Machine Learning

Lecture 2: Review of Linear Algebra and Python Libraries

Instructor: Dr. Farhad Pourkamali Anaraki

Norm and dot product for vectors

- Norm

```
import numpy as np
x = np.array([3,4])

def vector_norm(vector):
    squares = [element**2 for element in vector]
    return sum(squares)**0.5

print("||", x, "|| =")
vector_norm(x)
```

```
|| [3 4] || =
5.0
```

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

```
import numpy.linalg as LA
LA.norm(x)
```

```
5.0
```

- Dot product or inner product

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} x_i y_i$$

```
y = np.array([2,3])
```

```
np.dot(x,y)
```

```
18
```

```
x.dot(y)
```

```
18
```

Matrix multiplication

- Let us consider the following form

$$x_1 \begin{bmatrix} a_{11} \\ \vdots \\ a_{m1} \end{bmatrix} + \ldots + x_n \begin{bmatrix} a_{1n} \\ \vdots \\ a_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

```
A = np.array([
    [1,2,3],
    [4,5,6]])
A
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
x = np.array([2,4,5]).reshape(-1,1)
x
```

```
array([[2],
       [4],
       [5]])
```

```
np.matmul(A,x) # matrix multiplication
```

```
array([[25],
       [58]])
```

Matrix multiplication

- Another formula

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} a_{1i}x_i \\ \vdots \\ \sum_{i=1}^{n} a_{mi}x_i \end{bmatrix}$$

```
A = np.array([
    [1,2,3],
    [4,5,6]])
A
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
x = np.array([2,4,5]).reshape(-1,1)
x
```

```
array([[2],
       [4],
       [5]])
```

```
A.dot(x)
```

```
array([[25],
       [58]])
```

```
print(np.dot(A[0,:],x), np.dot(A[1,:],x))
```

```
[25] [58]
```

NumPy documentation

numpy.dot

numpy.**dot**(*a*, *b*, *out=None*)

Dot product of two arrays. Specifically,

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using matmul or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to multiply and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where `M>=2`), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

Matrix multiplication

- Matrices can be multiplied if their neighboring dimensions match

$$\underbrace{\mathbf{A}}_{n\times k}\ \underbrace{\mathbf{B}}_{k\times m} = \mathbf{C}$$

```
D = np.array([
        [ 2,  3,  5,  7],
        [11, 13, 17, 19],
        [23, 29, 31, 37]])
```

```
A = np.array([[3],[4],[5]])
```

```
try:
    D.dot(A)
except ValueError as e:
    print("ValueError:", e)
```

```
ValueError: shapes (3,4) and (3,1) not aligned: 4 (dim 1) != 3 (dim 0)
```

Matrix multiplication

```
F = np.array([
        [5,2],
        [4,1],
        [9,3]])
F
```

```
array([[5, 2],
       [4, 1],
       [9, 3]])
```

```
A = np.array([
    [10,20,30],
    [40,50,60]])
A
```

```
array([[10, 20, 30],
       [40, 50, 60]])
```

```
A.dot(F)
```

```
array([[400, 130],
       [940, 310]])
```

```
F.dot(A)
```

```
array([[130, 200, 270],
       [ 80, 130, 180],
       [210, 330, 450]])
```

Other operations

- Matrix transpose

```
A
```

```
array([[10, 20, 30],
       [40, 50, 60]])
```

```
A.T
```

```
array([[10, 40],
       [20, 50],
       [30, 60]])
```

- Matrix inverse

```
A = np.array([[1, 1.5],[0, 1]])
A
```

```
array([[1. , 1.5],
       [0. , 1. ]])
```

```
A_inv = LA.inv(A)
A_inv
```

```
array([[ 1. , -1.5],
       [ 0. ,  1. ]])
```

```
A.dot(A_inv)
```

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

Identity matrix

```
I = np.identity(3)
```

```
F = np.array([[2],[1],[3]])
F
```

```
array([[2],
       [1],
       [3]])
```

```
I.dot(F)
```

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

Pandas

Pandas

- The Pandas library provides high-performance and easy-to-use data structures
  - DataFrame: 2D table like a spreadsheet with column names and row labels
  - https://github.com/ageron/handson-ml2/blob/master/tools_pandas.ipynb
- Series: 1D array or a column in a spreadsheet

```
s = pd.Series([2,-1,3,5])
s
```

```
0    2
1   -1
2    3
3    5
dtype: int64
```

- Index label

```
s2 = pd.Series([68, 83, 112, 68], index=["alice", "bob", "charles", "darwin"])
s2
```

```
alice        68
bob          83
charles     112
darwin       68
dtype: int64
```

Series

- Finding elements

```
s2["bob"]
```

```
83
```

```
s2[1]
```

```
83
```

```
s2.loc["bob"] # accessing by label
```

```
83
```

```
s2.iloc[1] # accessing by integer location
```

```
83
```

- Create Series from a dictionary

```
temp = {"alice": 68, "bob": 83, "colin": 86, "darwin": 68}
pd.Series(temp)
```

```
alice     68
bob       83
colin     86
darwin    68
dtype: int64
```

Time range

```
dates = pd.date_range('2016/10/29 5:30pm', periods=12, freq='H')
dates
```

