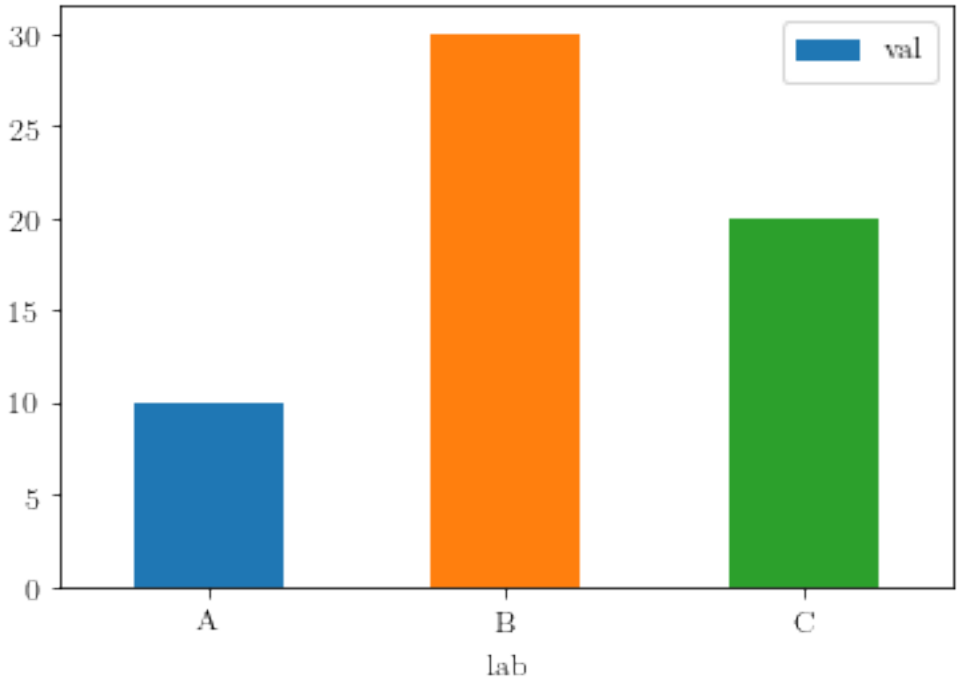


1.3.5 Bar Plot (with Pandas)

Pandas also provides visualisation, as well as on numpy. An example is a simple barchart below, visualising very simple generated data. You may also try different dataset forms and enhance the graphic!

Amazingly, Pandas is built on the top of numpy. This means that it works over numpy.

```
In [9]: #with pandas
        import pandas as pd
        df = pd.DataFrame({'lab':['A', 'B', 'C'], 'val':[10, 30, 20]})
        ax = df.plot.bar(x='lab', y='val', rot=0)
```

