IEML

Yongduek Seo (yndk@sogang.ac.kr)

Oct 15, 2019

CONTENTS:

CHAPTER

ONE

NUMPY INTRO

1.1 Key Examples

- All of the following problems are found in the text book.
- Try to answer the questions by searching for appropriate `numpy` functions.
- Use jupyter notebook

1.2 1. What is the average height of US presidents?

- plot a graph of the heights for visual display through time
- plot a histogram of the heights to examine the distribution of heights
- Write a python script to find
  1. the tallest president
  2. the shorted president
  3. the average of the heights
  4. the median of the heghts

```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
import numpy as np
import pandas as pd
```

```
data = pd.read_csv('data/president_heights.csv')
heights = np.array(data['height(cm)'])
names = np.array(data['name'])
```

```
names
```

```
array(['George Washington', 'John Adams', 'Thomas Jefferson',
       'James Madison', 'James Monroe', 'John Quincy Adams',
       'Andrew Jackson', 'Martin Van Buren', 'William Henry Harrison',
       'John Tyler', 'James K. Polk', 'Zachary Taylor',
       'Millard Fillmore', 'Franklin Pierce', 'James Buchanan',
       'Abraham Lincoln', 'Andrew Johnson', 'Ulysses S. Grant',
```

(continues on next page)

(continued from previous page)

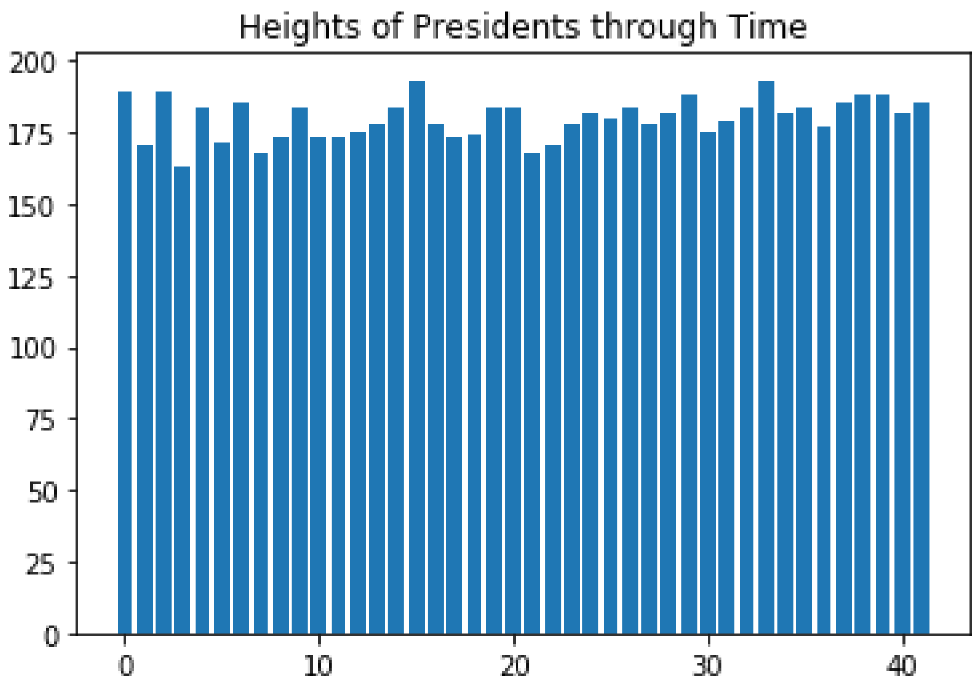
```
        'Rutherford B. Hayes', 'James A. Garfield', 'Chester A. Arthur',
        'Benjamin Harrison', 'William McKinley', 'Theodore Roosevelt',
        'William Howard Taft', 'Woodrow Wilson', 'Warren G. Harding',
        'Calvin Coolidge', 'Herbert Hoover', 'Franklin D. Roosevelt',
        'Harry S. Truman', 'Dwight D. Eisenhower', 'John F. Kennedy',
        'Lyndon B. Johnson', 'Richard Nixon', 'Gerald Ford',
        'Jimmy Carter', 'Ronald Reagan', 'George H. W. Bush',
        'Bill Clinton', 'George W. Bush', 'Barack Obama'], dtype=object)
```

```
data['name'][2]
```

```
'Thomas Jefferson'
```

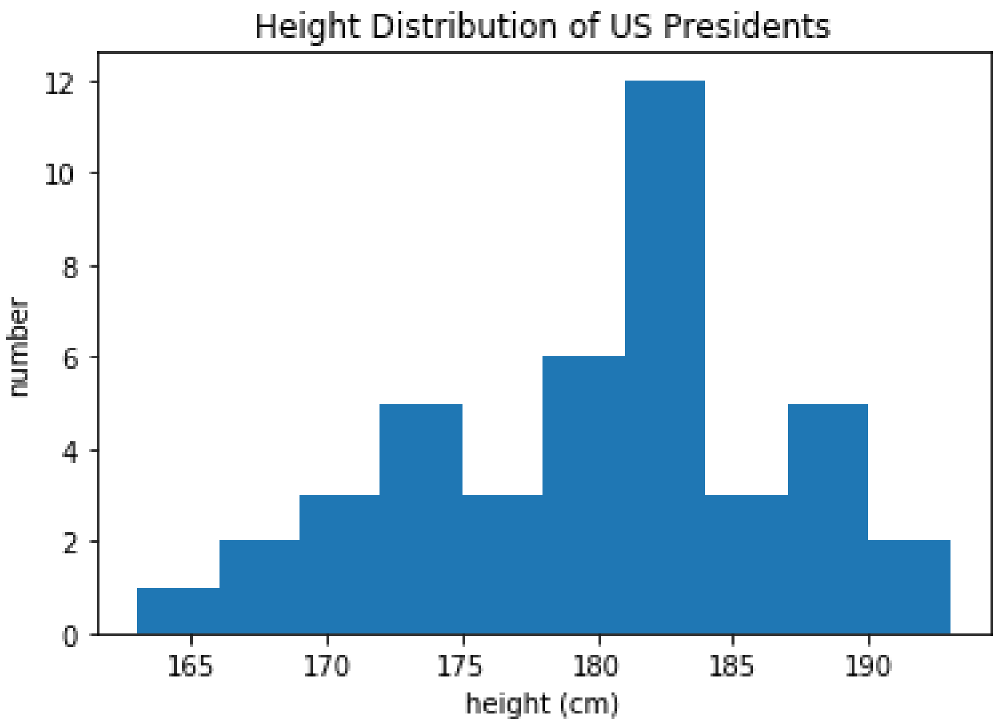
```
a = np.array (data['height(cm)'])

plt.bar(range(a.size), a)
plt.title('Heights of Presidents through Time')
```

```
Text(0.5, 1.0, 'Heights of Presidents through Time')
```



```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
plt.hist(heights)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number');
```



```
print("Maximum height:    ", heights.max())
```

```
Maximum height:     193
```

```
imax = heights.argmax()
```

```
print('The president of the tallest: ', data['name'][imax] )
```

```
The president of the tallest:  Abraham Lincoln
```

```
print("Minimum height:    ", heights.min())
```

```
Minimum height:     163
```

```
imin = heights.argmin()
```

```
print ('The president of the shortest: ', data['name'][imin])
```

```
The president of the shortest:  James Madison
```

```
print("Average height:    ", heights.std())
```

```
Average height:     6.931843442745892
```

```
print("Mean height:       ", heights.mean())
```

```
Mean height:         179.73809523809524
```

1.3 Count the number of alphabests used in the presidents' names

```
count = np.zeros(26, dtype=np.int)
type(count), count.shape, count.dtype, count
```

```
(numpy.ndarray,
 (26,),
 dtype('int64'),
 array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0]))
```

- Only capital letters will be used.
- The ASCII code of 'A' is 65. Try `print(ord('A'))`
- Therefore, the index to the array `count` is calculated by `ord(upper(c)) - 65`

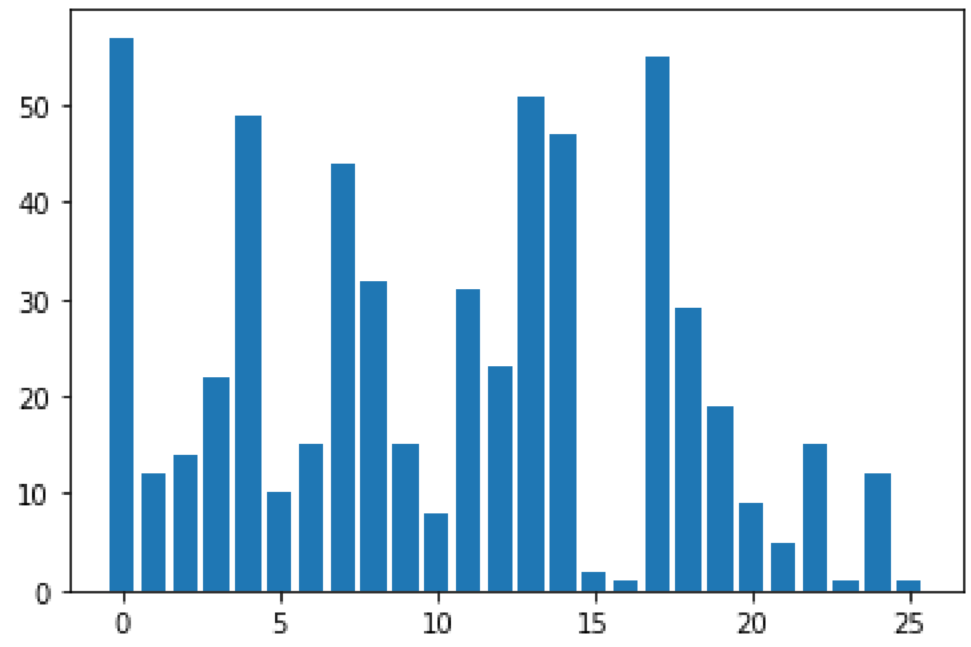
```
for name in data['name']:
    for c in name:
        if c == ' ': continue
        upper = c.upper()
        ind = ord(upper) - 65
        count[ind] += 1
```

```
count
```

```
array([57, 12, 14, 22, 49, 10, 15, 44, 32, 15,  8, 31, 23, 51, 47,  2,  1,
       55, 29, 19,  9,  5, 15,  1, 12,  1])
```

```
plt.bar(range(count.shape[0]), count)
```

```
<BarContainer object of 26 artists>
```



1.4 Compute score of each president according to the values in `count`

1.4.1 the array size must be the same as the size of presidents

```
score2 = np.zeros(names.shape[0], dtype=np.int)
score2
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

1.4.2 Another array to compute normalized `score3 = score2 / len(name)`

```
score3 = np.zeros(names.shape[0])
```

```
for i, name in zip(range(names.shape[0]), names):
   # print (i,name)
    score = 0
    for c in name:
        if c == ' ': continue
        upper = c.upper()
        ind = ord(upper) - 65
        score += count[ind]
    # print (name, score, score/len(name))
    score2[i] = score
    score3[i] = score/len(name)
```

```
score2
```

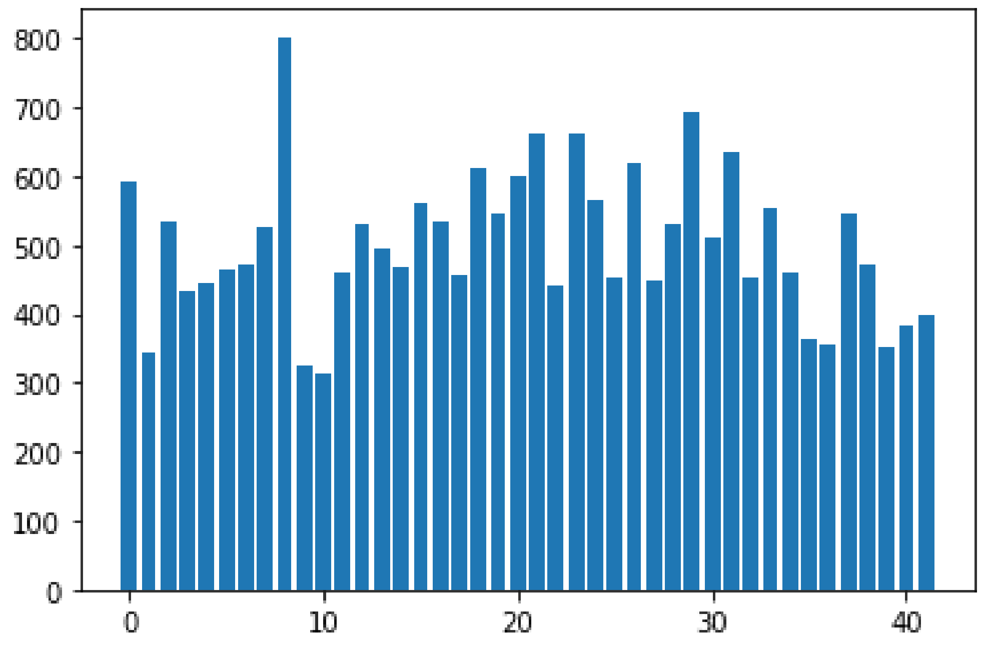
```
array([590, 345, 534, 434, 445, 464, 470, 526, 802, 323, 313, 461, 529,
       496, 468, 562, 533, 458, 612, 545, 599, 660, 441, 663, 566, 453,
       617, 447, 530, 692, 510, 633, 453, 554, 461, 363, 354, 547, 471,
       351, 383, 399])
```

```
score3
```

```
array([34.70588235, 34.5       , 33.375     , 33.38461538, 37.08333333,
       27.29411765, 33.57142857, 32.875     , 36.45454545, 32.3       ,
       24.07692308, 32.92857143, 33.0625    , 33.06666667, 33.42857143,
       37.46666667, 38.07142857, 28.625     , 32.21052632, 32.05882353,
       35.23529412, 38.82352941, 27.5625    , 36.83333333, 29.78947368,
       32.35714286, 36.29411765, 29.8       , 37.85714286, 32.95238095,
       34.        , 31.65      , 30.2       , 32.58823529, 35.46153846,
       33.        , 29.5       , 42.07692308, 27.70588235, 29.25      ,
       27.35714286, 33.25      ])
```

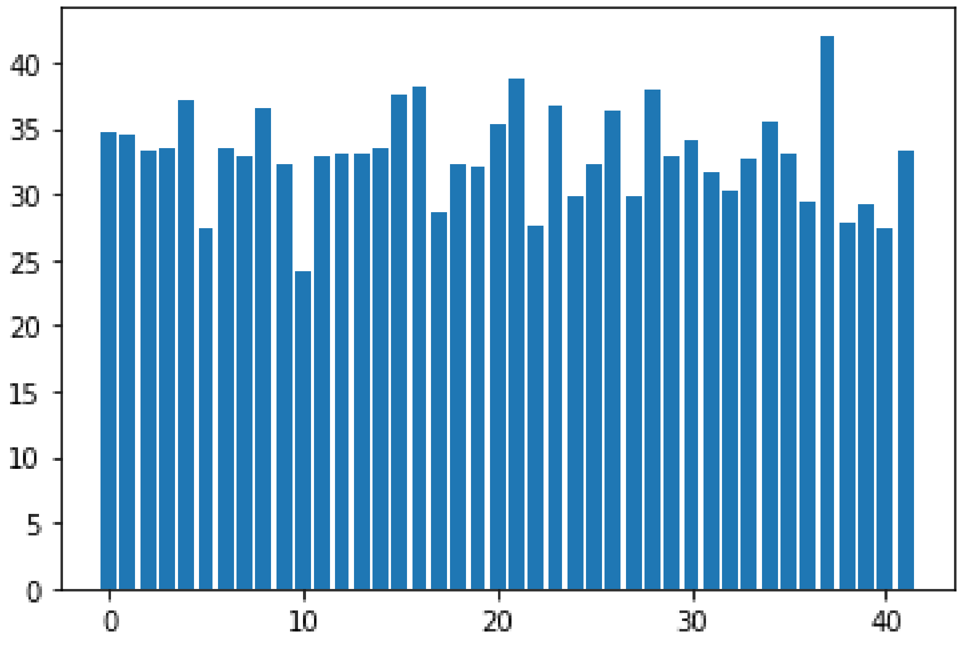
```
plt.bar(range(score2.shape[0]), score2)
```

```
<BarContainer object of 42 artists>
```



```
plt.bar(range(score3.shape[0]), score3)
```

```
<BarContainer object of 42 artists>
```



```
imax3 = score3.argmax()
```

```
print ('The president of the highest normalized score is ', names[imax3])
```

```
The president of the highest normalized score is  Ronald Reagan
```

CHAPTER

TWO

COMPUTE THE SCORE OF MY NAME

1. using the routines above, make a function that returns the score: `def get_score3 (name)`
2. test the function with `'Ronald Reagan'` and see if the score is the same as `score3[imax3]`
   - try other names too for sure to check the correctness of your function
3. try your name

```
def get_score3 (name):
    score = 0
    # your code here
    # -----

    # -----
    return score
#
```

CHAPTER

THREE

MATPLOTLIB PRIMER

```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
```

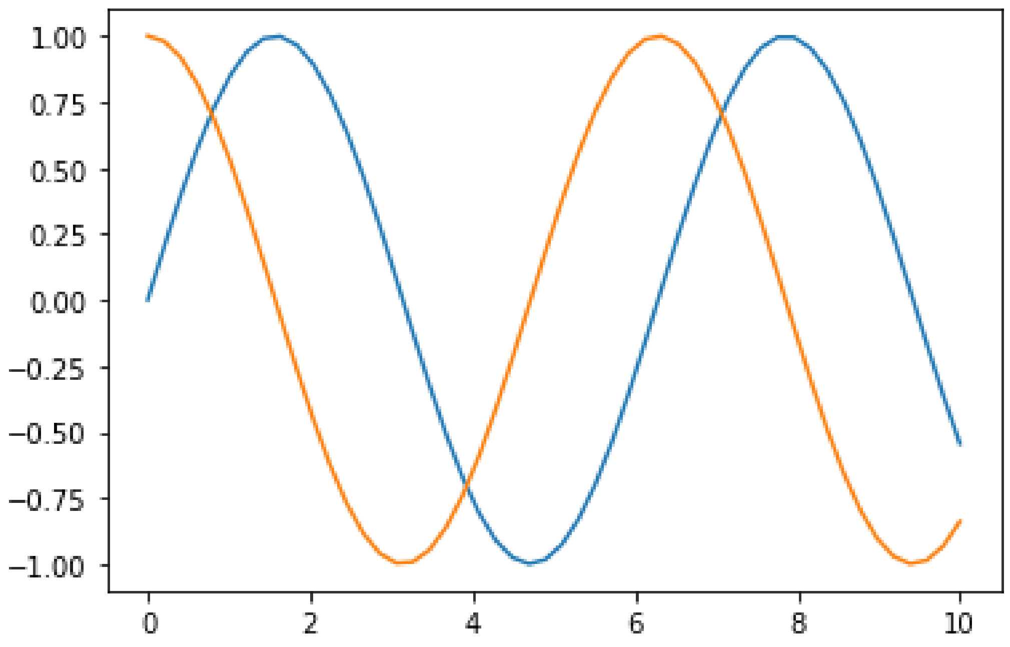
```
import numpy as np
```

```
x = np.linspace (0, 10, 50)
print ('x: ', x)
```

```
x:  [ 0.          0.20408163  0.40816327  0.6122449   0.81632653  1.02040816
  1.2244898   1.42857143  1.63265306  1.83673469  2.04081633  2.24489796
  2.44897959  2.65306122  2.85714286  3.06122449  3.26530612  3.46938776
  3.67346939  3.87755102  4.08163265  4.28571429  4.48979592  4.69387755
  4.89795918  5.10204082  5.30612245  5.51020408  5.71428571  5.91836735
  6.12244898  6.32653061  6.53061224  6.73469388  6.93877551  7.14285714
  7.34693878  7.55102041  7.75510204  7.95918367  8.16326531  8.36734694
  8.57142857  8.7755102   8.97959184  9.18367347  9.3877551   9.59183673
  9.79591837 10.        ]
```

```
plt.plot (x, np.sin(x))
plt.plot (x, np.cos(x))
```

```
[<matplotlib.lines.Line2D at 0x7f45e2b073c8>]
```

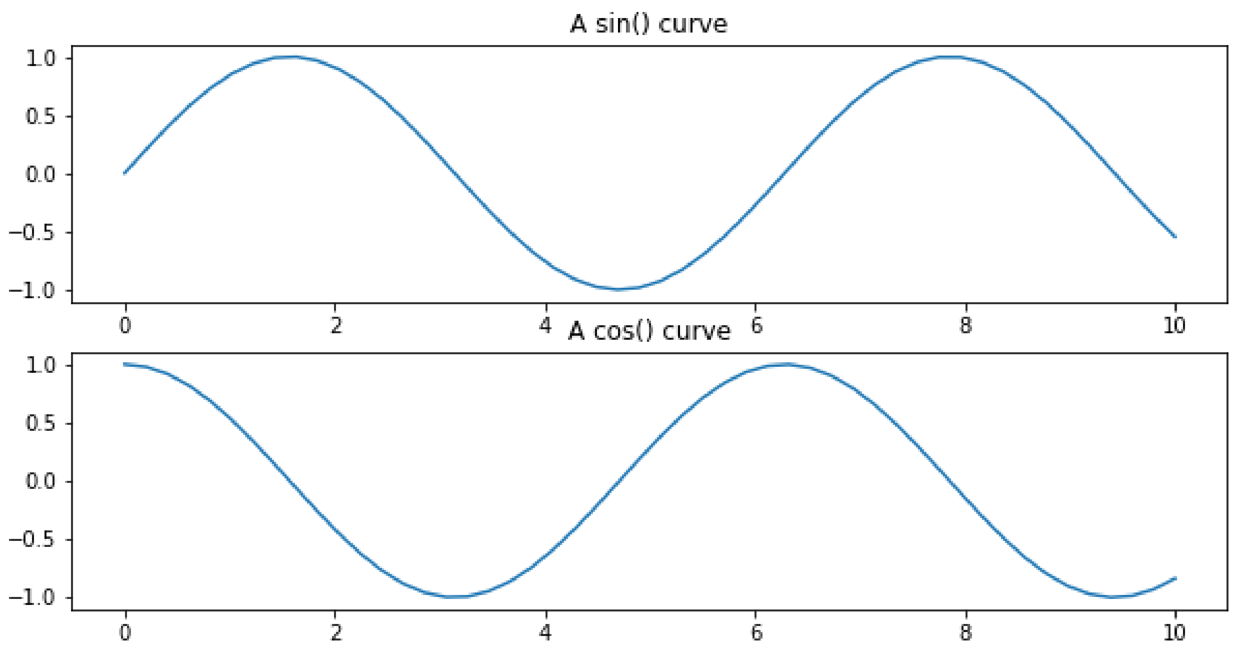


3.1 1. MATLAB-style interface

- Easy and simple plots
- this interface is **stateful**: it keeps track of the current figure and axes, which are where `plt` commands are applied. You can get a reference to these using the `plt.gcf()` to get the current figure and `plt.gca()` to get the current axes.