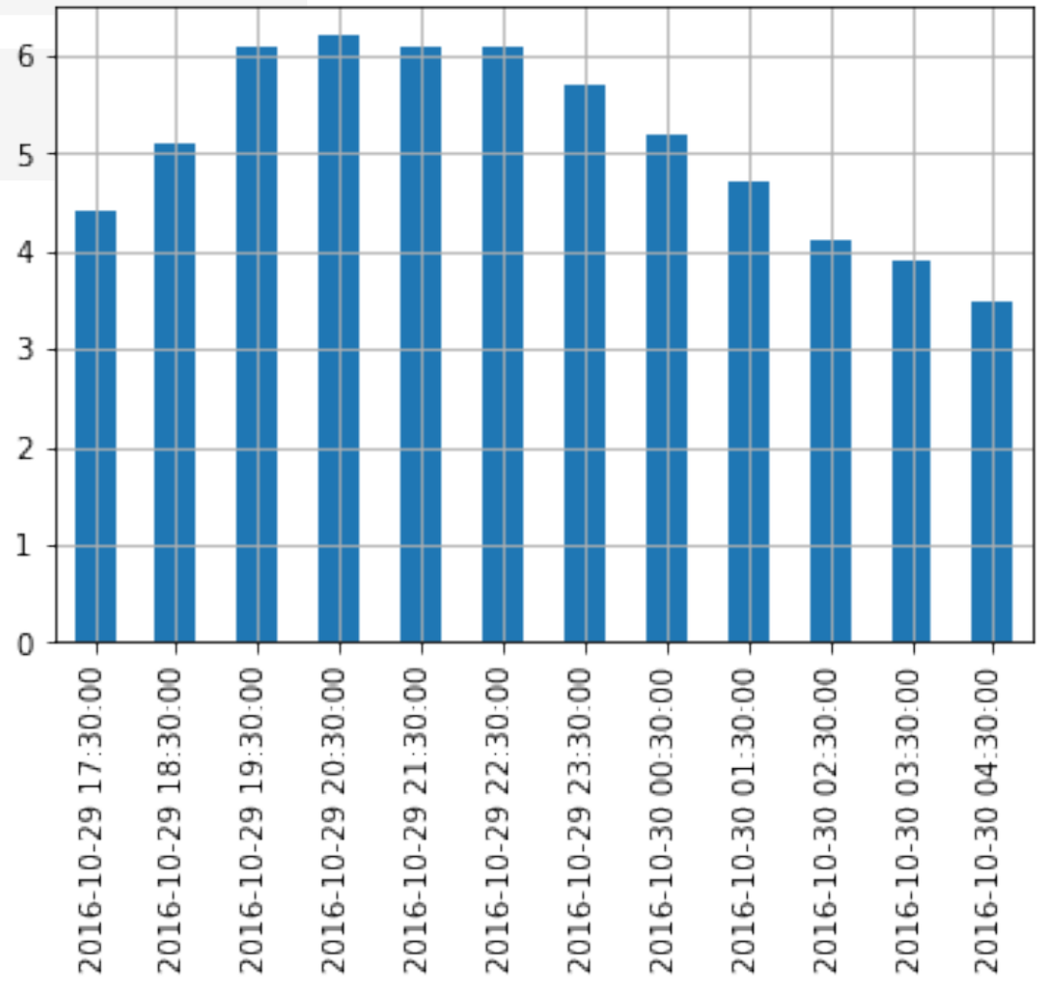
```
DatetimeIndex(['2016-10-29 17:30:00', '2016-10-29 18:30:00',
               '2016-10-29 19:30:00', '2016-10-29 20:30:00',
               '2016-10-29 21:30:00', '2016-10-29 22:30:00',
               '2016-10-29 23:30:00', '2016-10-30 00:30:00',
               '2016-10-30 01:30:00', '2016-10-30 02:30:00',
               '2016-10-30 03:30:00', '2016-10-30 04:30:00'],
              dtype='datetime64[ns]', freq='H')
```

```
temperatures = [4.4,5.1,6.1,6.2,6.1,6.1,5.7,5.2,4.7,4.1,3.9,3.5]
```

```
temp_series = pd.Series(temperatures, dates)
temp_series
```

```
temp_series = pd.Series(temperatures, dates)
temp_series
```

```
import matplotlib.pyplot as plt
temp_series.plot(kind="bar")

plt.grid(True)
plt.show()
```



DataFrame

```
people_dict = {
    "weight": pd.Series([68, 83, 112], index=["alice", "bob", "charles"]),
    "birthyear": pd.Series([1984, 1985, 1992], index=["bob", "alice", "charles"]),
    "children": pd.Series([0, 3], index=["charles", "bob"]),
    "hobby": pd.Series(["Biking", "Dancing"], index=["alice", "bob"])}
people = pd.DataFrame(people_dict)
```

	weight	birthyear	children	hobby
alice	68	1985	NaN	Biking
bob	83	1984	3.0	Dancing
charles	112	1992	0.0	NaN

missing values

How to access columns?

```
people[["birthyear", "hobby"]]
```

	birthyear	hobby
alice	1985	Biking
bob	1984	Dancing
charles	1992	NaN

Another way to create DataFrame

```
values = [[1985, np.nan, "Biking",   68],
          [1984, 3,      "Dancing",  83],
          [1992, 0,      np.nan,    112]]
d3 = pd.DataFrame(
        values,
        columns=["birthyear", "children", "hobby", "weight"],
        index=["alice", "bob", "charles"]
     )
d3
```

	birthyear	children	hobby	weight
alice	1985	NaN	Biking	68
bob	1984	3.0	Dancing	83
charles	1992	0.0	NaN	112

Another way to create DataFrame

```
people = pd.DataFrame({
    "birthyear": {"alice":1985, "bob": 1984, "charles": 1992},
    "hobby": {"alice":"Biking", "bob": "Dancing"},
    "weight": {"alice":68, "bob": 83, "charles": 112},
    "children": {"bob": 3, "charles": 0}
})
people
```

	birthyear	hobby	weight	children
alice	1985	Biking	68	NaN
bob	1984	Dancing	83	3.0
charles	1992	NaN	112	0.0

Accessing rows

```
people
```

	birthyear	hobby	weight	children
alice	1985	Biking	68	NaN
bob	1984	Dancing	83	3.0
charles	1992	NaN	112	0.0

```
people[people["birthyear"] < 1990]
```

	birthyear	hobby	weight	children
alice	1985	Biking	68	NaN
bob	1984	Dancing	83	3.0

Adding/removing columns

- Adding column

```
people["age"] = 2020 - people["birthyear"]  # adds a new column "age"
```

```
people
```

	birthyear	hobby	weight	children	age
alice	1985	Biking	68	NaN	35
bob	1984	Dancing	83	3.0	36
charles	1992	NaN	112	0.0	28