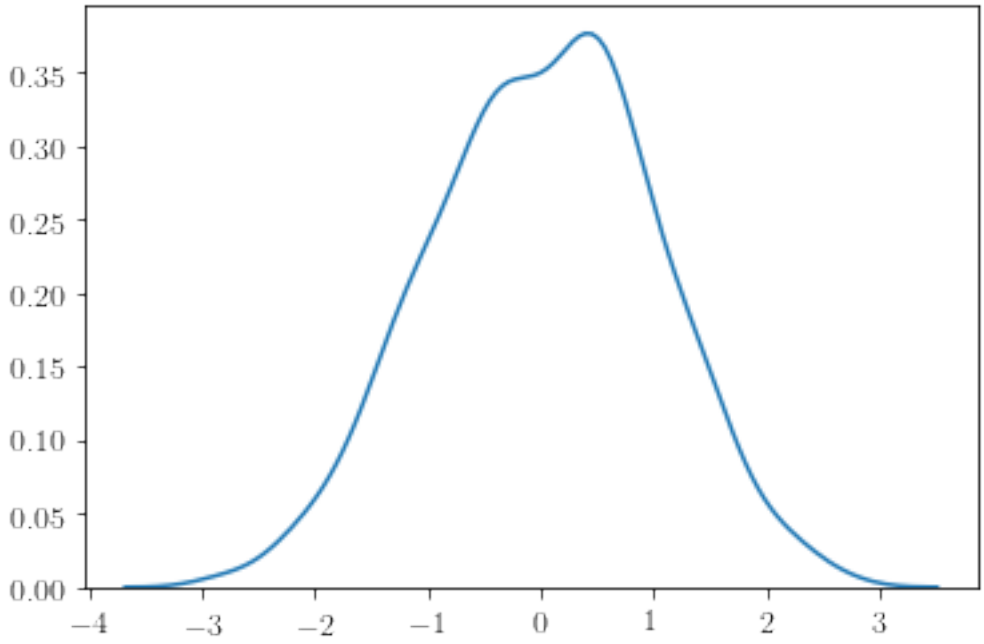


Task You may try another visualisation tools provided by Pandas! e.g. for visualising time series, composite data, or any other purpose.

1.3.6 Kernel Density Plot (with Seaborn)

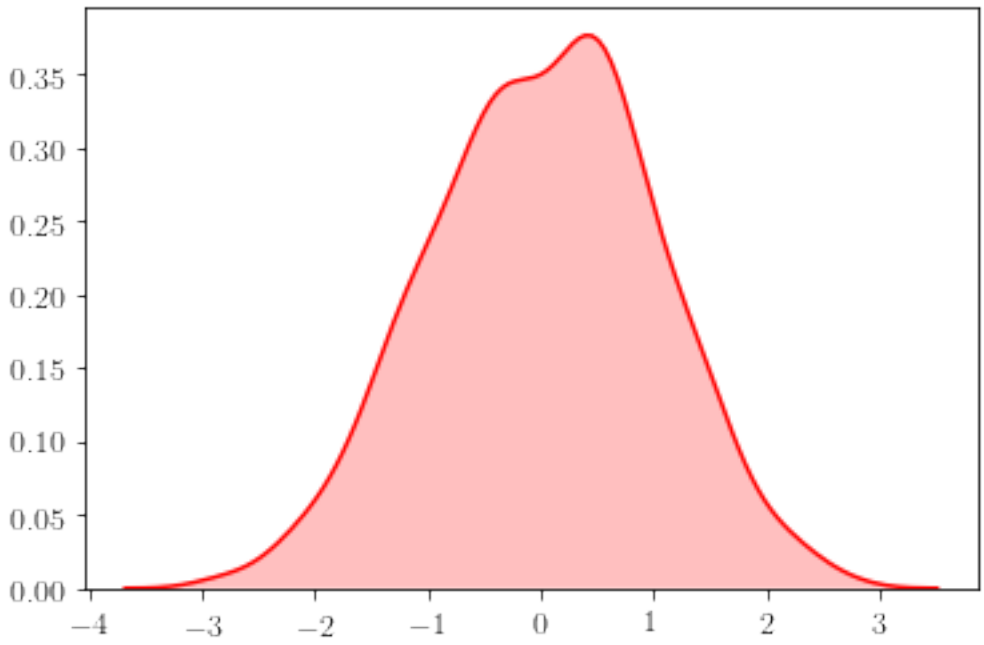
Another Python visualisation tool that can be used is Seaborn. The graphic below depicts a visualisation of kernel density estimate. Just like a histogram, a kernel density estimate is a density estimator for a given datapoints.

```
In [10]: #with seaborn https://seaborn.pydata.org/generated/seaborn.kdeplot.html
         import seaborn as sns
         ax = sns.kdeplot(x1_1)
```