```
plt.figure(figsize=(10,5)) # create a plot figure of size(width, height)

# two panels will be created
# first axis
plt.subplot(2,1, 1)
plt.plot(x, np.sin(x))
plt.title ('A sin() curve')

# second axis
plt.subplot(2,1, 2)
plt.plot(x, np.cos(x))
plt.title ('A cos() curve')
```

```
Text(0.5, 1.0, 'A cos() curve')
```

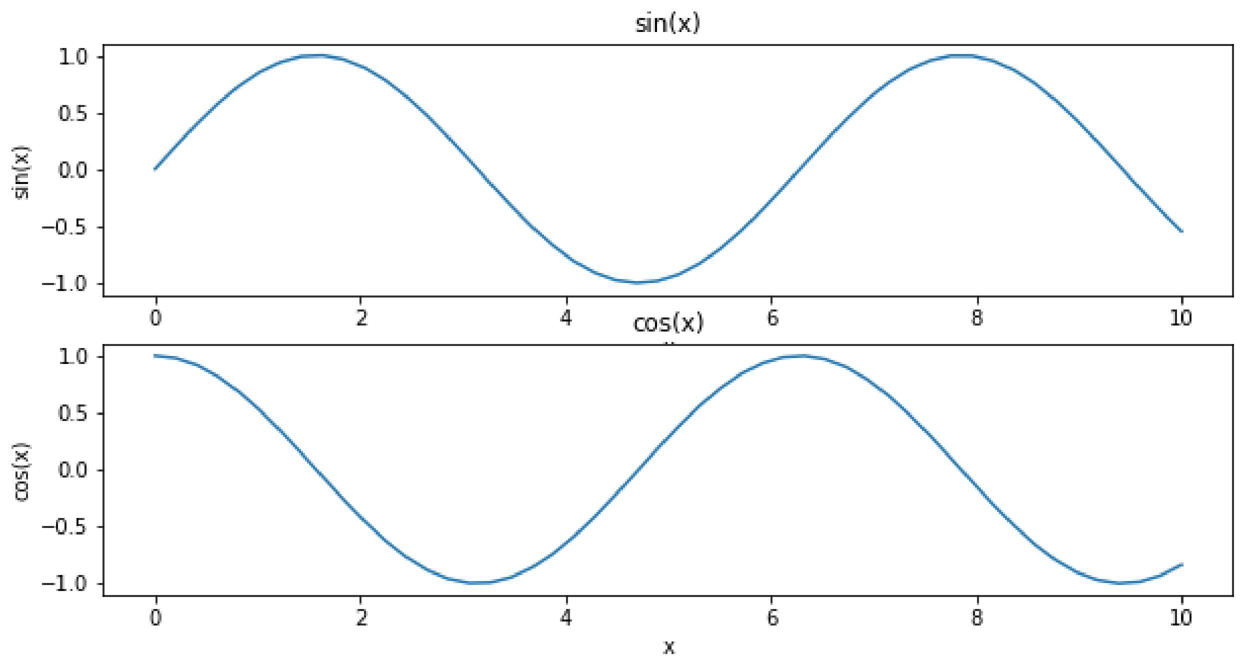


3.2 2. Object-oriented Interface

- more sophisticated

```
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(10,5))

axes[0].plot (x, np.sin(x))
axes[1].plot (x, np.cos(x))

axes[0].set(title='sin(x)', xlabel='x', ylabel='sin(x)')
axes[1].set(title='cos(x)', xlabel='x', ylabel='cos(x)')
```

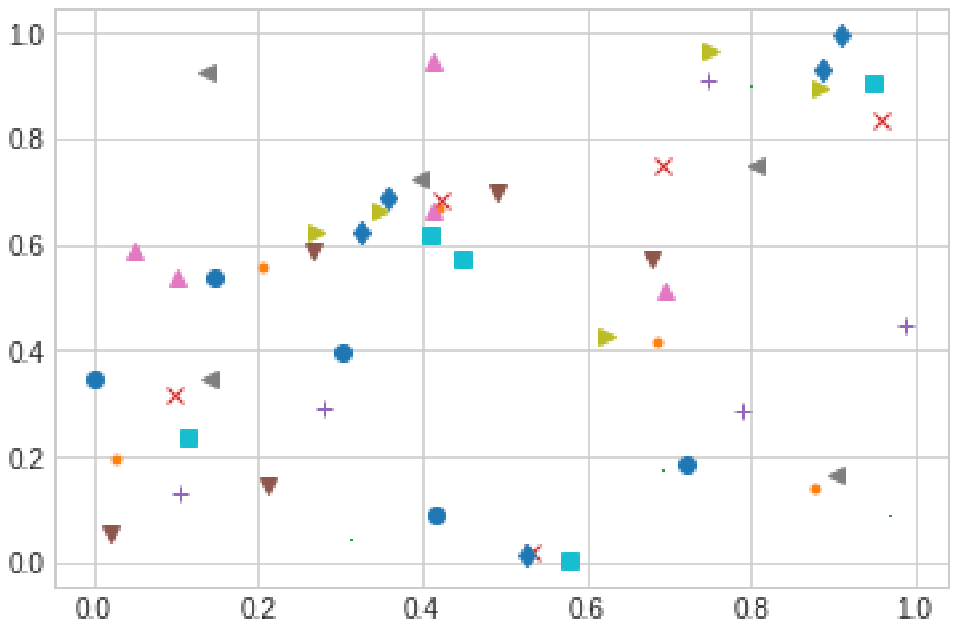
```
[Text(0, 0.5, 'cos(x)'), Text(0.5, 0, 'x'), Text(0.5, 1.0, 'cos(x)')]
```



3.3 3. Simple Scatter Plots

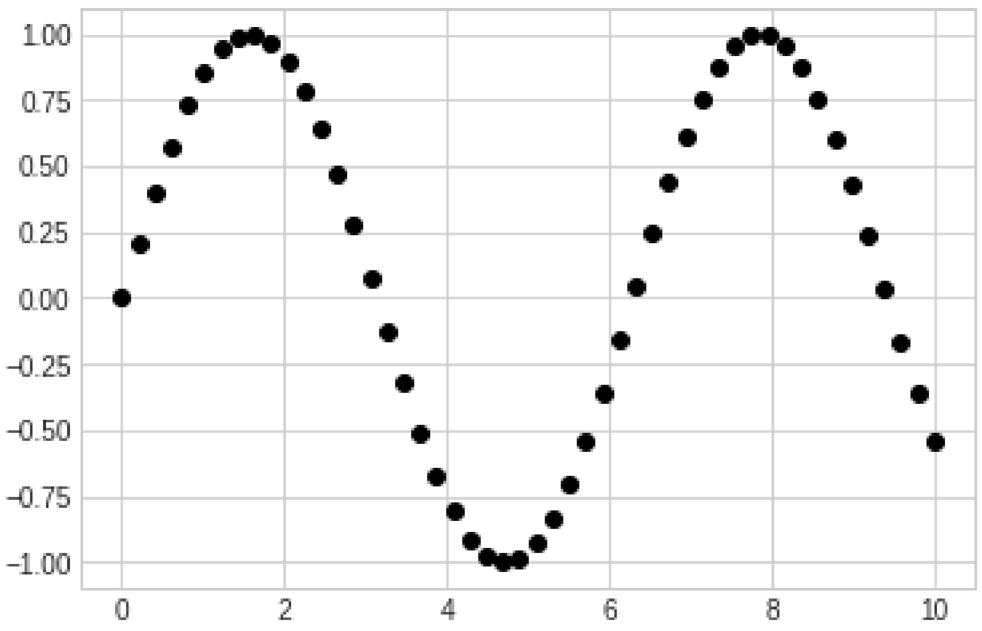
```
plt.style.use('seaborn-whitegrid')
```

```
rng = np.random.RandomState(1)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    xvalues = rng.rand(5) # random [0, 1)
    yvalues = rng.rand(5)
    plt.plot(xvalues, yvalues, marker, label="marker='{0}'".format(marker))
```

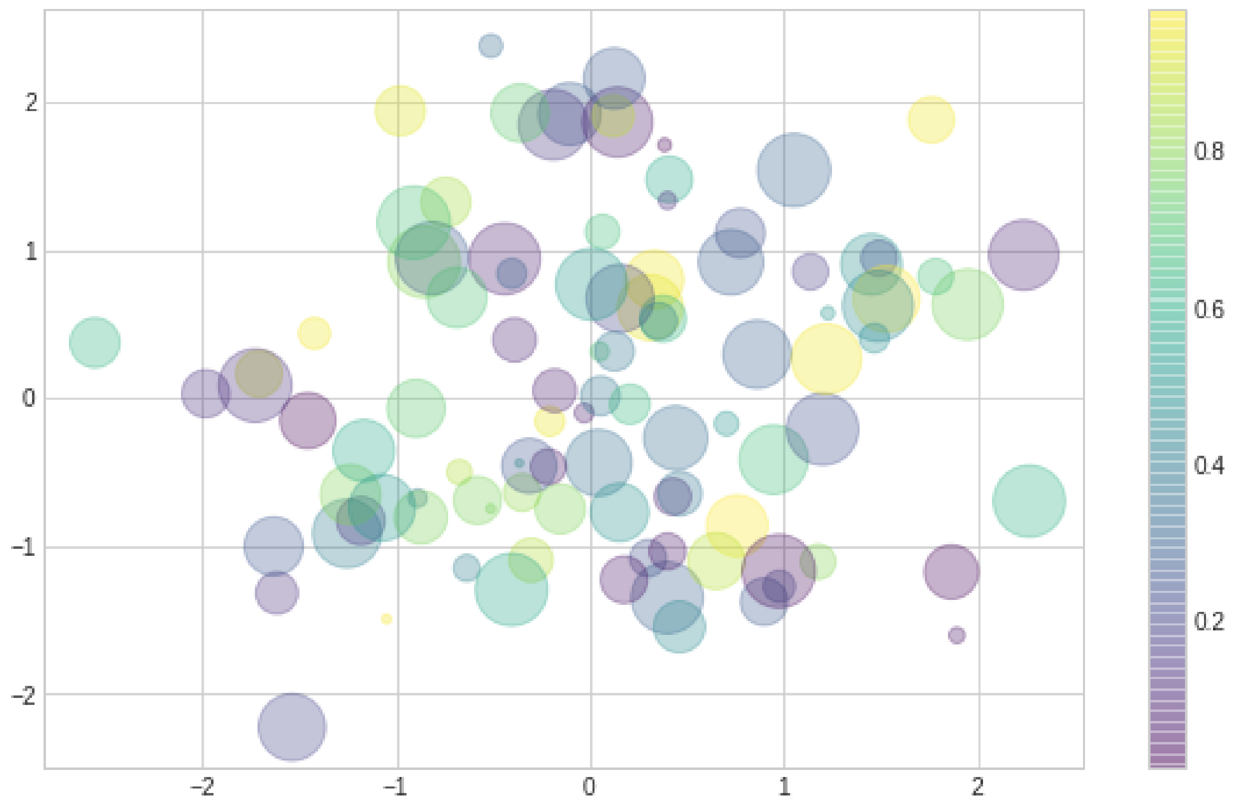


```
plt.plot(x,np.sin(x), 'o', color='black')
```

```
[<matplotlib.lines.Line2D at 0x7f45abfb5320>]
```



```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

# MATLAB-style
plt.figure (figsize=(10,6))
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3, cmap='viridis')
plt.colorbar(); # show color scale
```



```
from sklearn.datasets import load_iris
```

```
iris = load_iris()
features = iris.data
features.shape
```

```
(150, 4)
```

```
iris.target
```

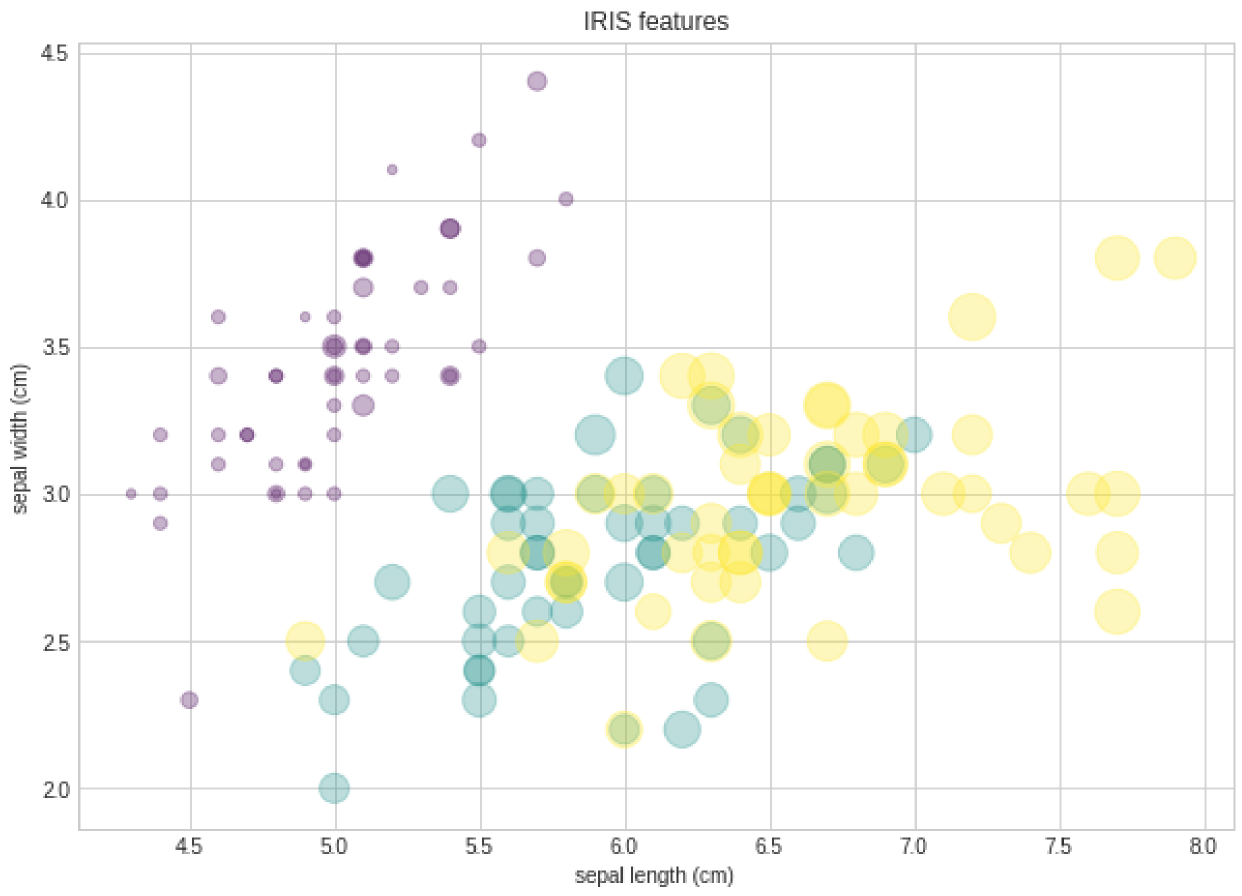
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

```
iris.feature_names
```

```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

```
# Object-oriented Style
fig, ax = plt.subplots(1, figsize=(10,7))
ax.scatter(features[:,0], features[:,1], alpha=0.3, s=200*features[:,3], c=iris.
↪target, cmap='viridis')
ax.set_xlabel (iris.feature_names[0])
ax.set_ylabel (iris.feature_names[1])
ax.set_title ('IRIS features')
```

```
Text(0.5, 1.0, 'IRIS features')
```



3.4 4. Density and Countour Plots

```
def func(x,y):
    return np.sin(x) ** 10. + np.cos(10 + x*y) * np.cos(x)
```

```
x = np.linspace(0,5, 5)
y = np.linspace(0,5, 4)
```

```
print ('x:\n', x, '\n', 'y:\n', y)
```

```
x:
 [0.   1.25 2.5  3.75 5.  ]
 y:
 [0.         1.66666667 3.33333333 5.        ]
```

```
X, Y = np.meshgrid(x, y)
X.shape, Y.shape
```

```
((4, 5), (4, 5))
```

```
print ('X:\n', X, '\nY: \n', Y)
```

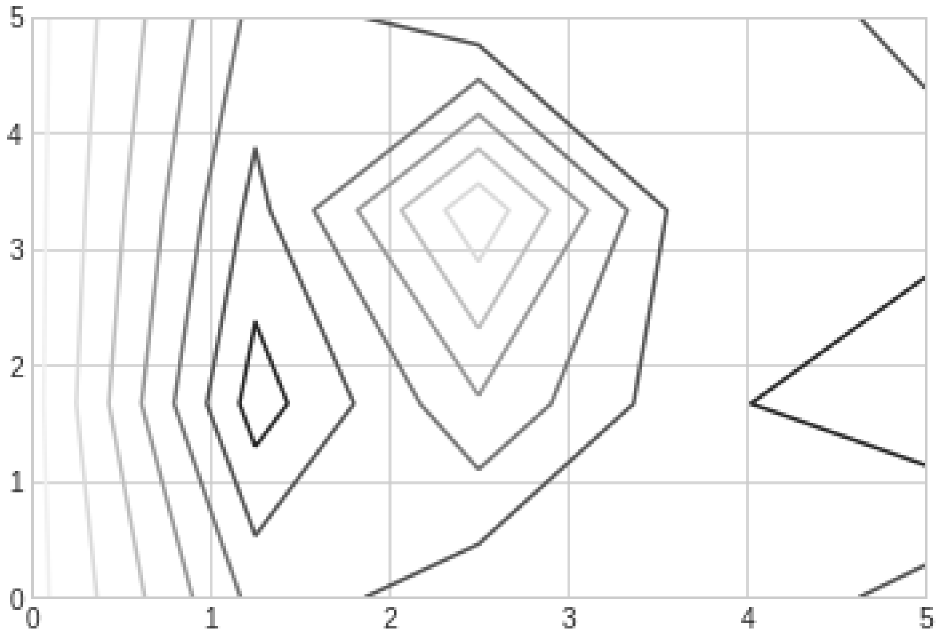
```
X:
 [[0.   1.25 2.5  3.75 5.  ]
 [0.   1.25 2.5  3.75 5.  ]
 [0.   1.25 2.5  3.75 5.  ]
 [0.   1.25 2.5  3.75 5.  ]]
Y:
 [[0.         0.         0.         0.         0.        ]
 [1.66666667 1.66666667 1.66666667 1.66666667 1.66666667]
 [3.33333333 3.33333333 3.33333333 3.33333333 3.33333333]
 [5.         5.         5.         5.         5.        ]]
```

```
Z = func(X, Y)
Z, Z.shape
```

```
(array([[-0.83907153,  0.32779018,  0.6781112 ,  0.69222873,  0.41940746],
        [-0.83907153,  0.87161399,  0.02952449,  0.7066609 ,  0.90411846],
        [-0.83907153,  0.58306763, -0.69085212,  0.72031903,  0.66787683],
        [-0.83907153,  0.32224423,  0.70553683,  0.73318806,  0.40107702]]),
 (4, 5))
```

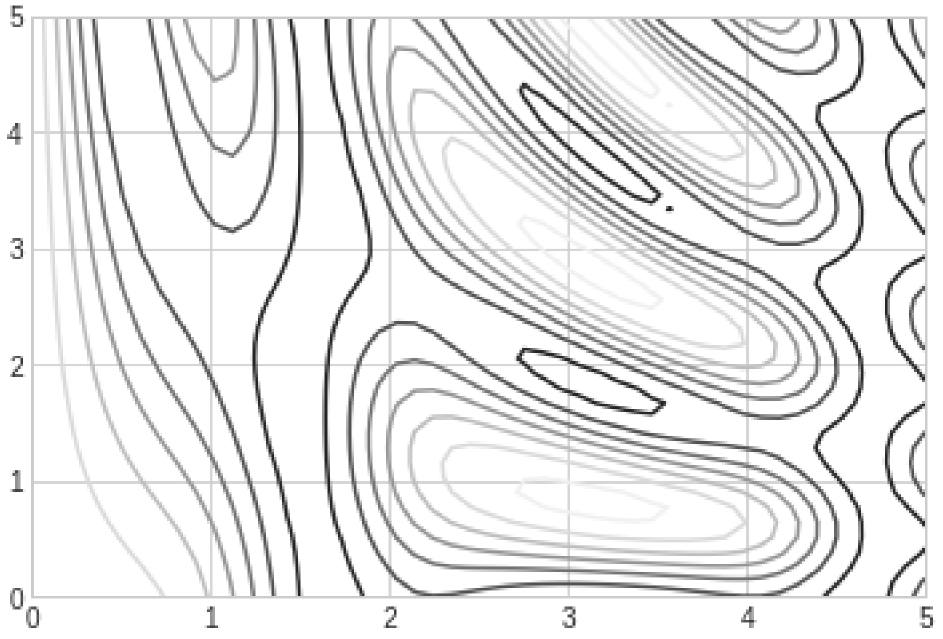
```
plt.contour (X, Y, Z)
```

```
<matplotlib.contour.QuadContourSet at 0x7f459bc12a90>
```



```
# Increase the resolution
x = np.linspace(0,5, 50)
y = np.linspace(0,5, 40)
X, Y = np.meshgrid(x,y)
Z = func(X,Y)
plt.contour (X,Y,Z)
```

```
<matplotlib.contour.QuadContourSet at 0x7f459bd28940>
```



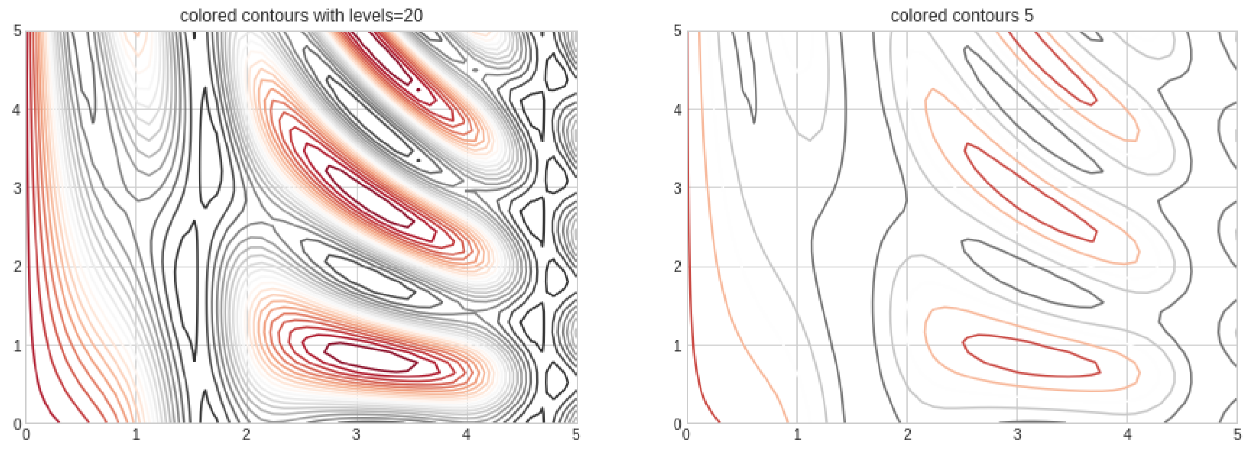
```
fig, axes = plt.subplots(1, 2, figsize=(15,5))
```

(continues on next page)

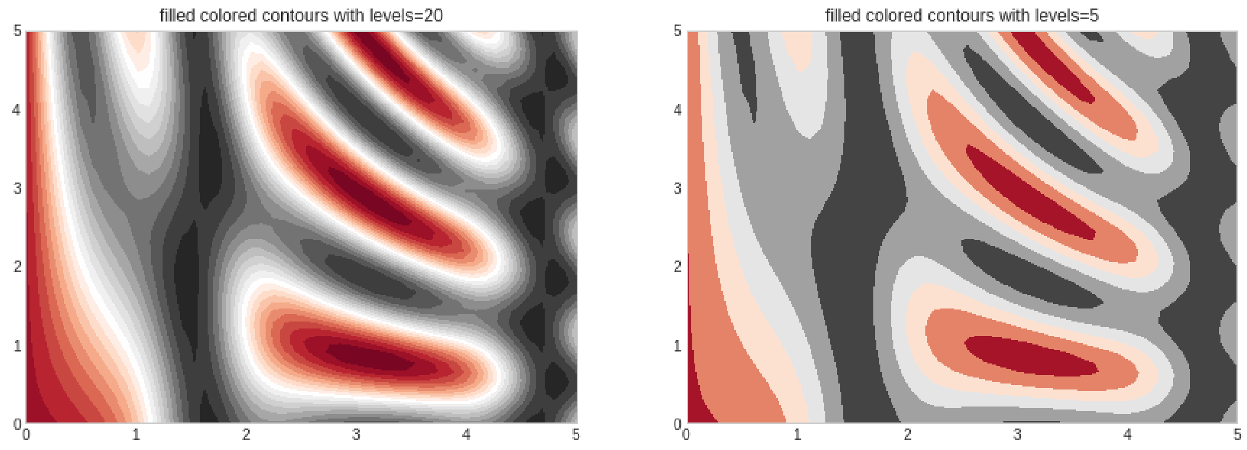
(continued from previous page)

```
axes[0].contour(X,Y,Z, levels=20, cmap='RdGy')
axes[0].set_title ('colored contours with levels=20')

axes[1].contour(X,Y,Z, levels=5, cmap='RdGy')
axes[1].set_title ('colored contours 5')
```

```
Text(0.5, 1.0, 'colored contours 5')
```

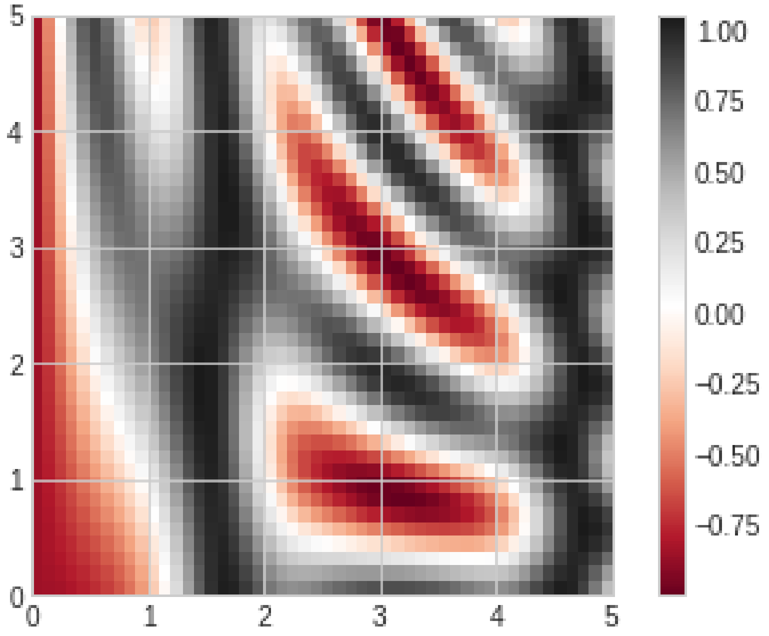


```
fig, axes = plt.subplots(1, 2, figsize=(15,5))
for ax, levels in zip(axes, [20, 5]):
    ax.contourf(X,Y,Z, levels=levels, cmap='RdGy')
    ax.set_title ('filled colored contours with levels={}'.format(levels))
```

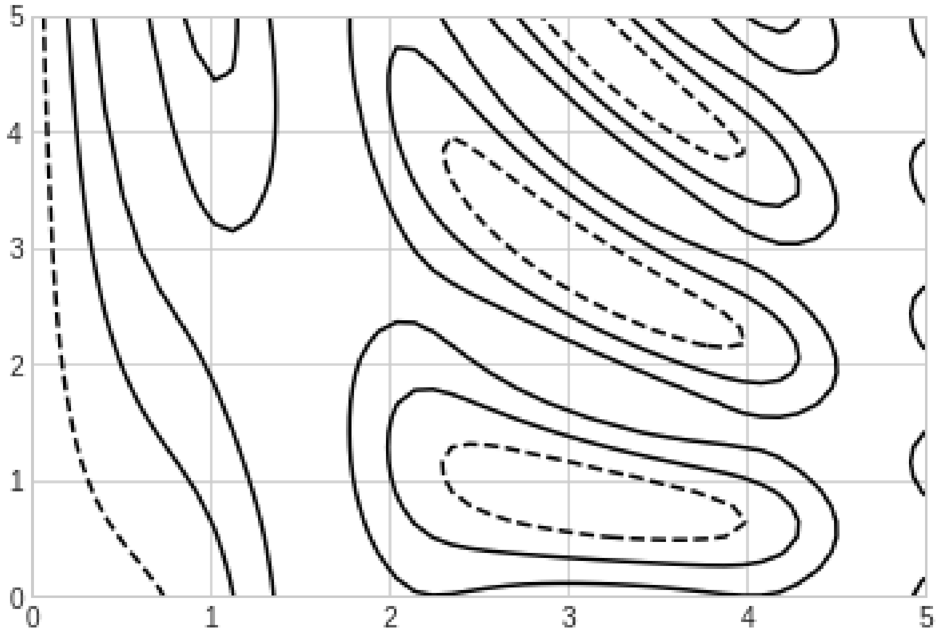


```
plt.imshow(Z, extent=[0,5,0,5], origin='lower', cmap='RdGy')
plt.colorbar()
plt.axis(aspect='image') # aspect ratio setting to make $x$ and $y$ units match
```

```
(0.0, 5.0, 0.0, 5.0)
```