- Removing column

```
birthyears = people.pop("birthyear")
```

```
people
```

	hobby	weight	children	age
alice	Biking	68	NaN	35
bob	Dancing	83	3.0	36
charles	NaN	112	0.0	28

Handling missing data

- Replace all NaNs by any value

```
bonus_array = np.array([[0,np.nan,2],[np.nan,1,0],[0, 1, 0], [3, 3, 0]])
bonus_points = pd.DataFrame(bonus_array, columns=["oct", "nov", "dec"],
                            index=["bob","colin", "darwin", "charles"])
bonus_points
```

	oct	nov	dec
bob	0.0	NaN	2.0
colin	NaN	1.0	0.0
darwin	0.0	1.0	0.0
charles	3.0	3.0	0.0

```
bonus_points.fillna(-1)
```

	oct	nov	dec
bob	0.0	-1.0	2.0
colin	-1.0	1.0	0.0
darwin	0.0	1.0	0.0
charles	3.0	3.0	0.0

Handling missing data

- Interpolation technique to fill in missing values

```
bonus_points
```

	oct	nov	dec
bob	0.0	NaN	2.0
colin	NaN	1.0	0.0
darwin	0.0	1.0	0.0
charles	3.0	3.0	0.0

```
bonus_points.interpolate(axis=0)
```

	oct	nov	dec
bob	0.0	NaN	2.0
colin	0.0	1.0	0.0
darwin	0.0	1.0	0.0
charles	3.0	3.0	0.0

```
bonus_points.interpolate(axis=1)
```

	oct	nov	dec
bob	0.0	1.0	2.0
colin	NaN	1.0	0.0
darwin	0.0	1.0	0.0
charles	3.0	3.0	0.0

Saving

- We usually save DataFrame to CSV

```
my_df = pd.DataFrame(
    [["Biking", 68.5, 1985, np.nan], ["Dancing", 83.1, 1984, 3]],
    columns=["hobby","weight","birthyear","children"],
    index=["alice", "bob"]
)
my_df
```

	hobby	weight	birthyear	children
alice	Biking	68.5	1985	NaN
bob	Dancing	83.1	1984	3.0

```
my_df.to_csv("my_df.csv")
```

```
!ls
```

```
my_df.csv  sample_data
```

```
!head my_df.csv
```

```
,hobby,weight,birthyear,children
alice,Biking,68.5,1985,
bob,Dancing,83.1,1984,3.0
```

Loading

- From saved CSV file

```
my_df_loaded = pd.read_csv("my_df.csv", index_ col=0)
my_df_loaded
```

	hobby	weight	birthyear	children
alice	Biking	68.5	1985	NaN
bob	Dancing	83.1	1984	3.0

- How can you upload your CSV file?

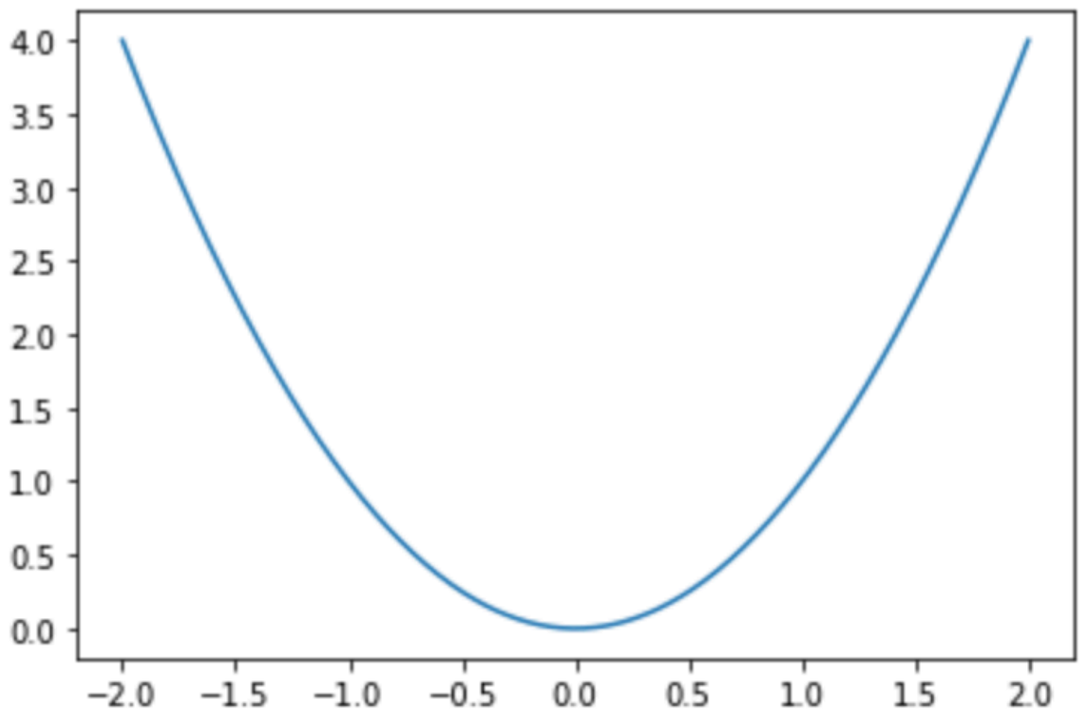
```
from google.colab import files
uploaded = files.upload()
```

Matplotlib

Matplotlib

- Comprehensive library for creating visualizations in Python
  - https://github.com/ageron/handson-ml2/blob/master/tools_matplotlib.ipynb