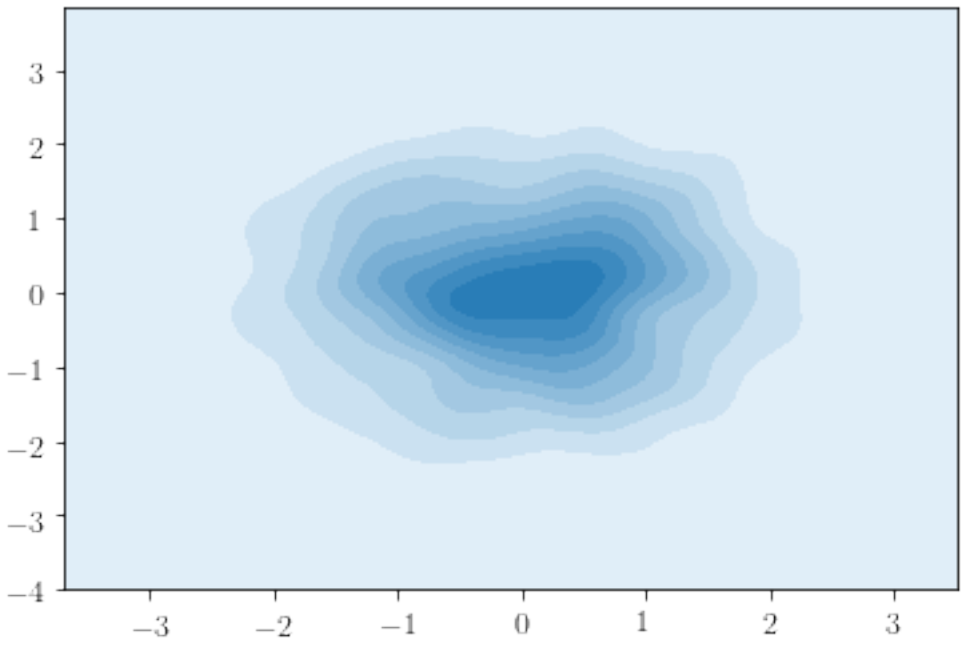
If we would like to add any shadow under the density curve, simply we specify `color` attribute becomes let us say "r" for red, or just use "red".

```
In [11]: ax = sns.kdeplot(x1_1, shade=True, color="r")
```



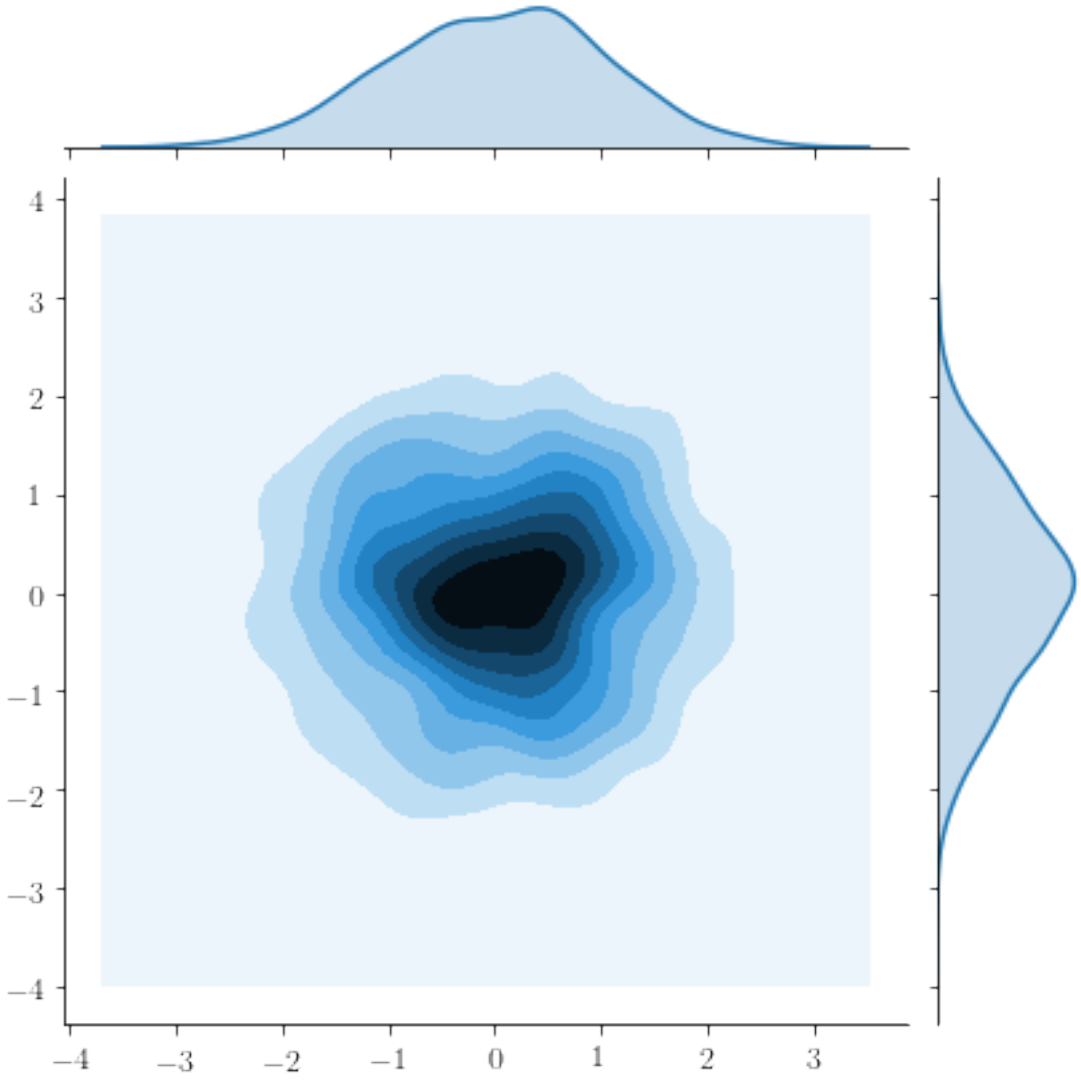
Previously, we visualise univariate data points, that is any $x \in \mathbb{R}$. Now, we would like show bivariate datapoints, that is any $x \in \mathbb{R}^2$. The plot can be seen as follows:

```
In [12]: ax = sns.kdeplot(x1_1, x1_2, shade=True)
```