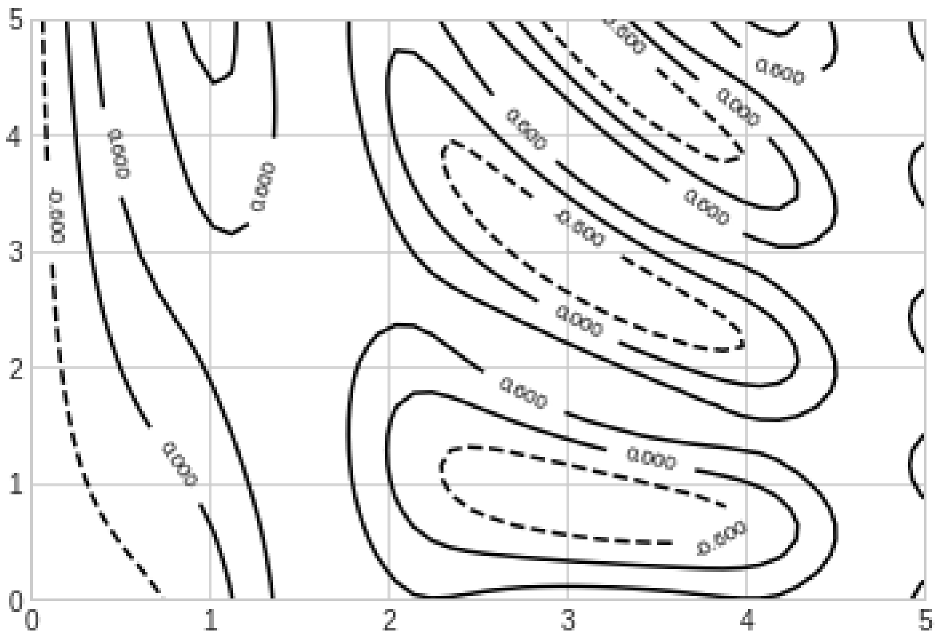


```
contours = plt.contour(X,Y,Z, 3, colors='k')
```



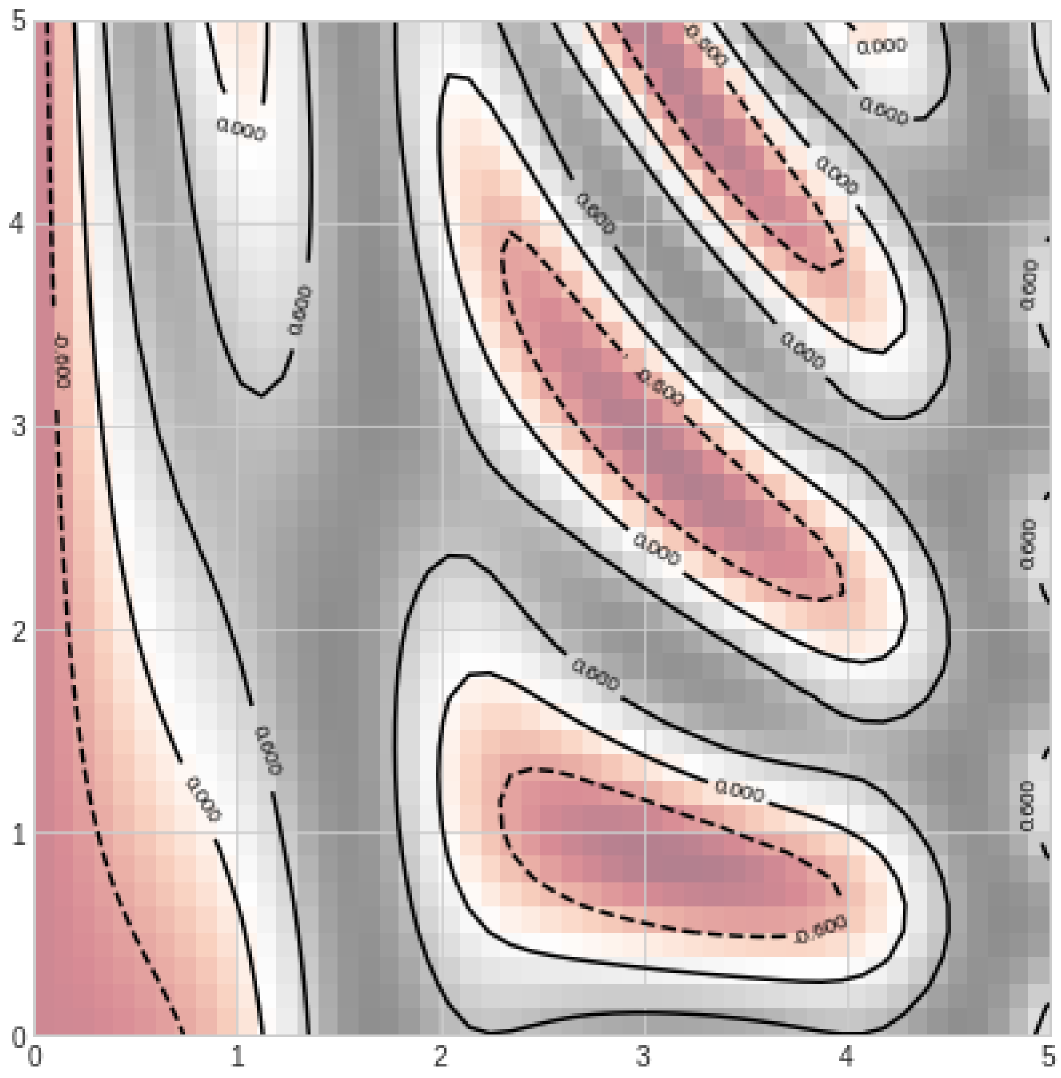
```
contours = plt.contour(X,Y,Z, 3, colors='k')
plt.clabel(contours, inline=True, fontsize=8)
```

```
<a list of 14 text.Text objects>
```



```
plt.figure(figsize=(10, 7))

contours = plt.contour(X,Y,Z, 3, colors='k')
plt.clabel(contours, inline=True, fontsize=8)
plt.imshow(Z, origin='lower', extent=[0,5,0,5], cmap='RdGy', alpha=0.5)
```

```
<matplotlib.image.AxesImage at 0x7f459b66db70>
```



3.5 5. Histograms

```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)
```

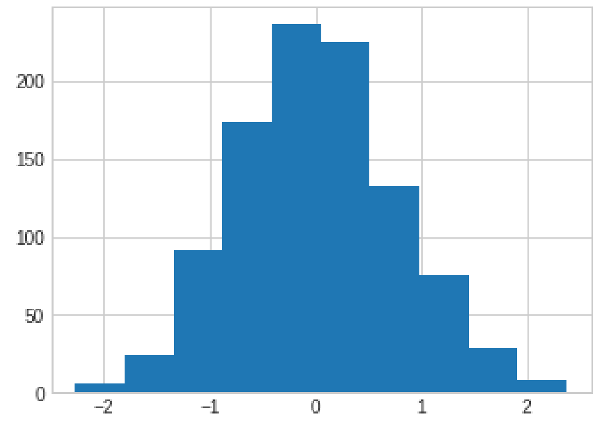
```
plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
plt.hist(x1)
plt.subplot(1,2,2)
plt.hist(x1, density=True)
```

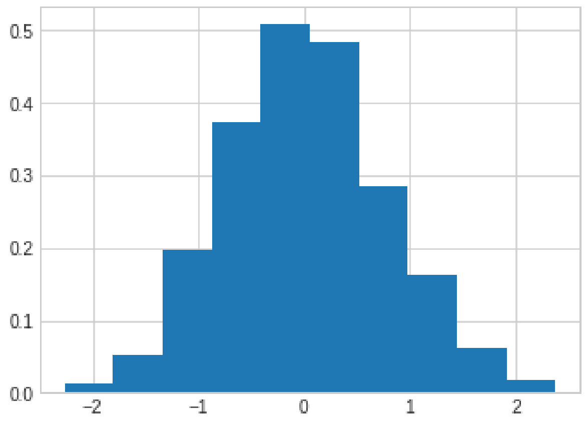
```
(array([0.00359923, 0.01979575, 0.10797684, 0.28973785, 0.49489384,
        0.41751044, 0.30233514, 0.13677066, 0.02159537, 0.00539884]),
 array([-2.81354059, -2.25786585, -1.70219112, -1.14651639, -0.59084166,
        -0.03516693,  0.5205078 ,  1.07618253,  1.63185726,  2.187532  ,
```

(continues on next page)

(continued from previous page)

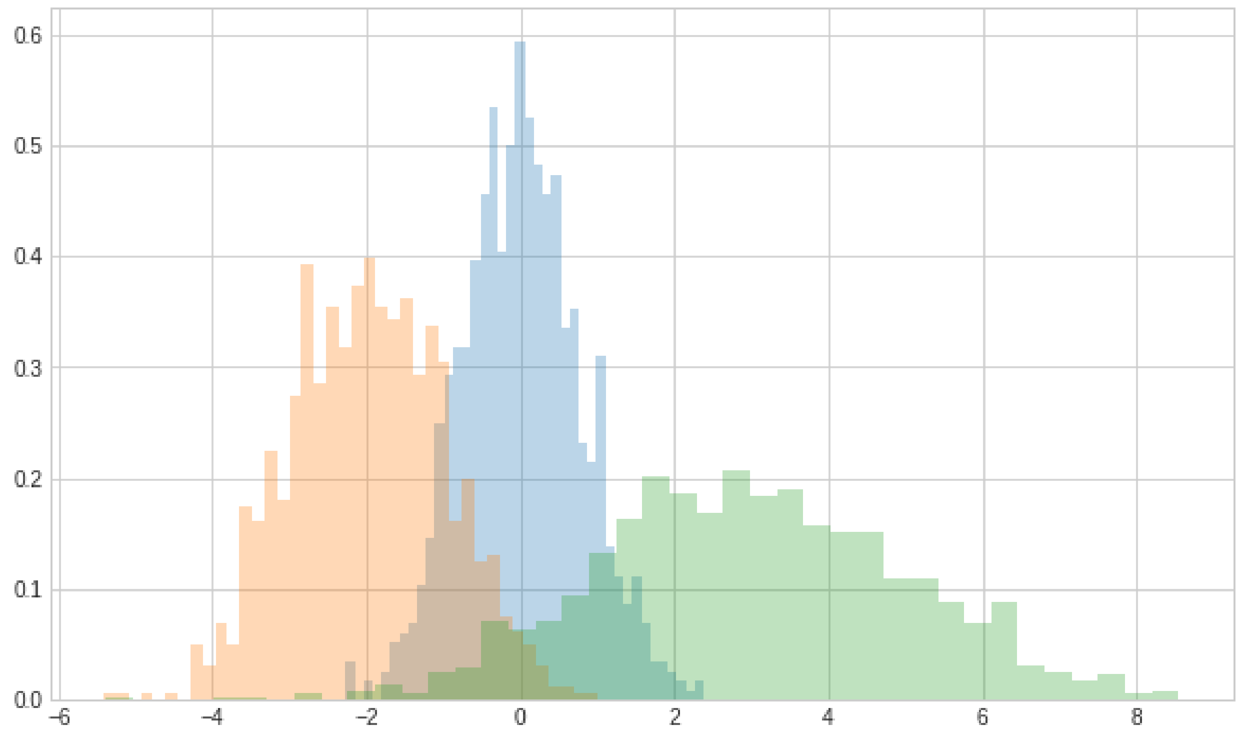
```
        2.74320673]),
 <a list of 10 Patch objects>)
```





```
plt.figure(figsize=(10,6))
kwargs = dict(histtype='stepfilled', alpha=0.3, density=True, bins=40)
plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs)
```

```
(array([0.00291683, 0.        , 0.        , 0.00291683, 0.00291683,
        0.00291683, 0.00291683, 0.01750101, 0.02041784, 0.0087505 ,
        0.04375252, 0.04666935, 0.06417036, 0.06125353, 0.11667339,
        0.10500605, 0.11375655, 0.16917641, 0.18959425, 0.16042591,
        0.20417843, 0.2187626 , 0.15750907, 0.23042994, 0.14584173,
        0.14875857, 0.16042591, 0.11959022, 0.10208921, 0.09333871,
        0.06125353, 0.04958619, 0.02625151, 0.02333468, 0.02041784,
        0.00583367, 0.00583367, 0.00291683, 0.00291683, 0.00583367]),
 array([-4.49099523, -4.14815784, -3.80532044, -3.46248304, -3.11964564,
        -2.77680824, -2.43397085, -2.09113345, -1.74829605, -1.40545865,
        -1.06262125, -0.71978386, -0.37694646, -0.03410906,  0.30872834,
         0.65156573,  0.99440313,  1.33724053,  1.68007793,  2.02291533,
         2.36575272,  2.70859012,  3.05142752,  3.39426492,  3.73710232,
         4.07993971,  4.42277711,  4.76561451,  5.10845191,  5.45128931,
         5.7941267 ,  6.1369641 ,  6.4798015 ,  6.8226389 ,  7.16547629,
         7.50831369,  7.85115109,  8.19398849,  8.53682589,  8.87966328,
         9.22250068]),
 <a list of 1 Patch objects>)
```

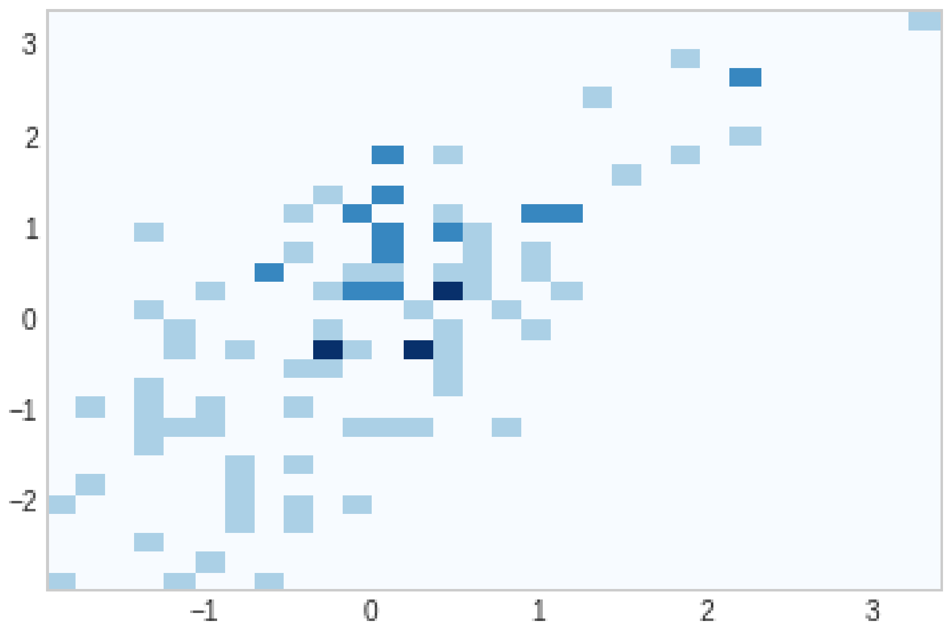


3.5.1 Two-dimensional histograms and binning

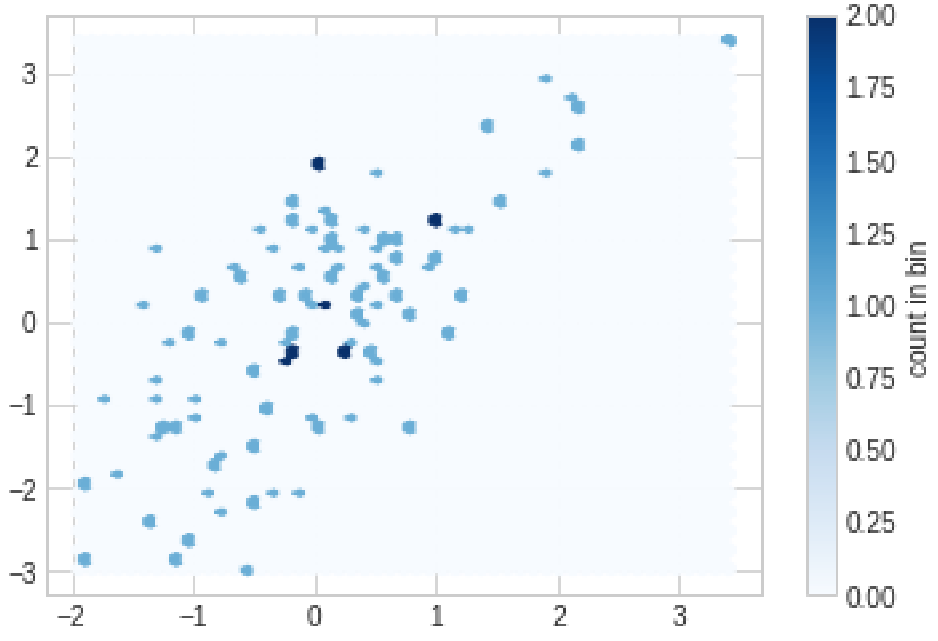
```
mean = [0, 0]
cov = [[1,1], [1,2]]
x = np.random.multivariate_normal (mean, cov, 100)
x.shape
```

```
(100, 2)
```

```
h = plt.hist2d(x[:,0], x[:,1], bins=30, cmap='Blues')
```



```
plt.hexbin(x[:,0], x[:,1], gridsize=50, cmap='Blues')
cb = plt.colorbar(label='count in bin')
```



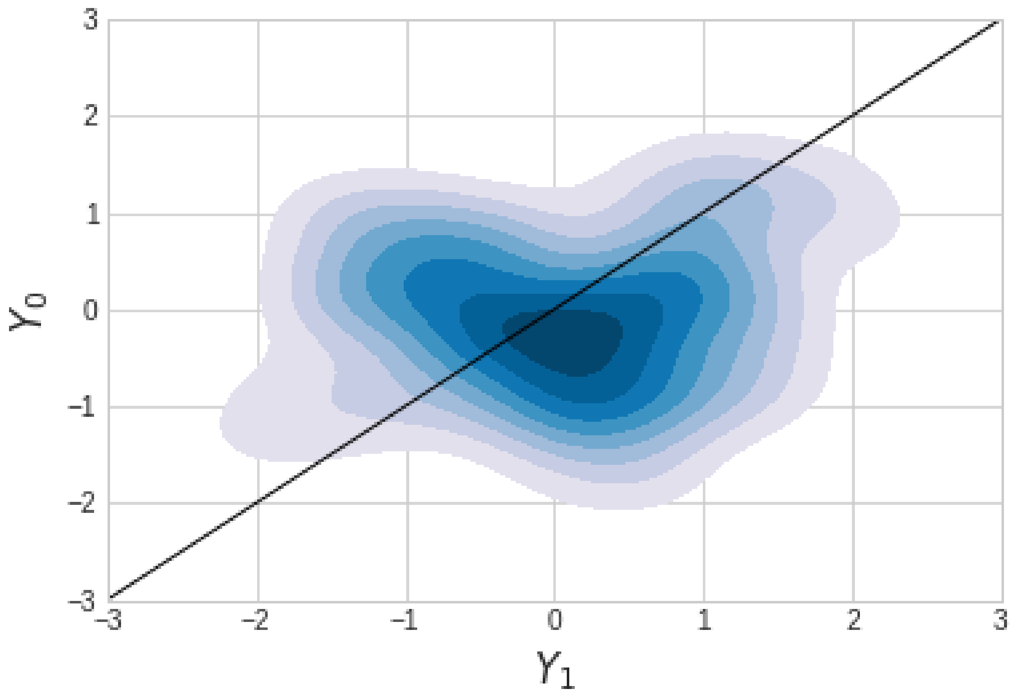
3.5.2 Using Kernel Density Estimation

```
from scipy.stats import gaussian_kde
```

```
kde_func = gaussian_kde(x)
```

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as pl

Y = np.random.multivariate_normal((0, 0), [[0.8, 0.05], [0.05, 0.7]], 100)
ax = sns.kdeplot(Y, shade = True, cmap = "PuBu")
ax.patch.set_facecolor('white')
ax.collections[0].set_alpha(0)
ax.set_xlabel('$Y_1$', fontsize = 15)
ax.set_ylabel('$Y_0$', fontsize = 15)
pl.xlim(-3, 3)
pl.ylim(-3, 3)
pl.plot([-3, 3], [-3, 3], color = "black", linewidth = 1)
pl.show()
```

```
/home/yndk/.local/lib/python3.6/site-packages/seaborn/distributions.py:679:␣
↪UserWarning: Passing a 2D dataset for a bivariate plot is deprecated in favor of␣
↪kdeplot(x, y), and it will cause an error in future versions. Please update your␣
↪code.
  warnings.warn(warn_msg, UserWarning)
```

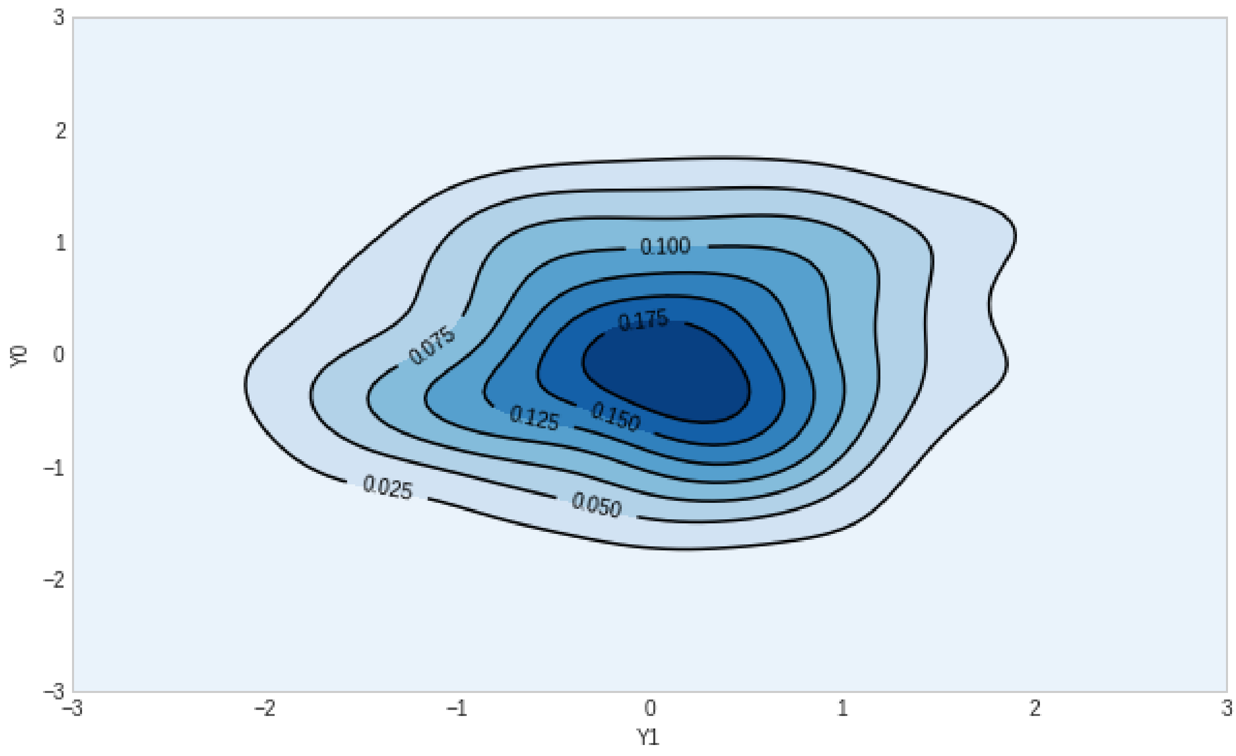


```
import numpy as np
import matplotlib.pyplot as pl
import scipy.stats as st

data = np.random.multivariate_normal((0, 0), [[0.8, 0.05], [0.05, 0.7]], 100)
x = data[:, 0]
y = data[:, 1]
xmin, xmax = -3, 3
ymin, ymax = -3, 3
```

(continues on next page)

(continued from previous page)

```
# Peform the kernel density estimate
xx, yy = np.mgrid[xmin:xmax:100j, ymin:ymax:100j]
positions = np.vstack([xx.ravel(), yy.ravel()])
values = np.vstack([x, y])
kernel = st.gaussian_kde(values)
f = np.reshape(kernel(positions).T, xx.shape)

fig = pl.figure(figsize=(10,6))
ax = fig.gca()
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)
# Contourf plot
cfset = ax.contourf(xx, yy, f, cmap='Blues')
## Or kernel density estimate plot instead of the contourf plot
#ax.imshow(np.rot90(f), cmap='Blues', extent=[xmin, xmax, ymin, ymax])
# Contour plot
cset = ax.contour(xx, yy, f, colors='k')
# Label plot
ax.clabel(cset, inline=1, fontsize=10)
ax.set_xlabel('Y1')
ax.set_ylabel('Y0')

pl.show()
```



```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

CHAPTER

FOUR

MATRIX OPERATION WITH NUMPY

4.1 1. definition

```
dim = 5
rows = 4
cols = 5
```

4.2 2. Vector

```
v1 = np.array([1,2,3,4,5.])
v2 = np.random.random(5)
print ('v1: ', v1, '\nv2: ', v2)
```

```
v1:  [1. 2. 3. 4. 5.]
v2:  [0.26240229 0.5259813  0.22681668 0.83282371 0.48881508]
```

4.3 2.1 Element-wise +, -, *, /

```
v = v1 + v2
v
```

```
array([1.26240229, 2.5259813 , 3.22681668, 4.83282371, 5.48881508])
```

```
v = v1 * v2
v
```

```
array([0.26240229, 1.0519626 , 0.68045003, 3.33129483, 2.44407541])
```

```
v = v1 / v2
v
```

```
array([ 3.8109424 ,  3.80241654, 13.22654061,  4.80293723, 10.22881696])
```

