Projet Analyse De Donnée

L3

Hossein Khani

Content:

- Why Data Analyses
- Data Manipulation (Pandas Library)
- Data Visualisation (Matplotlib,Pyplot, Seaborn)
- Linear Regression
- Principle Component Analysis
- Non-Negative Matrix Factorization
- Orthogonal Matching pursuit

NEEDS:

- Basic Python Skills (Lists, Dictionaries, Functions, methods,….)
- Working with DataFrames (Data Cleaning and manipulation with Pandas Library)
- Working with Matrices (Numpy Library)
- Mathematics behind Machine learning Techniques (Mostly probability and statistics)
- Machine learning library (Scipy or Sklearn)

QUESTION:

Input:

longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

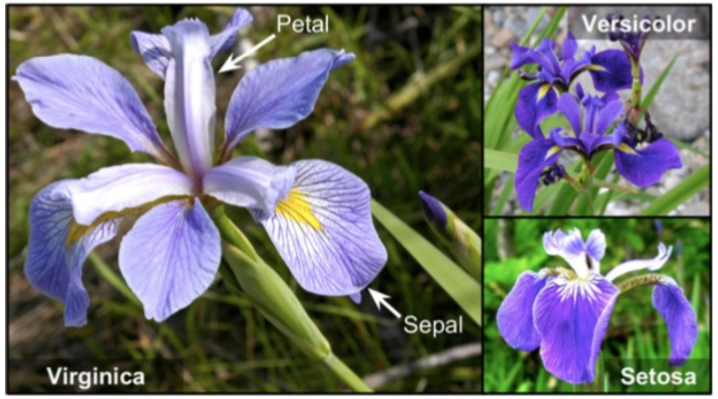
Output:

-121.32	39.43	18.0	1860.0	409.0	741.0	349.0	1.8672	??	INLAND

REGRESSION

QUESTION:

Input:



species	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	...	texture55	texture56	texture57	texture58
Acer_Opalus	0.007812	0.023438	0.023438	0.003906	0.011719	0.009766	0.027344	0.0	...	0.007812	0.000000	0.002930	0.002930
Pterocarya_Stenoptera	0.005859	0.000000	0.031250	0.015625	0.025391	0.001953	0.019531	0.0	...	0.000977	0.000000	0.000000	0.000977
Quercus_Hartwissiana	0.005859	0.009766	0.019531	0.007812	0.003906	0.005859	0.068359	0.0	...	0.154300	0.000000	0.005859	0.000977

Output:

??	0.000000	0.003906	0.023438	0.005859	0.021484	0.019531	0.023438	0.0	...	0.000000	0.000977	0.000000	0.000000

CLASSIFICATION

Install Python

- https://www.python.org/
- pip Python package management system : **python3 -m pip --version**
- install jupyter notebook: **python3 -m pip install -U jupyter**
- Install pandas: **pip install pandas**

- The Jupyter Notebook is the original web application for creating and sharing computational documents.
- Pandas the main tool of data analyse
- Pandas permits us to import data from various sources for example (CSV), and manipute them.

DataFrames:

- **https://insights.stackoverflow.com/survey**

How to use Pandas to work with DataFrame……

1. How to read data from csv file,
2. Take a look at the datafram,
3. Where dataframe comes from, its equivalent in python
4. Series objects and accessing multi-columns
5. Indexing
6. Accessing rows in DataFrames
7. Setting index for data frame
8. Changing columns’ names
9. Changing single row’s values

Numpy:

As a Data Analyst how to collect data?

List?

- Collection of values
- Hold different types
- Change, add, remove

What we need more?

- Mathematical operations over collections
- Speed

Numpy:

Body mass Index:

```
Height = [1.73, 1.68 , 1.71 , 1.89 , 1.79]
Weight = [65.4 , 59.2 , 63.6 , 88.4 , 68.7]
```

```
Weight / Height ** 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-43-0f6f8ba4f85f> in <module>
----> 1 Weight / Height ** 2

TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

To Solve: Looping over elements? → Not fast and efficient

Numpy (numeric python):

Solution?

nympy arrays:

- Alternative to python lists
- Calculations over entire arrays
- Easy and Fast

To Install:

pip3 install numpy

Numpy

```
import numpy as np

np_height = np.array(Height)
np_height
```

```
array([1.73, 1.68, 1.71, 1.89, 1.79])
```

```
np_weight = np.array(Weight)
np_weight
```

```
array([65.4, 59.2, 63.6, 88.4, 68.7])
```

```
bmi = np_weight / np_height**2
bmi
```

```
array([21.85171573, 20.97505669, 21.75028214, 24.7473475 , 21.44127836])
```

Numpy

Python is able to treat numpy arrays as single elements.

Where the speed comes from?

Numpy arrays collect values of the same type:

- Either integer
- Either float
- String
- …..

Numpy (Remarks)

```
np.array([1.0 , "Hossein" , True])
```

```
array(['1.0', 'Hossein', 'True'], dtype='<U32')
```

- Nympy array, is a data type in python.
- It has its own methods.
- These methods might act differently on arrays compared to other types.

Numpy (Remarks)

Example:

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

```
python_list + python_list
```

```
[1, 2, 3, 1, 2, 3]
```

```
numpy_array+numpy_array
```

```
array([2, 4, 6])
```

Numpy (Subsetting)

Example:

```
bmi
```

```
array([21.85171573, 20.97505669, 21.75028214, 24.7473475 , 21.44127836])
```

```
bmi[2]
```

Referring to specific index

```
21.750282138093777
```

```
bmi > 21
```

Looking for specific values

```
array([ True, False,  True,  True,  True])
```

```
bmi[bmi<21]
```

```
array([20.97505669])
```

Numpy (2D)

```
type(np_height)
```

```
numpy.ndarray
```

```
np_2d = np.array([[1.73, 1.68 , 1.71 , 1.89 , 1.79],
                  [65.4 , 59.2 , 63.6 , 88.4 , 68.7]])
```

```
np_2d
```

```
array([[ 1.73,  1.68,  1.71,  1.89,  1.79],
       [65.4 , 59.2 , 63.6 , 88.4 , 68.7 ]])
```

```
np_2d.shape
```

```
(2, 5)
```

Numpy (2D)

```
np_2d = np.array([[1.73, 1.68 , 1.71 , 1.89 , 1.79],
                  [65.4 , 59.2 , 63.6 , 88.4 , 68.7]])
np_2d
```

```
array([[ 1.73,  1.68,  1.71,  1.89,  1.79],
       [65.4 , 59.2 , 63.6 , 88.4 , 68.7 ]])
```

```
np_2d[0]
```

```
array([1.73, 1.68, 1.71, 1.89, 1.79])
```

```
np_2d[0][2]
```

```
1.71
```

```
np_2d[0,2]
```

```
1.71
```

Two ways to select

```
np_2d[0,1:3]
```

```
array([1.68, 1.71])
```

Numpy (Basic Statistics)

```
np_2d = np.array([Height,
                  Weight])
np_2d
```

```
array([[ 1.73,  1.68,  1.71,  1.89,  1.79],
       [65.4 , 59.2 , 63.6 , 88.4 , 68.7 ]])
```

```
np.mean(np_2d[0,:])
```

```
1.7600000000000002
```

```
np.median(np_2d[0,:])
```

```
1.73
```

```
np.sum(np_2d[0,:])
```

```
8.8
```

Numpy (Data Generation)

```
height = np.round(np.random.normal(1.75,2.0, 5000),2)
weight = np.round(np.random.normal(10.32,15.0, 5000),2)
```

```
np_city = np.column_stack((height,weight))
```

```
np_city.shape
```

```
(5000, 2)
```

Numpy (Data Generation)

```
np.zeros([4,5],dtype = int)
```

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

```
np.ones([4,5], dtype = int)
```

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

```
np.full((2,3), 6, dtype = int)
```

```
array([[6, 6, 6],
       [6, 6, 6]])
```

Numpy (Dtype)

- Python types: int, float, bool,...
  Their size depends on the platform they are applied to...

- Dtypes: numpy numerical types are instances of dtype objects. The numpy types have fixed-sizes.

  np.int32, np.int64, np.bool8, np.float32, np.float64

```
z = np.zeros([2,3], dtype = np.bool8)
```

```
z
```

```
array([[False, False, False],
       [False, False, False]])
```

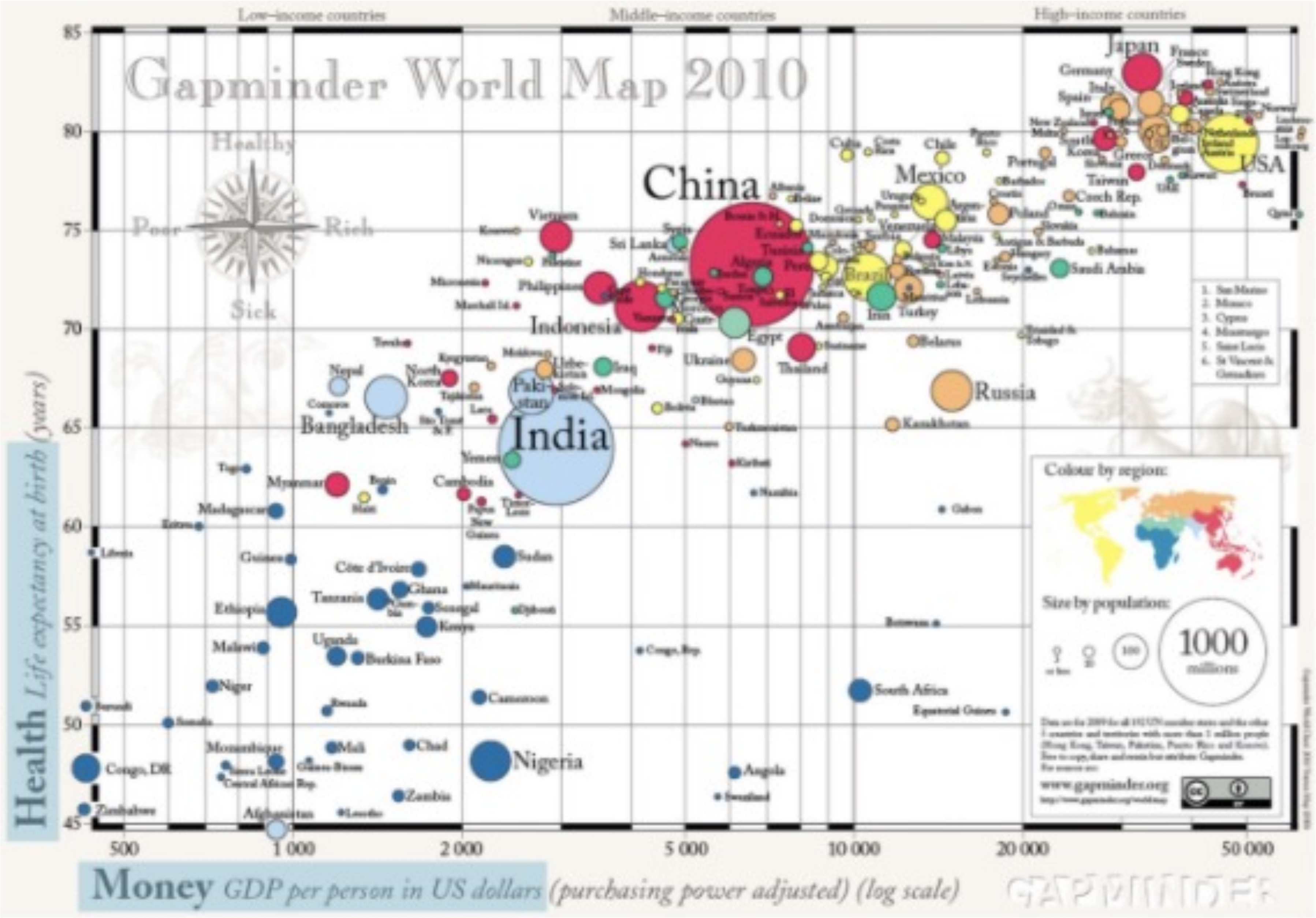
```
type(z)
```

```
numpy.ndarray
```

```
z.dtype
```

```
dtype('bool')
```

Data Visualisation:

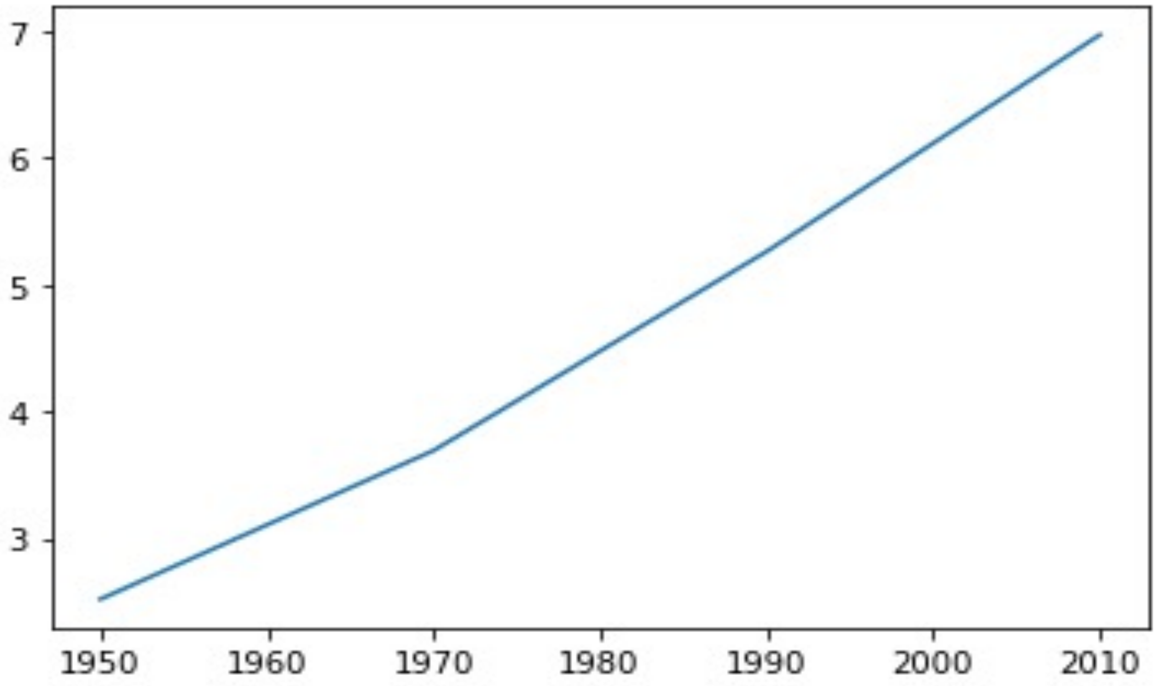


The most important visualization library : Matplotlib:

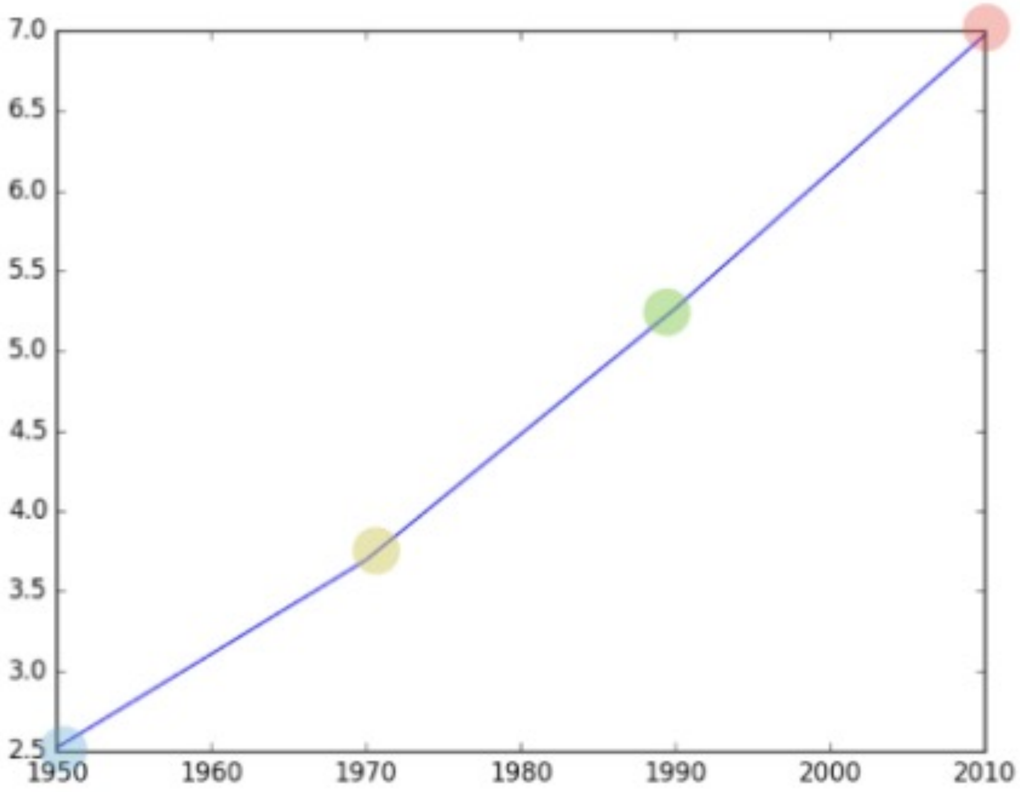
```python
import matplotlib.pyplot as plt
```

```python
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
```

```python
plt.plot(years , pop)
plt.show()
```



plt.plot() :

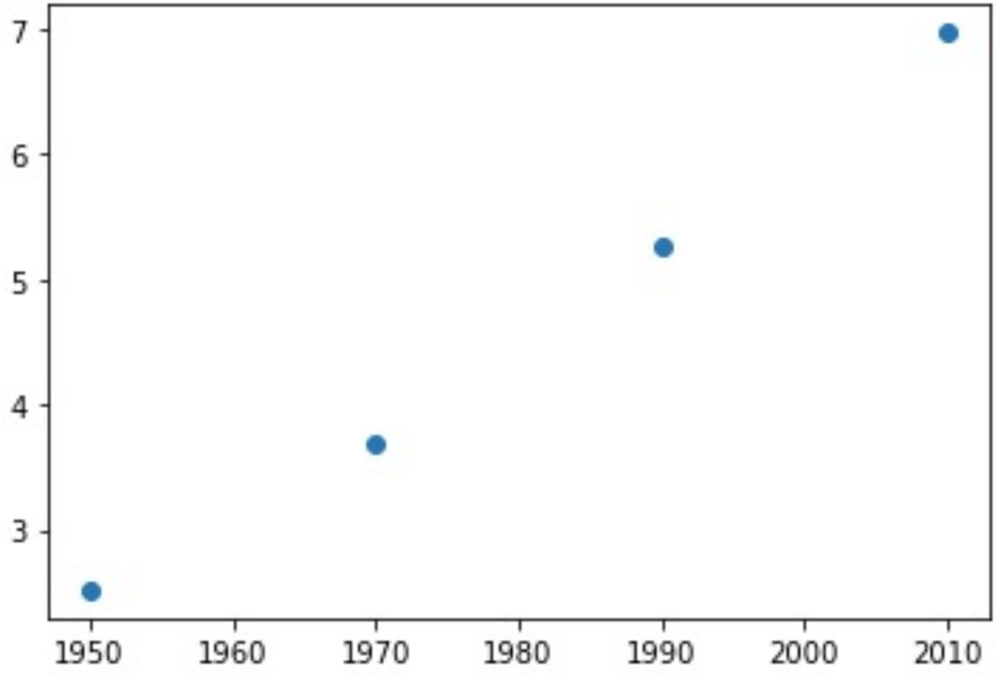



```
year = [1950 , 1970 , 1990 ,  2010]
pop  = [2.519, 3.692, 5.263, 6.972]
```

plt.scatter() :

```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
```

```
plt.scatter(years , pop)
plt.show()
```



Scatter plot is used when we need to measure the correlation between two attributes.

plt.scatter() :