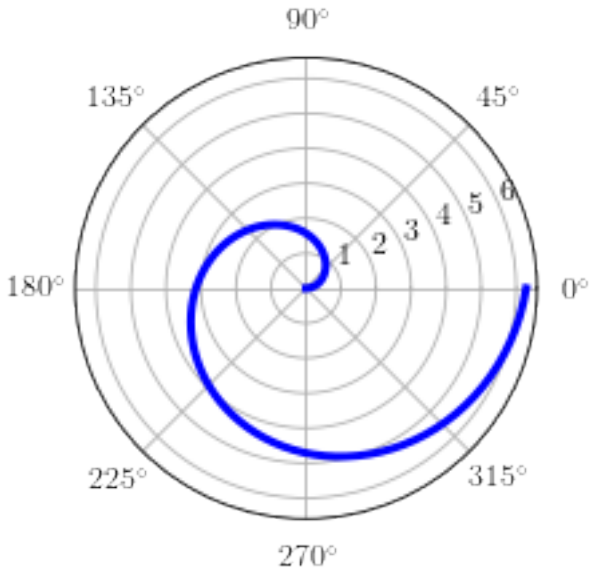
We can also make combine the two dimensional density estimation plot with one dimensional estimation on each. By using seaborn, we simply call the function `jointplot()`. The graphic can therefore be seen as the following:

```
In [13]: #https://seaborn.pydata.org/generated/seaborn.jointplot.html
         ax = sns.jointplot(x1_1,x1_2, kind='kde')
```