4.3.1 2.2 dot product

```
dp = np.dot(v1, v2)
dp
```

```
7.770185167242561
```

```
dot_sum = 0
for i in range (v1.shape[0]):
    dot_sum += v1[i]*v2[i]
print ('Dot product of two vectors: ', dot_sum, dp, ' and difference: ', dot_sum - dp)
```

```
Dot product of two vectors:  7.770185167242561 7.770185167242561  and difference:  0.0
```

4.4 3. Matrix

```
mat1 = np.random.random((rows,cols)) # random in [0, 1)
mat1
```

```
array([[6.76970484e-01, 8.06321304e-01, 9.62819702e-01, 7.96392874e-01,
        4.76598587e-01],
       [3.02337938e-01, 3.30836311e-01, 5.16417211e-02, 4.10603682e-01,
        4.99507594e-01],
       [4.49155020e-04, 8.98640804e-01, 3.08076609e-01, 6.12487407e-01,
        3.54063381e-01],
       [6.79148065e-01, 1.59594691e-01, 4.59115593e-02, 9.74920380e-01,
        1.63540092e-01]])
```

```
mat1.shape, mat1.dtype
```

```
((4, 5), dtype('float64'))
```

```
mat2 = np.random.random((rows, cols))
mat2
```

```
array([[0.48103878, 0.60140471, 0.11021118, 0.09678919, 0.22472494],
       [0.23439197, 0.34008368, 0.68202147, 0.66495444, 0.47735697],
       [0.92157035, 0.34285619, 0.40447625, 0.75938522, 0.19651801],
       [0.89248429, 0.42511928, 0.5780973 , 0.99868724, 0.21842825]])
```

4.4.1 3.1 Element-wise Matrix operations: +, -, *, /

```
m = mat1 + mat2
m
```

```
array([[1.15800926, 1.40772601, 1.07303088, 0.89318206, 0.70132353],
       [0.5367299 , 0.67091999, 0.73366319, 1.07555812, 0.97686456],
       [0.9220195 , 1.241497  , 0.71255286, 1.37187263, 0.55058139],
       [1.57163235, 0.58471397, 0.62400886, 1.97360762, 0.38196834]])
```

```
m = mat1 * mat2
m
```

```
array([[3.25649055e-01, 4.84925428e-01, 1.06113491e-01, 7.70822212e-02,
        1.07103588e-01],
       [7.08655839e-02, 1.12512031e-01, 3.52207626e-02, 2.73032740e-01,
        2.38443431e-01],
       [4.13927949e-04, 3.08104565e-01, 1.24609671e-01, 4.65113884e-01,
        6.95798317e-02],
       [6.06128978e-01, 6.78467797e-02, 2.65413486e-02, 9.73640544e-01,
        3.57217764e-02]])
```

```
m = mat1 / mat2
m
```

```
array([[1.40730958e+00, 1.34072995e+00, 8.73613494e+00, 8.22811798e+00,
        2.12080863e+00],
       [1.28988183e+00, 9.72808543e-01, 7.57186149e-02, 6.17491454e-01,
        1.04640264e+00],
       [4.87380070e-04, 2.62104294e+00, 7.61667986e-01, 8.06556924e-01,
        1.80168412e+00],
       [7.60963609e-01, 3.75411562e-01, 7.94183940e-02, 9.76201898e-01,
        7.48713094e-01]])
```

4.4.2 3.2 Matrix multiplication: @

- size constraint must be satisfied
- $ A(r,c) :raw-latex:‘times ‘B(c, d) = C(r,d) $

```
# matrix transpose operation
mat3 = mat2.T
print('A: ', mat1.shape, 'B: ', mat3.shape)
```

```
A:  (4, 5) B:  (5, 4)
```

```
mat3
```

```
array([[0.48103878, 0.23439197, 0.92157035, 0.89248429],
       [0.60140471, 0.34008368, 0.34285619, 0.42511928],
       [0.11021118, 0.68202147, 0.40447625, 0.5780973 ],
       [0.09678919, 0.66495444, 0.75938522, 0.99868724],
       [0.22472494, 0.47735697, 0.19651801, 0.21842825]])
```

```
mm = mat1 @ mat3
mm
```

```
array([[1.10087378, 1.8466295 , 1.98819506, 2.40302172],
       [0.50208809, 0.73007455, 0.82291141, 0.95950192],
       [0.71346539, 1.09212405, 0.96782188, 1.24954945],
       [0.55885081, 0.9711197 , 1.47164958, 1.70987943]])
```

```
mm.shape
```

```
(4, 4)
```

```
mat1[0], mat3[:,0]
```

```
(array([0.67697048, 0.8063213 , 0.9628197 , 0.79639287, 0.47659859]),
 array([0.48103878, 0.60140471, 0.11021118, 0.09678919, 0.22472494]))
```

```
mm00 = mat1[0].dot(mat3[:,0])
print ('This value mm00: {} must be the same as mm[0,0]: {}'.format(mm00, mm[0,0]))
```

```
This value mm00: 1.1008737824037702 must be the same as mm[0,0]: 1.1008737824037702
```

```
diff = mm00 - mm[0,0]
print ('this means zeros: ', diff)
```

```
this means zeros:  0.0
```

4.4.3 3.3 Matrix - Vector Multiplication

```
mat1.shape, v1.shape
```

```
((4, 5), (5,))
```

The dimension of v_1 must be 5×1 for the m-v multiplication

```
v5x1 = v1[:,np.newaxis]
v5x1
```

```
array([[1.],
       [2.],
       [3.],
       [4.],
       [5.]])
```

Now we can compute the multiplication as the definition says.

```
mv_5x1 = mat1 @ v5x1
mv_5x1
```

```
array([[10.74663663],
       [ 5.25888842],
       [ 6.94222712],
       [ 5.85345411]])
```

`Numpy` just does it without dimension extension. I don't recomment this way because it is quite confused.

```
mv1 = mat1 @ v1
mv1
```

```
array([10.74663663,  5.25888842,  6.94222712,  5.85345411])
```

Notice that the two variables `mv_5x1` and `mv1` have the same values, but different dimensions.

```
mv1.shape, mv_5x1.shape
```

```
((4,), (4, 1))
```

Warning!

```
# simple subtraction will result in something unexpected!
mv1 - mv_5x1
```

```
array([[ 0.        , -5.48774821, -3.8044095 , -4.89318252],
       [ 5.48774821,  0.        ,  1.6833387 ,  0.59456568],
       [ 3.8044095 , -1.6833387 ,  0.        , -1.08877302],
       [ 4.89318252, -0.59456568,  1.08877302,  0.        ]])
```

Warning 2

```
mv_5x1 - mv1
```

```
array([[ 0.        ,  5.48774821,  3.8044095 ,  4.89318252],
       [-5.48774821,  0.        , -1.6833387 , -0.59456568],
       [-3.8044095 ,  1.6833387 ,  0.        ,  1.08877302],
       [-4.89318252,  0.59456568, -1.08877302,  0.        ]])
```

Possibly use `squeez` or `newaxis`

```
# you can use squeeze
mv_5x1_sqz = mv_5x1.squeeze(axis=1)
mv_5x1_sqz - mv1
```

```
array([0., 0., 0., 0.])
```

4.5 3.4 Matrix Decompositions

```
mat = np.random.random((3,3))
sym = mat + mat.T # make it symmetric for better understanding
sym
```

```
array([[0.16132429, 0.05381911, 1.25636573],
       [0.05381911, 1.37489697, 1.31586585],
       [1.25636573, 1.31586585, 0.44275736]])
```

4.5.1 Eigen-decomposition

For a symmetric $n \times n$ matrix ($A^T = A$), it can be decomposed into the multiplication:

$$A = U\Lambda U^T$$
$$where : math : `U`isanorthogonalmatrix(i.e., : math : `UU^T = I`)and$$

$\Lambda = diag(\lambda_1, ..., \lambda_n)$ is a diagonal matrix.

The elements λ_i's are called the **eigen-values** of the matrix A, and the column vectors of $U = [u_1, ..., u_n]$ is called the **eigen-vectors** of A.

The pair (u_i, λ_i) satisfies:

$$Au_i = \lambda_i u_i$$

If A is positive-definite, then all the eigen-values are positive.

The following matrices are positive definite:

- the identity matix I
- diagonal matix with positive elements $diag(1, 2, 3)$
- any matrix that has positive eigen-values.
- sample covariance matrix

$$A = \sum_{i=1}^{N} (x_i - m)^T (x_i - m)$$

- a matrix of proper ellipse in $\mathbb{R}^n$

$$C = diag(1/a_1^2, 1/a_2^2, ..., 1/a_n^2), \text{ for } x^T C x = 1$$

  – e.g. an ellipse in 2D:

$$x^2/a^2 + y^2/b^2 = 1$$

Eigen decomposition with `numpy.linalg.eig()`

```
ev, U = np.linalg.eig (sym)
print (ev, '\n', U)
```

```
[-1.26949146  0.66240804  2.58606204]
 [[ 0.61271707 -0.71276347 -0.34138838]
 [ 0.34204838  0.62858669 -0.69848528]
 [-0.71244698 -0.31120251 -0.62894538]]
```

orthogonality of U

```
print (U.transpose() @ U, '\n', U @ U.T)
```

```
[[ 1.00000000e+00 -2.22044605e-16 -1.66533454e-16]
 [-2.22044605e-16  1.00000000e+00 -4.16333634e-16]
 [-1.66533454e-16 -4.16333634e-16  1.00000000e+00]]
 [[ 1.00000000e+00  2.77555756e-17 -2.77555756e-16]
 [ 2.77555756e-17  1.00000000e+00  1.66533454e-16]
 [-2.77555756e-16  1.66533454e-16  1.00000000e+00]]
```

```
recon = U @ np.diag (ev) @ U.T
error = recon - sym
print (error, '\n', np.round(error, 7))
```

```
[[-4.44089210e-16 -3.33066907e-16 -4.44089210e-16]
 [-3.33066907e-16 -4.44089210e-16 -2.22044605e-16]
 [-4.44089210e-16  0.00000000e+00 -5.55111512e-16]]
 [[-0. -0. -0.]
 [-0. -0. -0.]
 [-0.  0. -0.]]
```

4.5.2 Singular Value Decomposition (SVD)

```
u, s, vt = np.linalg.svd(mat)
```

```
s
```

```
array([1.34430748, 0.67592205, 0.28419014])
```

```
recon = u @ np.diag(s) @ vt
recon - mat
```

```
array([[ 1.38777878e-17, -7.63278329e-17, -1.11022302e-16],
       [ 3.46944695e-17,  2.22044605e-16,  1.11022302e-16],
       [ 0.00000000e+00,  1.11022302e-16, -2.22044605e-16]])
```

SVD for a $m \times n$ matrix ($m > n$)

```
mat43 = np.random.random((4,3))
mat43
```

```
array([[0.72952583, 0.16585428, 0.88240786],
       [0.31319221, 0.55959353, 0.01489713],
       [0.64731371, 0.29531888, 0.31787163],
       [0.61200016, 0.70302823, 0.7042169 ]])
```

```
u, s, vt = np.linalg.svd (mat43, full_matrices=False)
```

```
u.shape, s.shape, vt.shape
```

```
((4, 3), (3,), (3, 3))
```

```
recon = u @ np.diag(s) @ vt
recon - mat43
```

```
array([[-2.22044605e-16, -2.49800181e-16, -1.11022302e-16],
       [ 2.22044605e-16,  1.11022302e-16, -4.18068358e-16],
       [ 1.11022302e-16,  0.00000000e+00, -5.55111512e-17],
       [ 3.33066907e-16,  0.00000000e+00, -1.11022302e-16]])
```

```
u
```

```
array([[-0.60264689,  0.63273472, -0.01282194],
       [-0.2558916 , -0.72325577, -0.16503826],
       [-0.41333623, -0.06191151, -0.78768909],
       [-0.6328423 , -0.26965687,  0.59341711]])
```

```
# u is a unitary (or orthogonal) matrix: the columns are orthogonal to each other,␣
↪and mag = 1
u.T @ u
```

```
array([[ 1.00000000e+00,  1.39272183e-16, -6.92178737e-17],
       [ 1.39272183e-16,  1.00000000e+00,  1.23581554e-16],
       [-6.92178737e-17,  1.23581554e-16,  1.00000000e+00]])
```

```
# but this is not!
u @ u.T
```

```
array([[ 0.76370091, -0.30130065,  0.22002194,  0.20315042],
       [-0.30130065,  0.61581704,  0.28054596,  0.25903339],
       [ 0.22002194,  0.28054596,  0.79513398, -0.18915667],
       [ 0.20315042,  0.25903339, -0.18915667,  0.82534807]])
```

```
# vt is a unitary and square matrix
vt.T @ vt
```

```
array([[ 1.00000000e+00,  1.37273771e-16,  1.31282456e-16],
       [ 1.37273771e-16,  1.00000000e+00, -1.69170765e-16],
       [ 1.31282456e-16, -1.69170765e-16,  1.00000000e+00]])
```

```
# so this too results in id. matrix
vt @ vt.T
```

```
array([[1.00000000e+00, 9.26373212e-17, 3.72389816e-17],
       [9.26373212e-17, 1.00000000e+00, 4.87390640e-17],
       [3.72389816e-17, 4.87390640e-17, 1.00000000e+00]])
```

```
for i in range(vt.shape[0]):
    print ('mag: ', np.linalg.norm(vt[i]))
```

```
mag:  1.0
mag:  1.0
mag:  0.9999999999999999
```

```
for i in range(vt.T.shape[0]):
    print ('mag: ', np.linalg.norm(vt.T[i]))
```

```
mag:  1.0
mag:  1.0
mag:  1.0
```

4.5.3 Determinant

```
np.linalg.det(vt), np.linalg.det(vt.T)
```

```
(1.0, 1.0)
```

4.5.4 Inverse of non-singular square matrix

```
print ('Invertible matrix does have non-zero determinant:', np.linalg.det( mat ) )
```

```
Invertible matrix does have non-zero determinant: -0.25822853811120877
```

$A^{-1} \times A = A \times A^{-1} = I$

```
inv = np.linalg.inv (mat)
ii = inv @ mat
np.linalg.det(ii) # must be 1 since ii is identity matrix
```

```
1.0
```

```
ii
```

```
array([[ 1.00000000e+00, -1.15191601e-16, -5.92759050e-17],
       [ 1.50077222e-18,  1.00000000e+00, -1.98459262e-18],
       [-4.96225481e-18, -2.00389893e-17,  1.00000000e+00]])
```

4.6 How to draw an ellipse & Level contours of a 2D Gaussian

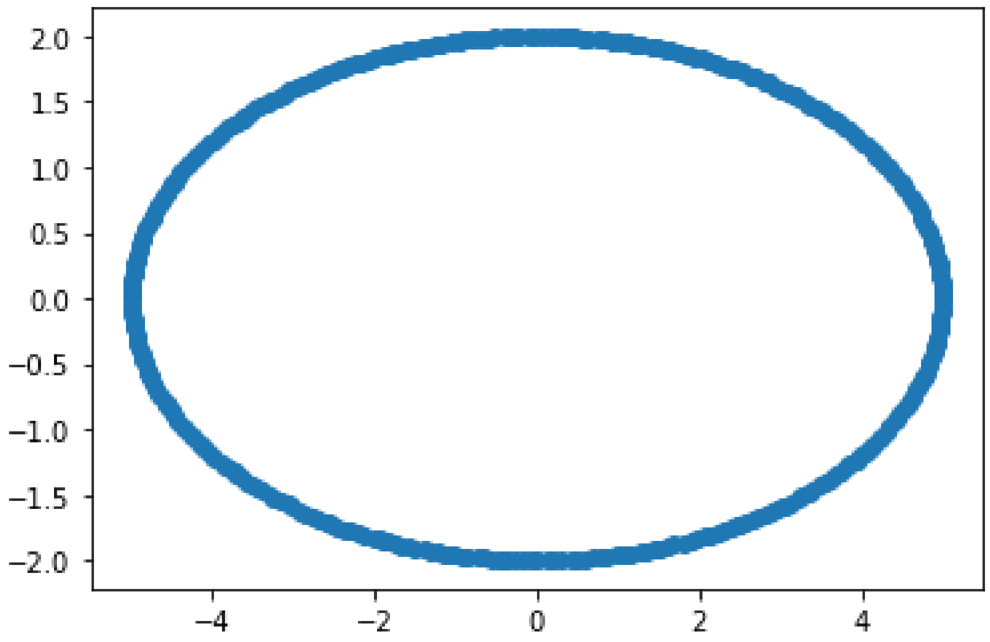
The equation of an ellipse with $a^2 = 5^2, b^2 = 2^2$:

$$x^T diag(1/25, 1/4)x = 1$$

```
a, b = 5, 2
C = np.diag([1/a**2, 1/b**2])
print ('C =\n', C)
xcoord = np.array([ a*np.cos(np.rad2deg(r)) for r in range(360)])
ycoord = np.array([ b*np.sin(np.rad2deg(r)) for r in range(360)])
X = np.vstack((xcoord, ycoord))
plt.scatter(X[0], X[1])
```

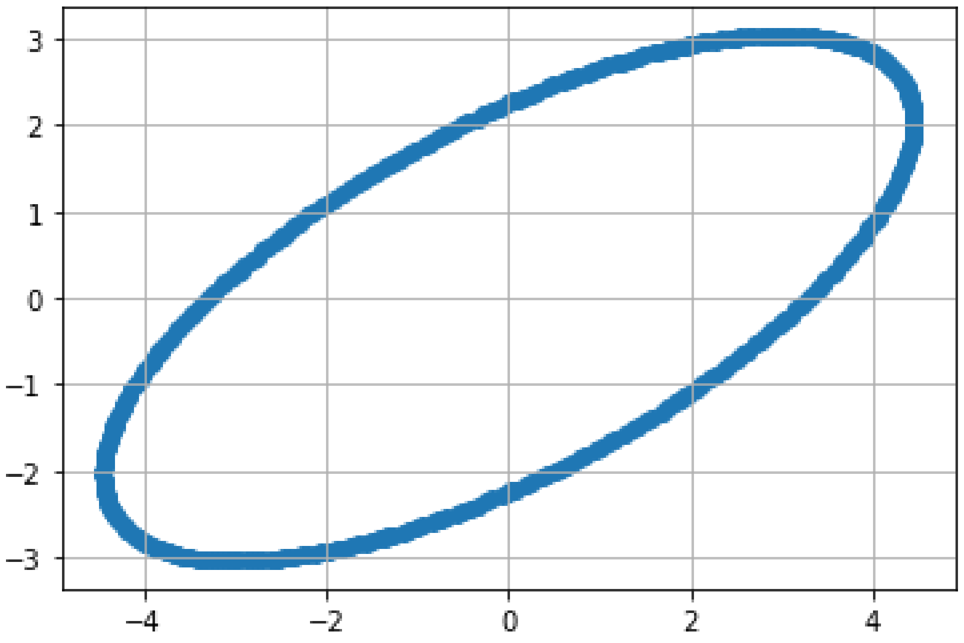
```
C =
 [[0.04 0.  ]
 [0.   0.25]]
```

```
<matplotlib.collections.PathCollection at 0x7f23cda2a4e0>
```



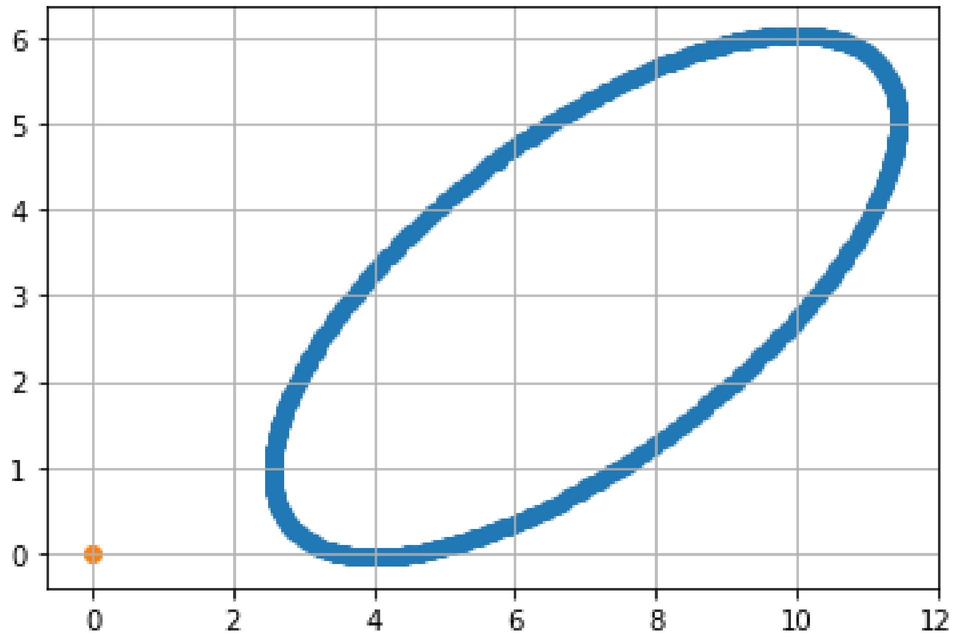
rotate it 30 degrees

```
r = np.deg2rad(30)
R = np.array([ [np.cos(r), -np.sin(r)], [np.sin(r), np.cos(r)] ])
Y = R @ X
plt.scatter(Y[0], Y[1])
plt.grid(True)
```



now translate it

```
translation = [7, 3] # tx=7, ty=3
Z = Y.copy()
for i in range(Z.shape[1]):
    Z[:,i] += translation
plt.scatter(Z[0], Z[1])
plt.scatter(0,0) # origin point
plt.grid(True)
```



1. orignally, $x^T C x = 1$, where $C = diag(1/25, 1/4)$

2. rotate it, $y = Rx$. So, $x = R^T y$
   - original equation changes to: $x^T RR^T CRR^T x = 1$
   - we can write $y^T Hy = 1$ where $H = R^T CR$
3. translate it.

$$z = y + m = Rx + m$$

   - so $x = R^T(z - m)$
   - $x^T Cx = (z - m)^T RCR^T(z - m) = 1$
   - using z variable $(z - m)^T H(z - m) = 1$

```
H = R.T @ C @ R
m = np.array(translation)
print ('R:', R)
print ('C: ',C)
print ('H: ', H)
print ('m: ', m)
```

```
R: [[ 0.8660254 -0.5      ]
 [ 0.5        0.8660254]]
C:  [[0.04 0.  ]
 [0.   0.25]]
H:  [[0.0925     0.09093267]
 [0.09093267 0.1975    ]]
m:  [7 3]
```

4.7 Now, how to draw the ellipse given H and m?

1. perform eigen decomposition to H

```
ev, U = np.linalg.eig(H)
```

```
print (U)
```

```
[[-0.8660254 -0.5      ]
 [ 0.5       -0.8660254]]
```

```
# really correct?
U @ np.diag(ev) @ U.T - H
```

```
array([[ 0.00000000e+00, -1.38777878e-17],
       [-1.38777878e-17,  0.00000000e+00]])
```

```
# U the same as R?
print (U - R)
```

```
[[-1.73205081e+00  5.55111512e-17]
 [-5.55111512e-17 -1.73205081e+00]]
```

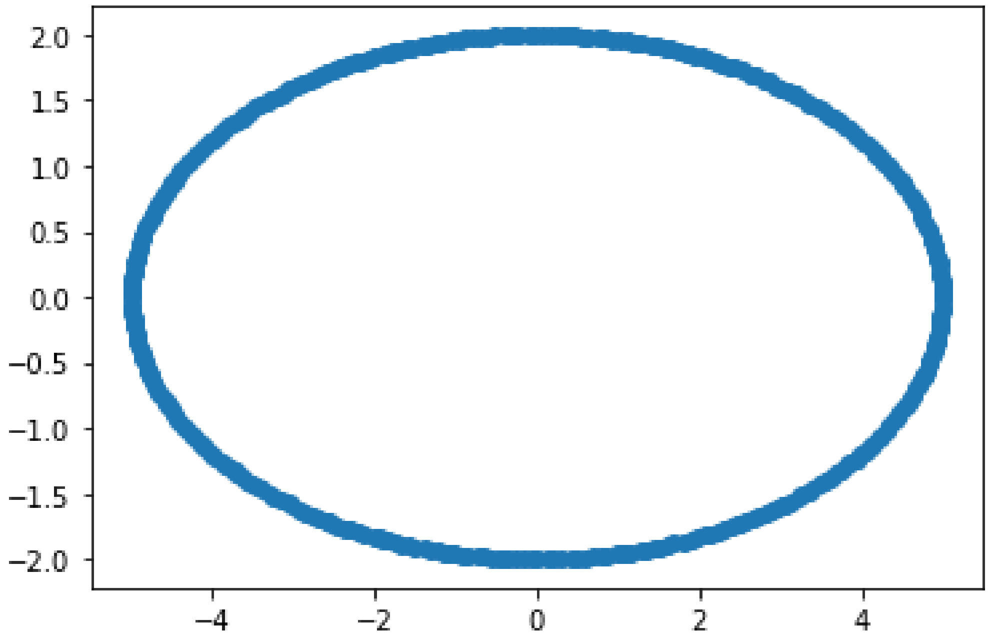
Q. Thr orthogoanl matrix U may not be the same as R. Why?

```
aa2 = ev[0]
bb2 = ev[1]
print(ev)
```

```
[0.04 0.25]
```

```
aa = 1/np.sqrt(aa2)
bb = 1/np.sqrt(bb2)
xxcoord = np.array([ aa*np.cos(np.rad2deg(r)) for r in range(360)])
yycoord = np.array([ bb*np.sin(np.rad2deg(r)) for r in range(360)])
XX = np.vstack((xxcoord, yycoord))
plt.scatter(XX[0], XX[1])
```