```
np.corrcoef(years, pop)
```

```
array([[1.        , 0.99664316],
       [0.99664316, 1.        ]])
```

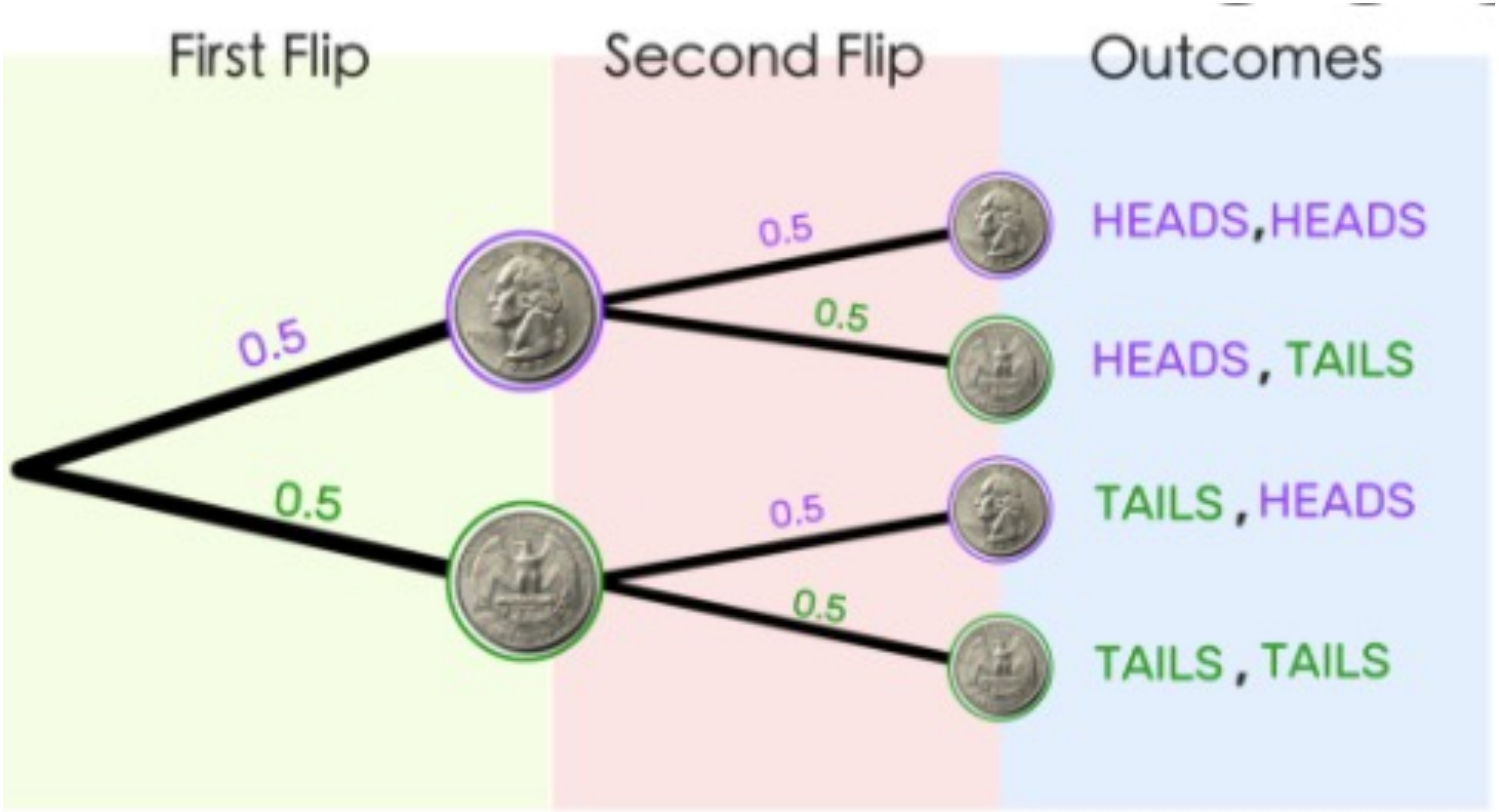
```
import scipy.stats  as st
st.pearsonr(years, pop)
```

```
(0.9966431630322238, 0.0033568369677620113)
```

Correlation

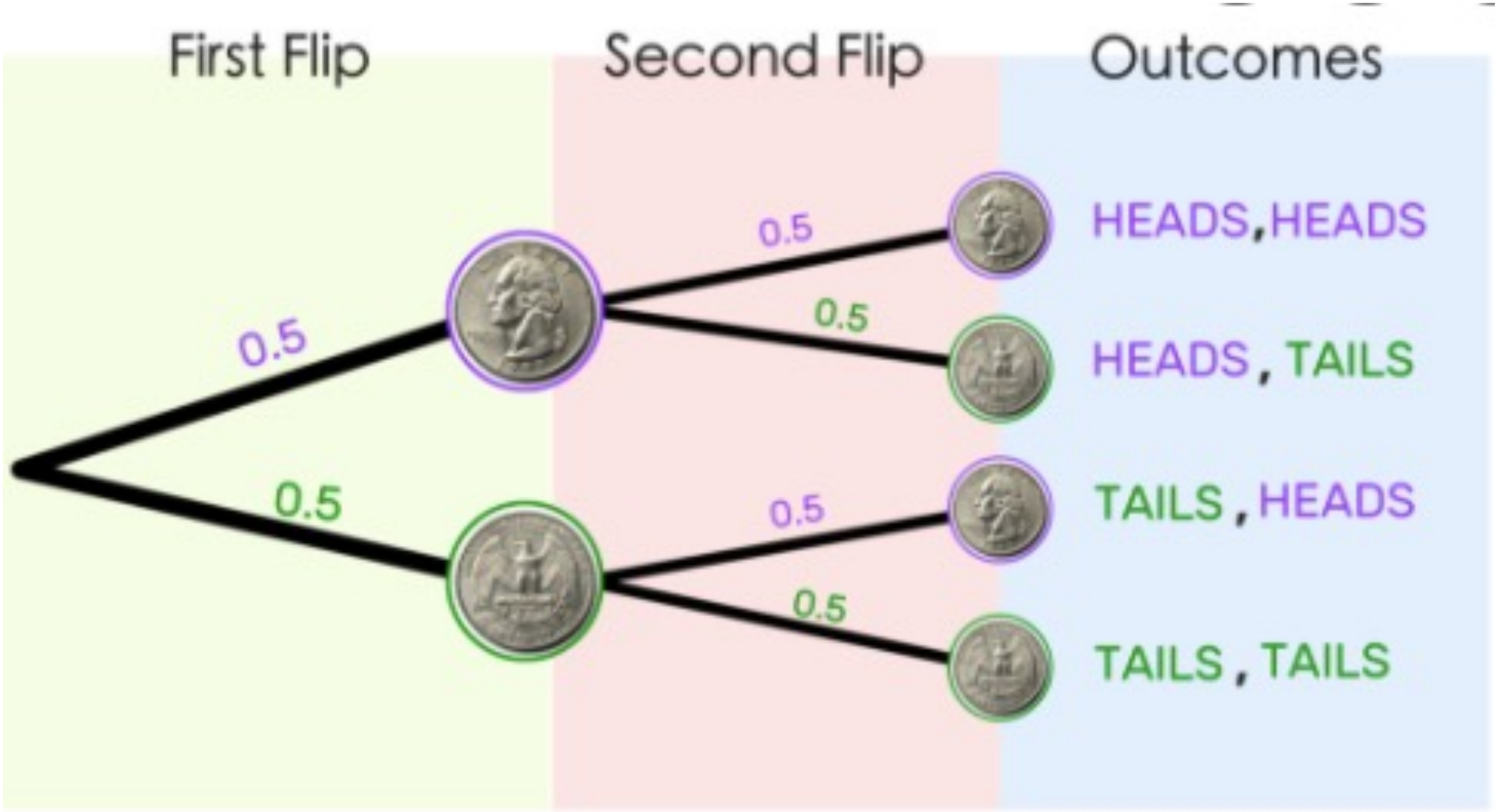
P-value

P-Value:



Probabilities?

P-Value:



$\frac{1}{4}$

$\frac{1}{2}$ one T one H

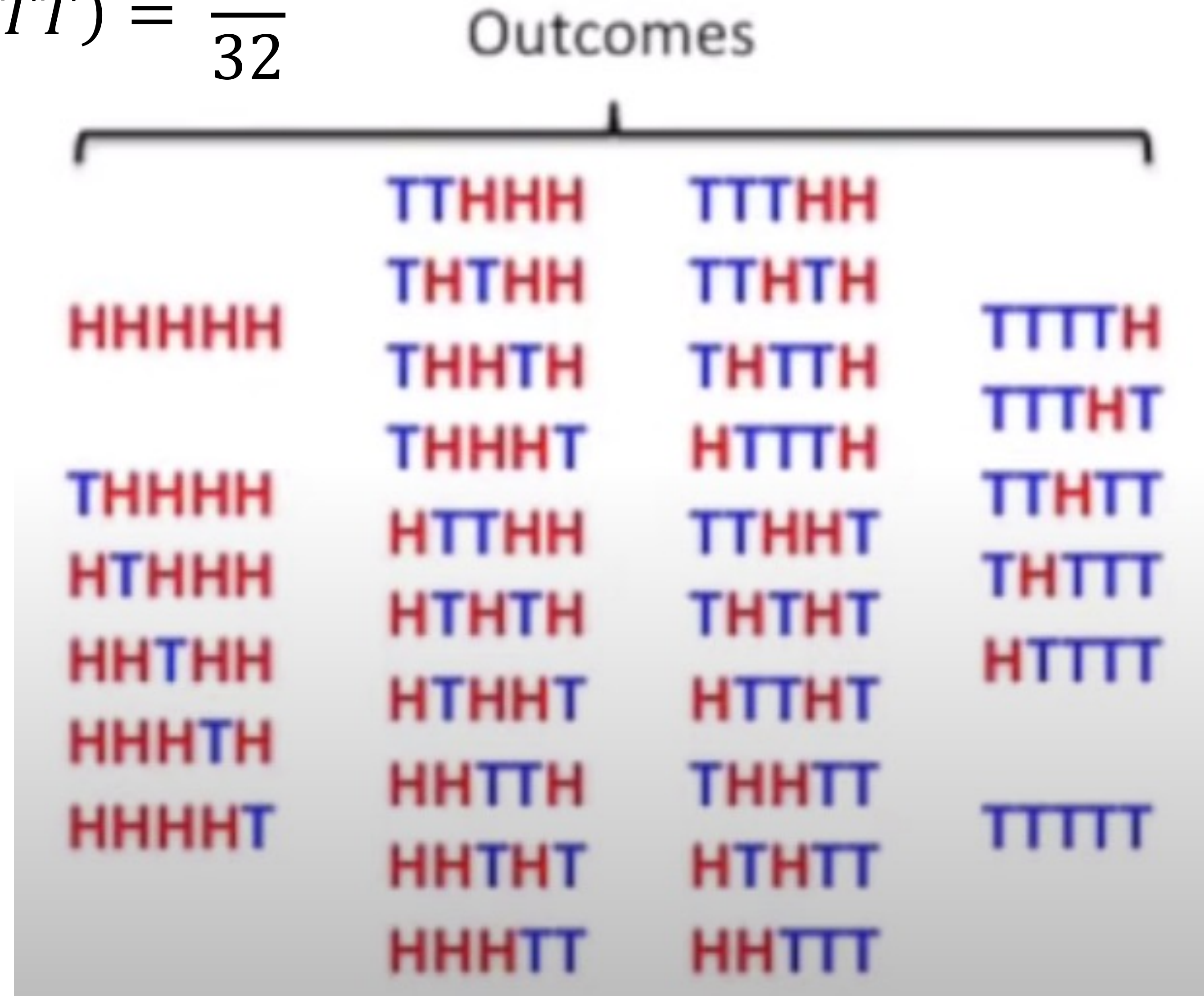
$\frac{1}{4}$

P-Value (flipping a coin 5 times):

$$p(HHHHH) = \frac{1}{32}$$

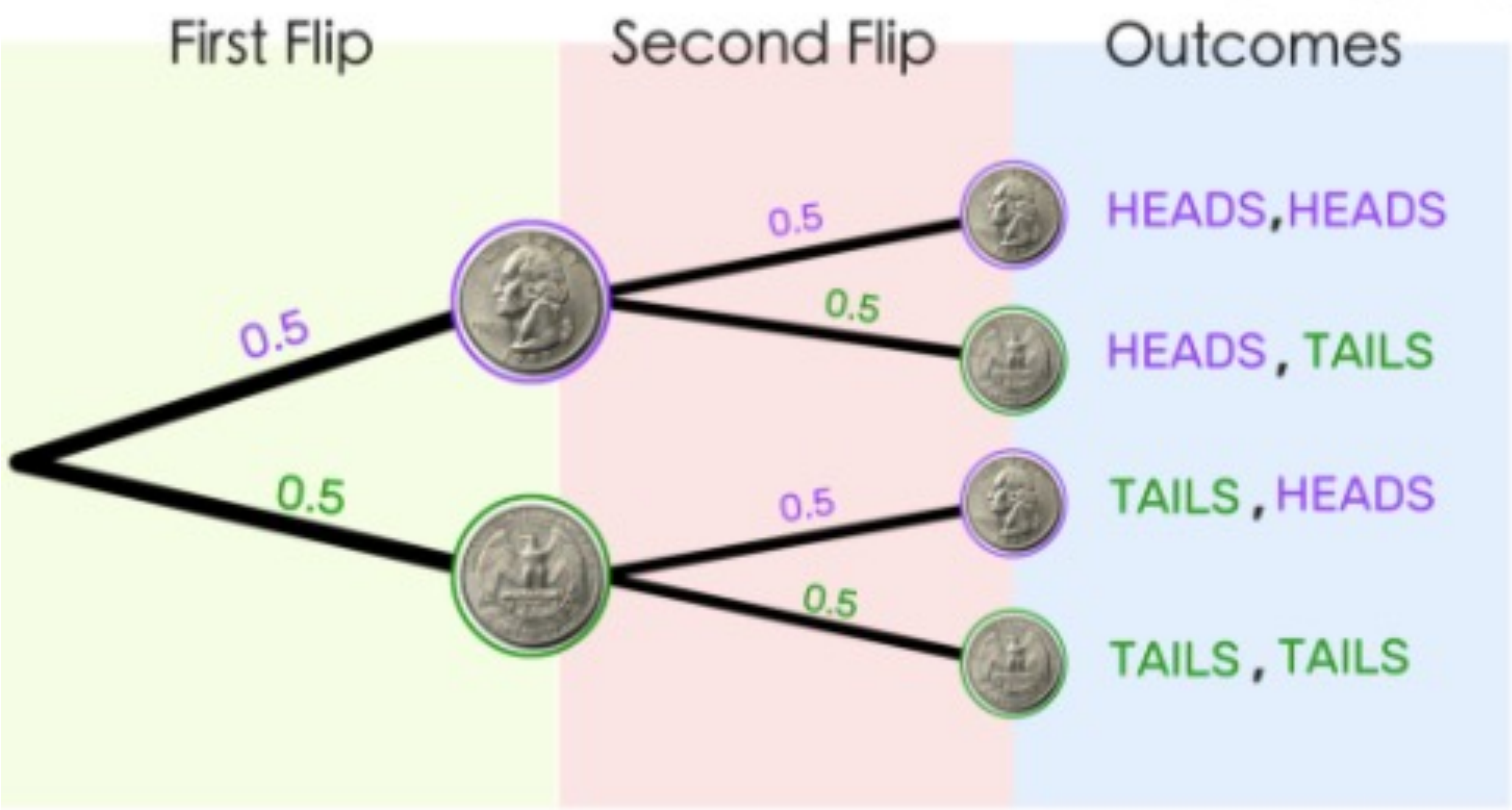
$$p(TTTTT) = \frac{1}{32}$$

$$p - value(HHHHH) = \frac{1}{32} + \frac{1}{32} = 0.0625$$



P-Value:

probability that random chance generated the data or somethig else that is equal or rarer.



P-values:

$$\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$$

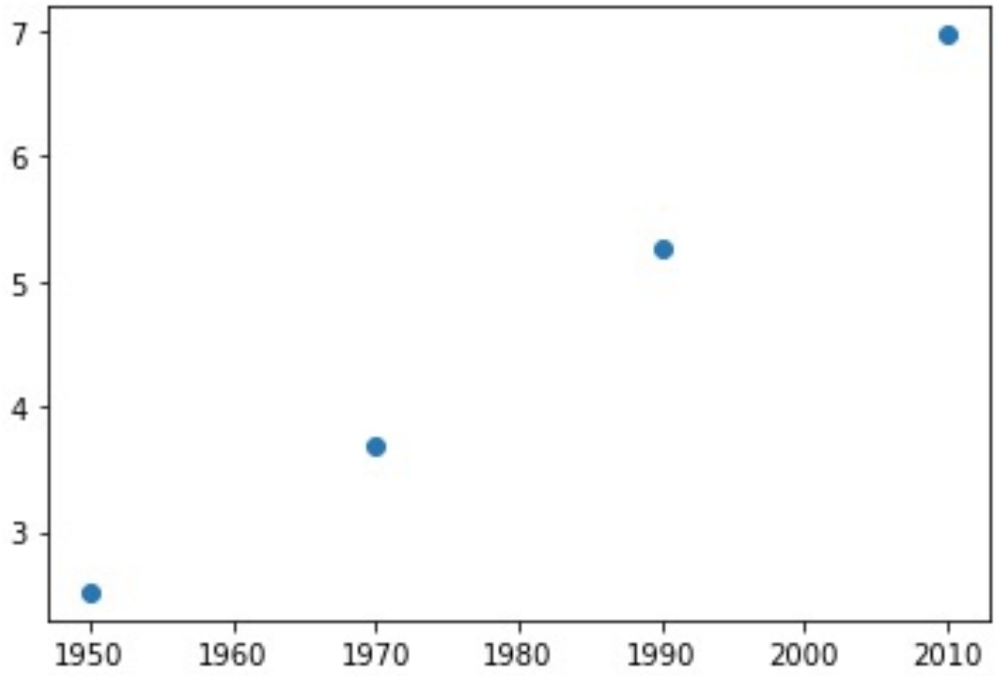
$$1$$

$$\frac{1}{2}$$

plt.scatter() :

```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
```

```
plt.scatter(years , pop)
plt.show()
```



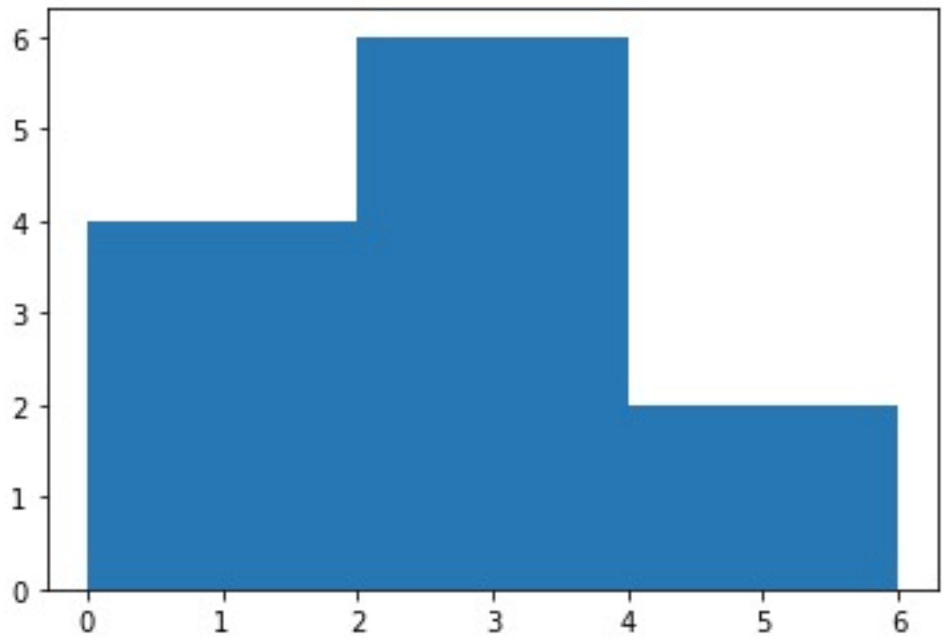
Scatter plot is used when we need to measure the correlation between two attributes.

plt.hist() :

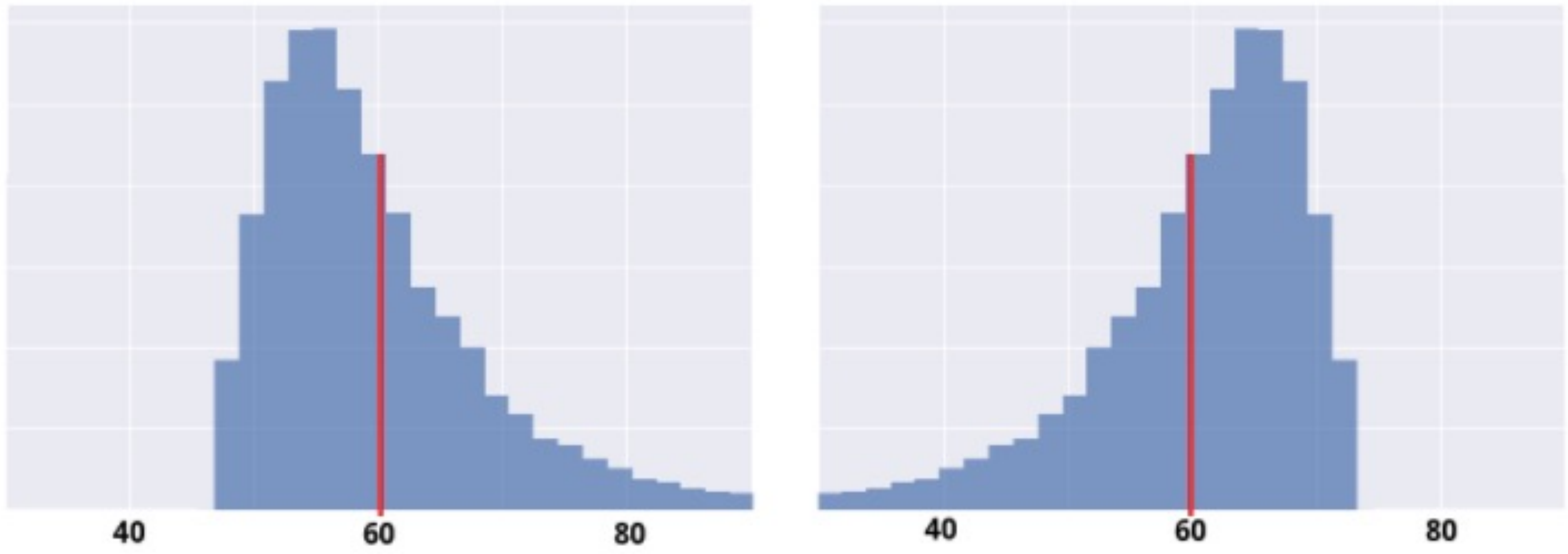
```
values = [0,0.6,1.4,1.6,2.2,2.5,2.6,3.2,3.5,3.9,4.2,6]
```