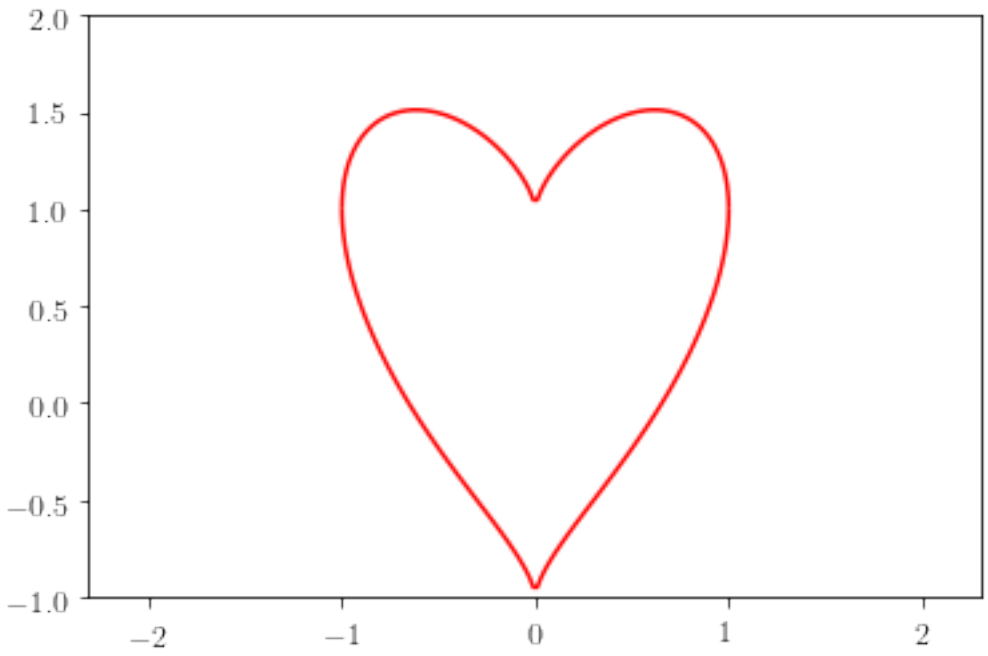

4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3

1.3.7 Miscellaneous

```
In [14]: fig = plt.figure()
         ax = fig.add_axes([0.0, 0.0, .6, .6], polar=True)
         t = np.linspace(0, 2 * np.pi, 100)
         ax.plot(t, t, color='blue', lw=3);
```




```
In [15]: y, x = np.ogrid[-1:2:100j, -1:1:100j]
         plt.contour(x.ravel(),
                     y.ravel(),
                     x**2 + (y-((x**2)**(1.0/3)))**2,
                     [1],
                     colors='red',)
         plt.axis('equal')
         plt.show()
```



2 Preprocessing

We will do some simple preprocessing with Python.

2.1 Scaling

2.1.1 MinMax Scaling

%%latex

The MinMax scaling is simply convert linearly the original value x, into the new nother value x_{new}, with the folling:

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

. By following that formula, all the x's will be forced to fall in the range $[0, 1]$. The generalisation for MinMax scaling is:

$$x_{new} = x_{std} * (b - a) + a$$

where $x_{std} = \frac{x - x_{min}}{x_{max} - x_{min}}$ and a, b are the minimum and maximum value for the range $[a, b]$.

Now, suppose that we have synthetic data, say uniformly generated from **np.random.rand**, with the shape 10 x 3 (which means that we have 10 data points, each has 3 attributes). We then do the MinMaxScaler to the data, conditional on each column. First, import MinMaxScaler, and specify the setting as the following:

```
In [16]: from sklearn.preprocessing import MinMaxScaler
         minmaxscaler = MinMaxScaler(feature_range=(0, 1))
         #if not specified, default feature range will be in [0,1]
```

Now, generate random number with the shape 10 x 3

```
In [17]: np.random.seed(1500)
         X = np.random.rand(10,3)
         print(X)
```

```
[[0.98451502 0.10019498 0.71348497]
 [0.14298264 0.6412398  0.90647641]
 [0.58426722 0.35536841 0.47612775]
 [0.44096615 0.65555366 0.94144979]
 [0.78338986 0.9915377  0.04527771]
 [0.65264265 0.71571167 0.04051945]
 [0.32555806 0.69552568 0.4198415 ]
 [0.15818161 0.98608914 0.17239575]
 [0.08682796 0.46574264 0.64864652]
 [0.10907593 0.92440577 0.2639907 ]]
```