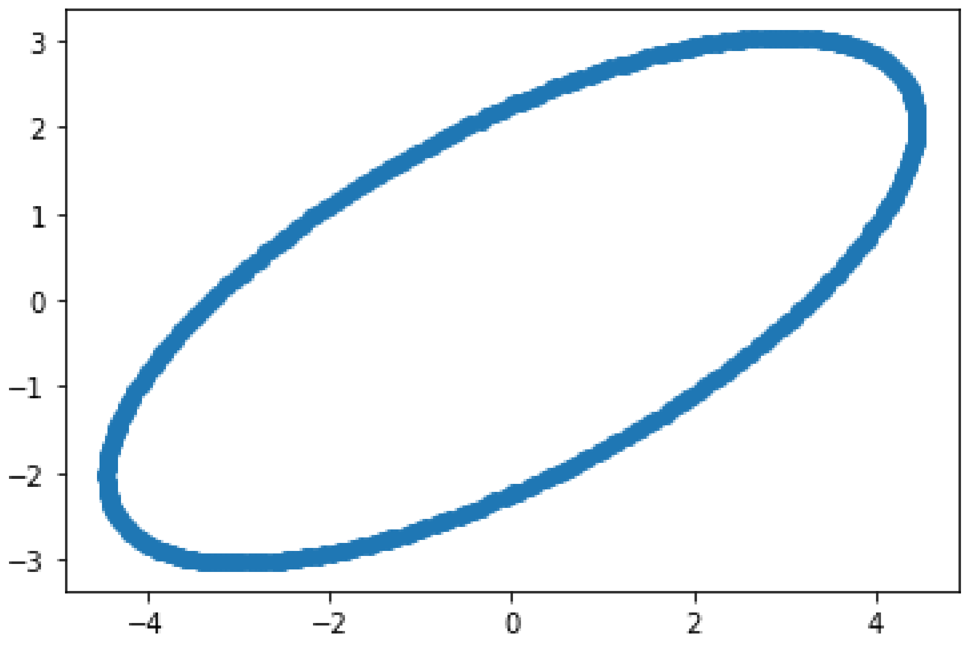
```
<matplotlib.collections.PathCollection at 0x7f23cd4565c0>
```



Rotate it by U

```
YY = R @ XX
plt.scatter(YY[0], YY[1])
```

```
<matplotlib.collections.PathCollection at 0x7f23cd3bc390>
```

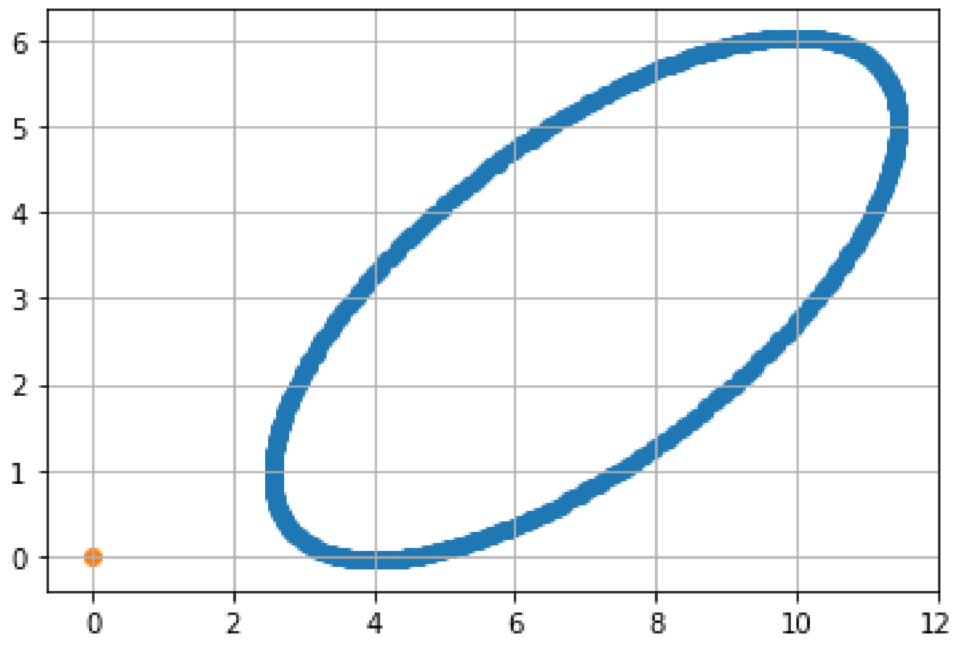


Translate it by m

```
m.shape
```

```
(2,)
```

```
#ZZ = YY + np.vstack( (m for i in range(360)) ).T
# use the trick of numpy array broadcasting for short code length.
# Not recommended if you are suspicious of the broadcasting mechanism.
# Using for-loop is safe every time.
ZZ = YY + m.reshape(2,1)
#
plt.scatter (ZZ[0], ZZ[1])
plt.scatter (0,0)
plt.grid(True)
```



4.8 2D Gaussian with $(x-m)^T\Sigma^{-1}(x-m)$ in the exponent

- The equi-level contour of 1-sigma is given by $(x-m)^T\Sigma^{-1}(x-m) = 1^2$
- The equi-level contour of 2-sigam is given by $(x-m)^T\Sigma^{-1}(x-m) = 2^2$
- Here $H = \Sigma^{-1}$
- 1D case:
  - 1-sigma point means when $x = 1\sigma$, in which case $\frac{x^2}{\sigma^2} = \frac{\sigma^2}{\sigma^2} = 1$
  - 2-sigma point means when $x = 2\sigma$

4.9 Q. Plot three contour lines (1,2,3 sigma)

1. for the first two features (sepal length and width) of setosa in the iris dataset.
   - First, you need to compute the sample mean m and sample covariace Σ of the data points.
   - Then, draw three ellipses using the abovementioned method.
2. Do the same for the two features of the other species.

```
i = 0
while i < 5:
    print (i); i+=1
```

```
0
1
2
3
4
```

CHAPTER

FIVE

END

CHAPTER

SIX

RANDOM SAMPLE GENERATION AND PROBABILITY DISTRIBUTIONS

```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
import numpy as np
```

6.1 Notations/Formulas from Probability & Statistics

Mean and Variance of a discrete random varaible X whose probability is given by $p_i = P(X = x_i)$

$$\mathbb{E}[X] = \sum_i p(x_i)x_i$$

$$\mathbb{E}[(X-\mu)^2] = \sum_i p(x_i)(x_i-\mu)^2$$

6.2 Gaussian Distribution

- PDF (Probability Density Function) is given by

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{ -\frac{1}{2}\frac{(x-m)^2)}{\sigma^2} \right\}$$

- CDF (Cumulative distribution function) is given by

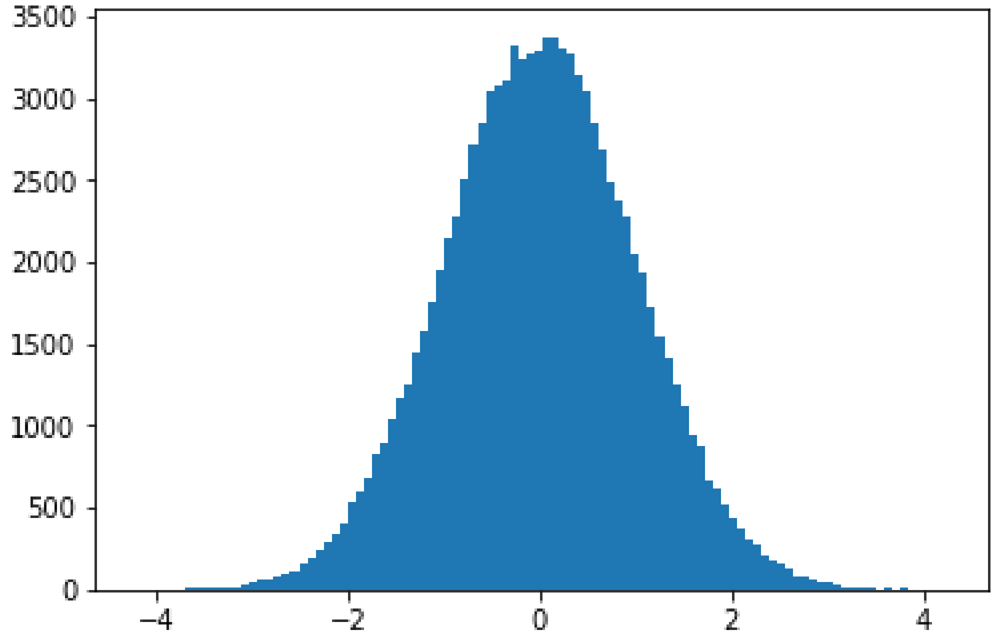
$$Prob[X < x_0] = \int_{-\infty}^{x_0} p(x)dx$$

```
mean = 0
std = 1
N = 100000
```

```
x = np.random.normal(mean, std, N)
x.shape, x[:10]
```

```
((100000,),
 array([-1.07067485, -1.40613318, -1.35403904,  0.88580919,  0.76958017,
         0.892695  , -0.7011873 ,  1.30072567,  0.64376671, -1.94480456]))
```

```
h = plt.hist (x, bins=100)
```



```
len(h), h[0].shape, h[1].shape
```

```
(3, (100,), (101,))
```

```
h[0][:10], h[1][:10]
```

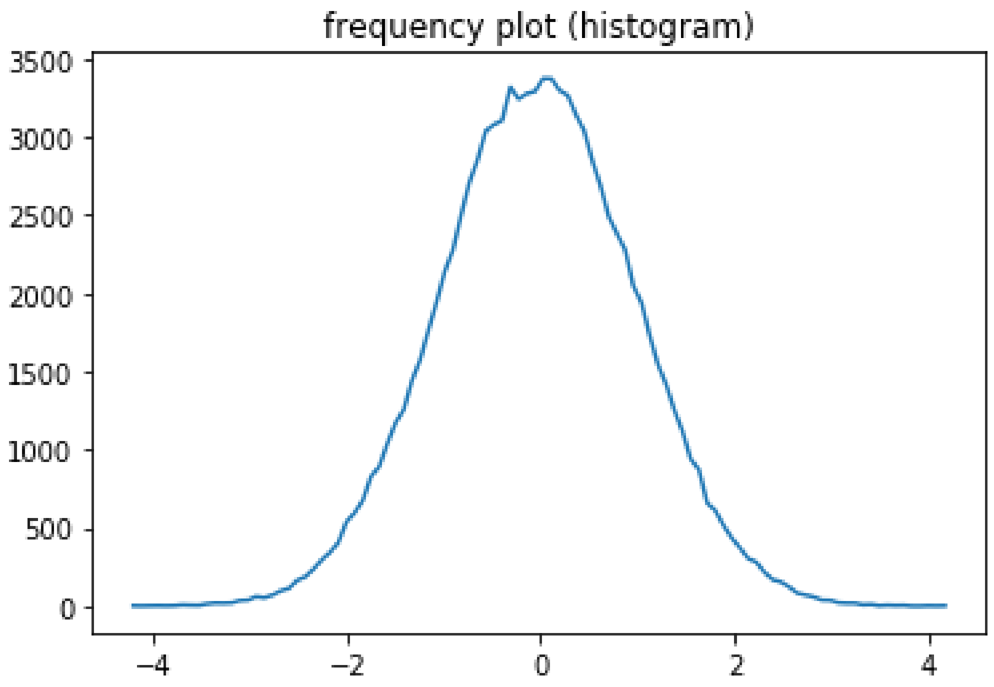
```
(array([ 2.,  1.,  2.,  3.,  2.,  3.,  8.,  6.,  5., 15.]),
 array([-4.20944681, -4.12476529, -4.04008377, -3.95540224, -3.87072072,
        -3.7860392 , -3.70135767, -3.61667615, -3.53199463, -3.4473131 ]))
```

6.2.1 The frequency for the bins is recorded in h[0]

```
freq = h[0]
domain = h[1]
```

```
plt.plot (domain[:-1], freq)
plt.title ('frequency plot (histogram)')
```

```
Text(0.5, 1.0, 'frequency plot (histogram)')
```



Notice that the x-axis shows a difference range. It is the number of items in the arrqy `freq`

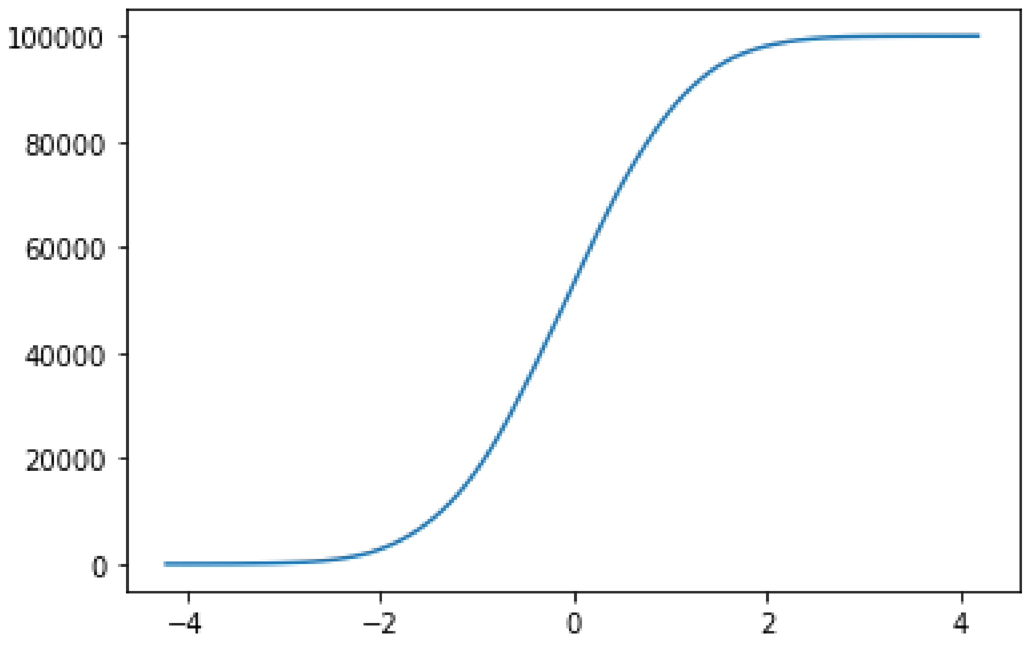
Cumulative Frequency Computation

```
cumfreq = np.zeros_like (freq)
cumfreq[0] = freq[0]
for i in range(1, cumfreq.size):
    cumfreq[i] = cumfreq[i-1] + freq[i]
print ('The last term of CF must be equal to N :', cumfreq[-1] == N)
```

```
The last term of CF must be equal to N : True
```

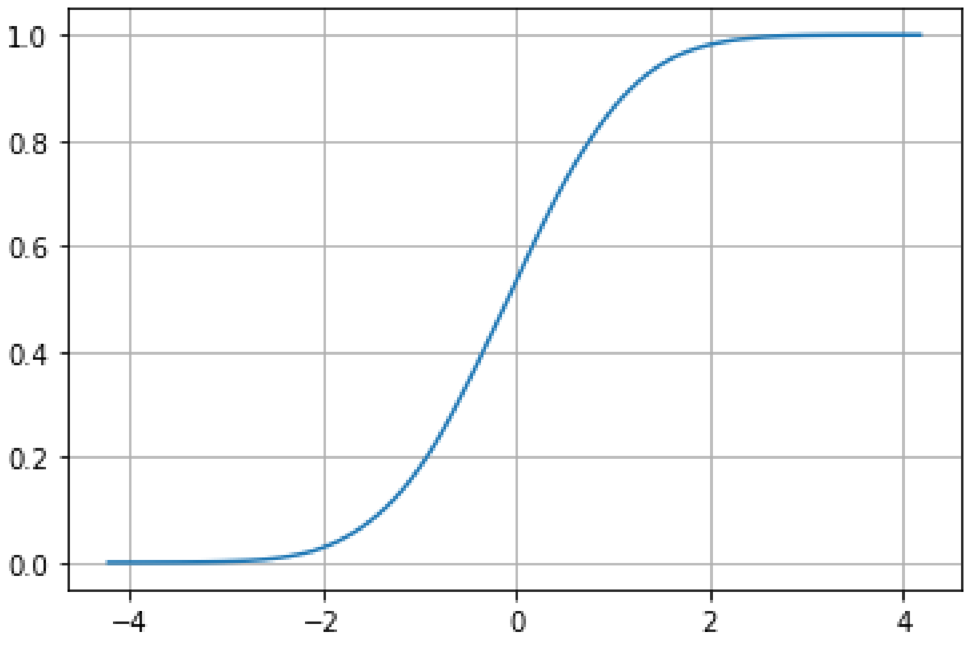
```
plt.plot (domain[:-1], cumfreq)
```

```
[<matplotlib.lines.Line2D at 0x7f50fc36d898>]
```



```
normalized_cumfreq = cumfreq / N
```

```
plt.plot (domain[:-1], normalized_cumfreq)
plt.grid(True)
```

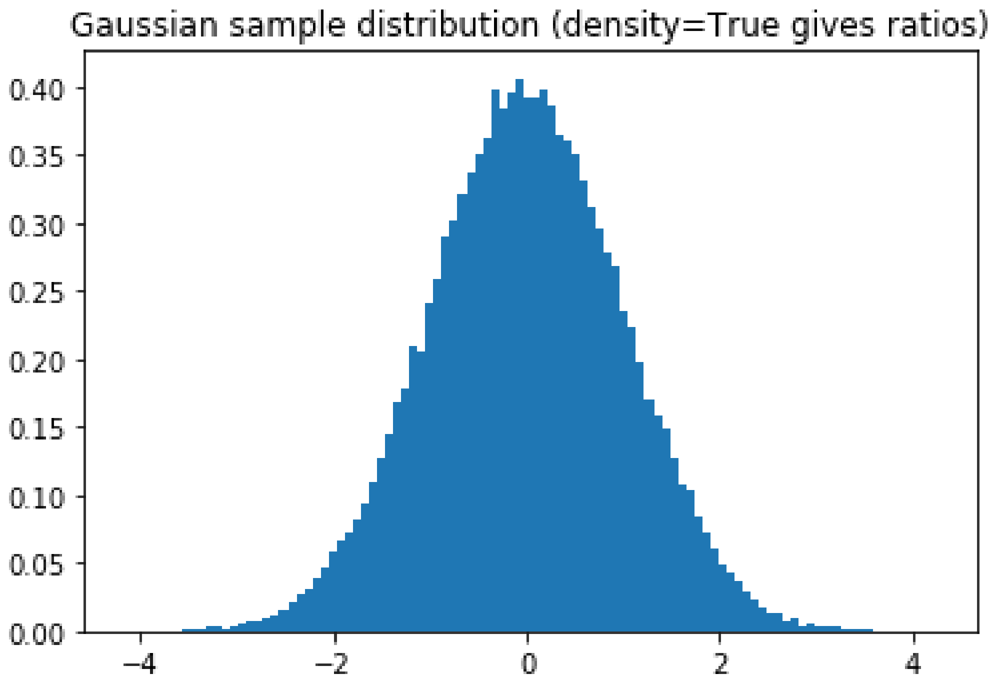


Samples from the standard normal distribution can also be obtained by `np.random.randn(size)`

```
y = np.random.randn(N)
y.shape, y[:10]
```

```
((100000,),
 array([-1.1958721 , -0.71605046,  1.75105571,  1.20057299,  0.69904783,
        -2.18675664, -0.79896062, -0.28236177, -0.36000248,  0.02559079]))
```

```
h = plt.hist (y, bins=100, density=True)
_ = plt.title ('Gaussian sample distribution (density=True gives ratios)')
```



6.2.2 Cumulative Frequence

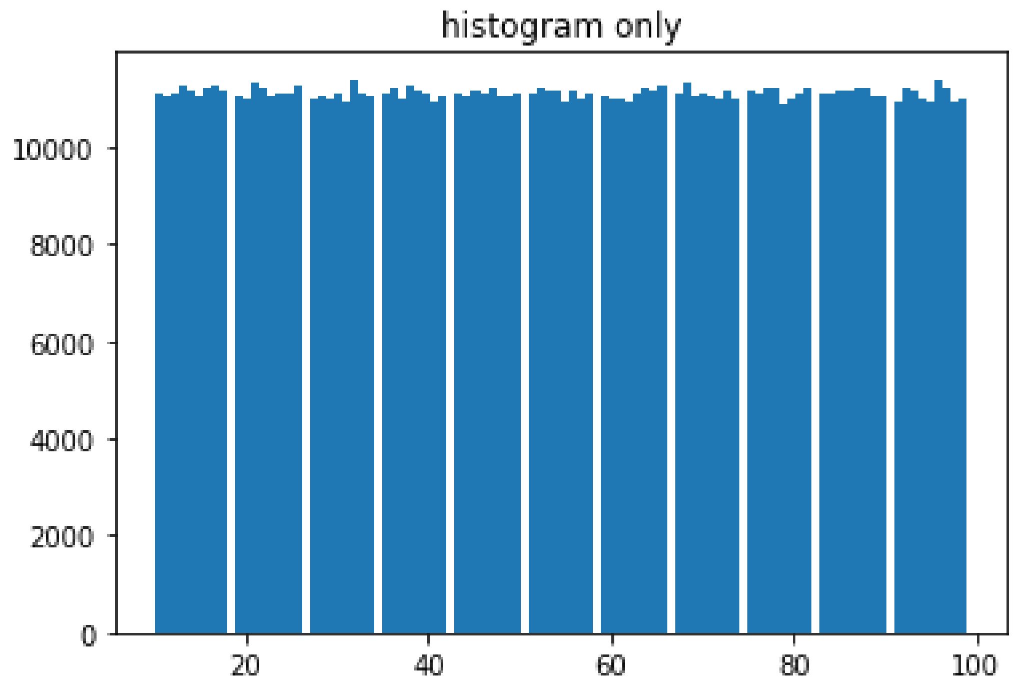
```
def cumfreq(f, density=False):
    cdf = np.zeros_like (f)
    cdf[0] = f[0]
    for i in range(1, cdf.size):
        cdf[i] = cdf[i-1] + f[i]
    #
    if density:
        cdf /= cdf[-1]
    #
    return cdf
```

6.3 Random Integer Generation

```
xint = np.random.randint(10, high=100, size=1000000)
```

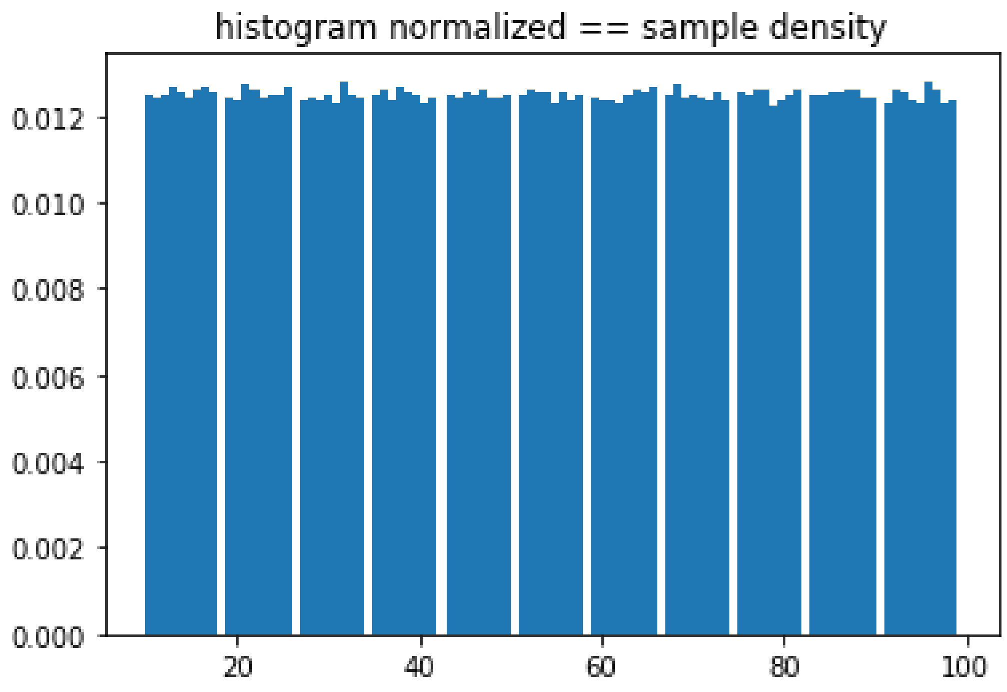
```
h = plt.hist(xint, bins=100, histtype='stepfilled')
plt.title ('histogram only')
print ('histogram bin size = ', h[1].size)
```

```
histogram bin size =  101
```



```
h = plt.hist(xint, bins=100, histtype='stepfilled', density=True)
plt.title ('histogram normalized == sample density')
```

```
Text(0.5, 1.0, 'histogram normalized == sample density')
```

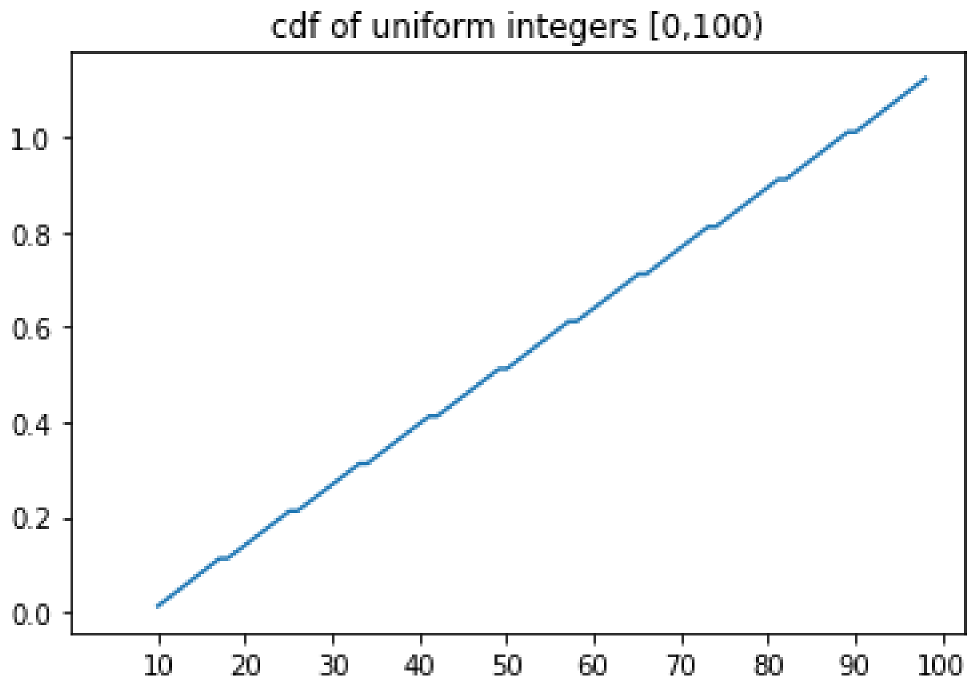


```
h[0][:10]
```

```
array([0.01247528, 0.01243596, 0.01246067, 0.01265281, 0.01251798,
       0.0124427 , 0.01262022, 0.01265169, 0.01255281, 0.        ])
```

```
freq = h[0] # this is already normalized. So, the cumulative plot will have 1 at the␣
↪last index
cdf = cumfreq (freq)
plt.plot (h[1][:-1], cdf)
plt.xlim(left=0)
plt.xticks(np.arange(10, 110, 10))
plt.title ('cdf of uniform integers [0,100)')
```

```
Text(0.5, 1.0, 'cdf of uniform integers [0,100)')
```