```
plt.hist(values , bins = 3)
```

```
(array([4., 6., 2.]),
 array([0., 2., 4., 6.]),
 <BarContainer object of 3 artists>)
```



plt.hist() :



Distribution of Data:

- Which is the most frequent data? statistics.mode()
- The data is centered around which point? Nupmy.mean()
- What is the value observed in 50% of the time? Numpy.median()
- How vary the values are ? np.std()

Distribution of Data:

Most of the time it takes 80 mins

Half of the times it takes 80 mins

On average it takes 80 mins

How long does it take
to go from
City A to city B

Distribution of Data:

```
values
```

[0, 0.6, 1.4, 1.6, 2.2, 2.5, 2.6, 3.2, 3.5, 3.9, 4.2, 6]

```
import numpy as np
np.mean(values)
```

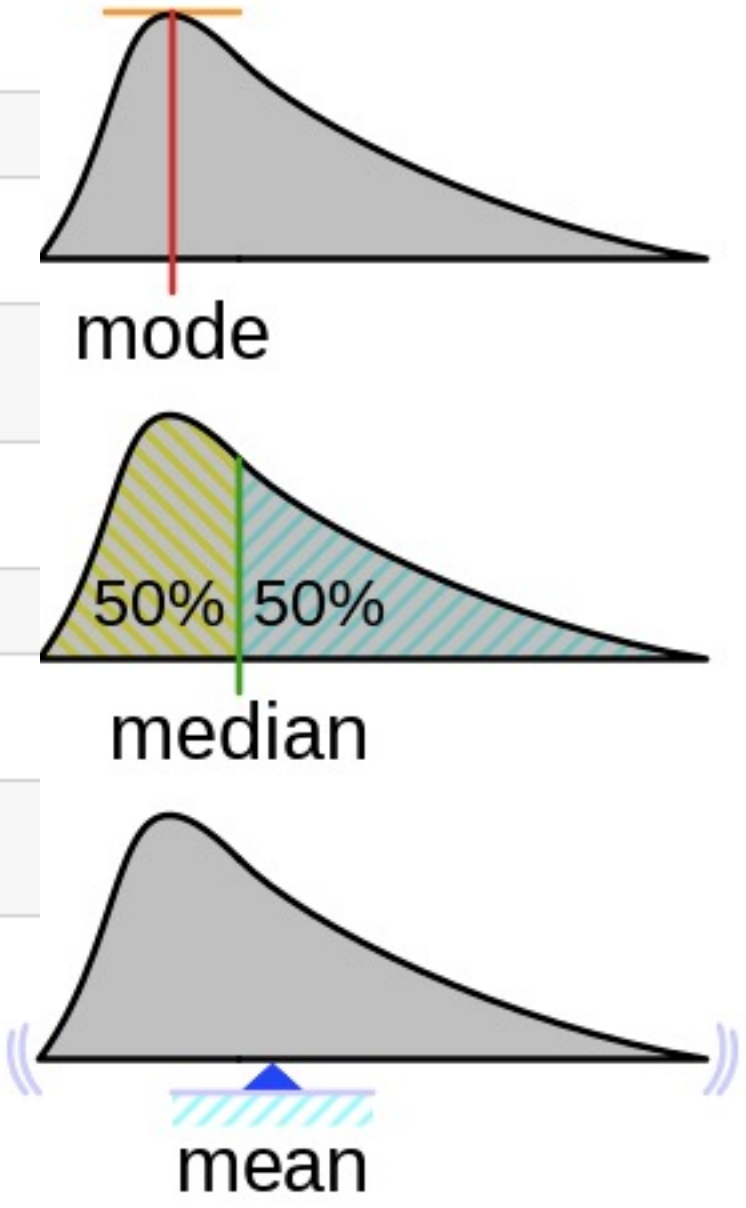
2.6416666666666666

```
np.median(values)
```

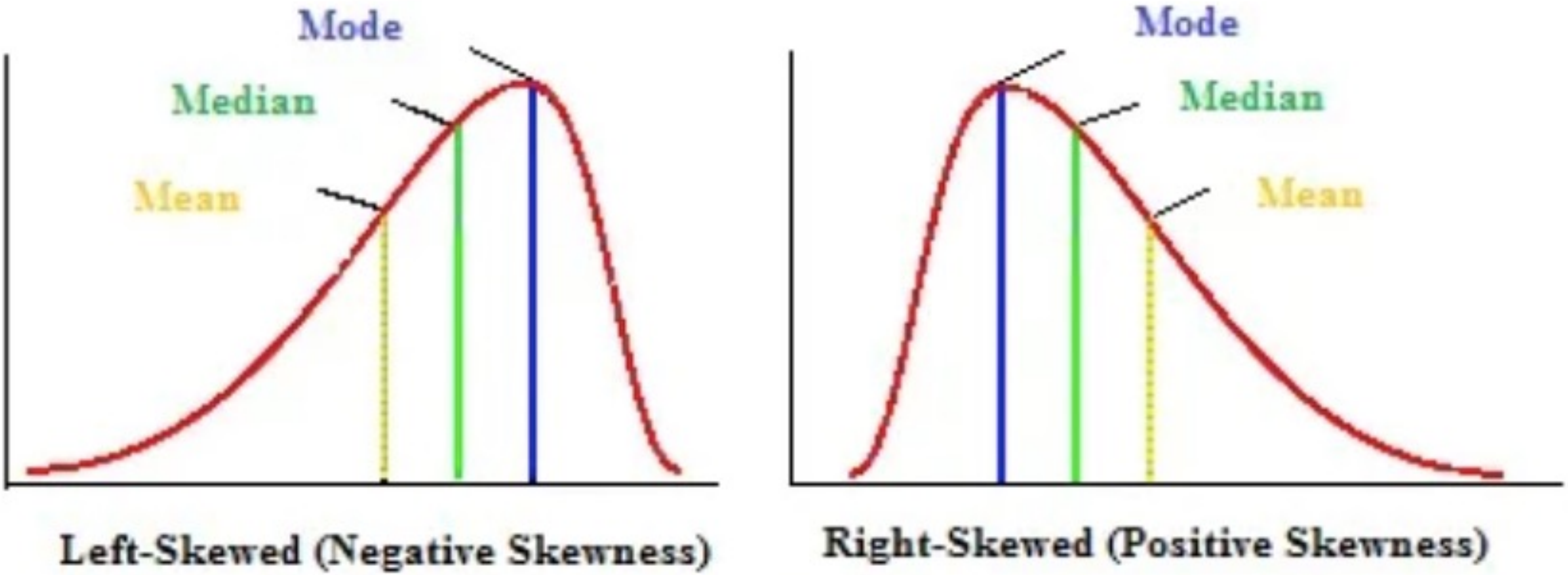
2.55

```
import statistics as sts
sts.mode(values)
```

0

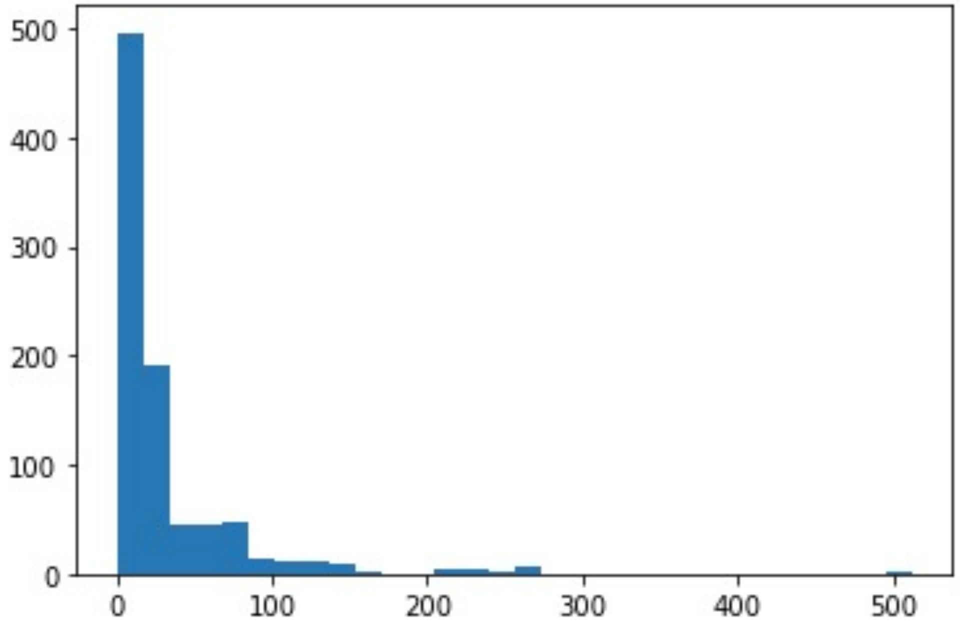


Distribution of Data:



Distribution of Data:

```
plt.hist(df['Fare'], bins = 30)
plt.show()
```

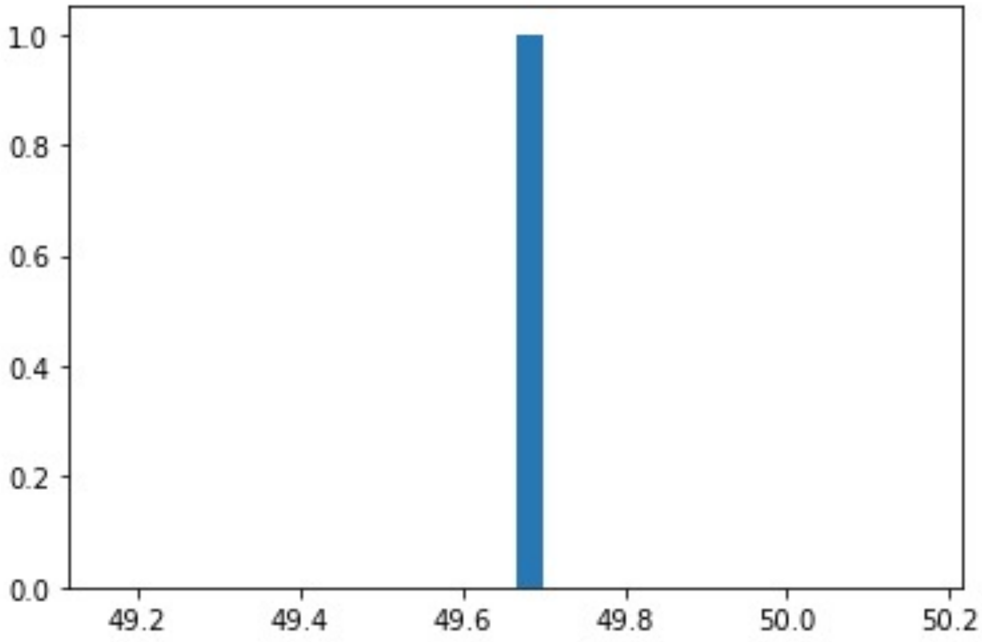


$$z = \frac{x - \mu}{\sigma}$$

$\mu =$ Mean

$\sigma =$ Standard Deviation

```
plt.hist(np.std(df['Fare']), bins = 30)
plt.show()
```



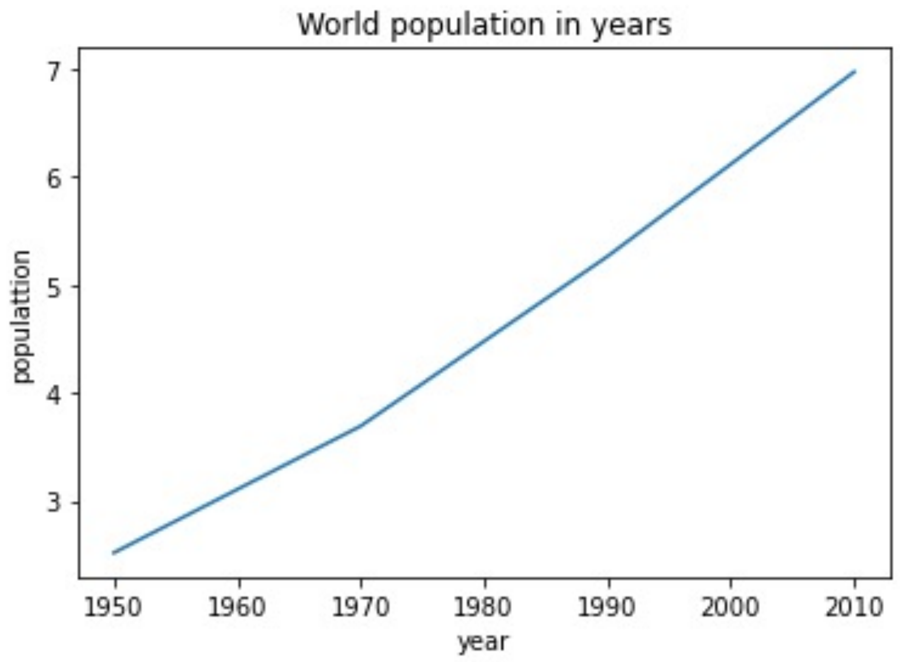
Customization:

- Add labels to the axis: plt.xlabel() , plt.ylabel()
- Add Title to the plot : plt.title()
- Changing values one the axis: plt.xticks() , plt.yticks()
- Labeling values on the axis

Customization:

```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
np_years = np.array(years)
np_pop = np.array(pop)
plt.plot(np_years , np_pop)
plt.xlabel('year')
plt.ylabel('populattion')
plt.title('World population in years')
```

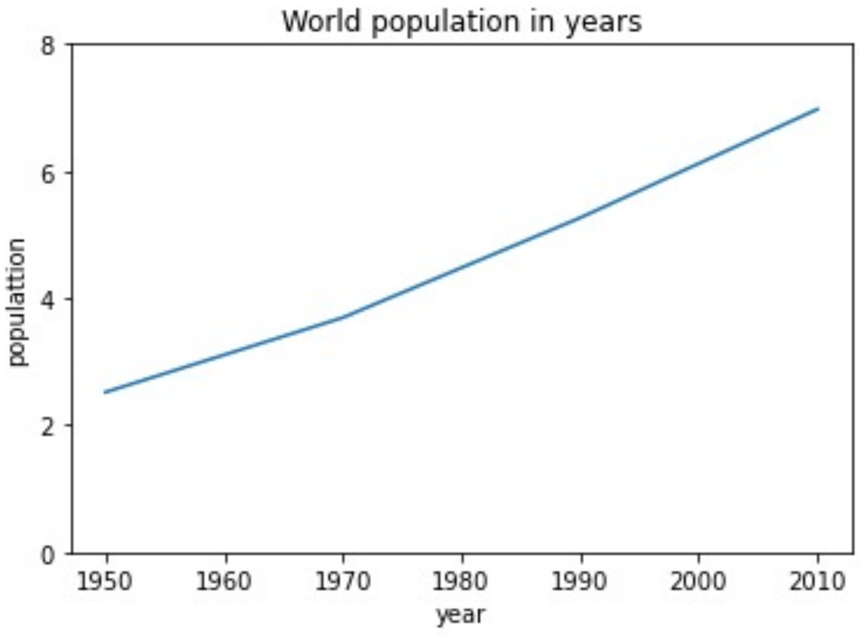
```
Text(0.5, 1.0, 'World population in years')
```



Customization:

```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
np_years = np.array(years)
np_pop = np.array(pop)
plt.plot(np_years , np_pop)
plt.xlabel('year')
plt.ylabel('populattion')
plt.yticks([0,2,4,6,8])
plt.title('World population in years')
```

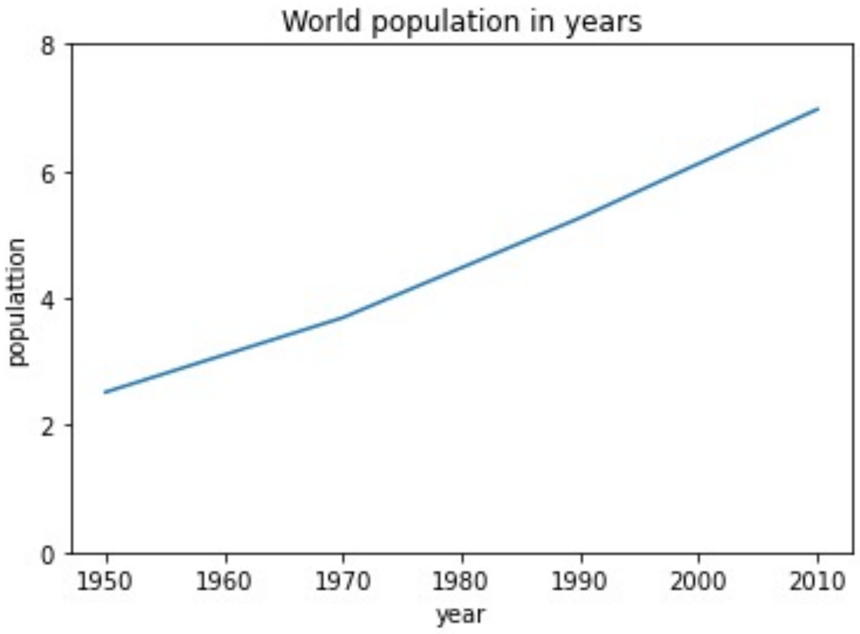
```
Text(0.5, 1.0, 'World population in years')
```



Customization:

```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
np_years = np.array(years)
np_pop = np.array(pop)
plt.plot(np_years , np_pop)
plt.xlabel('year')
plt.ylabel('populattion')
plt.yticks([0,2,4,6,8])
plt.title('World population in years')
```

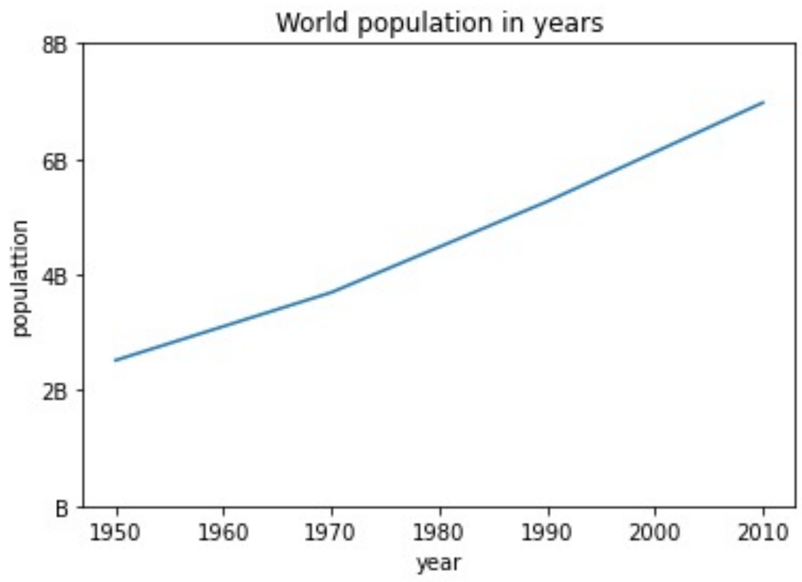
```
Text(0.5, 1.0, 'World population in years')
```



Customization:

```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
np_years = np.array(years)
np_pop = np.array(pop)
plt.plot(np_years , np_pop)
plt.xlabel('year')
plt.ylabel('populattion')
plt.yticks([0,2,4,6,8],['B','2B','4B','6B','8B'])
plt.title('World population in years')
```

```
Text(0.5, 1.0, 'World population in years')
```



Machine Learning:

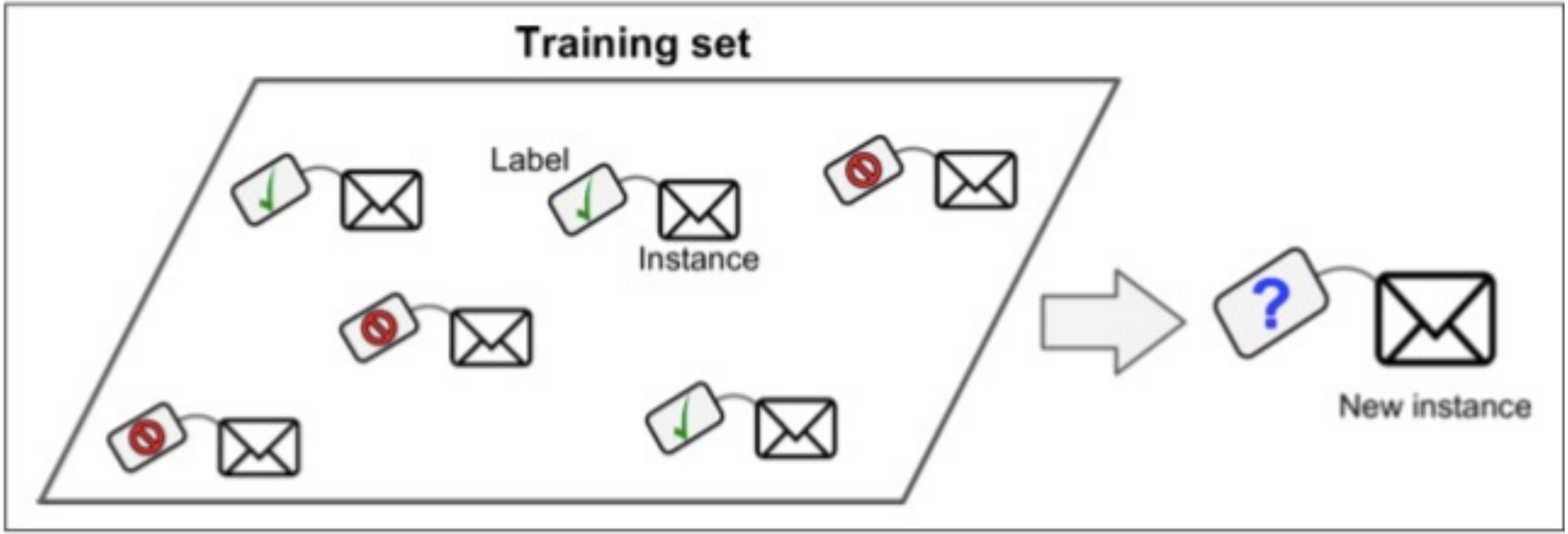
How you write a code with traditional programming technique to detect spams?

- What a spam looks like, what are the patterns,
- Write a detection algorithm for each pattern,… .

Problem??

There is an infinite number of patterns!

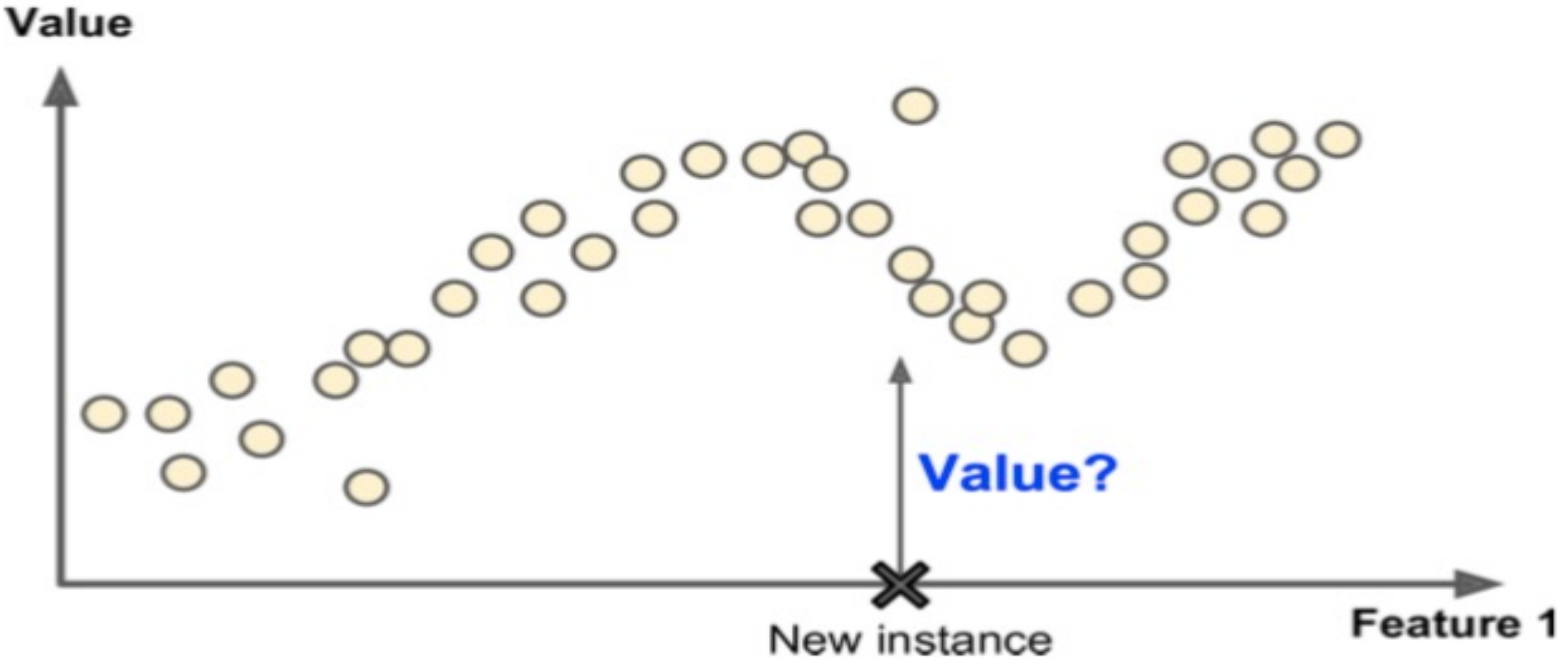
ML (Supervised):



In ML, the model will learn (based on some examples) which patterns are representative of a spam.

Classification

ML (Supervised):



Given a set of Instances and their corresponging value, we can quess what is the value of a newly entered instance.

Regression

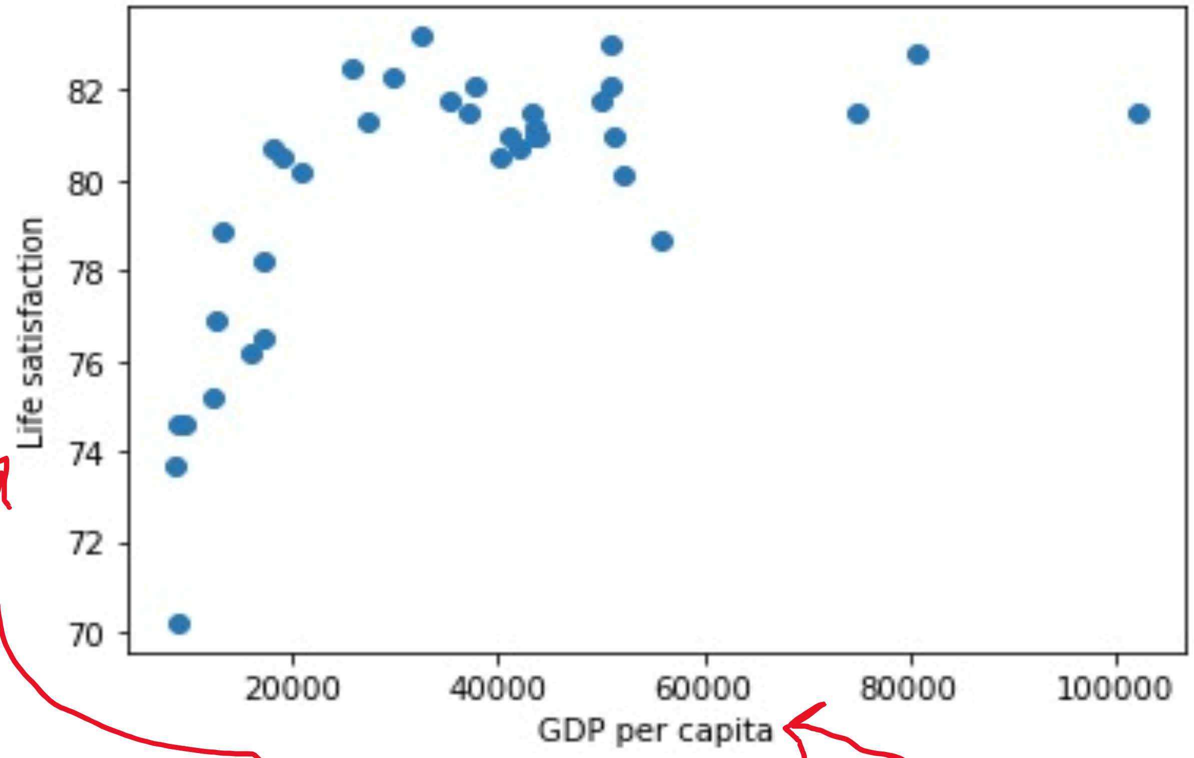
ML (Example):

	Country	GDP per capita	Life satisfaction
0	Australia	50961.865	82.1
1	Austria	43724.031	81.0
2	Belgium	40106.632	80.5
3	Brazil	8669.998	73.7
4	Canada	43331.961	81.5
5	Chile	13340.905	78.9
6	Czech Republic	17256.918	78.2

Given a GDP per capita in a country, can you guess what is the life satisfaction index?

ML (Example):

```
import numpy as np
plt.scatter(data['GDP per capita'] , data['Life satisfaction'])
x = np.array([1000 , 100000])
plt.xlabel('GDP per capita')
plt.ylabel('Life satisfaction')
plt.show()
```



What is the simplest and common pattern in the scatter plot?

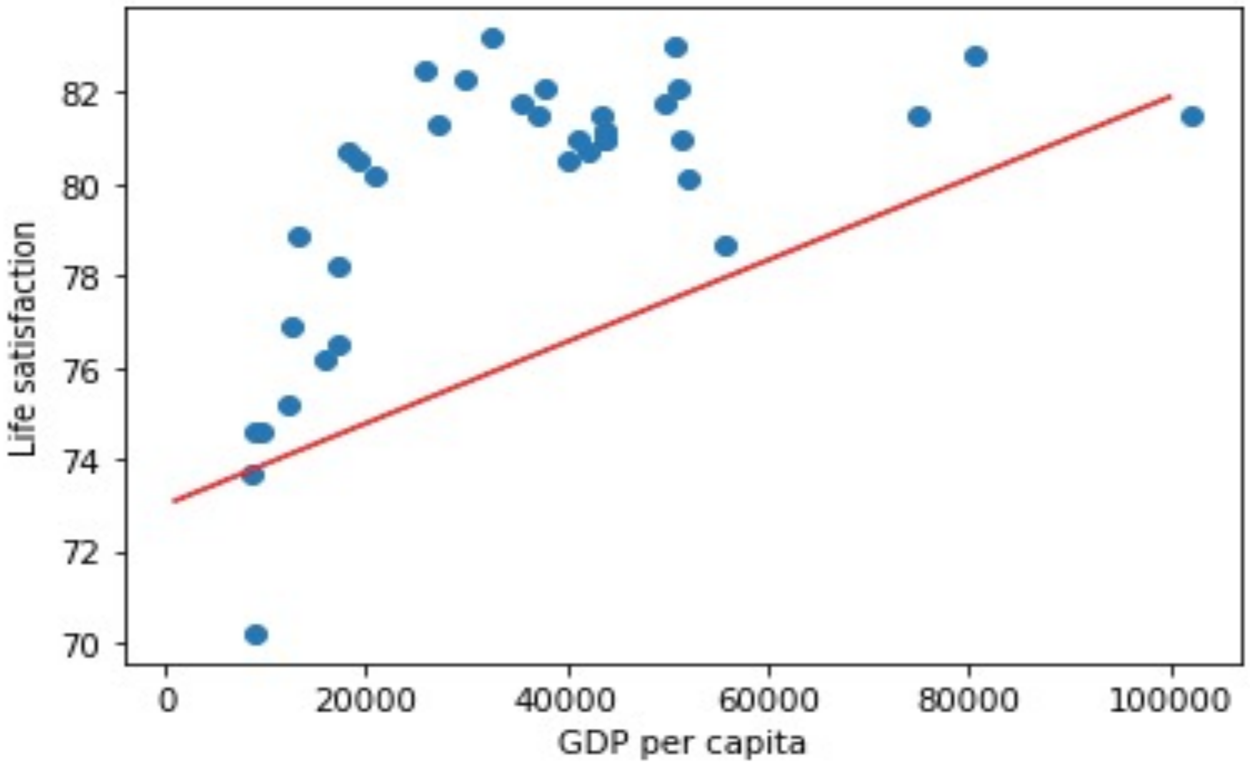
$$y = \theta_0 + \theta_1 x$$

ML (Example):

θ_0 → `t_0`

θ_1 → `t_1`

```
import numpy as np
plt.scatter(data['GDP per capita'] , data['Life satisfaction'])
x = np.array([1000 , 100000])
t_0 = 73
t_1 = 8.9e-05
plt.plot(x , t_0 + t_1*x , c = 'red')
plt.xlabel('GDP per capita')
plt.ylabel('Life satisfaction')
plt.show()
```

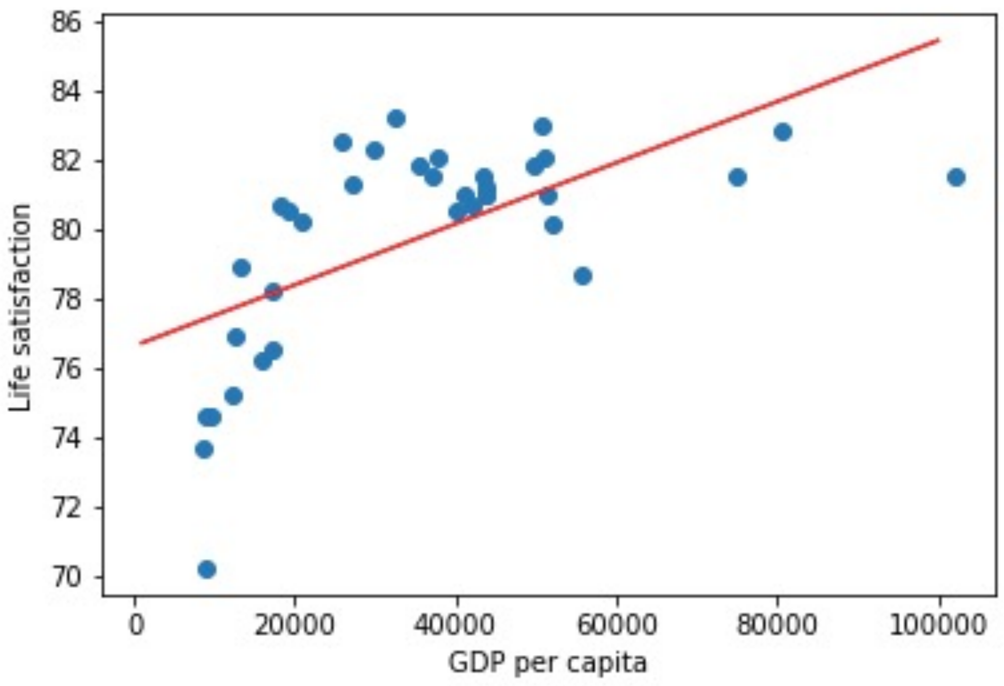


$$life\ satisfaction = \theta_0 + \theta_1 * \ GDP\ per\ Capita$$

ML (Example):

θ_0 → `t_0`
θ_1 → `t_1`