do the transformation:

```
In [18]: X_scaled = minmaxscaler.fit_transform(X)
         print(X_scaled)
```

```
[[1.         0.         0.74696731]
 [0.06255486 0.60699976 0.96118082]
 [0.55413438 0.28627981 0.48350942]
 [0.39450073 0.62305852 1.        ]
 [0.77595181 1.         0.00528149]
 [0.63030283 0.69054997 0.        ]
 [0.26593912 0.66790325 0.42103372]
 [0.07948611 0.99388725 0.14637791]
 [0.         0.41010899 0.67499898]
 [0.02478366 0.9246845  0.24804498]]
```

2.1.2 Standard Scaling

%%latex The standard scaling aims to transform the data into the "original" position. The transformation follows the equation below:

$$x_{new} = \frac{x - \mu}{\sigma}$$

where

$$\mu = \frac{1}{N}\sum_{i}^{N} x_i$$

and

$$\sigma = \frac{1}{N}\sum_{i}^{N} (x_i - \mu)^2$$

The "pipeline" is same as previous MinMax scaling method. Specifically, if we set `with_mean = False`, and `with_std = False`, the mean and standard deviation become 0 and 1, thus, there will be no difference between x and x_{new}.

```
In [19]: from sklearn.preprocessing import StandardScaler
         standardscaler = StandardScaler(with_mean=True, with_std=True)
         np.random.seed(1500)
         X = np.random.rand(10,3)
         print(X)
         X_scaled = standardscaler.fit_transform(X)
         print('-'*20)
         print(X_scaled)
```

```
[[0.98451502 0.10019498 0.71348497]
 [0.14298264 0.6412398  0.90647641]
 [0.58426722 0.35536841 0.47612775]
 [0.44096615 0.65555366 0.94144979]
 [0.78338986 0.9915377  0.04527771]
 [0.65264265 0.71571167 0.04051945]
 [0.32555806 0.69552568 0.4198415 ]
 [0.15818161 0.98608914 0.17239575]
 [0.08682796 0.46574264 0.64864652]
 [0.10907593 0.92440577 0.2639907 ]]
--------------------
[[ 1.86546379 -2.04046469  0.79193083]
 [-0.94952724 -0.04390281  1.40165512]
 [ 0.52660387 -1.09882451  0.0420403 ]
 [ 0.0472507   0.00891815  1.51214768]
 [ 1.19268454  1.24876539 -1.31915858]
 [ 0.755325    0.23091305 -1.3341915 ]
 [-0.3387983   0.15642278 -0.13578673]
 [-0.89868552  1.22865916 -0.91755033]
 [-1.13736902 -0.69152187  0.58708456]
 [-1.06294783  1.00103537 -0.62817135]]
```

2.2 Missing Data Imputation

```
In [20]: #check your scikit-learn version, make sure yours is 0.20 version.
         import sklearn
         print(sklearn.__version__)
```

```
0.20.1
```

In scikit learn, there is an option for imputation strategy, that is:

- "mean", replace missing value with the mean of the corresponding whole feature

- "median", replace missing value with the median of the corresponding whole feature
- "most_frequent" replace missing value with the modus of the corresponding whole feature
- "constant" replace missing value with the desired value that we specify

```
In [21]: from sklearn.impute import SimpleImputer
         imp = SimpleImputer(missing_values=np.nan, strategy='mean')
         imp.fit([[1, 2], [np.nan, 3], [7, 6]])

Out[21]: SimpleImputer(copy=True, fill_value=None, missing_values=nan, strategy='mean',
                verbose=0)

In [22]: #suppose we have a synthetic dataset that contains any
         #missing value, we can use pre-defined impute model
         import numpy as np
         X = [[np.nan, 2], [6, np.nan], [7, 6]]
         print(imp.transform(X))

[[4.         2.        ]
 [6.         3.66666667]
 [7.         6.        ]]
```

Task. Try different missing data imputation methods.