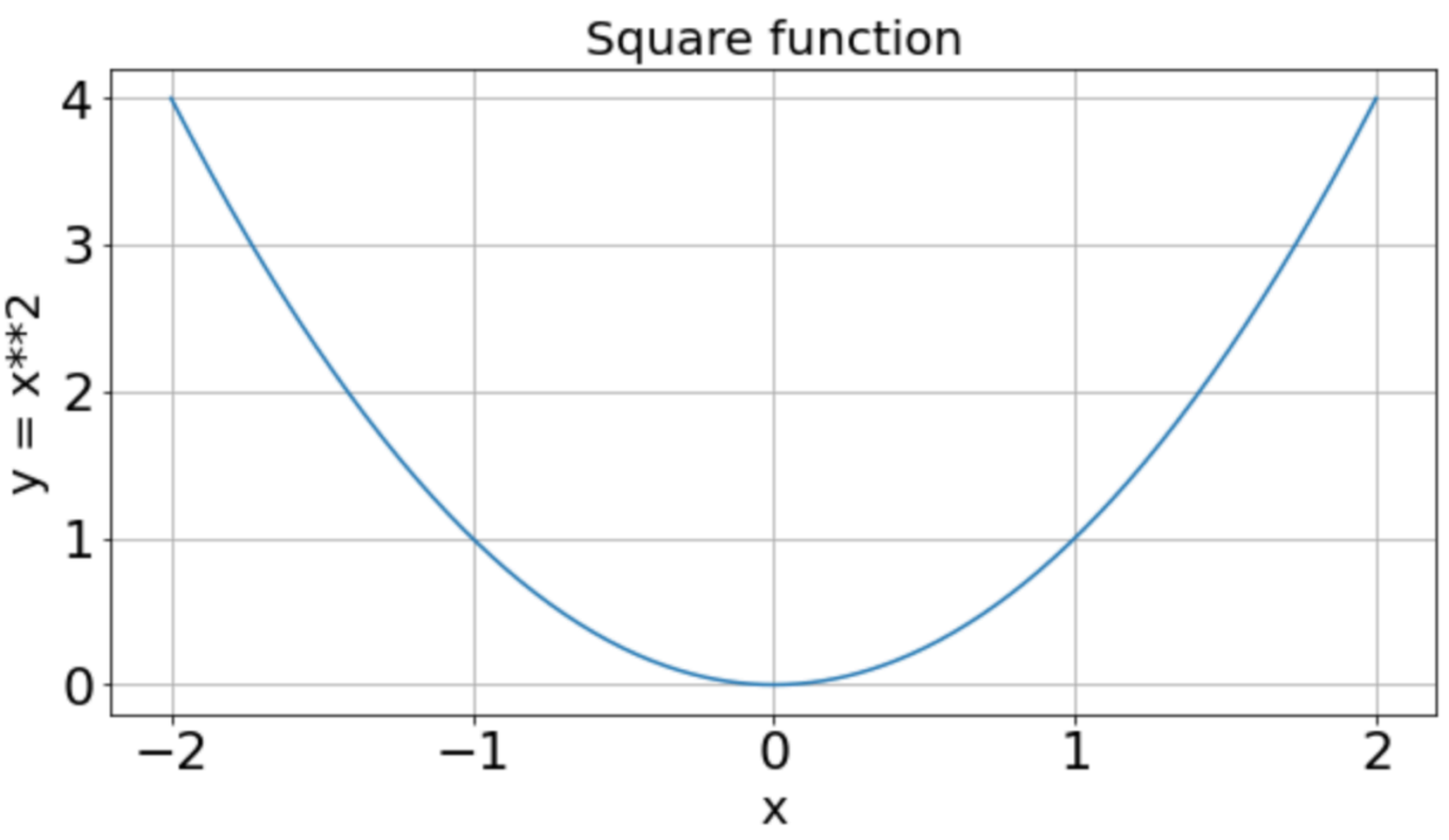
```
%matplotlib inline
import matplotlib.pyplot as plt

x = np.linspace(-2, 2, 500)
y = x**2

plt.plot(x, y)
plt.show()
```



Customization

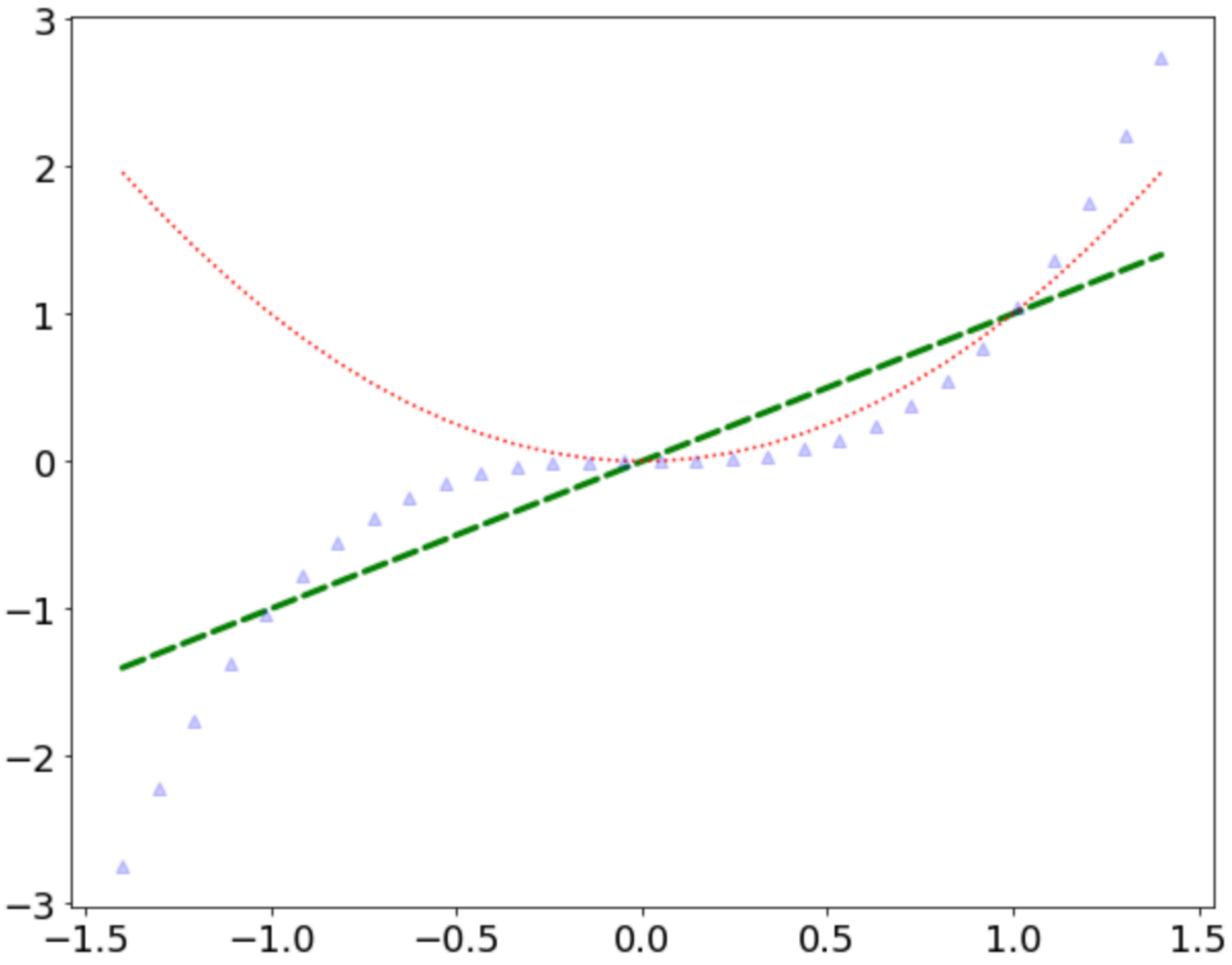
- We can add title, axes labels, etc.

```
plt.rcParams['figure.figsize'] = (10, 5)
plt.rcParams.update({'font.size': 22})
plt.plot(x, y)
plt.title("Square function", fontsize=20)
plt.xlabel("x", fontsize=20)
plt.ylabel("y = x**2", fontsize=20)
plt.grid(True)
plt.show()
```