```
import numpy as np
plt.scatter(data['GDP per capita'] , data['Life satisfaction'])
x = np.array([1000 , 100000])
t_0 = 76.61443338
t_1 = 8.82017196e-05
plt.plot(x , t_0 + t_1*x , c = 'red')
plt.xlabel('GDP per capita')
plt.ylabel('Life satisfaction')
plt.show()
```



$$life\ satisfaction = \theta_0 + \theta_1 * GDP\ per\ Capita$$

ML (Linear Assumption):

	GDP per capita	Life satisfaction	
$x^{(1)}$ →	50961.865	82.1	$y^{(1)}$
$x^{(2)}$ →	43724.031	81.0	$y^{(2)}$
$x^{(3)}$ →	40106.632	80.5	$y^{(3)}$
$x^{(4)}$ →	8669.998	73.7	$y^{(4)}$
	43331.961	81.5	
	13340.905	78.9	
	17256.918	78.2	
	52114.165	80.1	
	17288.083	76.5	
	41973.988	80.7	

$$\hat{y}^{(i)} = \theta_0 + \theta_1 \times x^{(i)}$$

ML (Example):

Main assumption: The data follows a linear model:

$$life\ satisfaction = \theta_0 + \theta_1 * GDP\ per\ Capita$$

➢ How you know which values make your model perform best?

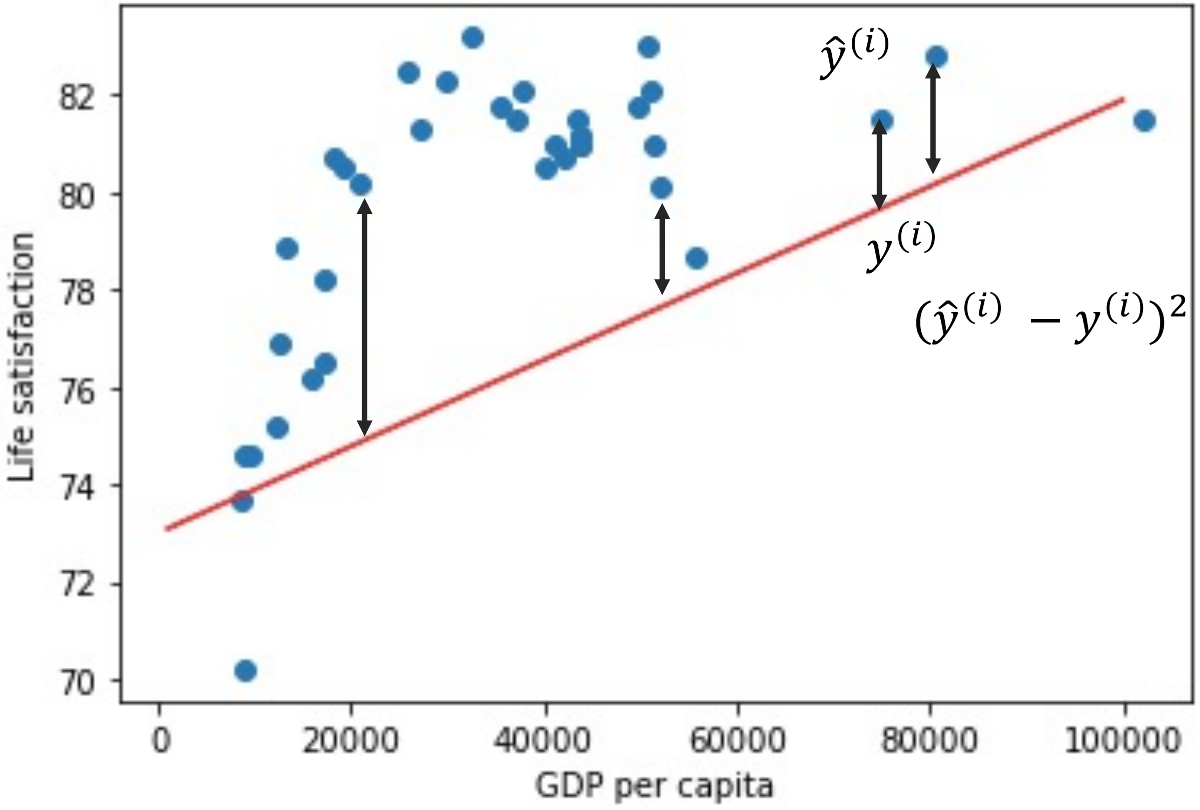
- Fitness Function
- Cost Function (typically used for linear regression problems.)

Linear Regression algorithm comes into play:

you feed it your training examples and it finds the parameters that make the linear model fit best to your data. This is called *training* the model.

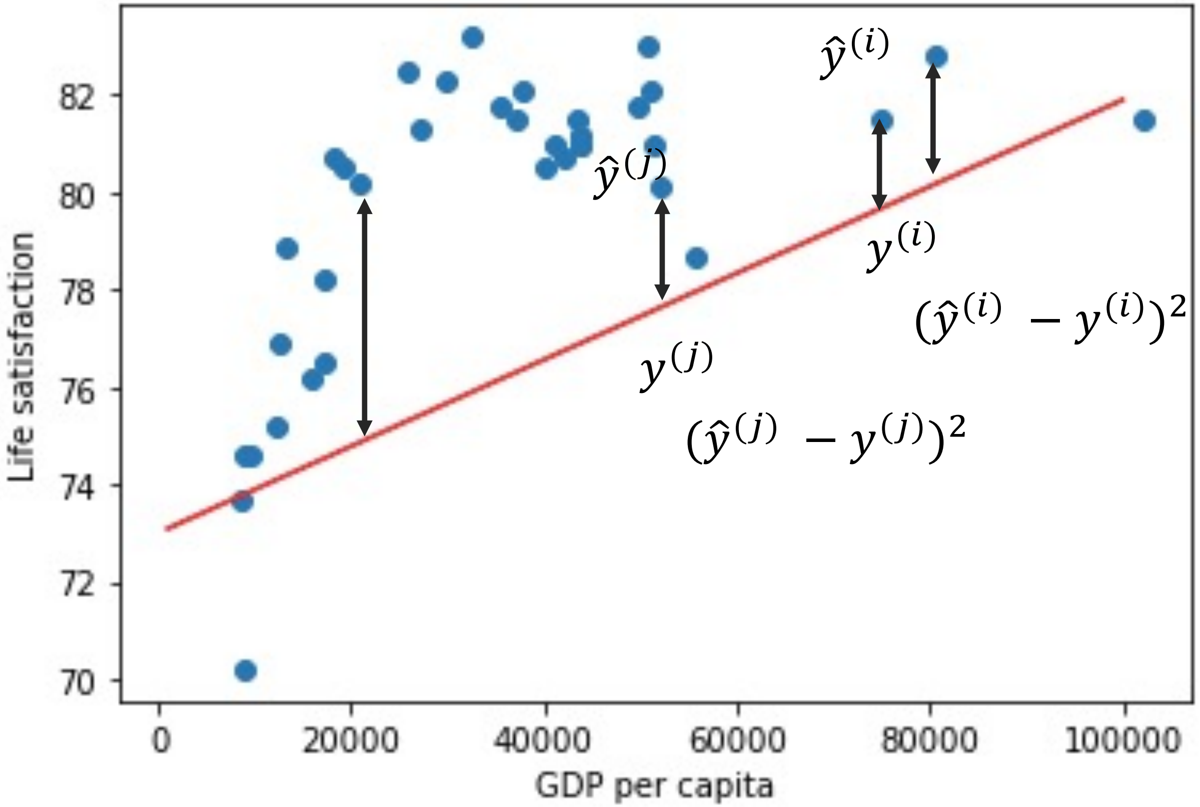
ML (Example):

Cost Function:



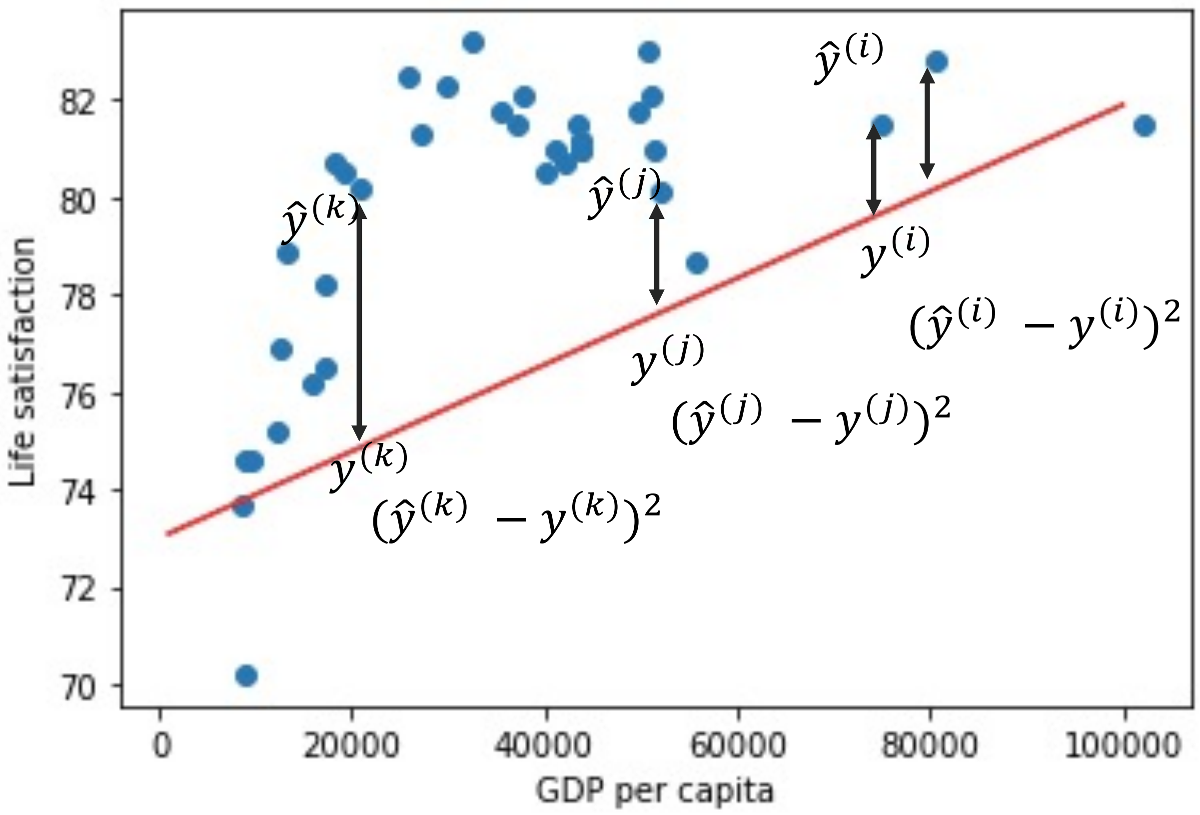
ML (Example):

Cost Function:



ML (Example):

Cost Function: Root Mean Square Error



$$\mathrm{RMS}E = \sqrt{\frac{1}{m}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2}$$

Always positive

ML (Example):

Sklearn (Python library for sklearn)

```
from sklearn.linear_model import LinearRegression
```

```
model = LinearRegression()
```

```
model.fit(data[['GDP per capita']], data[['Life satisfaction']])
```

Model Training

ML (Example):

Sklearn (Python library for sklearn)

```
model.predict(data[['GDP per capita']])
```

```
array([[81.10935751],
       [80.47096811],
       [80.15190729],
       [77.37914212],
       [80.43638686],
       [77.79112415],
       [78.13652323],
       [81.21099235],
       [78.13927203],
       [80.3166113 ],
       [79.9374337 ],
       [80.23039615],
       [78.20773465],
       [77.69401308],
       [81.09989505],
```

Model Prediction

```
data[['Life satisfaction']]
```

	Life satisfaction
0	82.1
1	81.0
2	80.5
3	73.7
4	81.5
5	78.9
6	78.2
7	80.1
8	76.5
9	80.7
10	82.1
11	81.0
12	80.7
13	75.2
14	83.0
15	81.0

ML (Example):

Predictor (A combination of attributes)

Target variable

GDP per capita	Life satisfaction
50961.865	82.1
43724.031	81.0
40106.632	80.5
8669.998	73.7
43331.961	81.5
13340.905	78.9
17256.918	78.2
52114.165	80.1
17288.083	76.5
41973.988	80.7

Samples

Assumption of Linearity

Training (Minimizing RMSE)

Predictions

```
[[81.10935751],
 [80.47096811],
 [80.15190729],
 [77.37914212],
 [80.43638686],
 [77.79112415],
 [78.13652323],
 [81.21099235],
 [78.13927203],
 [80.3166113 ],
```

ML :

Note: In general their might be more than one attribute:

- In this case, the first attribute of simple (i) is represented by variable $x_1^{(i)}$,
- The second attribute would be $x_2^{(i)}$
- ....
- The attribute p would be $x_p^{(i)}$

The linear assumption: $\hat{y}^{(i)} = \underbrace{\theta_0 + \theta_1 \times x_1^{(i)} + \theta_2 \times x_2^{(i)} + \ldots + \theta_p \times x_p^{(i)}}_{\text{Hyperplane}}$

Note: in some texts instead of $\hat{y}^{(i)}$, they use $h_\theta(x^i)$

ML (Example):

How the training part works? (The minimization of RMSE)

Min $\quad \mathrm{RM}SE = \sqrt{\frac{1}{m} \sum_{i=1}^{n} (\hat{y}^{(i)} - y^{(i)})^2}$

Is equal to Min $\quad (\theta^T X - y)^2$

$$\theta = [\theta_0 \quad \theta_1 \quad \theta_2 \ \dots \ \theta_p]$$

$$X = \begin{pmatrix} 1 & x_1^1 & x_2^1 & \cdots & x_p^1 \\ 1 & x_1^2 & x_2^2 & \cdots & x_p^2 \\ \vdots & \vdots & \cdots & \vdots & \\ 1 & x_1^n & x_2^n & \cdots & x_p^n \end{pmatrix}$$

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ \vdots \\ y^{(n)} \end{bmatrix}$$

ML (Example):

How the training part works? (The minimization of RMSE)

Min $(\theta^T X - y)^2$

$$\arg min_{\theta \in \mathbb{R}^{p+1}} (\theta X - Y)^T (\theta X - Y)$$

$$\nabla_\theta (\theta X - Y)^T (\theta X - Y) = 0$$

$$-2 X^T (y - \theta X) = 0$$

Normal Equation

$$\theta = (X^T X)^{-1} X^T y$$

It has a solution only when $(X^T X)^{-1}$ is inversible (when it's determinant is non-zero).

ML (Example):

Let’s test the normal equation by generating random data that follow linear pattern:

```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

Predictor

Target Variable

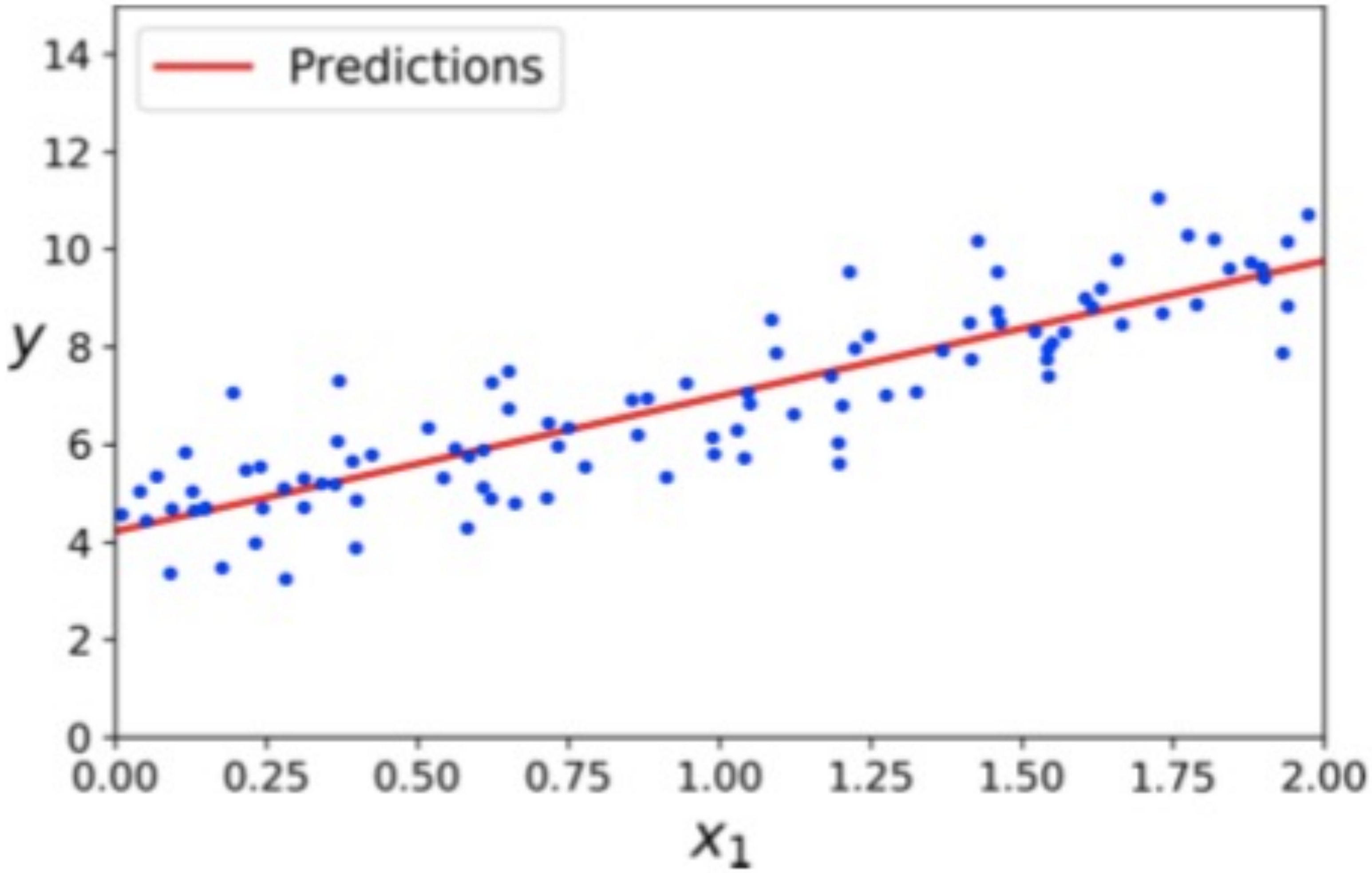
```
X
```

```
array([[0.10797752],
       [0.72385904],
       [0.42813739],
       [1.41056299],
       [1.28648117],
       [0.21776623],
       [1.03717871],
       [0.365245  ],
       [0.20016469],
       [0.20727274],
```

```
y
```

```
array([[ 3.5573093 ],
       [ 7.87380775],
       [ 7.2598704 ],
       [ 6.19588811],
       [ 9.13845766],
       [ 4.23094532],
       [ 8.61517587],
       [ 4.21443654],
       [ 6.3025794 ],
       [ 4.38441334],
```

ML (Example):



Predictions
14
12
10
8
6
4
2
0
y
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
x_1

ML (Example):

```
X_b = np.c_[np.ones((100, 1)), X] # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

Create matrix X

Predictor

```
X_b
```

```
array([[1.        , 0.10797752],
       [1.        , 0.72385904],
       [1.        , 0.42813739],
       [1.        , 1.41056299],
       [1.        , 1.28648117],
       [1.        , 0.21776623],
       [1.        , 1.03717871],
       [1.        , 0.365245  ],
       [1.        , 0.20016469],
       [1.        , 0.20727274],
```

```
theta_best
```

```
array([[4.06669028],
       [2.9236695 ]])
```

ML (Example):