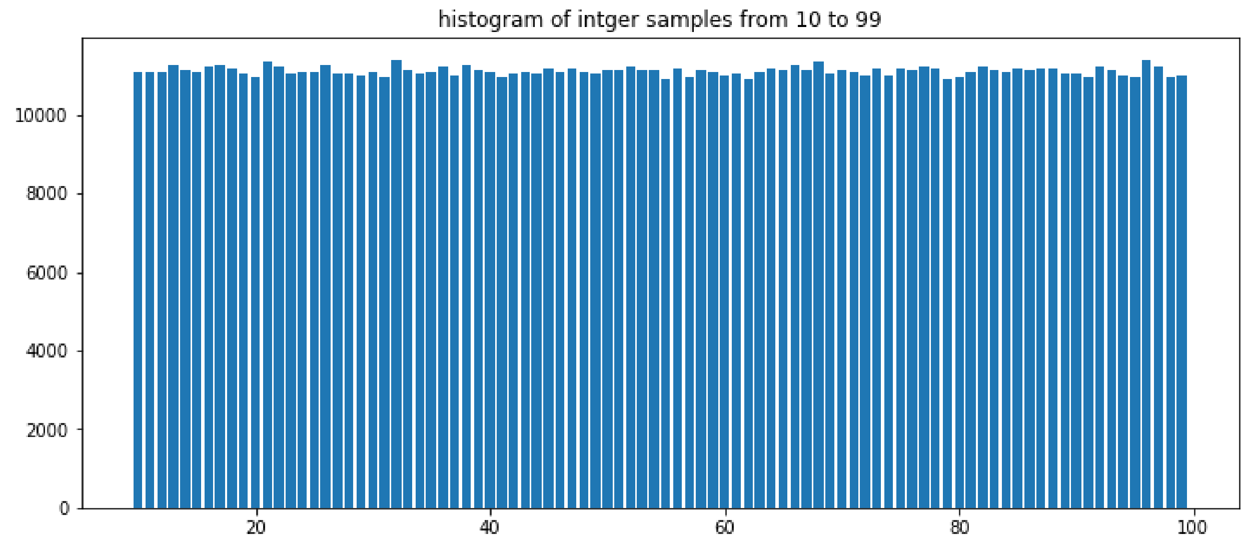


6.3.1 Histogram using `np.unique()`

```
unique, count = np.unique(xint, return_counts=True)
```

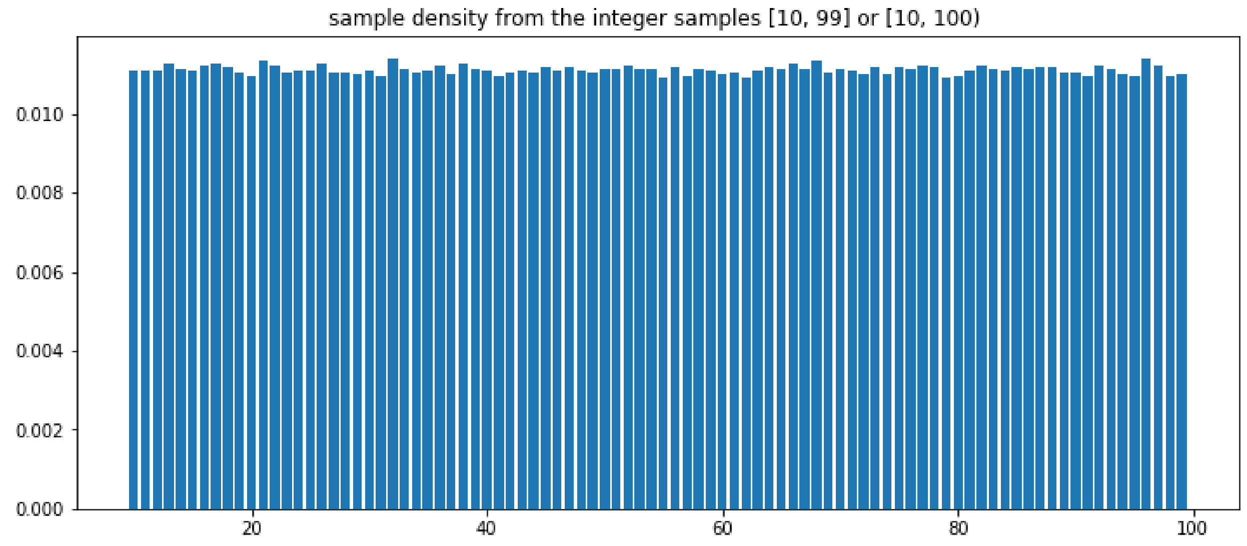
```
plt.figure(figsize=(12, 5))
plt.bar (unique, count)
plt.title ('histogram of intger samples from 10 to 99')
```

```
Text(0.5, 1.0, 'histogram of intger samples from 10 to 99')
```



```
plt.figure(figsize=(12, 5))
plt.bar (unique, count/xint.shape[0])
plt.title ('sample density from the integer samples [10, 99] or [10, 100) ')
```

```
Text(0.5, 1.0, 'sample density from the integer samples [10, 99] or [10, 100) ')
```



6.4 Bernoulli Distribution

- https://en.wikipedia.org/wiki/Bernoulli_distribution
- Parameter

$$Pr(X = 1) = p$$

- Probability Mass Function (PMF)

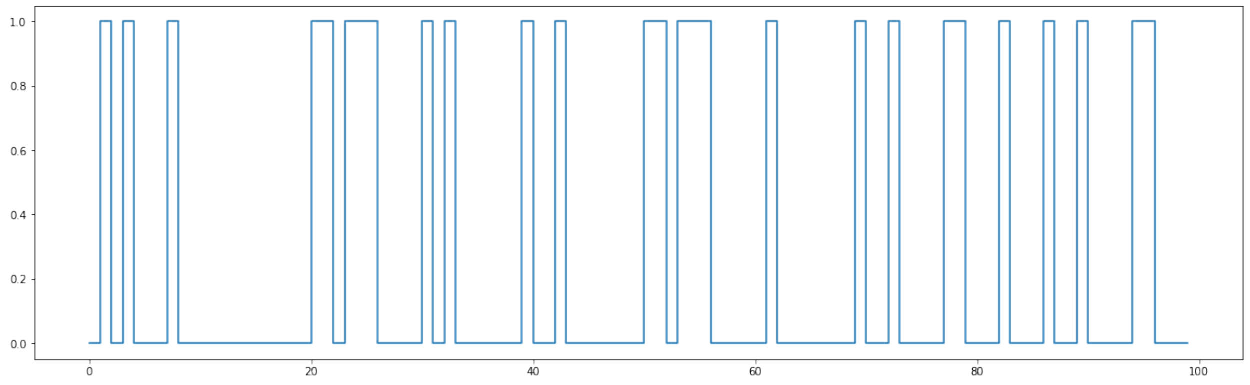
$$Pr(X = y) = p^y(1 - p)^{(1-y)}, \quad y \in \{0, 1\}$$

```
ntrial = 1
p = 0.3

# To examine the outpus
x = np.random.binomial(ntrial, p, 100)
x
```

```
array([0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
       1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1,
       0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1,
       0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0])
```

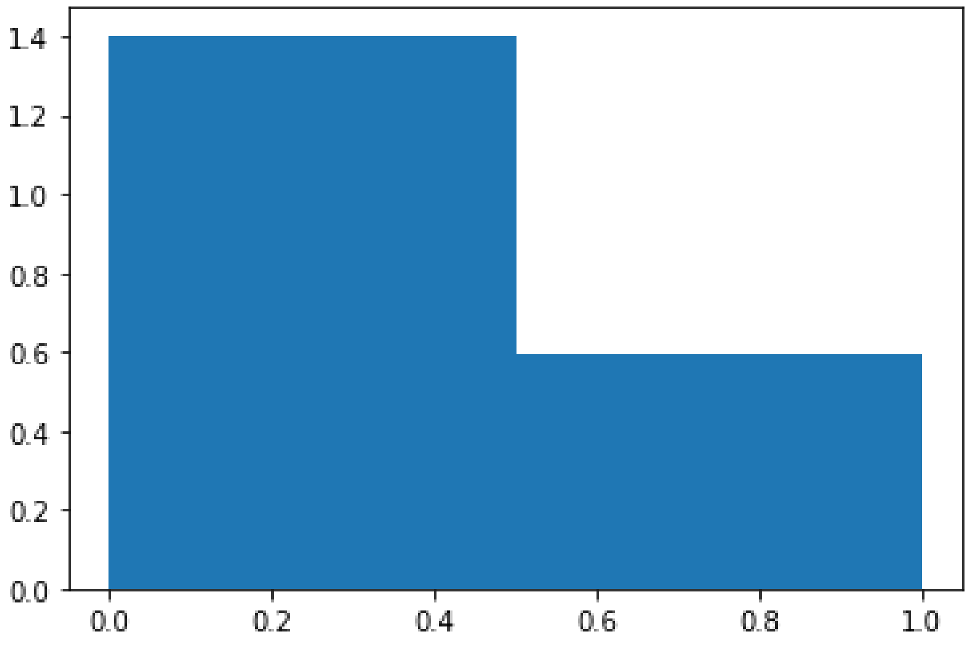
```
plt.figure(figsize=(20,6))
plt.plot (x, drawstyle='steps')
```

```
[<matplotlib.lines.Line2D at 0x7f50fb806be0>]
```



```
x = np.random.binomial (ntrial, p, 100000)
```

```
h = plt.hist (x, bins=2, density=True)
```



```
h[0], h[1]
```

```
(array([1.40266, 0.59734]), array([0. , 0.5, 1. ]))
```

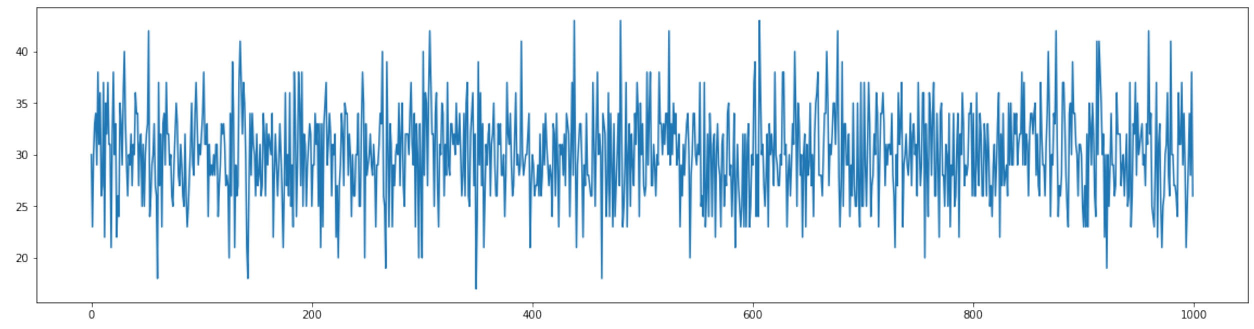
6.5 Binomial Distribution

- https://en.wikipedia.org/wiki/Binomial_distribution
  - In general, if the random variable X follows the binomial distribution with parameters $n \in \mathbb{N}$ and $p \in [0, 1]$, we write $X \sim B(n, p)$.
- PMF

$$Pr(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

```
ntrial = 100
p = 0.3

xbino = np.random.binomial(n=ntrial, p=p, size=1000)
```

```
plt.figure (figsize=(20,5))
h = plt.plot (xbino)
```

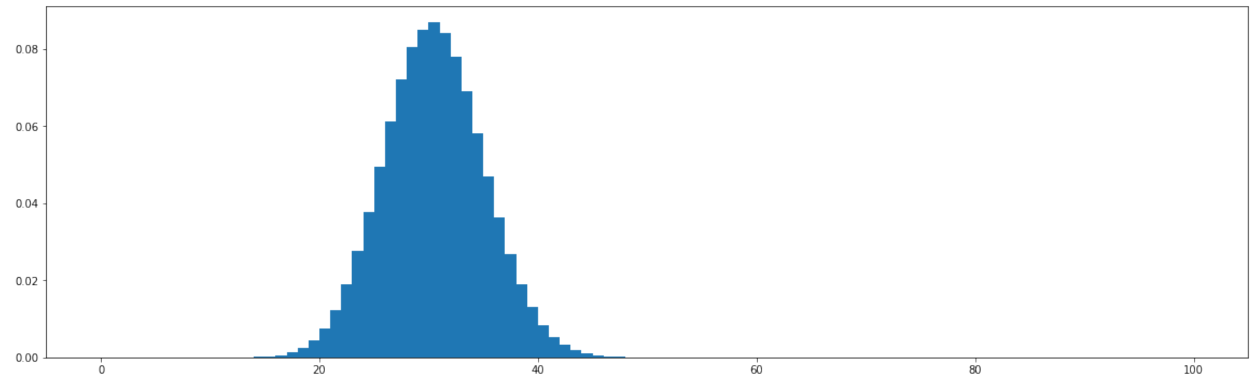


```
# summary of the generated numbers
np.unique (xbino)
```

```
array([17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43])
```

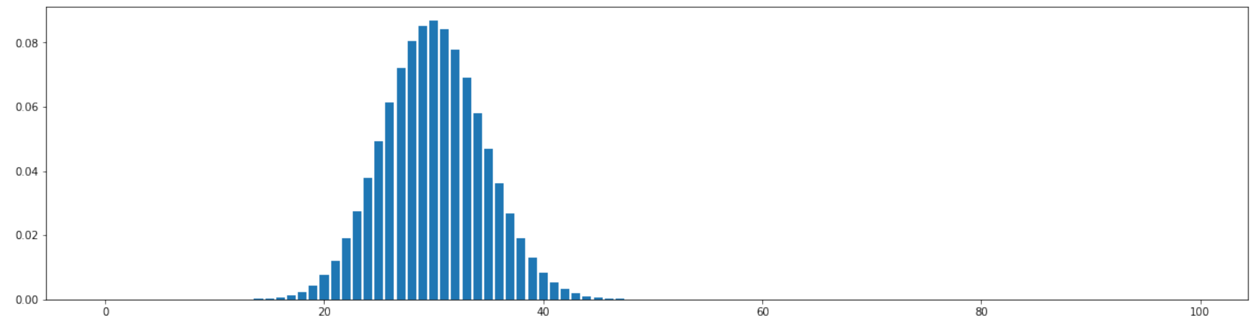
```
xbino = np.random.binomial (n=ntrial, p=p, size=1000000)
```

```
bins = [i for i in range(101)] # discrete bins

plt.figure (figsize=(20,6))
h = plt.hist (xbino, bins=bins, density=True, histtype='stepfilled')
```



```
xcoord = [i for i in range(h[0].size)]

plt.figure (figsize=(20,5))
plt.bar (xcoord, h[0])
```

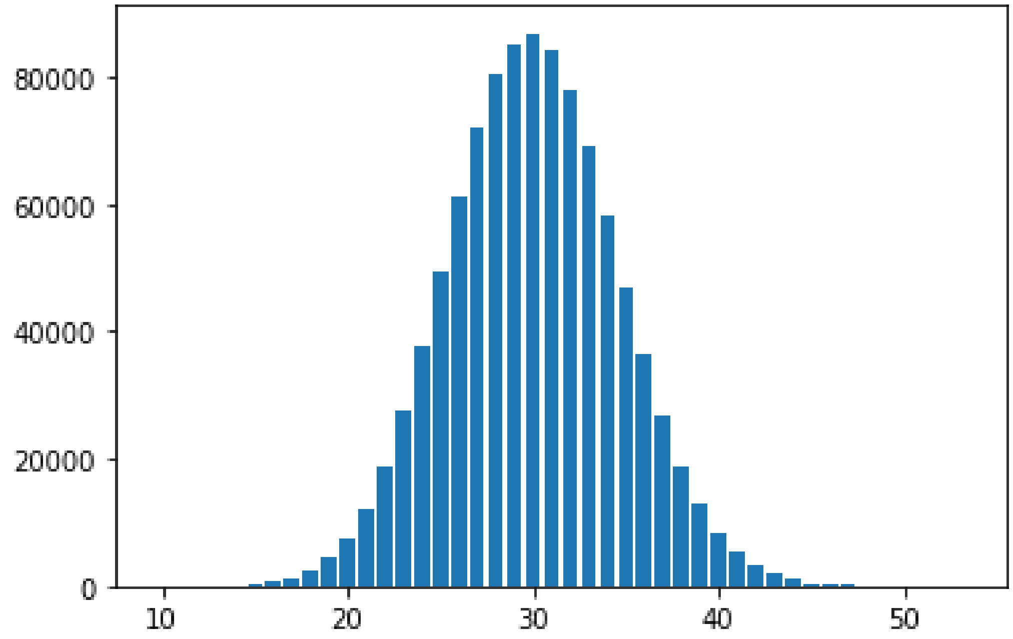
```
<BarContainer object of 100 artists>
```



```
unique, counts = np.unique (xbino, return_counts=True)
unique, counts
```

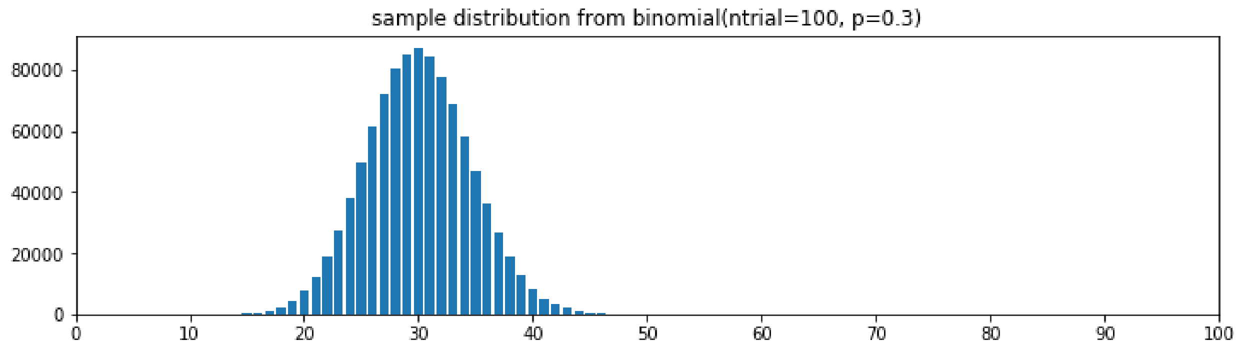
```
(array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
        27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
        44, 45, 46, 47, 48, 49, 50, 51, 52, 53]),
 array([    2,      3,     10,     45,    105,    241,    559,   1227,   2343,
         4437,   7585, 12175, 18954, 27563, 37825, 49405, 61284, 72054,
        80544, 85029, 86869, 84196, 77890, 68944, 58026, 46888, 36275,
        26713, 18991, 12988,  8372,  5198,  3248,  1878,  1037,    523,
          316,    155,     67,     19,     12,      2,      2,      1]))
```

```
_ = plt.bar (unique, counts)
```



```
plt.figure (figsize=(12,3))
plt.bar (unique, counts)
plt.xticks (np.arange(start=0, stop=ntrial+1, step=10))
plt.title ('sample distribution from binomial(ntrial={}, p={})'.format(ntrial, p))
```

```
Text(0.5, 1.0, 'sample distribution from binomial(ntrial=100, p=0.3)')
```

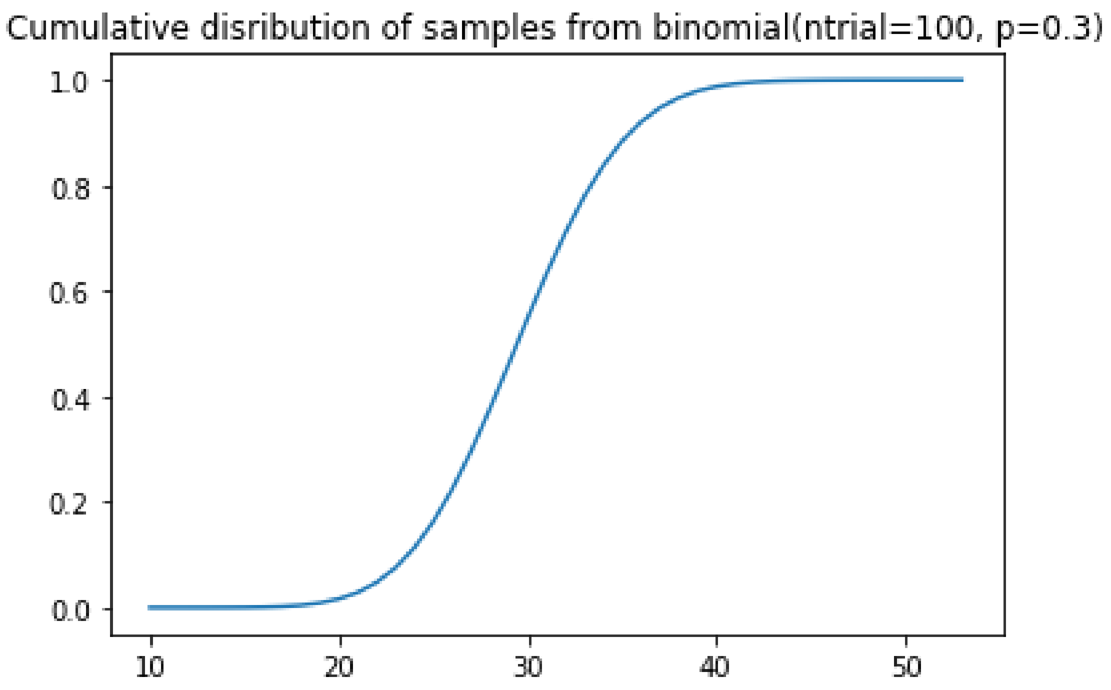


6.5.1 Sample CDF

```
cdf = np.zeros_like (counts)
cdf[0] = counts[0]
for i in range(1, unique.shape[0]):
    cdf[i] += cdf[i-1] + counts[i]
```

```
plt.plot (unique, cdf/cdf[-1])
plt.title ('Cumulative disribution of samples from binomial(ntrial={}, p={})'.
↪format(ntrial, p))
```

```
Text(0.5, 1.0, 'Cumulative disribution of samples from binomial(ntrial=100, p=0.3)')
```



6.6 Histogram for Discrete Variables using python dictionary

```
xbino.min(), xbino.max()
```

```
(10, 53)
```

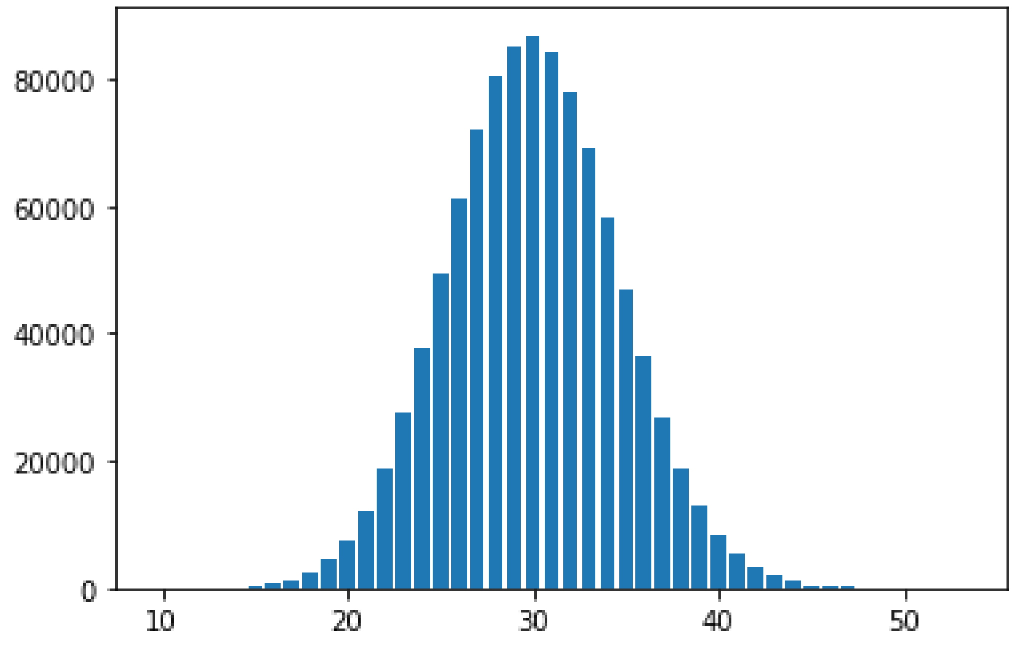
```
hist = { i : 0 for i in range (xbino.min(), xbino.max()+1, 1) }
for n in xbino:
    hist[n] += 1
```

```
hist.keys()
```

```
dict_keys([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
↪ 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
↪ 50, 51, 52, 53])
```

```
# with proper x-range
plt.bar(hist.keys(), hist.values())
```

```
<BarContainer object of 44 artists>
```