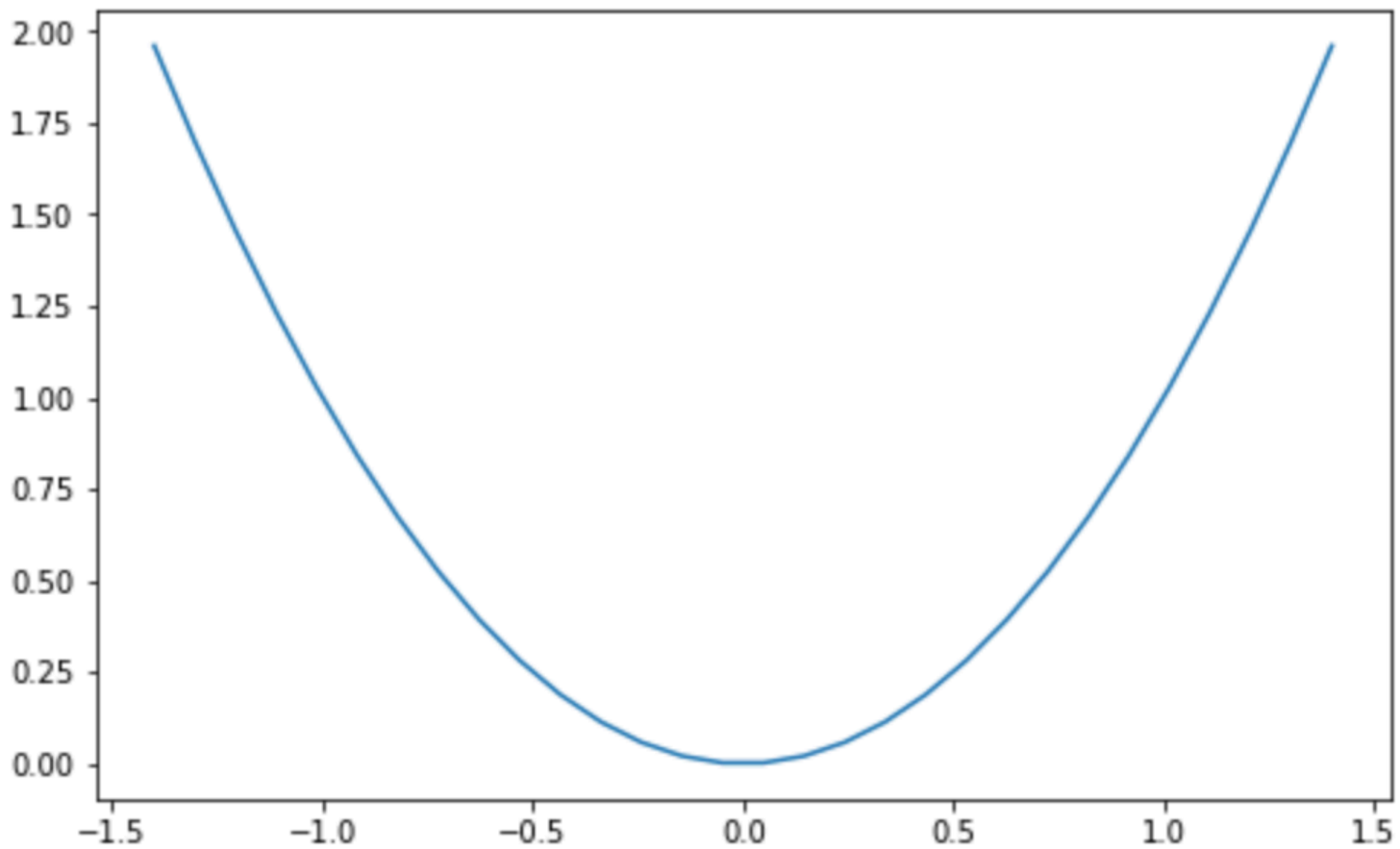
Customization

```
plt.rcParams['figure.figsize'] = (10, 8)
plt.rcParams.update({'font.size': 18})
x = np.linspace(-1.4, 1.4, 30)
line1, line2, line3 = plt.plot(x, x, 'g--', x, x**2, 'r:', x, x**3, 'b^')
line1.set_linewidth(3.0)
line1.set_dash_capstyle("round")
line3.set_alpha(0.2)
plt.show()
```