```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta_best)
y_predict
```

```
array([[4.06669028],
       [9.91402929]])
```

ML (Exercise):

Calculate normal equation for the dataset of GDP per capita / Life satisfaction.

ML (Gradient Descent):

➢ Problems with normal equations:

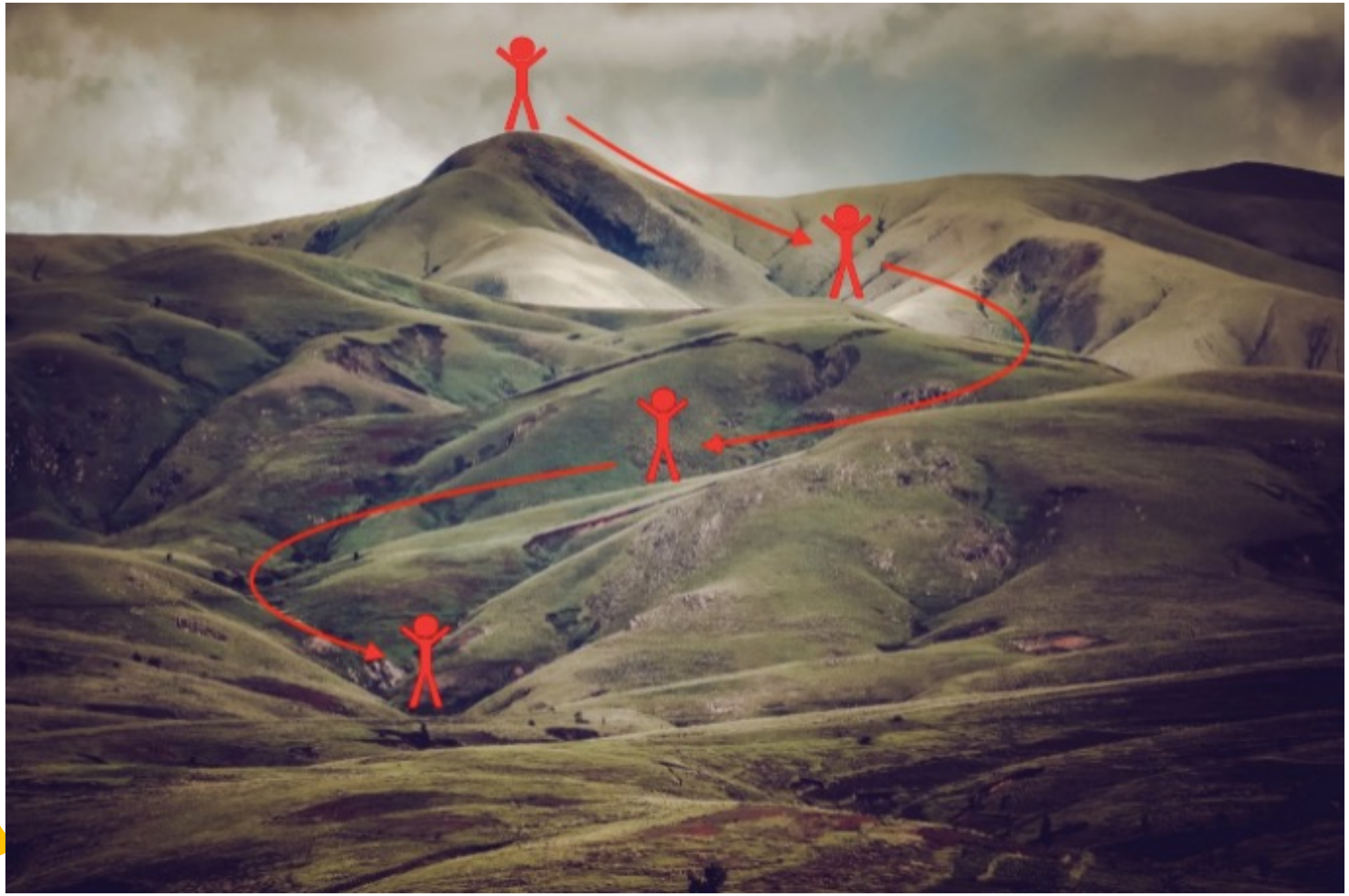
1. In many real cases $(X^T X)^{-1}$ is not invertible,
2. Even if it is for bid data sets the computational cost is $O(n^3)$ or $O(n^{2.4})$.

➢ So, instead of calculating θ from the normal equation, the learning algorithms use a technique to estimate this value which is called Gradient Descent.

1. Start with some initial parameters θ,

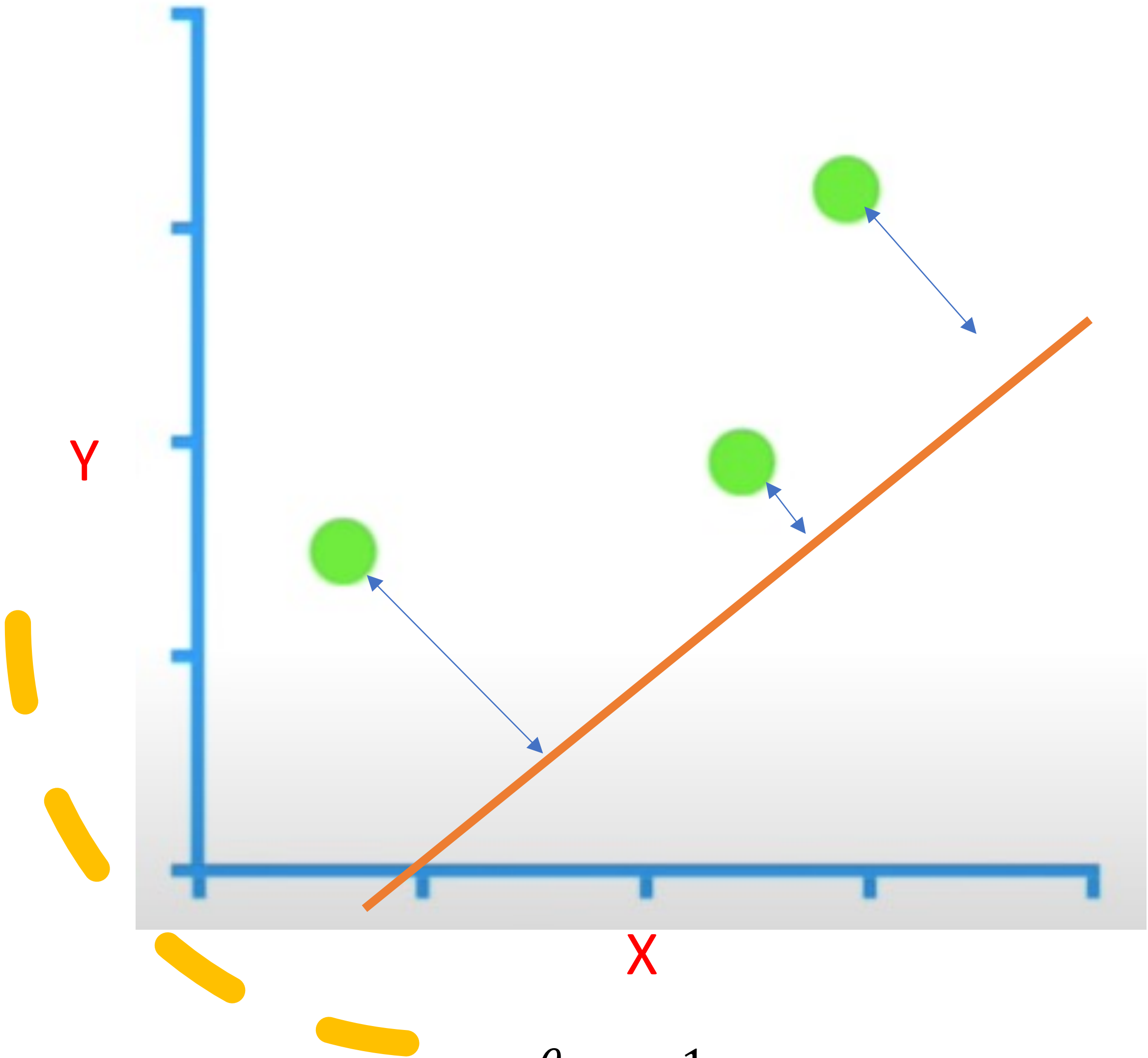
2. Tweaking the parameters (θ) iteratively, in a way that it reduces the cost function.

ML (Gradient Descent-Example):

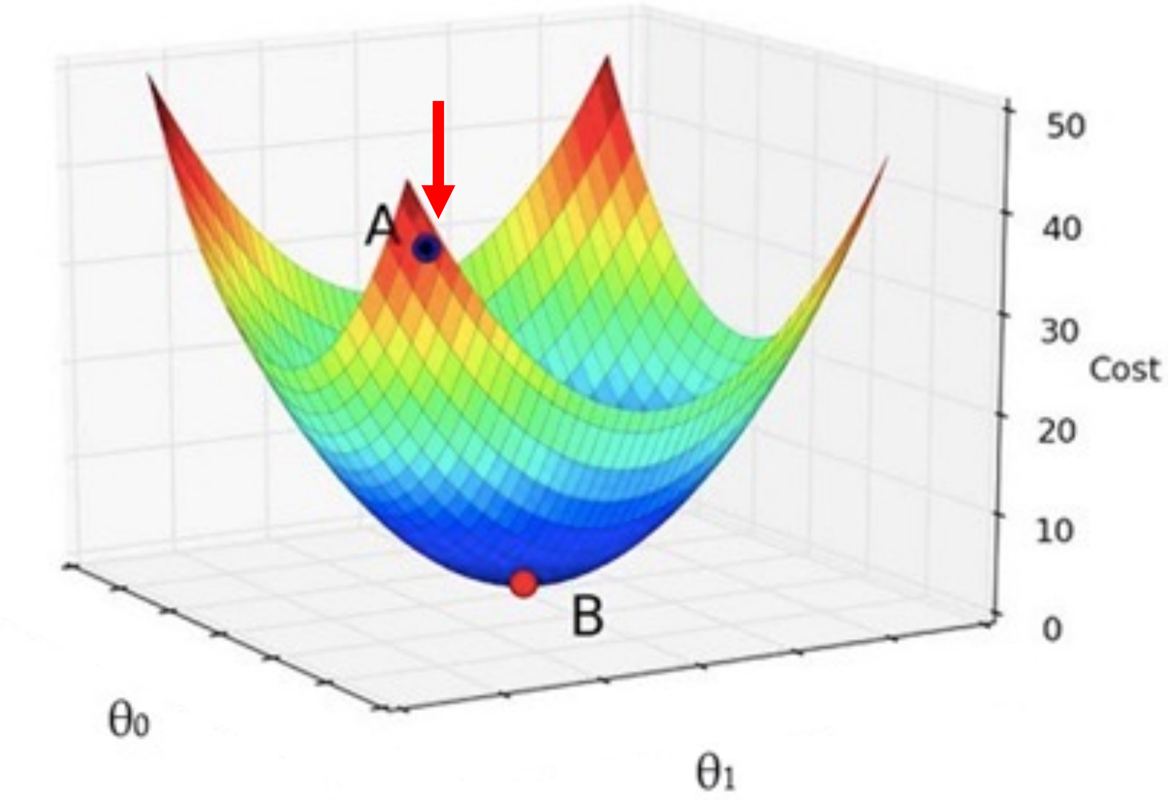


1. Start from a position (x, y),
2. Find the negative slope (to descend)
3. Take appropriate size step.

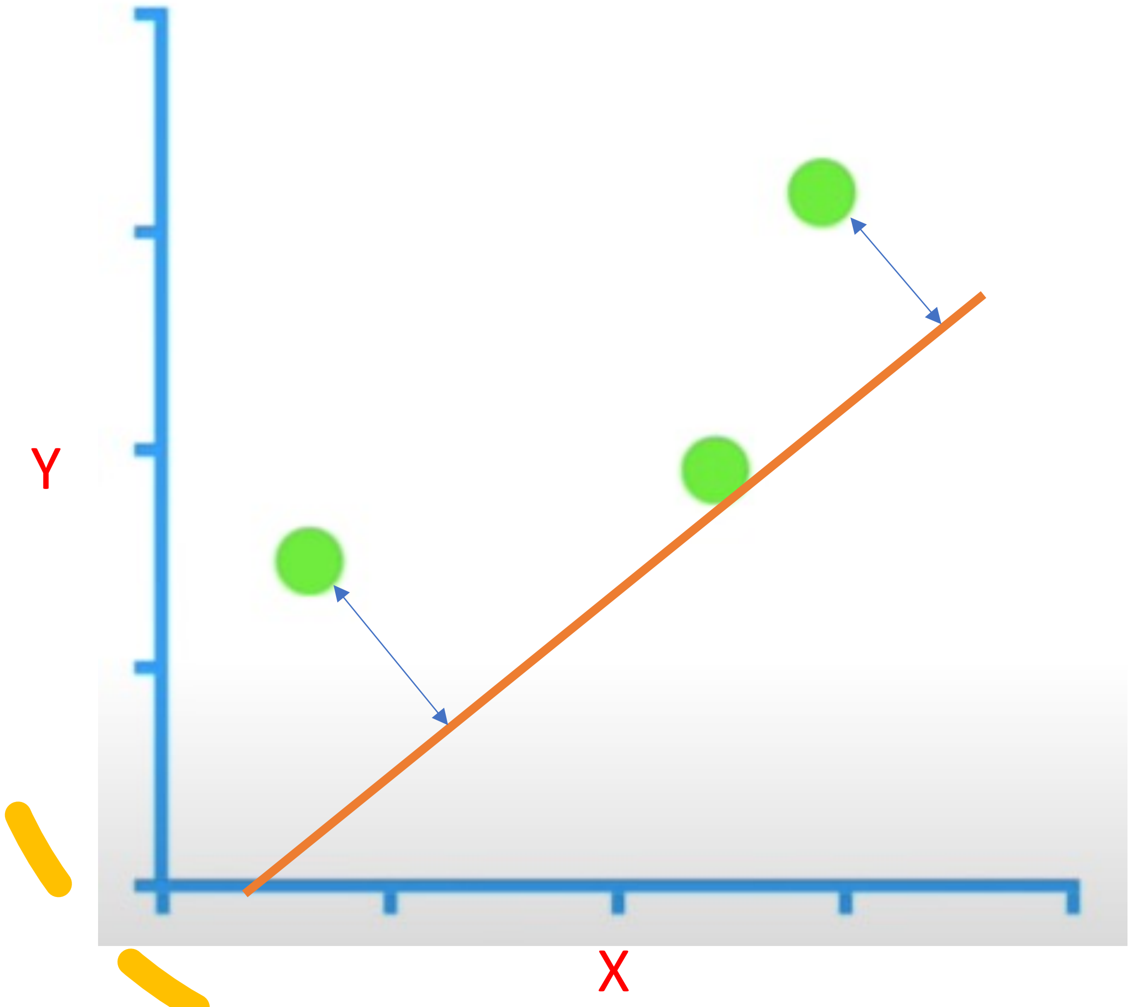
ML (Gradient Descent-Example):



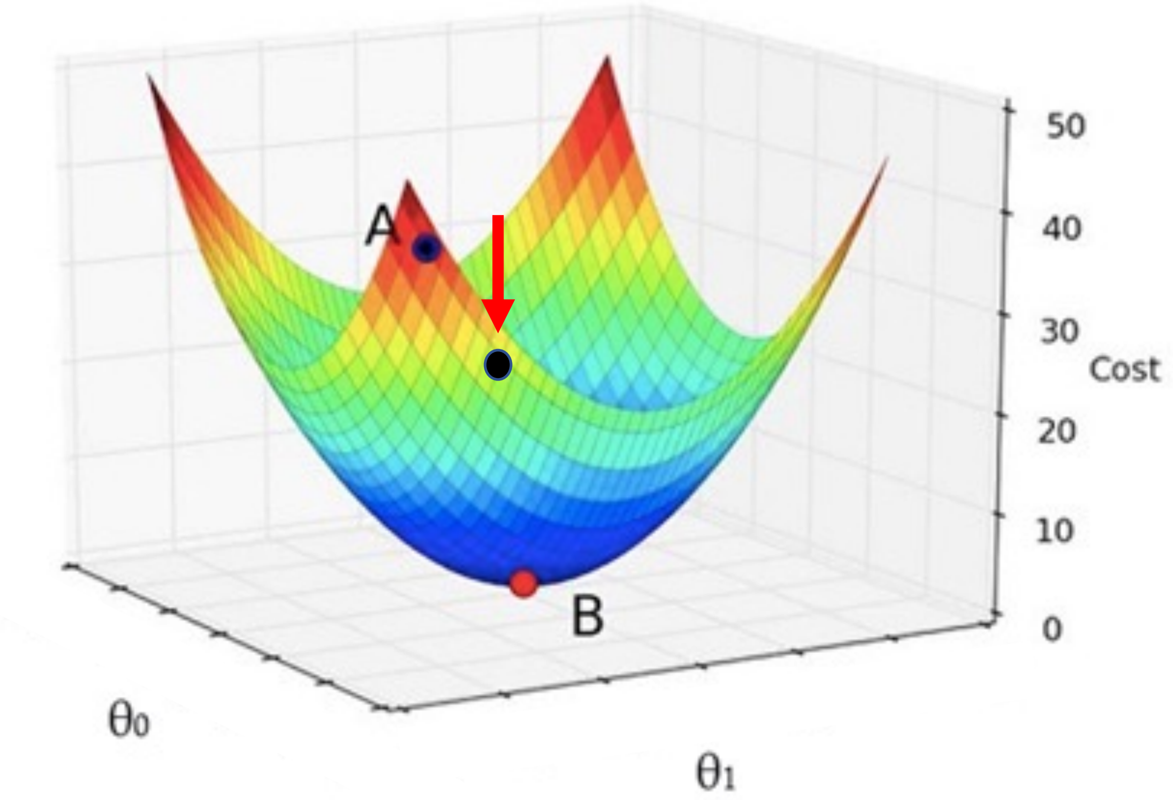
$\theta_0 = -1$
$\theta_1 = 1$



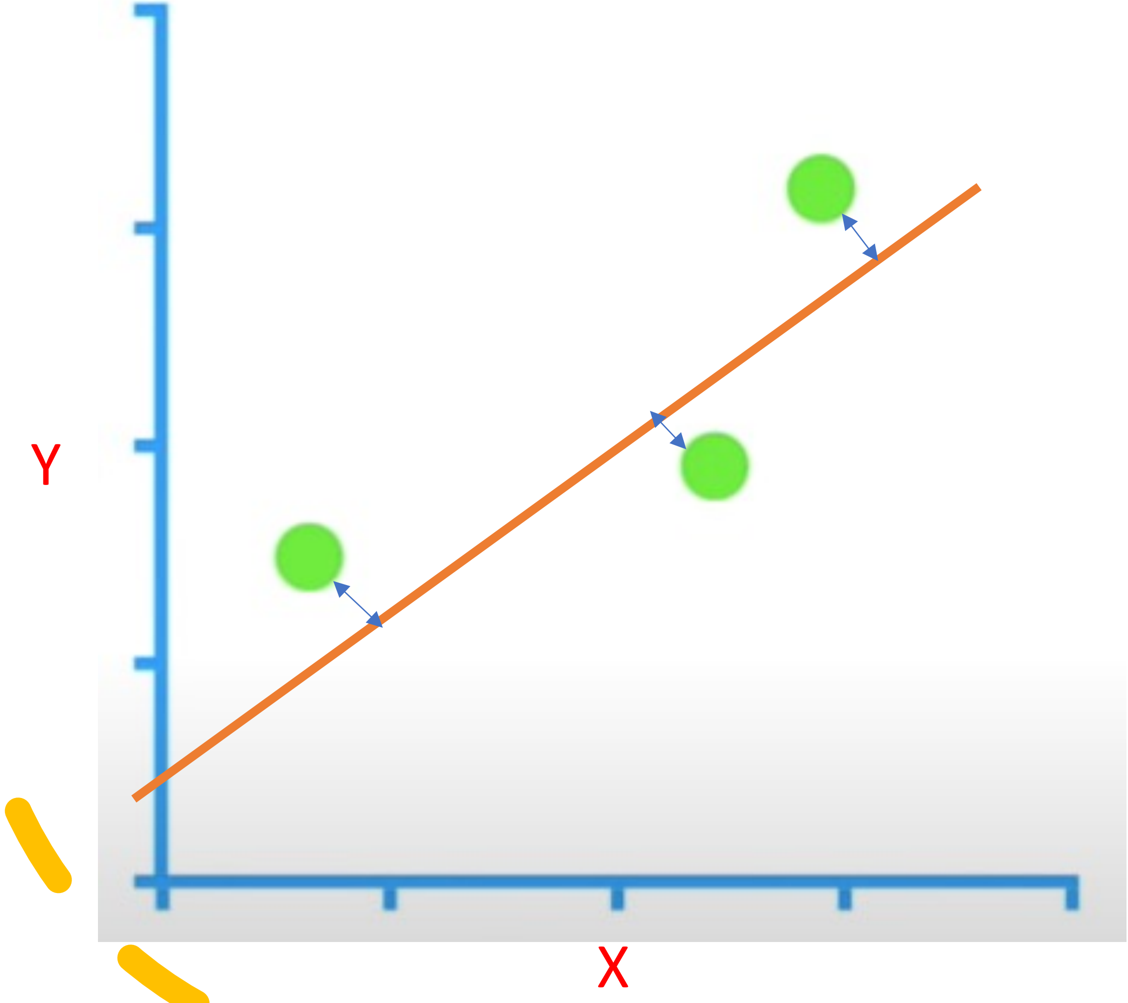
ML (Gradient Descent-Example):

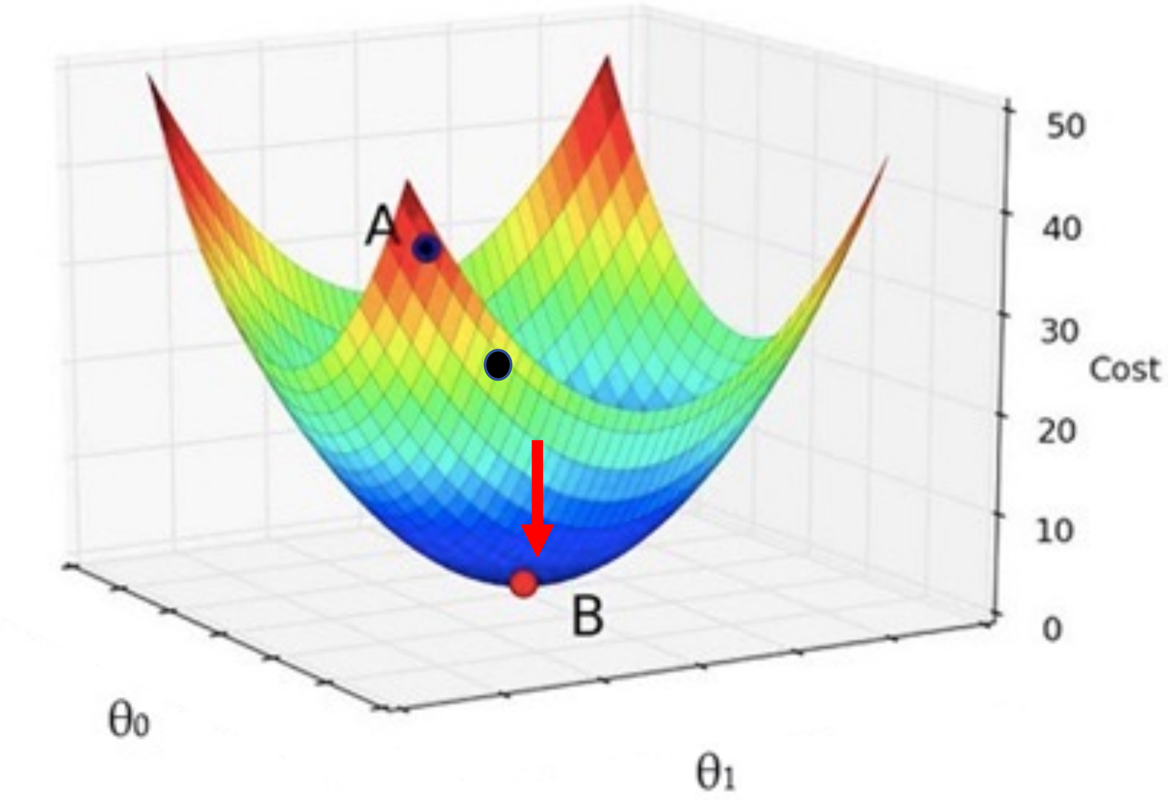


$\theta_0 = -0.5$
$\theta_1 = 1$



ML (Gradient Descent-Example):





$\theta_0 = 0.5$

$\theta_1 = 0.6$

ML (Gradient Descent-Example):

- Suppose we start with initial θ_0 and θ_1 .
- In which direction we should go to reduce the cost function?

$$MSE = (h_\theta(x^{(1)}) - y^{(1)})^2 + (h_\theta(x^{(2)}) - y^{(2)})^2 + (h_\theta(x^{(3)}) - y^{(3)})^2$$

$$\frac{d(MSE)}{d(\theta_0)} = \frac{d(MSE)}{d(h_\theta)} \times \frac{d(h_\theta)}{d(\theta_0)}$$

$$\frac{d(MSE)}{d(\theta_0)} = 2\,(h_\theta(x^{(1)}) - y^{(1)}) \times 1 + 2\,(h_\theta(x^{(2)}) - y^{(2)}) \times 1 + 2\,(h_\theta(x^{(3)}) - y^{(3)}) \times 1$$

$$\frac{d(MSE)}{d(\theta_1)} = \frac{d(MSE)}{d(h_\theta)} \times \frac{d(h_\theta)}{d(\theta_1)}$$

$$\frac{d(MSE)}{d(\theta_1)} = 2\,(h_\theta(x^{(1)}) - y^{(1)}) \times x^{(1)} + 2\,(h_\theta(x^{(2)}) - y^{(2)}) \times x^{(2)} + 2\,(h_\theta(x^{(3)}) - y^{(3)}) \times x^{(3)}$$

ML (Gradient Descent-Example):

- Step sizes:

$$\frac{d(MSE)}{d(\theta_0)}$$

Step size (θ_0) = $\text{slope} \times \textit{learning rate}$

Step size (θ_1) = $\text{slope} \times \textit{learning rate}$

$$\frac{d(MSE)}{d(\theta_1)}$$

Update Functions:

new θ_0 = previous θ_0 - step size θ_0
new θ_1 = previous θ_1 - step size θ_1

ML (Gradient Descent-summary):

1. Random initialization of parameters ($\theta_0, \theta_1, \theta_2$,…),
2. Calculate the slopes using gradient descent and chain rule,
3. Compute the steps using a pre-defined learning rate,
4. Update the parameters,

Note: In Machine learning, all the parameters that should be pre-defined in order to use the model are called hyper parameters. Hyper parameters are different from parameters. The parameters will be learned during training,…

ML (Error Calculation):

Calculate the Root Mean Square Error for the predictions you made using linear regression for the dataframe GDP per capita/ life satisfaction(Use the fonctionalities of numpy).

ML (Error Calculation):

Sub-module metrics and the function mean_squared_error of Sklearn.

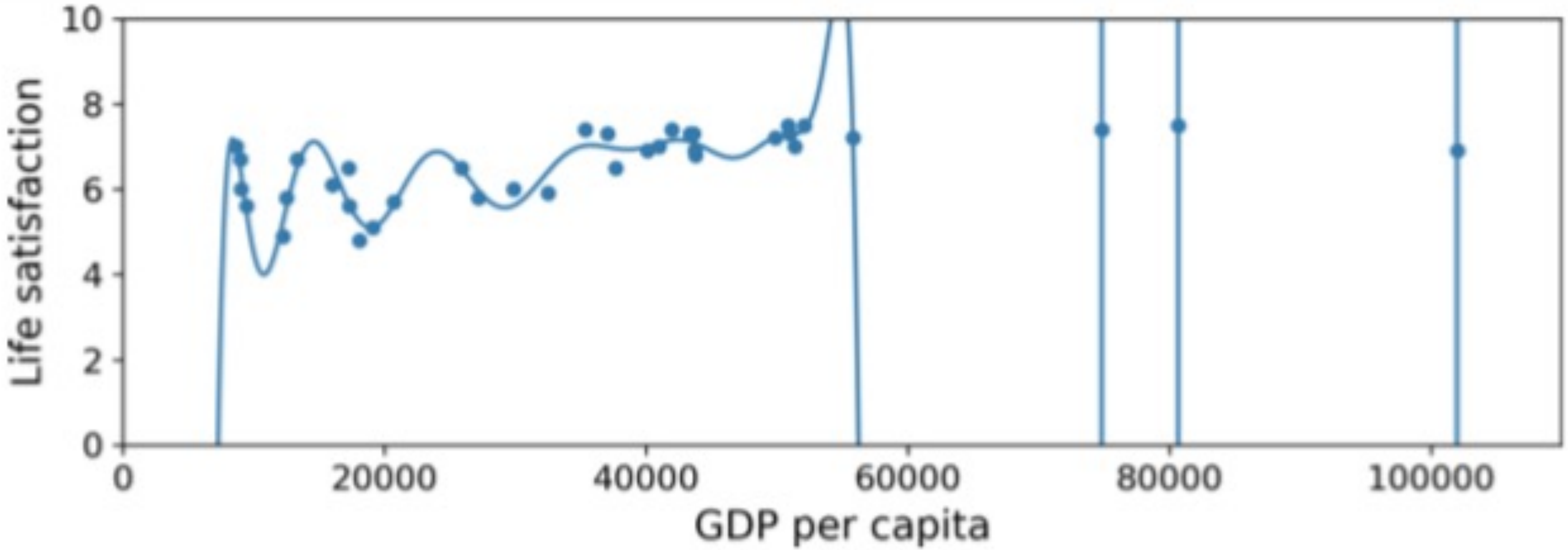
```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(np.array(data['Life satisfaction']) , predictions)
rmse = np.sqrt(mse)
```

ML (Question):

- Is this error reliable for future predictions?
- Does it mean that our model will perform the best to predict?

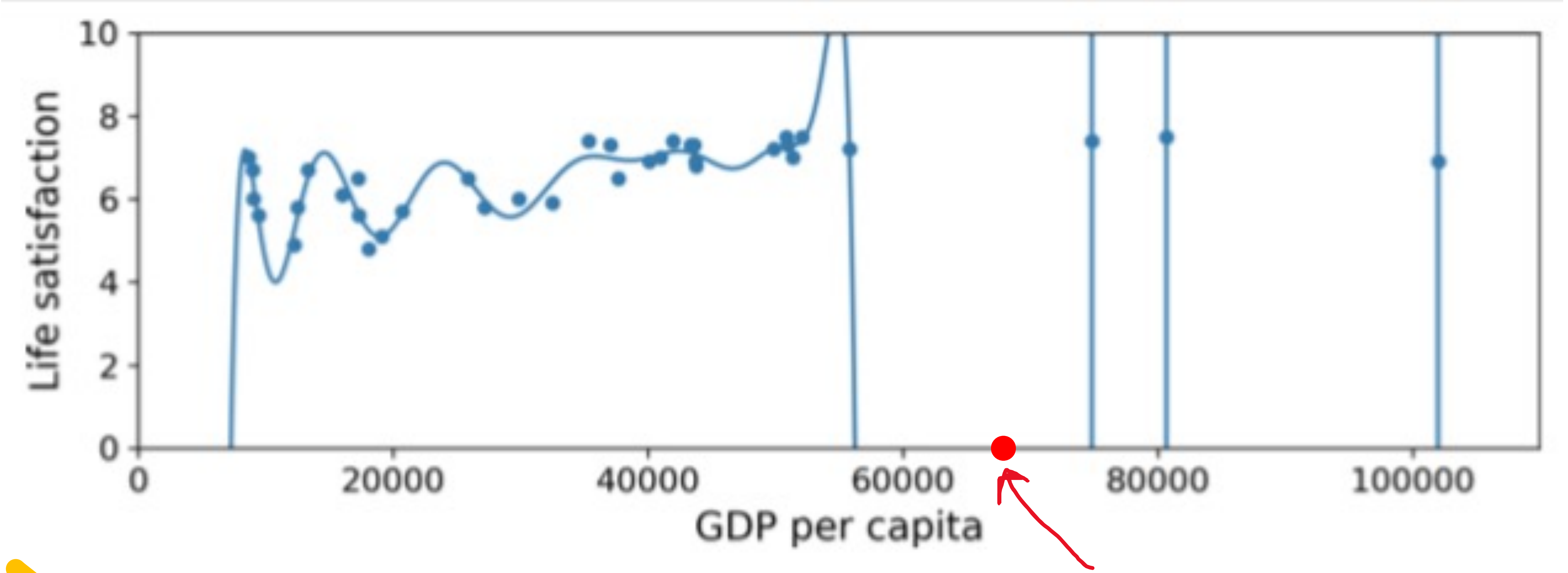
ML (Challenges of training):

Overfitting



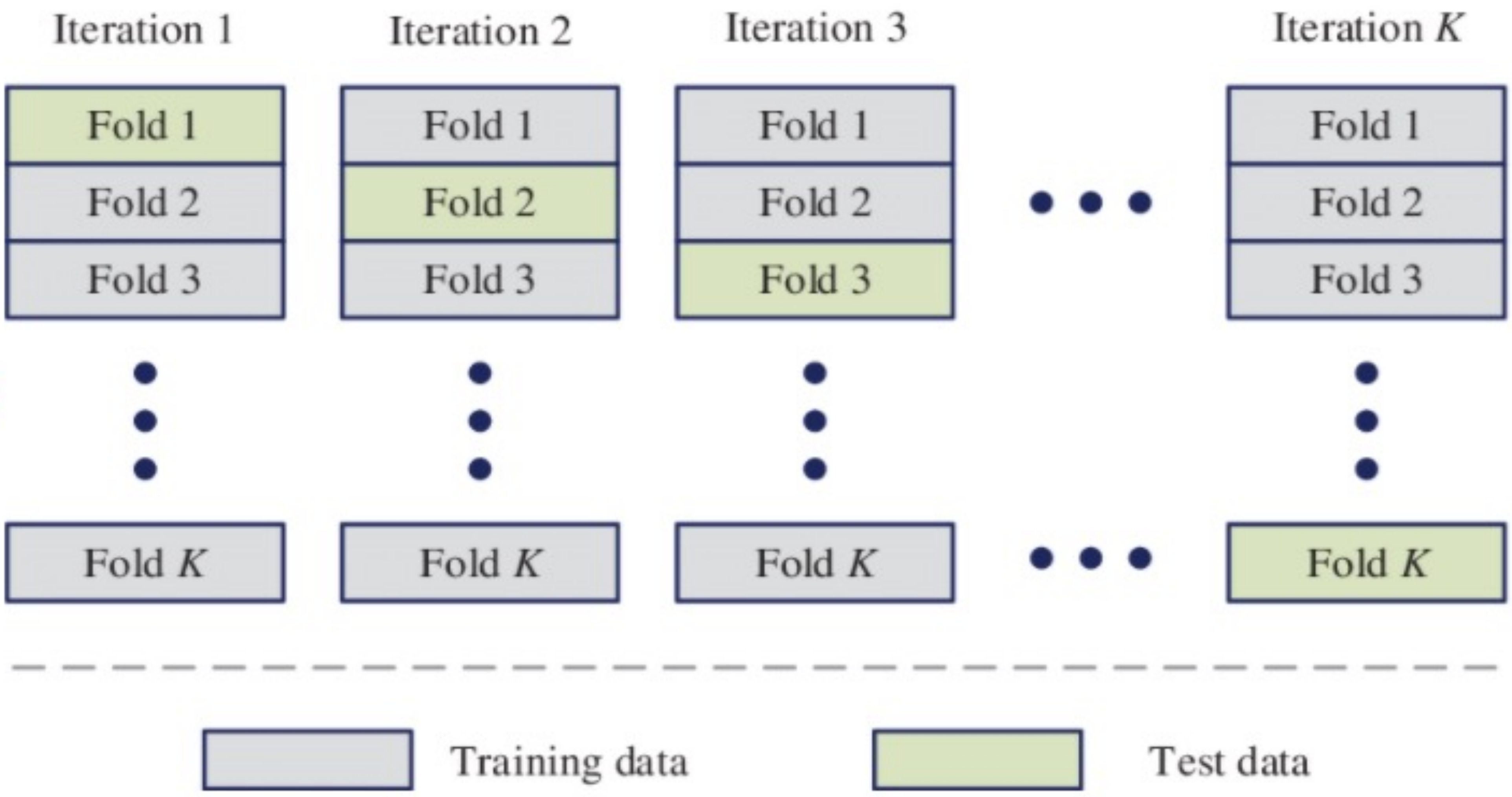
ML (Challenges of training):

Overfitting



What would be the prediction value for this point?

ML (Concept of cross validation):



K-fold cross-validation method.

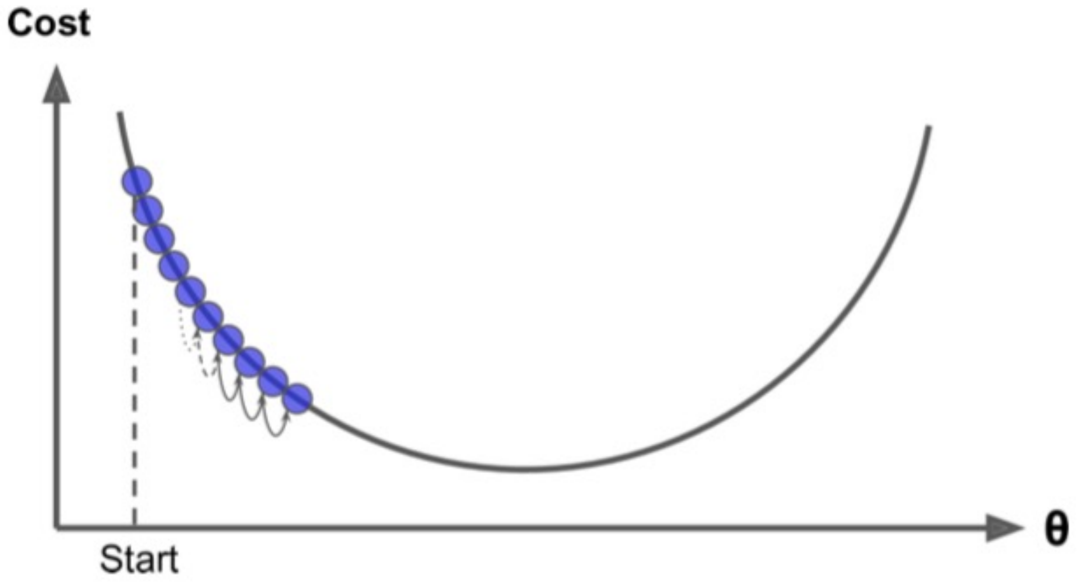
ML (Concept of cross validation):

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, data[['GDP per capita']], data[['Life satisfaction']],
                         scoring="neg_mean_squared_error", cv=10)
rmse_scores = np.sqrt(-scores)
```

```
rmse_scores
```