6.7 scipy.stats

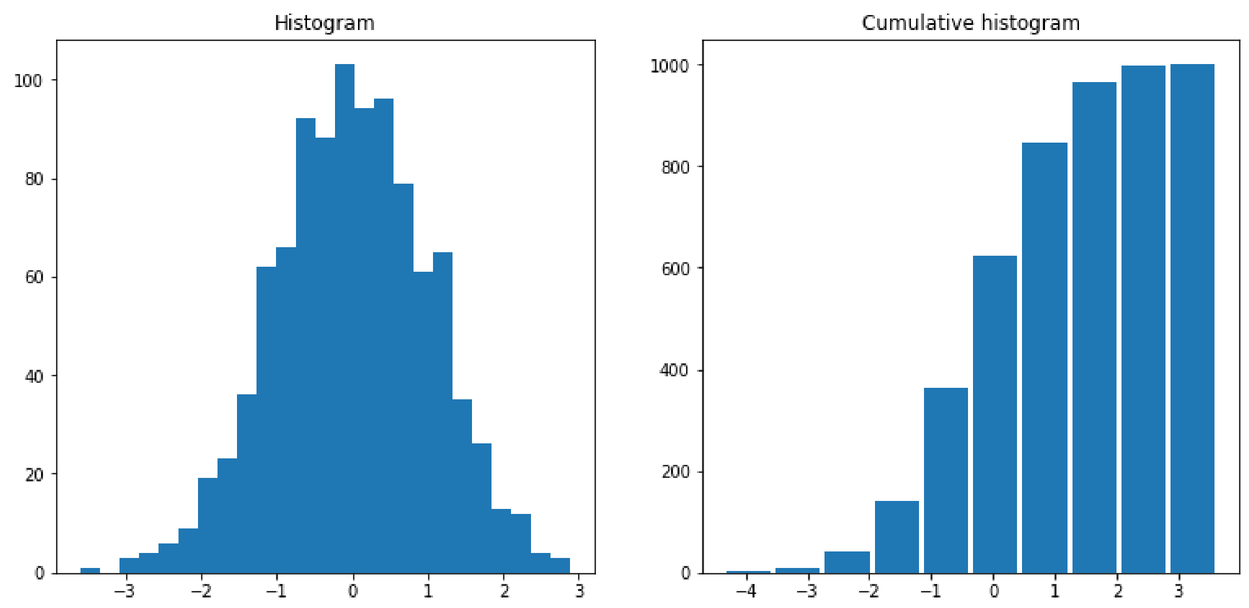
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.cumfreq.html

```
import scipy.stats
```

```
samples = np.random.randn (1000)

res = scipy.stats.cumfreq (samples)
x = res.lowerlimit + np.linspace(0, res.binsize * res.cumcount.size, res.cumcount.
↪size)
```

```
fig = plt.figure (figsize=(13,6))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)
ax1.hist (samples, bins=25)
ax1.set_title ('Histogram')
ax2.bar (x, res.cumcount, width=res.binsize)
ax2.set_title ('Cumulative histogram')
#ax2.set_xlim ([x.min(), x.max()])
```

```
Text(0.5, 1.0, 'Cumulative histogram')
```



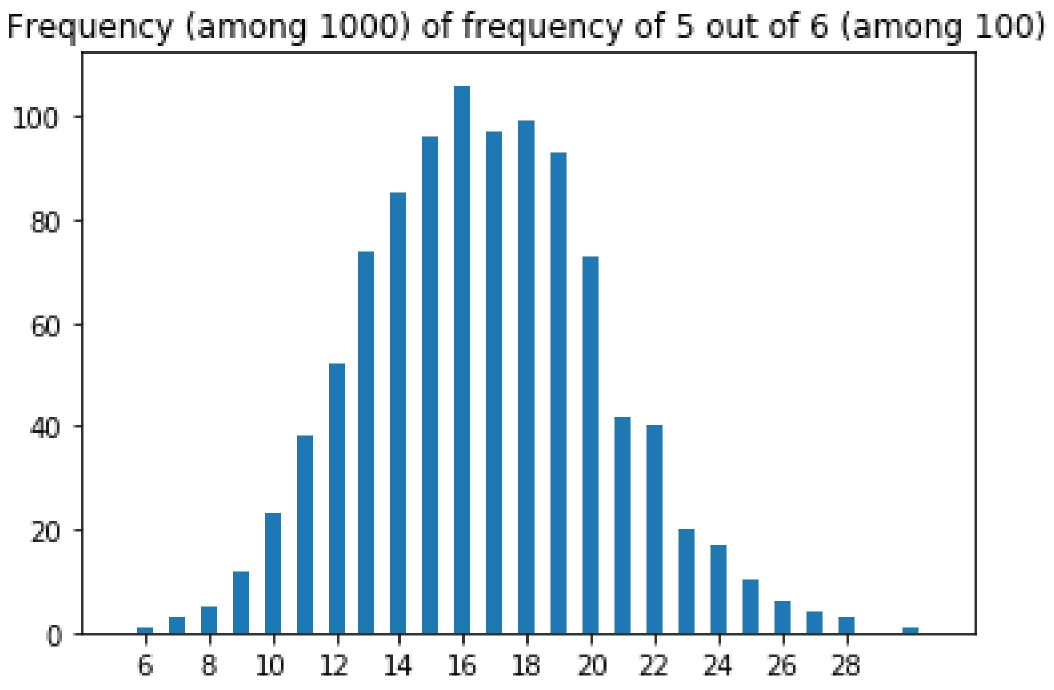
6.8 (extra) How many times will a fair die land on the same number (e.g. 5) out of 100 trials.

```
- use 'np.random.binomial(n=100, p=1/6., size=N)` to generate the samples.
- base event set = { face is 5, face is not 5 }, so it is binary. (the same applies␣
↪to other numbers each)
```

```
import numpy as np
import matplotlib.pyplot as plt

# great seed
np.random.seed(1337)
```

```
# how many times will a fair die land on the same number out of 100 trials.
data = np.random.binomial(n=100, p=1/6, size=1000)
```

```
values, counts = np.unique(data, return_counts=True) # can be used to construct a␣
↪histogram of discrete dataset

plt.vlines(values, 0, counts, color='C0', lw=6)

plt.title ('Frequency (among 1000) of frequency of 5 out of 6 (among 100)')
# optionally set y-axis up nicely
plt.ylim(0, max(counts) * 1.06)
plt.xlim(values.min()-2, values.max()+2)
_ = plt.xticks(values[::2])
```



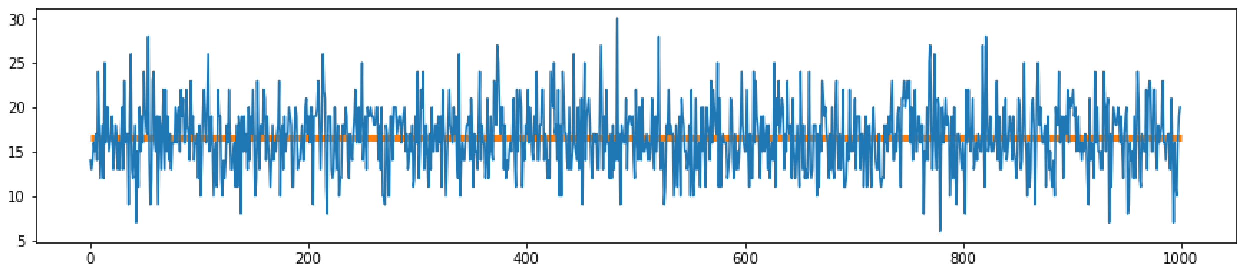
```
print ('Sample mean says that 5 will appear {} times on average out of 100 trials'.
↪format(data.mean()))
```

```
Sample mean says that 5 will appear 16.651 times on average out of 100 trials
```

```
print ('Sample probability: p = ', data.mean() / 100, '\nNotice: True p = ', 1/6)
```

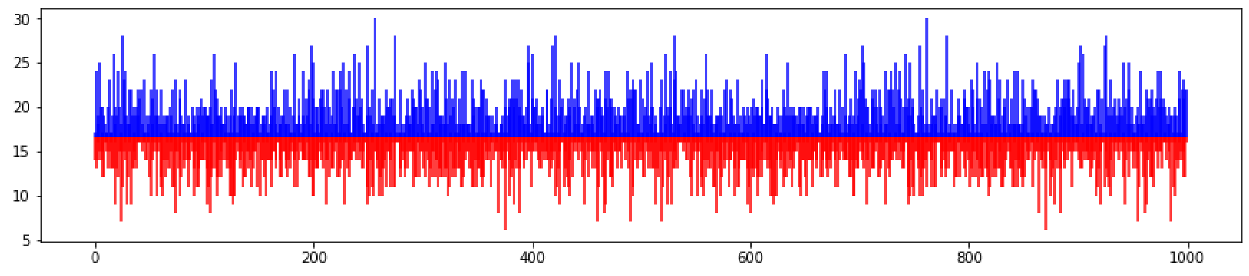
```
Sample probability: p =  0.16651
Notice: True p =  0.16666666666666666
```

```
plt.figure(figsize=(15,3))
plt.hlines(data.mean(), xmin=0, xmax=data.size, color='C1', lw=5)
_ = plt.plot (data)
```



```
plt.figure(figsize=(15,3))
plt.vlines (np.arange(data.size), data.mean(), data[data>data.mean()], color='b')
plt.vlines (np.arange(data.size), data.mean(), data[data<=data.mean()], color='r')
```

```
<matplotlib.collections.LineCollection at 0x7f50fb75ea58>
```



6.9 END

CHAPTER

SEVEN

IRIS DATA CLASSIFICATION

1. closest to the class center
2. Full Bayes with Gaussian Model
3. Naive Bayes with conditionally independent Gaussian

```
%matplotlib inline
```

install sklearn if not installed

```
!pip3 install sklearn
```

```
import matplotlib.pyplot as plt
import numpy as np
import sklearn.datasets
```

```
iris = sklearn.datasets.load_iris()
dir(iris)
```

```
['DESCR', 'data', 'feature_names', 'filename', 'target', 'target_names']
```

7.1 We use the first two features in the beginning

- Binary classification: Setosa vs The Others

```
data = iris.data[:, :2]
y = iris.target
print (data.shape)
```

```
(150, 2)
```

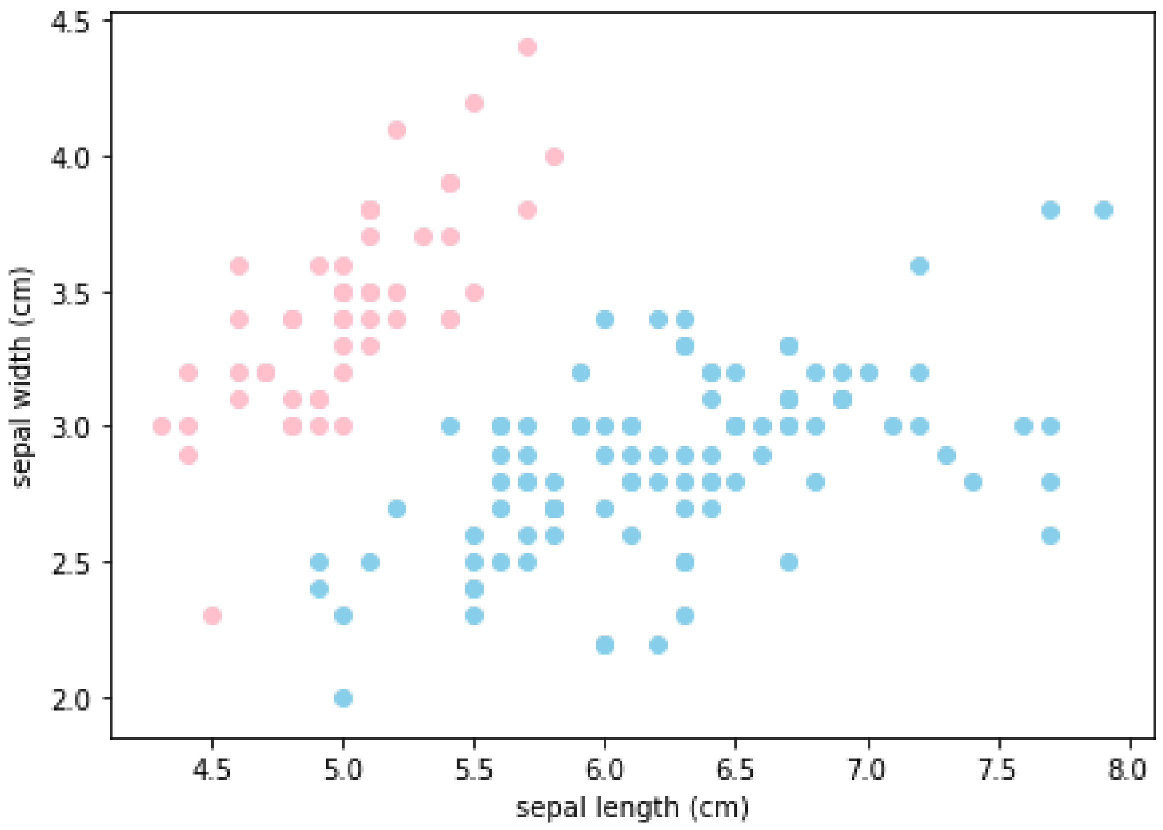
```
sel0 = y == 0 # setosa
sel1 = sel0 == False # the others
groups = [sel0, sel1]
nclasses = 2
nfeatures = 2
```

7.1.1 scatter plot

```
from matplotlib.patches import Ellipse

def scatter(centers=None, covs=None, scale=None):
    fig, ax = plt.subplots(figsize=(7,5))
    colors = ['pink', 'skyblue']
    for k in range (nclasses):
        s = groups[k]
        ax.scatter (data[s,0], data[s,1], color=colors[k])
    ax.set_xlabel (iris.feature_names[0])
    ax.set_ylabel (iris.feature_names[1])
    #
    if centers is not None:
        center_color = ['r', 'b']
        for k in range(nclasses):
            ax.scatter(centers[k,0], centers[k,1], color=center_color[k], s=200,␣
↪marker='*')
    #
    if covs is not None:
        colors = ['r', 'b']
        for k in range(nclasses):
            w, v = np.linalg.eig(covs[k]) # C = V W V'
#             print ('eig: ', w, v)
            if w[0] < w[1]:
                w = np.sort(w)[::-1] #
                v = v[:, w.argsort()] #
#                 print ('\teig: ', w, v)

            xaxis = v[:,0] #
#             print ('xaxis: ', xaxis, '  w: ', w)
            theta = np.arctan2(xaxis[1], xaxis[0])
#             print ('theta: ', np.degrees(theta))
            scale = 2 if scale is None else scale
            el = Ellipse (centers[k],
                          width=2*np.sqrt(w[0]*scale), # reverse x, y since the last␣
↪is the greatest
                          height=2*np.sqrt(w[1]*scale),
                          angle=np.degrees(theta),
                          alpha=0.3, color=colors[k])
            ax.add_artist(el)
    #
    return fig, ax
```

```
scatter()
```

```
(<Figure size 504x360 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7fd094a60c18>)
```



7.2 Method 1: Closest to the cluster center

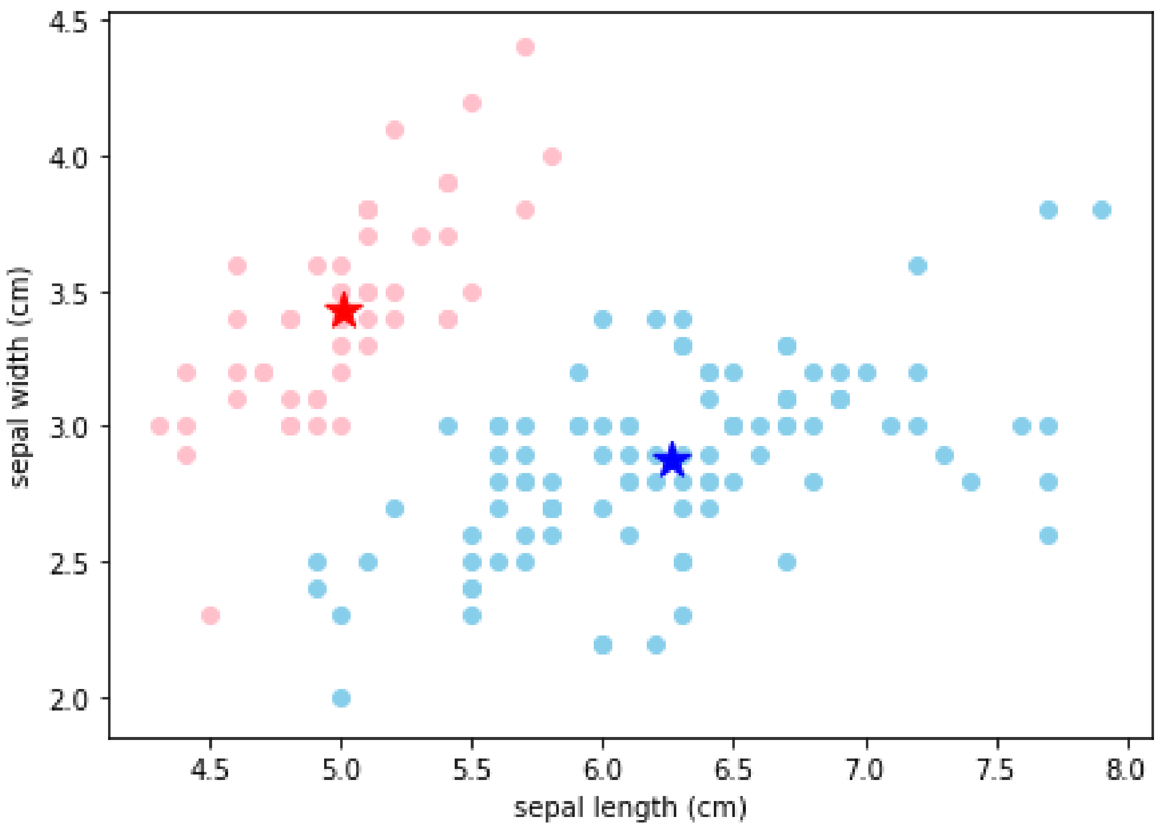
7.2.1 compute cluster centers

```
means = np.zeros ((nclasses, nfeatures))
for k in range(nclasses):
    means[k] = data[groups[k]].mean(axis=0)
print ('means = ', means)
```

```
means =  [[5.006 3.428]
 [6.262 2.872]]
```

```
scatter(centers=means)
```

```
(<Figure size 504x360 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7fd05aedfd68>)
```



7.2.2 distance() and predict()

```
def distance(xy, ab):
    return np.sqrt( np.power(xy[0]-ab[0],2.) + np.power(xy[1]-ab[1],2.))
#
def closest_predict(xy):
    d0 = distance (xy, means[0])
    d1 = distance (xy, means[1])
    if d0 < d1: return 0
    else: return 1
#
```

7.2.3 prediction map computation

```
step=0.1
xx = np.arange(4.0, 8.2, step=step)
yy = np.arange(1.8, 4.6, step=step)
print ('xx.shape={}  yy.shape={}'.format(xx.shape, yy.shape))
# make grid
xgrid = np.empty((yy.shape[0], xx.shape[0]))
ygrid = np.empty((yy.shape[0], xx.shape[0]))
pgrid = np.empty((yy.shape[0], xx.shape[0]))
for i in range(pgrid.shape[0]):
    for j in range(pgrid.shape[1]):
```

(continues on next page)

(continued from previous page)

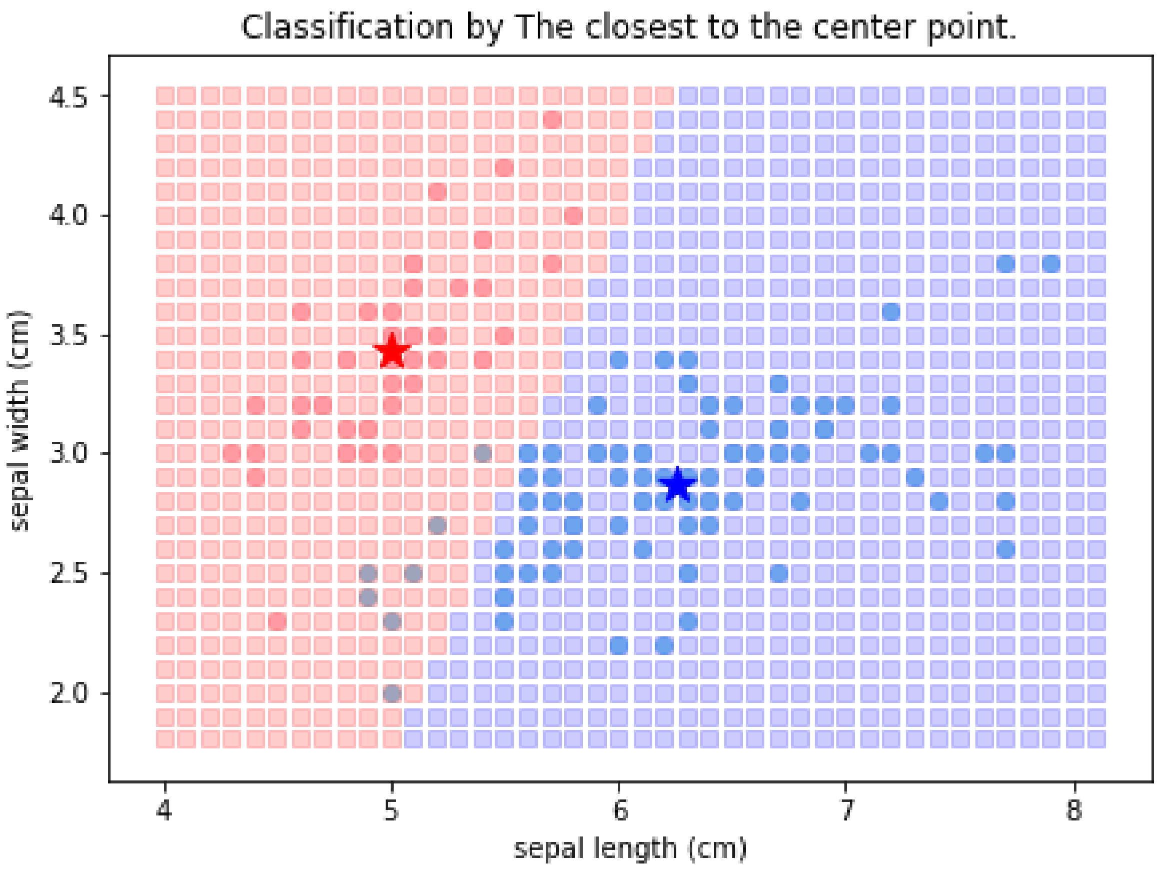
```
        ygrid[i,j] = yy[i]
        xgrid[i,j] = xx[j]
        pgrid[i,j] = closest_predict ((xx[j], yy[i]))
#
```

```
xx.shape=(42,)  yy.shape=(28,)
```

7.2.4 classification map

```
fig, ax = scatter(centers=means)
setosa = pgrid < 1
theothers = pgrid > 0
ax.scatter (xgrid[setosa].ravel(), ygrid[setosa].ravel(), color='r', alpha=0.2,␣
↪marker='s')
ax.scatter (xgrid[theothers].ravel(), ygrid[theothers].ravel(), color='b', alpha=0.2,␣
↪marker='s')
ax.set_title ('Classification by The closest to the center point.')
```

```
Text(0.5, 1.0, 'Classification by The closest to the center point.')
```



7.3 Method 2: Full Gaussian modeling and closest to the center with scale

- Naive Bayes: https://scikit-learn.org/stable/modules/naive_bayes.html
- What we do below is full Bayesian

7.3.1 Sample Covariance Matrix

```
def cov (X, mean):
    C = np.zeros((X.shape[1], X.shape[1]))
    for i in range(X.shape[0]):
        z = X[i] - mean
        a = z[:, np.newaxis]
        C += a @ a.transpose()
    C /= (X.shape[0] - 1) # unbiased. But -1 is not necessary when we have many data␣
↪like this example.
    return C
```

```
covs = [cov(data[groups[k]], means[k]) for k in range(nclasses) ]
covs
```

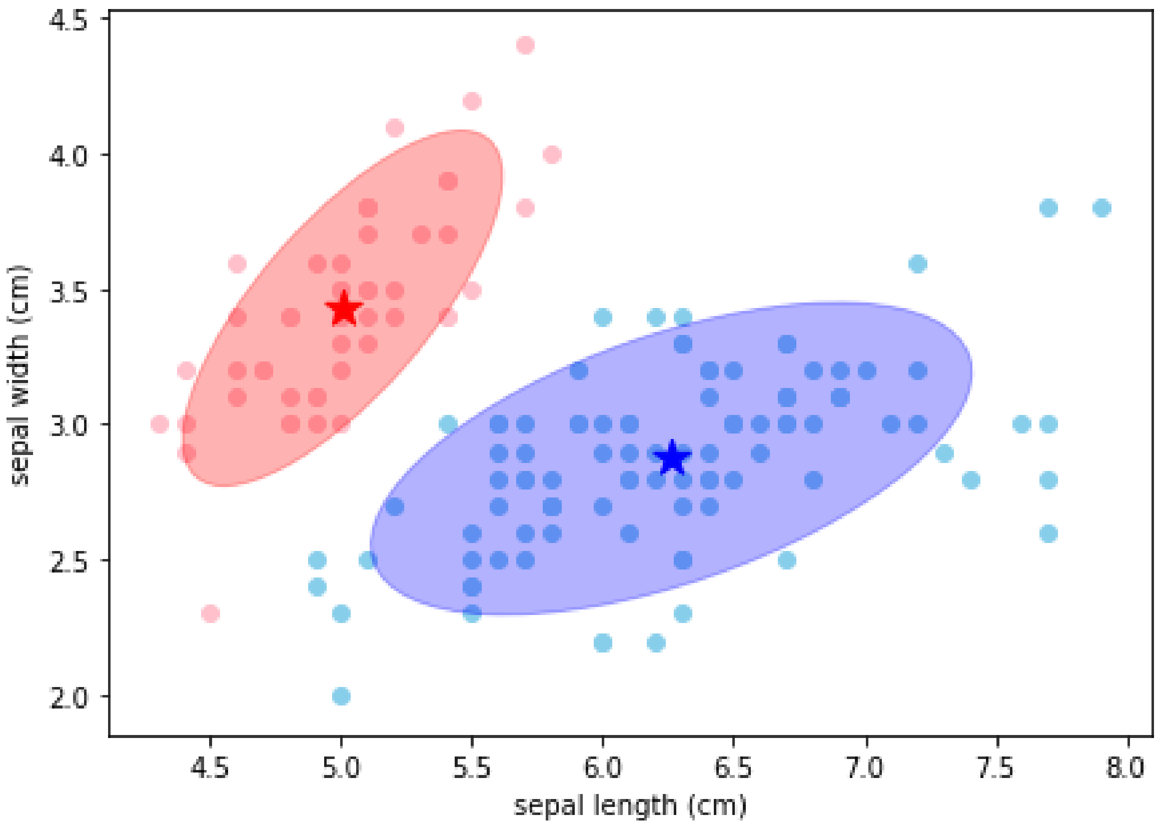
```
[array([[0.12424898, 0.09921633],
        [0.09921633, 0.1436898 ]]), array([[0.43934949, 0.12215758],
        [0.12215758, 0.11072323]])]
```

```
# numpy funtion for covariance
covs_np = [ np.cov( data[groups[k]].transpose() ) for k in range(nclasses) ]
covs_np
```

```
[array([[0.12424898, 0.09921633],
        [0.09921633, 0.1436898 ]]), array([[0.43934949, 0.12215758],
        [0.12215758, 0.11072323]])]
```

```
scatter(centers=means, covs=covs, scale=3)
```

```
(<Figure size 504x360 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7fd05a98f908>)
```