Saving a figure

```
plt.rcParams['figure.figsize'] = (8, 5)
plt.rcParams.update({'font.size': 10})
x = np.linspace(-1.4, 1.4, 30)
plt.plot(x, x**2)
plt.savefig("my_square_function.png", transparent=True)
```

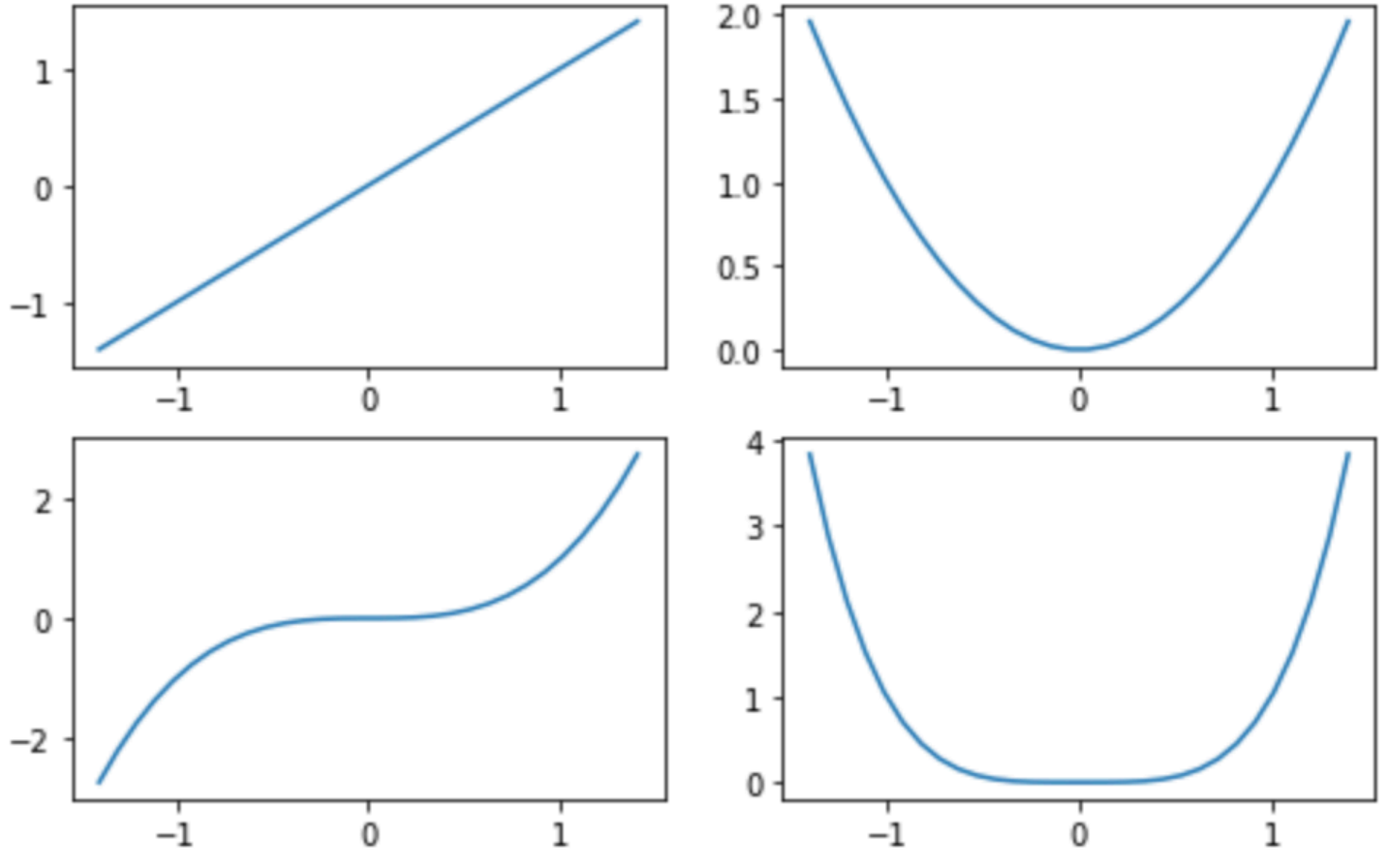



```
!ls
```

```
my_df.csv  my_square_function.png  sample_data
```