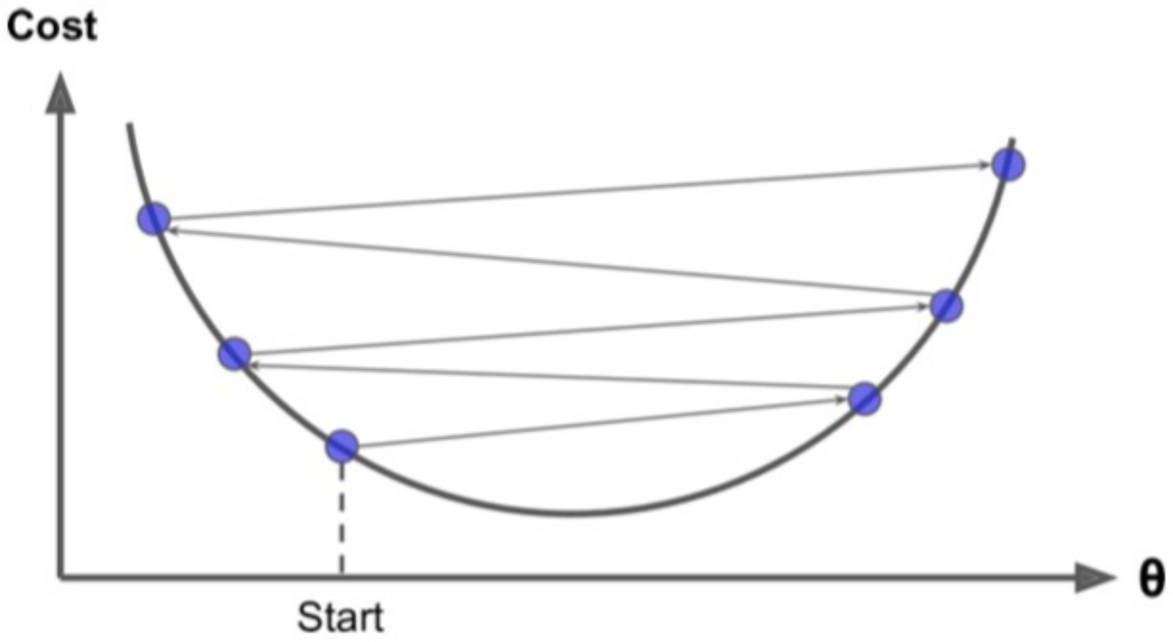
```
array([2.06550957, 0.98307636, 1.45811595, 2.0309329 , 3.211669  ,
       3.09781241, 1.78440725, 4.78091017, 2.29082548, 2.46068562])
```

ML (Challenges of gradient descent)



Small learning rate

Learning rate often is a value between 0 and 1, to find the best learning rate, we need to test the validation error of the candidate models with different learning rate.



Large learning rate

ML (Challenges of gradient descent)

Different versions of Gradient Descent

➢ **Batch Gradient descent**

$$\nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta})$$

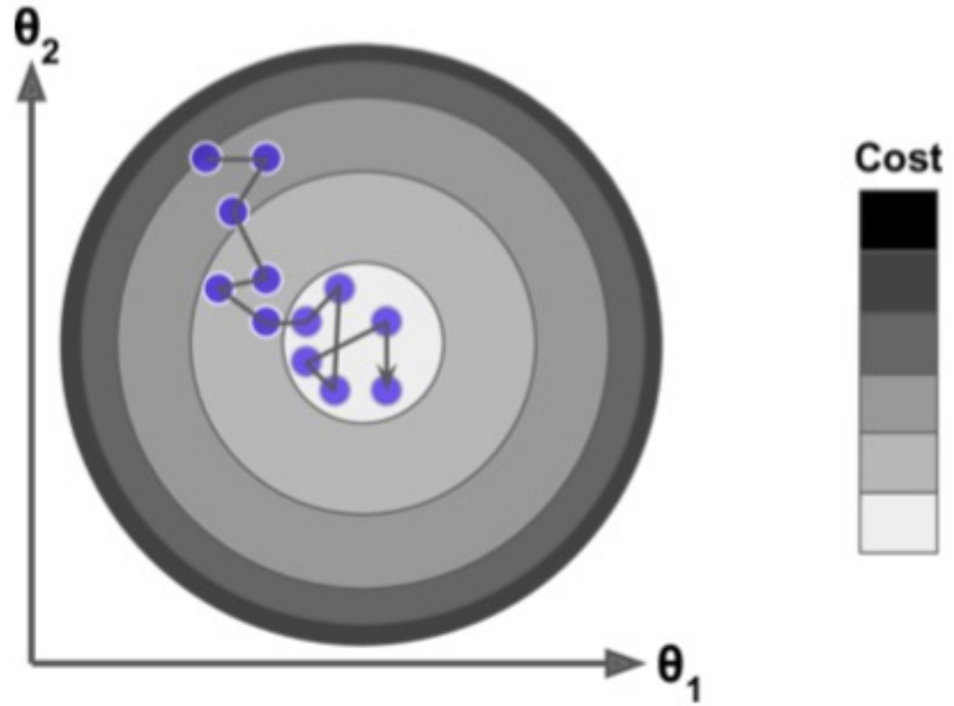
disadvantage??