7.3.2 2D Gaussain PDF as Conditional Likelihood Functions

- given the class label

```
def Gaussian2d(x, mean, cov):
    a = np.dot( np.linalg.inv(cov), x-mean)
    f = np.exp( np.dot(x-mean, a) / -2) / np.sqrt( (2*np.pi)**2 * np.linalg.det(cov) )
    return f
```

```
x = np.array([[4.6, 2.6]])
Gaussian2d(x[0], means[0], covs[0])
```

```
0.12792515163397944
```

```
# scipy function
import scipy.stats
gpdf2d = scipy.stats.multivariate_normal.pdf
print('Gaussian2d: ', gpdf2d([4.6, 2.6], mean=means[0], cov=covs[0]) )
```

```
Gaussian2d:  0.12792515163397936
```

7.3.3 Prior Distribution

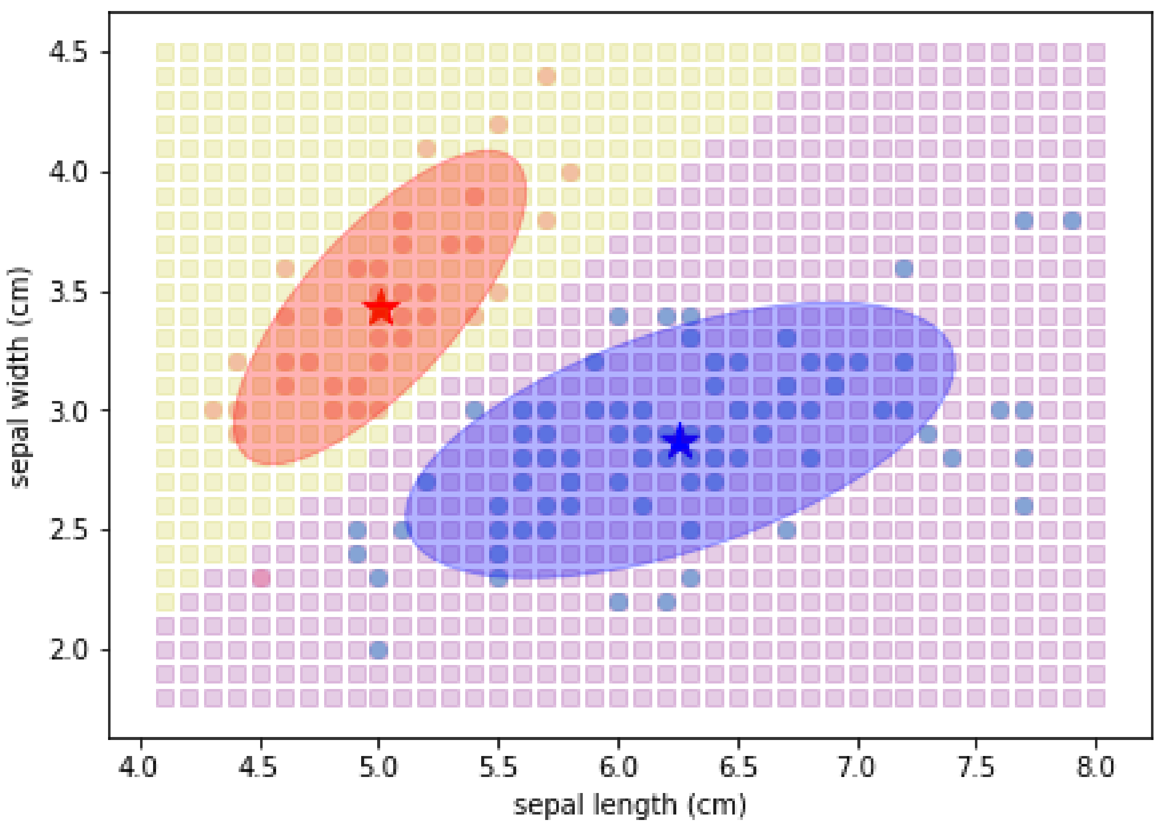
```
Prior = np.array([ groups[k].mean() for k in range(nclasses)])
Prior
```

```
array([0.33333333, 0.66666667])
```

7.3.4 Feature space separation

```
setosa = []
theothers = []

fig, ax = scatter (centers=means, covs=covs, scale=3)
step = 0.1
xx = np.arange(4.1, 8.1, step=step)
yy = np.arange(1.8, 4.6, step=step)
for x in xx:
    for y in yy:
        xy = np.array([x, y])
        g0 = Gaussian2d(xy, means[0], covs[0])
        g1 = Gaussian2d(xy, means[1], covs[1])
        posterior0 = g0 * Prior[0] # / evidence
        posterior1 = g1 * Prior[1]

        if posterior0 > posterior1: # this is setosa
            setosa.append(xy)
#             ax.scatter(xy[0], xy[1], marker='s', alpha=0.2, color='y')
        else:
            theothers.append(xy)
#             ax.scatter(xy[0], xy[1], marker='s', alpha=0.2, color='purple')
setosa = np.array(setosa)
theothers = np.array(theothers)
ax.scatter(setosa[:,0], setosa[:,1], marker='s', alpha=0.2, color='y')
ax.scatter(theothers[:,0], theothers[:,1], marker='s', alpha=0.2, color='purple')
```

```
<matplotlib.collections.PathCollection at 0x7fd05a8ce400>
```



7.4 Method 3: Naive Bayes

- conditional indepence assumption for the features

$$p(x_1, x_2|C) = p(x_1|C)p(x_2|C)$$

```
import sklearn.naive_bayes
gnb = sklearn.naive_bayes.GaussianNB()
```

7.4.1 Naive Bayes fit (sklearn)

```
gnb = gnb.fit (data, groups[0])
```

```
gnb.predict([[5.0, 3.0]])
```

```
array([ True])
```

7.4.2 grid generation for classification map

```
xg, yg = np.meshgrid(xx, yy)
```

```
grid = np.concatenate ((xg[np.newaxis,:], yg[np.newaxis,:]))
```

```
xg.shape, grid.shape
```

```
((28, 40), (2, 28, 40))
```

```
grid = grid.transpose((1,2,0))
grid.shape
```

```
(28, 40, 2)
```

```
gridarr = grid.reshape(-1,2)
gridarr.shape
```

```
(1120, 2)
```

```
gridarr[0], gridarr[-1]
```

```
(array([4.1, 1.8]), array([8. , 4.5]))
```

```
pred = gnb.predict(gridarr)
```

```
pred.sum() / pred.shape[0]
```

```
0.29017857142857145
```

```
setosa_gnb = gridarr[pred]
theothers_gnb = gridarr[pred==False]
```

```
# mean locations
gnb.theta_
```

```
array([[6.262, 2.872],
       [5.006, 3.428]])
```

```
means # our computation of the mean location
```

```
array([[5.006, 3.428],
       [6.262, 2.872]])
```

```
# isotropic, diagonal. sigam is the variance of each feature per class
gnb.sigma_
```

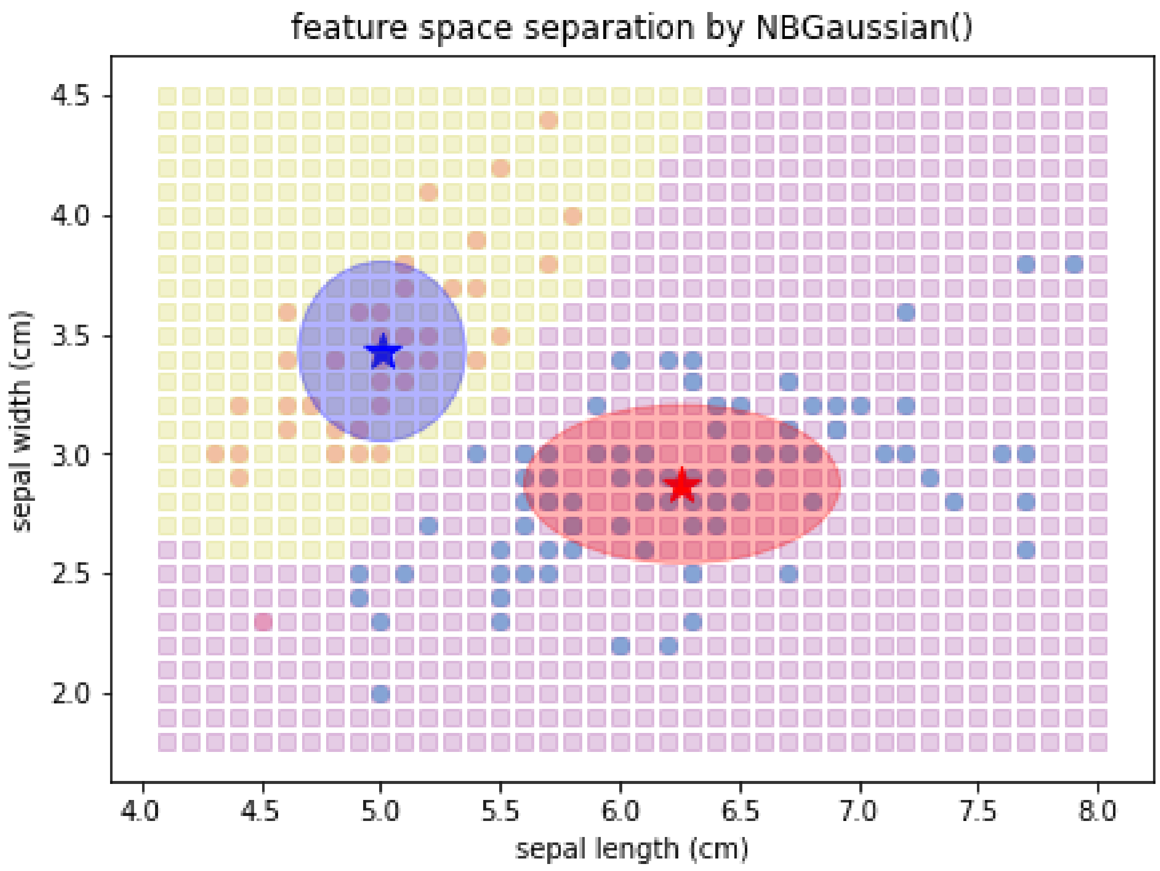
```
array([[0.434956, 0.109616],
       [0.121764, 0.140816]])
```

```
# class prior probabilities
gnb.class_prior_
```

```
array([0.66666667, 0.33333333])
```

```
covs0 = [np.diag(gnb.sigma_[k]) for k in range(nclasses)]
fig, ax = scatter (centers=gnb.theta_, covs=covs0, scale=1)
ax.scatter(setosa_gnb[:,0], setosa_gnb[:,1], marker='s', alpha=0.2, color='y')
ax.scatter(theothers_gnb[:,0], theothers_gnb[:,1], marker='s', alpha=0.2, color=
↪'purple')
ax.set_title('feature space separation by NBGaussian()')
```

```
Text(0.5, 1.0, 'feature space separation by NBGaussian()')
```



Conclusion?

Naive Bayes is not bad. Actually it seems quite good in spite of its simplicity.

CHAPTER

EIGHT

EOF

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
import sklearn.datasets
breast_cancer = sklearn.datasets.load_breast_cancer()
```

```
data = breast_cancer.data
y = breast_cancer.target
data.shape, y.shape
```

```
((569, 30), (569,))
```

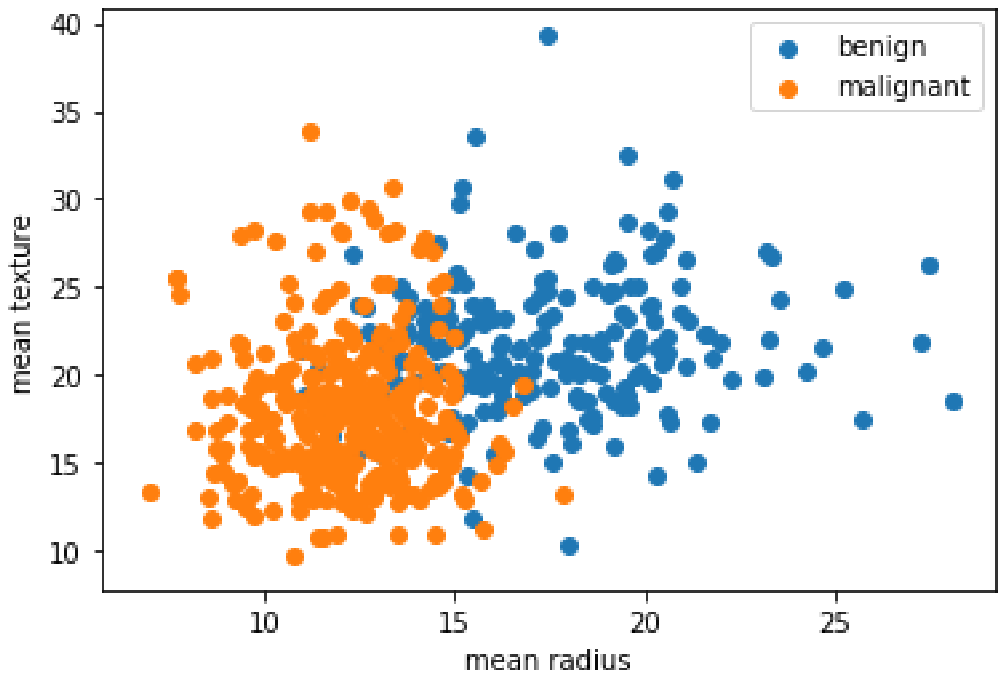
```
np.unique (breast_cancer.target_names)
```

```
array(['benign', 'malignant'], dtype='<U9')
```

```
benign = y == 0
malig = y == 1
```

```
def scatter_plot(i, j):
    plt.scatter(data[benign,i], data[benign,j], label='benign')
    plt.scatter(data[malig,i], data[malig,j], label='malignant')
    plt.xlabel (breast_cancer.feature_names[i])
    plt.ylabel (breast_cancer.feature_names[j])
    plt.legend()
```

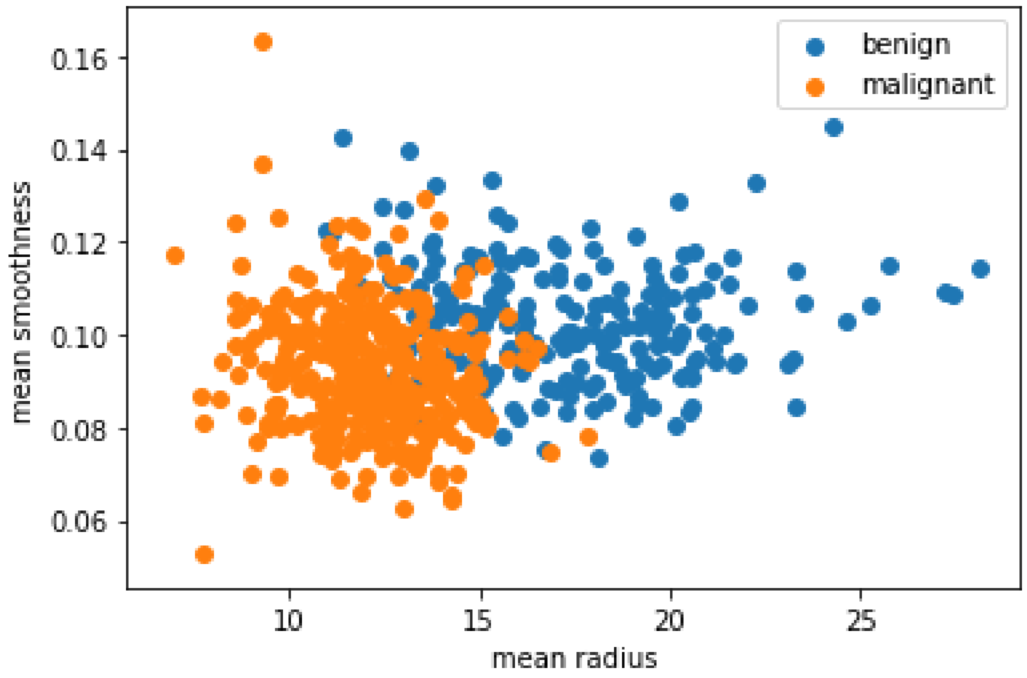
```
scatter_plot (0,1)
```



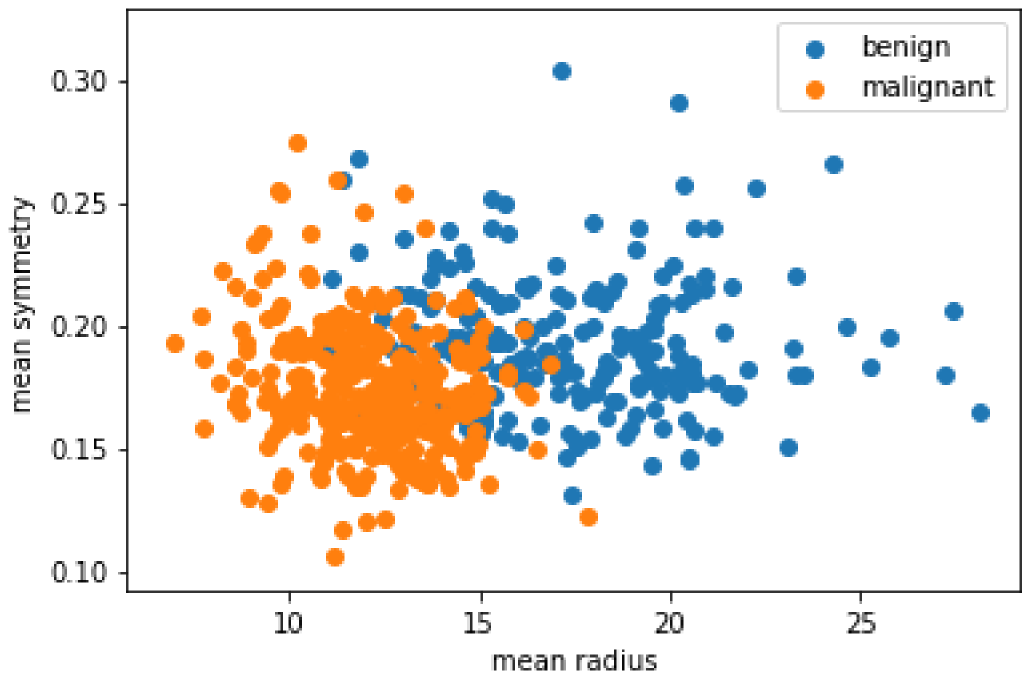
```
data.shape
```

```
(569, 30)
```

```
scatter_plot(0,4)
```



```
scatter_plot(0,8)
```


benign
malignant
0.30
0.25
0.20
0.15
0.10
mean symmetry
10
15
20
25
mean radius

CHAPTER

NINE

COMPUTATIONAL PROBABILITY THEORY

- Basic concepts of probability models needs to be revised to solve the problems below.
- Ask google.com, wikipedia.org, or any other helpful source to derive the correct answers
- Due: Oct. 14 Monday, 2019

You need to use `scipy` package to solve the problems. Install `scipy` if you don't have it installed in your computer by

```
pip3 install scipy
```

```
import scipy
print (scipy.__version__)
```

```
1.3.1
```

```
import numpy as np
print (np.__version__)
```

```
1.17.2
```

9.1 We use sub-moduels `scipy.stats` and `scipy.special` as given below.

```
import scipy.stats
import scipy.special
```

9.1.1 Normal distribution

The function `scipy.stats.norm.cdf(a)` gives the probability $Prob[X < a]$ for the standard normal distribution.

$$Prob[X < a] = \int_{-\infty}^{a} p(x)dx$$

$$where : math : `p(x)`isthepdfofthestandardnormaldistribution$$

$(m = 0, \sigma = 1)$

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{ -\frac{1}{2}\frac{(x-m)^2}{\sigma^2} \right\}$$

For example, $Prob[X < 0]$ for the standard normal distribution is 0.5 and can be obtained as follows

```
scipy.stats.norm.cdf(0)
```

```
0.5
```

The point x where cdf(x) == p can be obtained using `scipy.stats.norm.ppf(p).` - that is, the function cdf(x) is 0.5 at x = 0 - which is given below

```
scipy.stats.norm.ppf(0.5)
```

```
0.0
```

9.2 Binomial distribution

- https://en.wikipedia.org/wiki/Binomial_distribution
- PMF (Probability Mass Function)

$$Pr[k|n,p] = \binom{n}{k} p^k (1-p)^{n-k}$$

- use `scipy.stats.binom.cdf()` for CDF
- use `scipy.stats.binom.pmf()` for PMF

```
scipy.stats.binom.cdf(k=3, n=10, p=0.4)
```

```
0.3822806016000001
```

```
scipy.stats.binom.pmf(k=3, n=10, p=0.4)
```

```
0.21499084800000012
```

9.3 Poisson distribution

- https://en.wikipedia.org/wiki/Poisson_distribution
- PMF

$$Pr[k|\mu] = Pr[k \text{ events in interval }] = \frac{\mu^k \exp^{-\mu}}{k!}$$
$$where$$

  - μ is the average number of events per interval
  - wikipedia.org uses λ instead of μ
  - k takes values 0,1,2, …
- use `scipy.stats.poisson.cdf()` for CDF
- use `scipy.stats.poisson.pmf()` for PMF

```
scipy.stats.poisson.pmf(k=90, mu=100)
```

```
0.025038944623030353
```

```
scipy.stats.poisson.cdf(k=90, mu=100)
```

```
0.17138511932176148
```

9.4 Bayes' Theorem

- https://en.wikipedia.org/wiki/Bayes%27_theorem
- product rule of probability

$$P(X = x_i, Y = y_j) = P(Y = y_j|X = x_i)p(X = x_i)$$

  – $P(X = x_i, Y = y_j)$ is called the joint probability.

- sum rule of probability

$$P(X = x_i) = \sum_{j=1}^{L} P(X = x_i, Y = y_j)$$

  – $P(X = x_i)$ is called the marginal probability.

- Bayes' theorem

$$P(Y = y_j|X = x_i) = \frac{P(X = x_i|Y = y_j)P(Y = y_j)}{P(X = x_i)} = \frac{P(X = x_i|Y = y_j)P(Y = y_j)}{\sum_k P(X = x_i|Y = y_k)P(Y = y_k)}$$

```
A survey is conducted in a large office building. It is found that 30% of the office␣
↪workers weigh less than 62kg and that 25% of the office workers weigh more than␣
↪98kg.

The weights of the office workers may be modelled by a normal distribution with mean
↪$m$ and standard deviation $s$.
```

(a)

1. Determine two simultaneous linear equations satisfied by $m$ and $s$.
2. Compute the values of $m$ and $s$

(b) Find the probability that an office worker weighs more than 100kg.

```
There are elevators in the office building that take the office workers to their␣
↪offices.
Given that there are 10 workers in a particular elevator,
```

(c) find the probability that at least four of the workers weigh more than 100kg.

```
Given that there are 10 workers in an elevator and at least one weighs more than␣
↪100kg,
```

(d) find the probability that there are fewer than four workers exceeding 100kg.

```
The arrival of the elevators at the ground floor between 08:00 and 09:00 can be
↪modelled by a Poisson distribution.

Elevators arrive on average every 36 seconds.
```

(e) Find the probability that in any half hour period between 08:00 and 09:00 more than 60 elevators arrive at the ground floor.

```
An elevator can take a maximum of 10 workers. Given that 400 workers arrive in a half
↪hour period independently of each other,
```

(f) find the probability that there are sufficient elevators to take them to their offices.

9.4.1 END.

```
import scipy.stats as st
import scipy.special as ssp
```

```
a, b = st.norm.ppf(0.3), st.norm.ppf(0.75)
```

```
s = 36 / (b - a)
s
```

```
30.02776910773775
```

```
m = 62 - a * s
m
```

```
77.74657751557635
```

```
z0 = (100 - m ) / s
z0
```

```
0.7410947648018662
```

```
Pz0 = st.norm.cdf(z0)
```

```
t = 1 - Pz0
t
```

```
0.22931799182323287
```

```
ssp.comb(10, 4) * t**4 * (1-t)**6
```

```
0.12168110266158293
```

```
sum = 0
for i in range (4):
    sum += ssp.comb(10,i) * t**i * (1-t)**(10-i)
1 - sum
```

```
0.1779522510824021
```

```
sum = 0
for i in range (4,11):
    sum += ssp.comb(10,i) * t**i * (1-t)**(10-i)
```

```
sum
```

```
0.17795225108240206
```

```
1 - st.binom.cdf(3, n=10, p=t)
```

```
0.17795225108240198
```

```
sum123 = 0
for i in [1,2,3]:
    sum123 += ssp.comb(10,i) * t**i * (1-t)**(10-i)
sum123
```

```
0.7481294092068188
```

```
nu= st.binom.cdf(3, n=10, p=t) - st.binom.cdf(0, n=10, p=t)
```

```
deno = 1 - st.binom.cdf(0, n=10, p=t)
```

```
nu, deno
```

```
(0.7481294092068189, 0.9260816602892209)
```

```
nu / deno
```

```
0.807843888165514
```

```
1 - st.poisson.cdf(60, mu=50)
```

```
0.07216017981325695
```

```
1 - st.poisson.cdf(39, mu=50)
```

```
0.935429631078867
```

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search