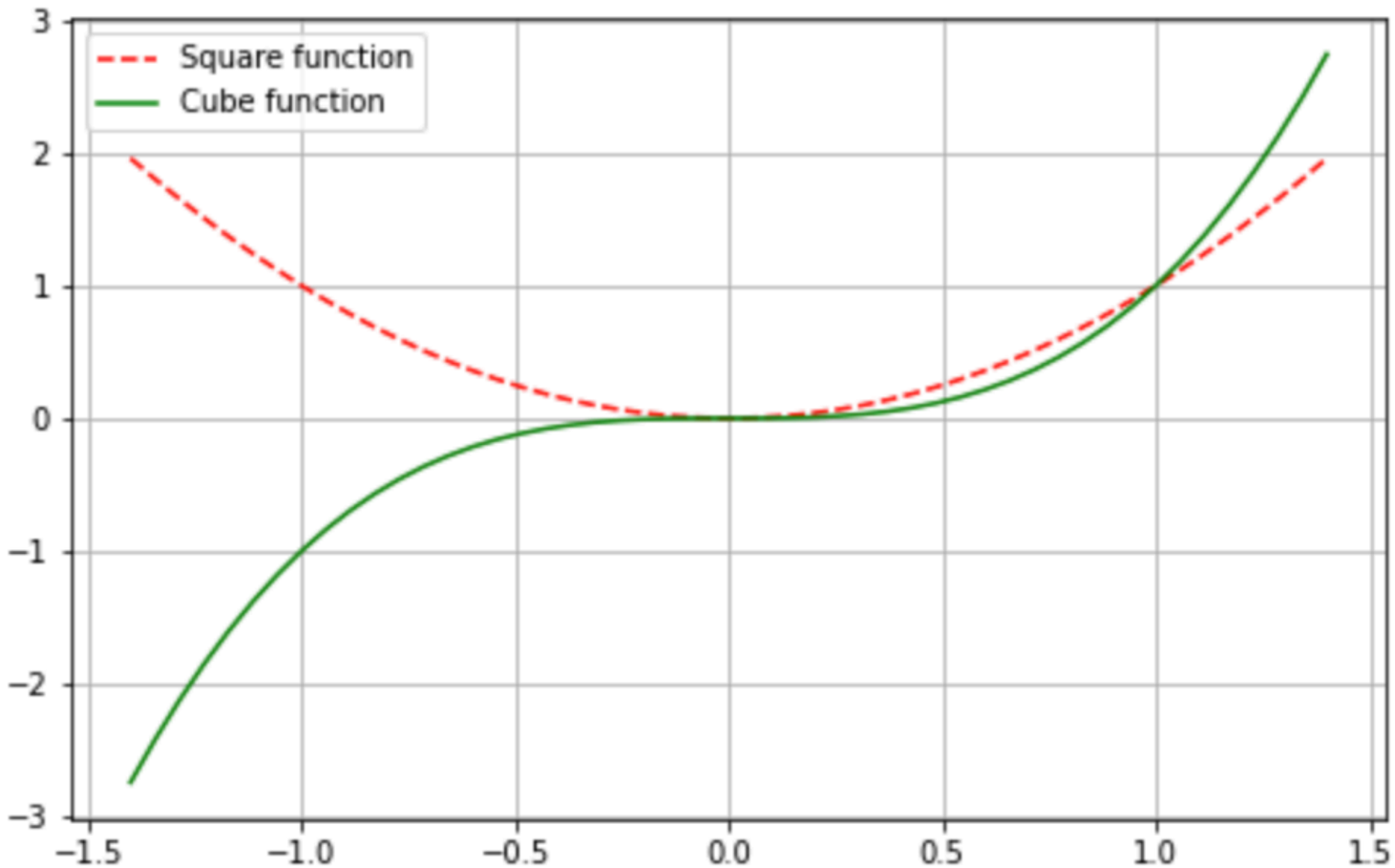
Subplots

```
x = np.linspace(-1.4, 1.4, 30)
plt.subplot(2, 2, 1)  # 2 rows, 2 columns, 1st subplot = top left
plt.plot(x, x)
plt.subplot(2, 2, 2)  # 2 rows, 2 columns, 2nd subplot = top right
plt.plot(x, x**2)
plt.subplot(2, 2, 3)  # 2 rows, 2 columns, 3rd subplot = bottow left
plt.plot(x, x**3)
plt.subplot(2, 2, 4)  # 2 rows, 2 columns, 4th subplot = bottom right
plt.plot(x, x**4)
plt.show()
```



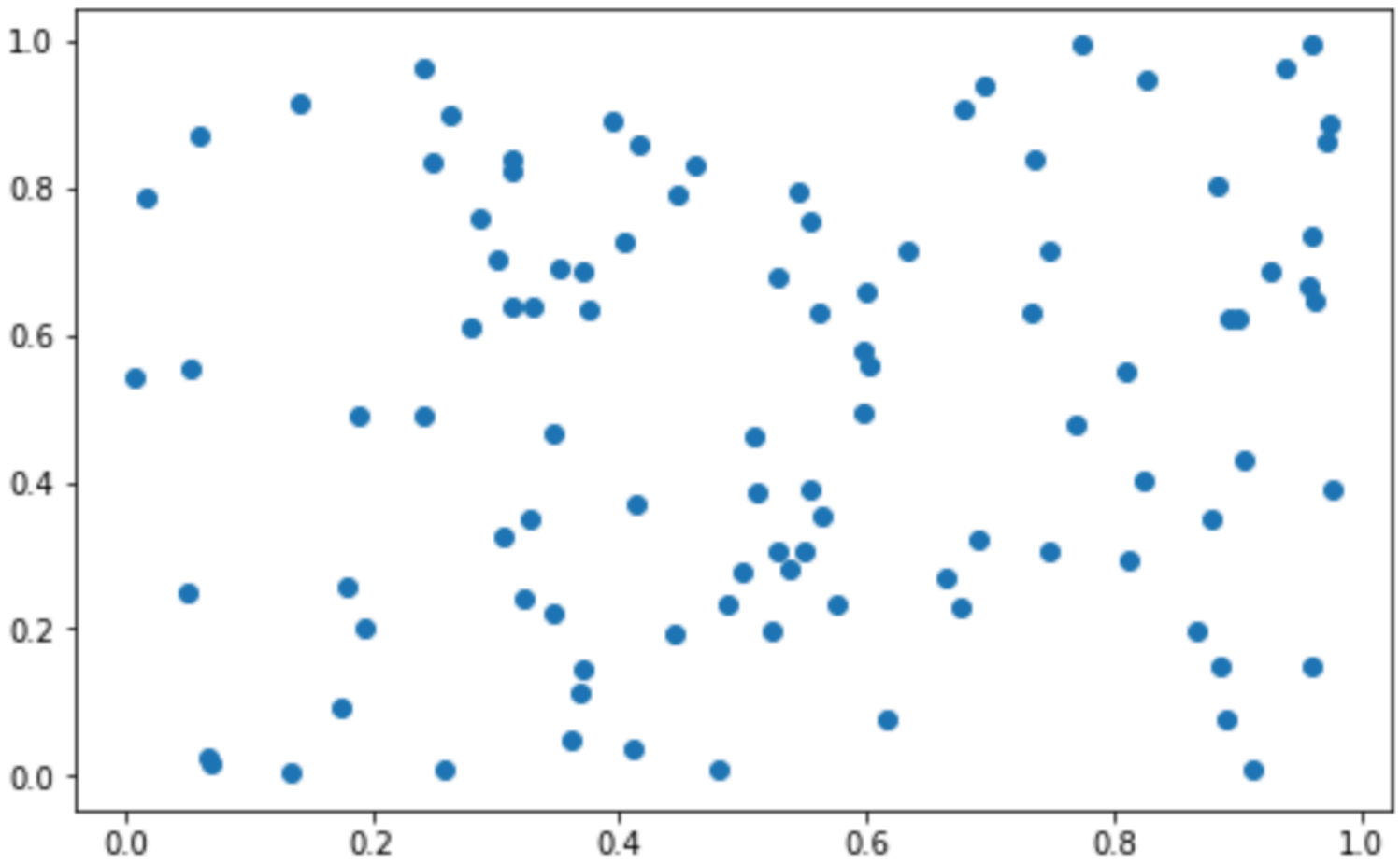
Legends

```
x = np.linspace(-1.4, 1.4, 50)
plt.plot(x, x**2, "r--", label="Square function")
plt.plot(x, x**3, "g-", label="Cube function")
plt.legend(loc="best")
plt.grid(True)
plt.show()
```



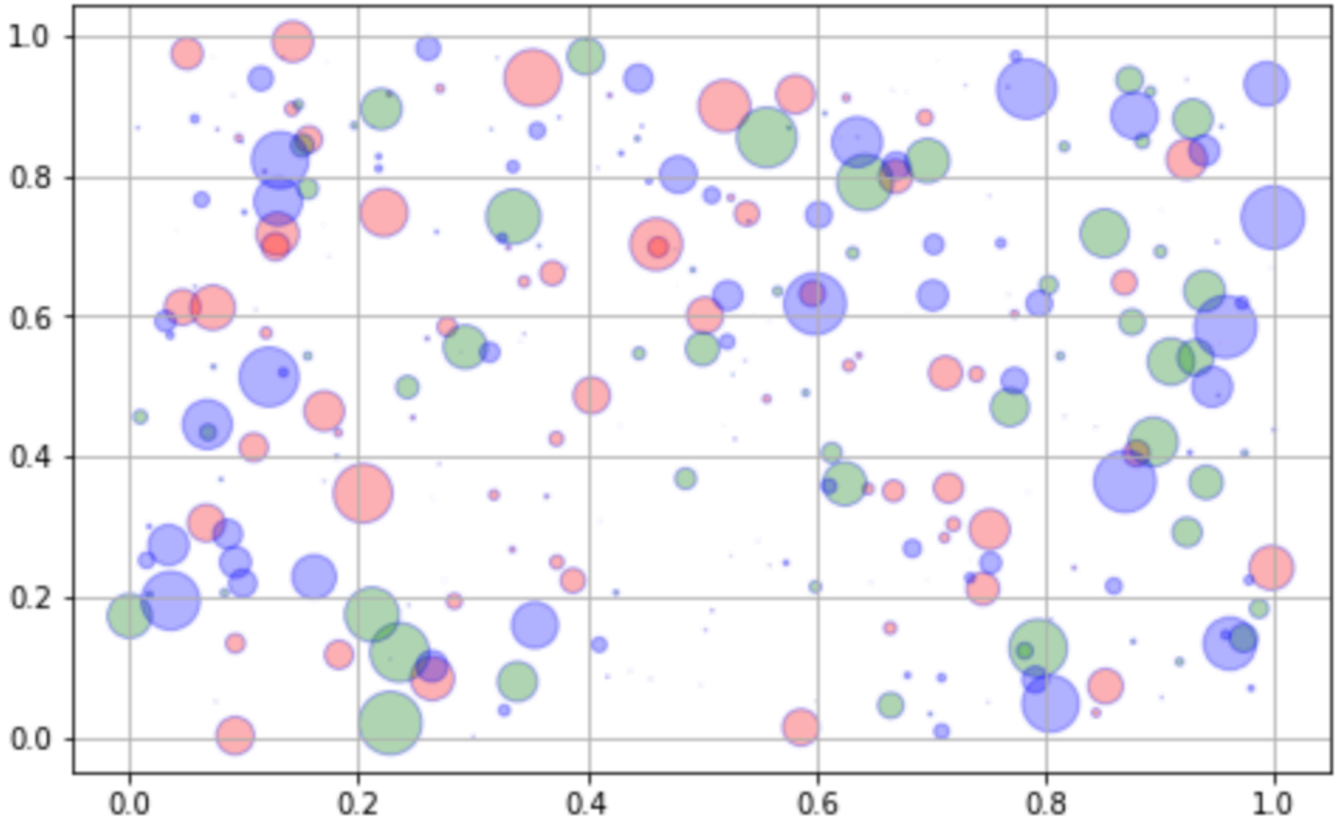
Scatter plot

```
from numpy.random import rand
x, y = rand(2, 100)
plt.scatter(x, y)
plt.show()
```



Scatter plot

```
for color in ['red', 'green', 'blue']:
    n = 100
    x, y = rand(2, n)
    scale = 500.0 * rand(n) ** 5
    plt.scatter(x, y, s=scale, c=color, alpha=0.3, edgecolors='blue')

plt.grid(True)

plt.show()
```



Utility function for plots

- It is often convenient to create a utility function

```
from numpy.random import randn

def plot_line(axis, slope, intercept, **kargs):
    xmin, xmax = axis.get_xlim()
    plt.plot([xmin, xmax], [xmin*slope+intercept, xmax*slope+intercept], **kargs)

x = randn(1000)
y = 0.5*x + 5 + randn(1000)*2
plt.axis([-2.5, 2.5, -5, 15])
plt.scatter(x, y, alpha=0.2)
plot_line(axis=plt.gca(), slope=0.5, intercept=5, color="magenta")
plt.grid(True)
plt.show()
```