ML (Challenges of gradient descent)

Different versions of Gradient Descent

➢ Stochastic Gradient descent

- Choose a sample randomly,
- Update the parameters based on the randomly selected sample,

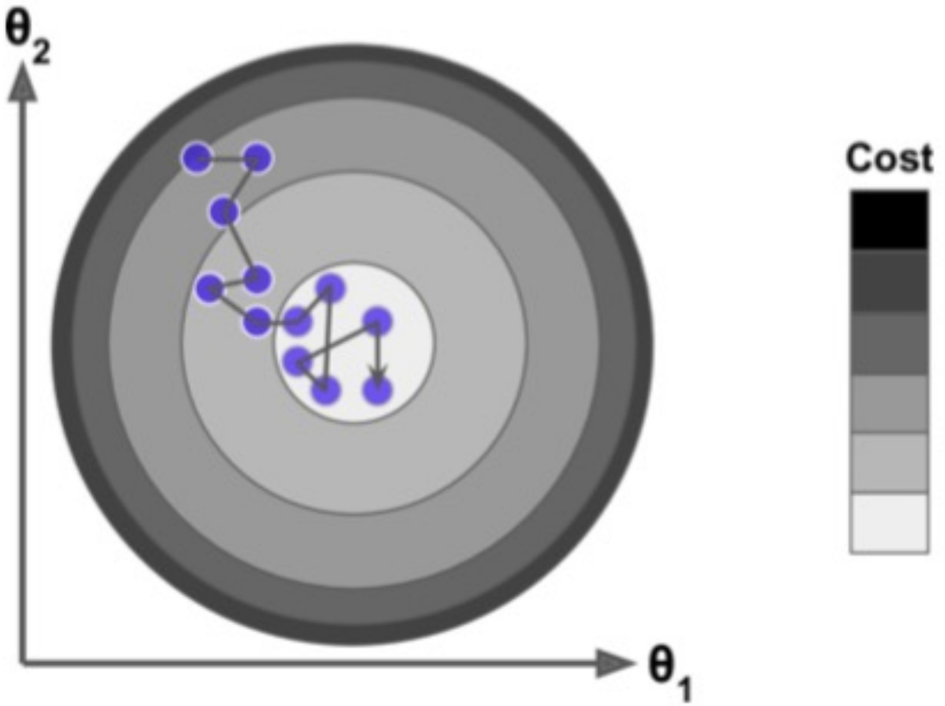


ML (Challenges of gradient descent)

Different versions of Gradient Descent

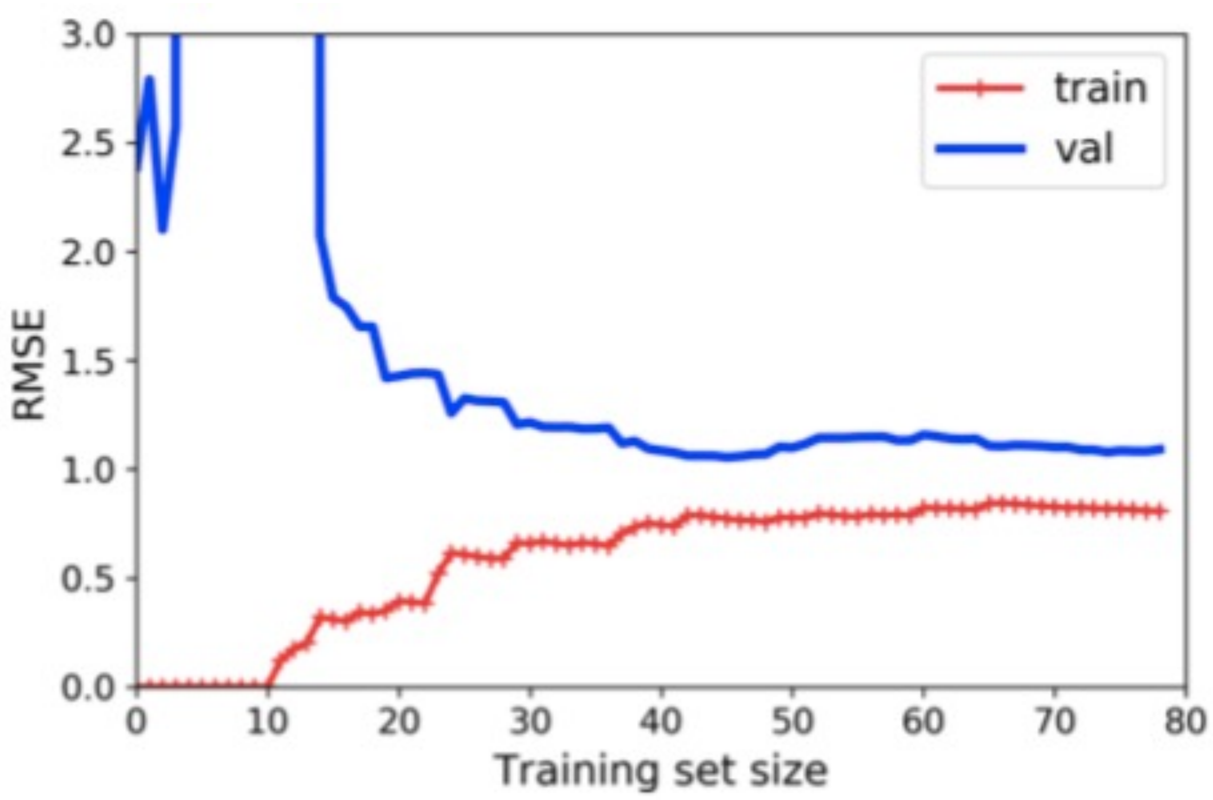
➢ **Stochastic Gradient descent**

- Choose a sample randomly,
- Update the parameters based on the randomly selected sample,



Solution:
Use mini-batch Gradient Descent

ML (Training Error- Validation Error):



ML (Regularization):

Problems with mean squared error:

$$\theta = (X^T X)^{-1} X^T y$$

- If the attributes are correlated, there is no unique solution(),
- If the number of attributes are more than the number of observation there is a risk of over-fitting.

To reduce the risk of over-fitting there are techniques to control the complexity of the model.

Control the increase in parameters θ

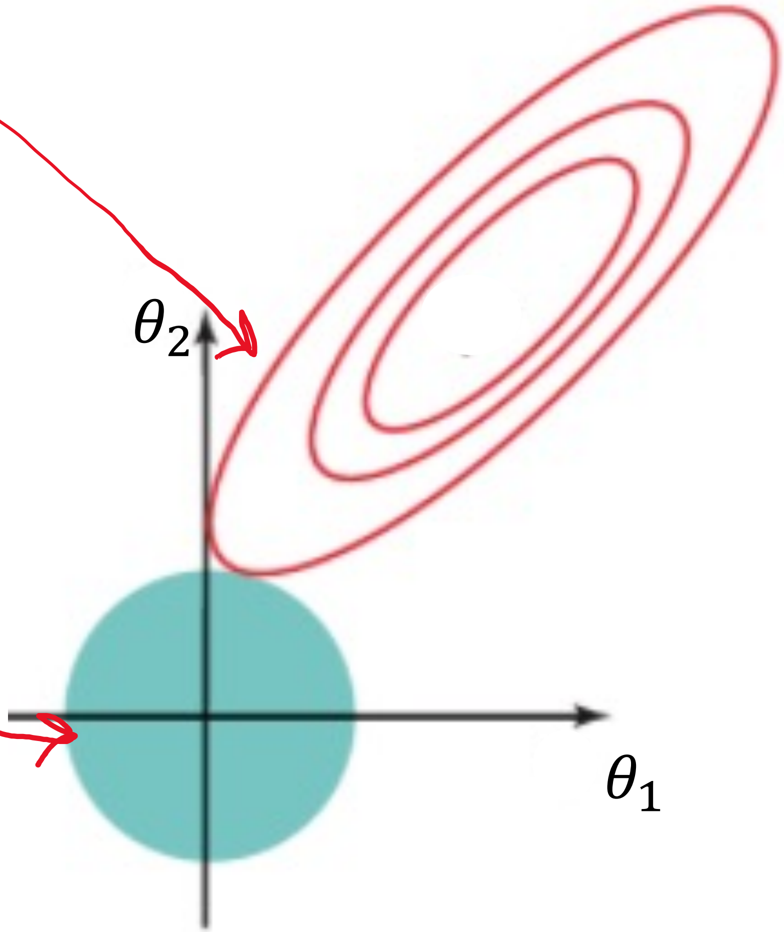
ML (Regularization-Lasso):

➢ Minimise Mean square error, but in addition take care of parameters not to be too big,

$$\theta = (\lambda I + X^T X)^{-1} X^T y$$

$$Error = MSE(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

$$\arg min_\theta\ MSE(\theta)\ s.t \sum_{i=1}^{n} \theta_i^2 \leq t$$

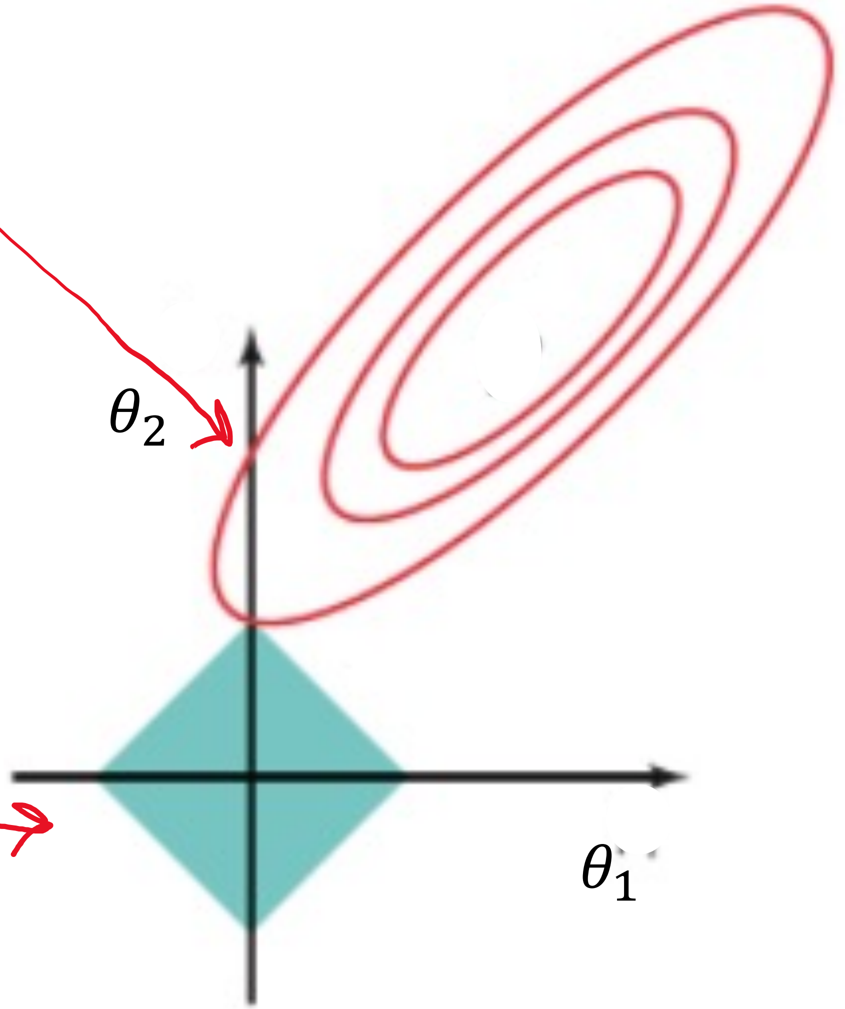


ML (Regularization-Lasso):

- Minimise Mean square error, but in addition take care of parameters not to be too big,

$$Error = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

$$\arg min_{\theta}\ MSE(\theta)\ s.t \sum_{i=1}^{n} |\theta_i| \le t$$



ML (Regularization-Implementation):

```python
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
ridge_reg = Ridge(alpha=1, solver="cholesky")
lasso_reg = Lasso(alpha = 0.1)
```

ML (Linear Regression on housing):

```
import pandas as pd
```

```
data = pd.read_csv('housing.csv')
```

```
data.head(2)
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY

ML (Linear Regression on housing):

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Numerical Variables

Categorical Variable

ML (Linear Regression on housing):

```
data.info()
```

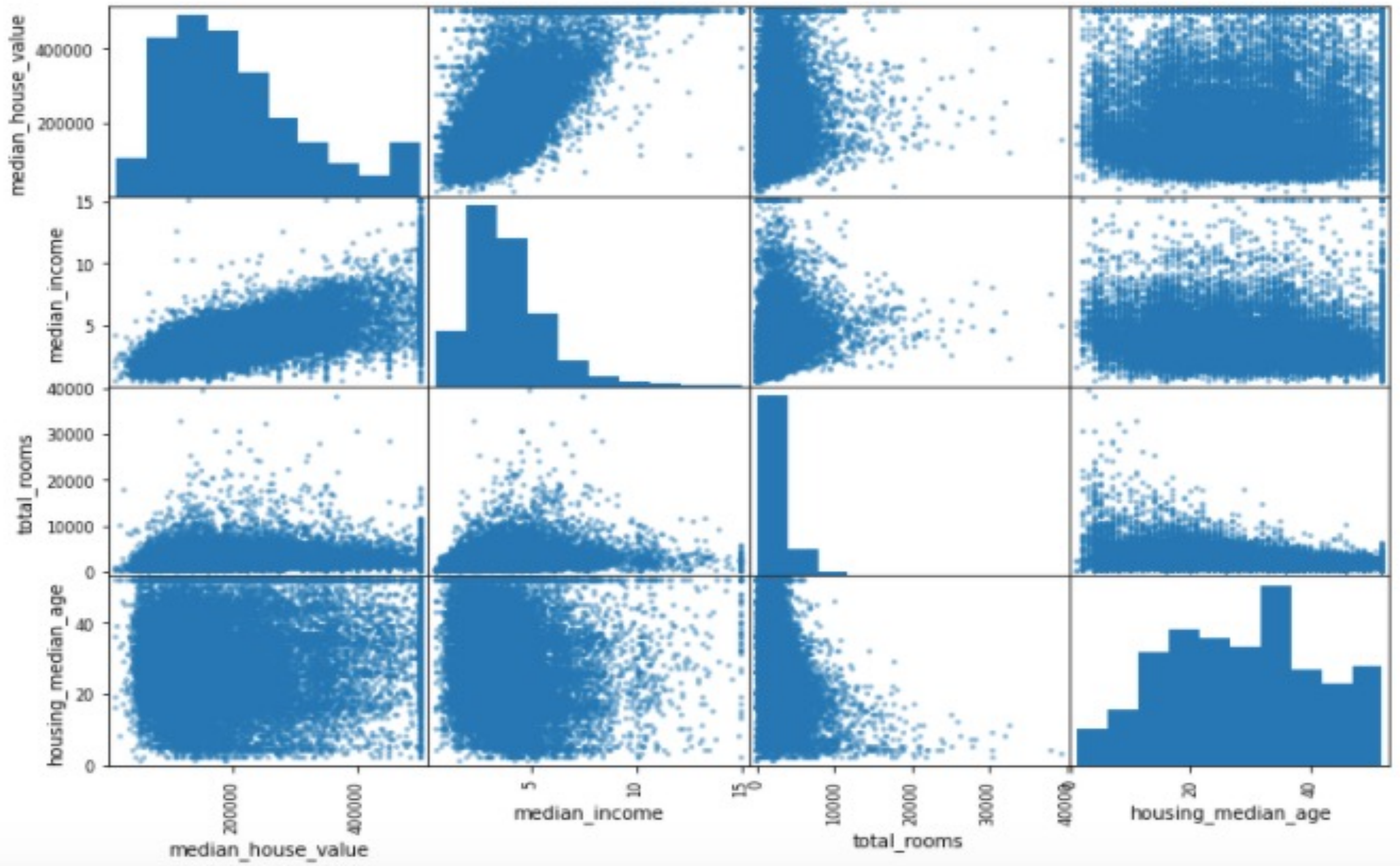
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object 
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Numerical Variables

Categorical Variable

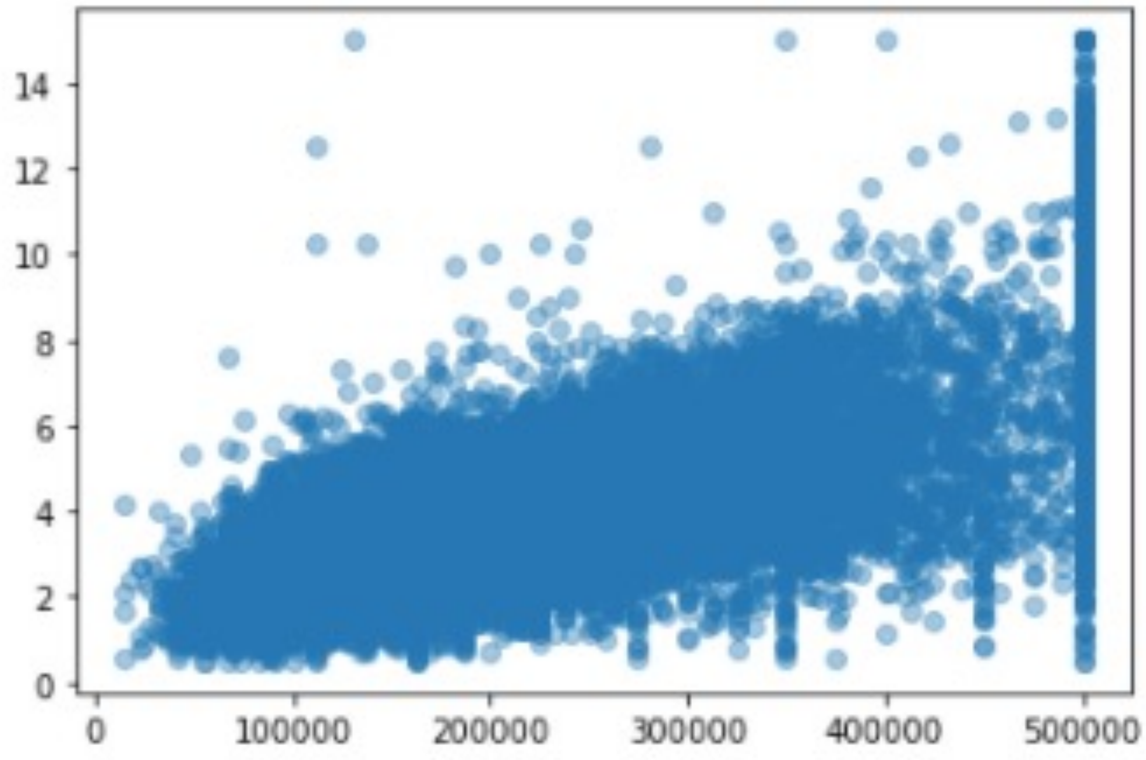
ML (Linear Regression on housing):

```
from matplotlib import pyplot as plt
from pandas.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(data[attributes], figsize = (12, 8 ))
plt.show()
```



ML (Linear Regression on housing):

```
plt.scatter(data['median_house_value'] , data['median_income'], alpha = 0.4)
plt.show()
```



ML (Linear Regression on housing):

```
corr_matrix = data.corr()
corr_matrix
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
longitude	1.000000	-0.924664	-0.108197	0.044568	0.069608	0.099773	0.055310	-0.015176	-0.045967
latitude	-0.924664	1.000000	0.011173	-0.036100	-0.066983	-0.108785	-0.071035	-0.079809	-0.144160
housing_median_age	-0.108197	0.011173	1.000000	-0.361262	-0.320451	-0.296244	-0.302916	-0.119034	0.105623
total_rooms	0.044568	-0.036100	-0.361262	1.000000	0.930380	0.857126	0.918484	0.198050	0.134153
total_bedrooms	0.069608	-0.066983	-0.320451	0.930380	1.000000	0.877747	0.979728	-0.007723	0.049686
population	0.099773	-0.108785	-0.296244	0.857126	0.877747	1.000000	0.907222	0.004834	-0.024650
households	0.055310	-0.071035	-0.302916	0.918484	0.979728	0.907222	1.000000	0.013033	0.065843
median_income	-0.015176	-0.079809	-0.119034	0.198050	-0.007723	0.004834	0.013033	1.000000	0.688075
median_house_value	-0.045967	-0.144160	0.105623	0.134153	0.049686	-0.024650	0.065843	0.688075	1.000000

ML (Linear Regression on housing):

```
corr_matrix['median_house_value'].sort_values(ascending = False)
```

```
median_house_value    1.000000
median_income         0.688075
total_rooms           0.134153
housing_median_age    0.105623
households            0.065843
total_bedrooms        0.049686
population           -0.024650
longitude            -0.045967
latitude             -0.144160
Name: median_house_value, dtype: float64
```

ML (Linear Regression on housing):

```
data["rooms_per_household"] = data["total_rooms"]/data["households"]
data["bedrooms_per_room"] = data["total_bedrooms"]/data["total_rooms"]
data["population_per_household"]=data["population"]/data["households"]
```

```
corr_matrix2 = data.corr()
corr_matrix2['median_house_value'].sort_values(ascending = False)
```

```
median_house_value          1.000000
median_income               0.688075
rooms_per_household         0.151948
total_rooms                 0.134153
housing_median_age          0.105623
households                  0.065843
total_bedrooms              0.049686
population_per_household   -0.023737
population                 -0.024650
longitude                  -0.045967
latitude                   -0.144160
bedrooms_per_room          -0.255880
Name: median_house_value, dtype: float64
```

ML (Linear Regression on housing):

Option 1:

Fill out missing values

```
df = data.copy()
```

```
median_nbbedrooms = df['total_bedrooms'].median()
```

```
median_nbbedroom_per_room = df['bedrooms_per_room']
```

```
df['total_bedrooms'].fillna(median_nbbedrooms , inplace = True)
```

```
df['bedrooms_per_room'].fillna(median_nbbedroom_per_room , inplace = True)
```

ML (Linear Regression on housing):

Option 2:

Fill out missing values

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
data.drop('ocean_proximity',axis = 1, inplace = True)
imputer.fit(data)
```

```
SimpleImputer(strategy='median')
```

```
imputer.statistics_
```

```
array([-1.18490000e+02,  3.42600000e+01,  2.90000000e+01,  2.12700000e+03,
        4.35000000e+02,  1.16600000e+03,  4.09000000e+02,  3.53480000e+00,
        1.79700000e+05,  5.22912879e+00,  2.03162434e-01,  2.81811565e+00])
```

```
X = imputer.transform(data)
```

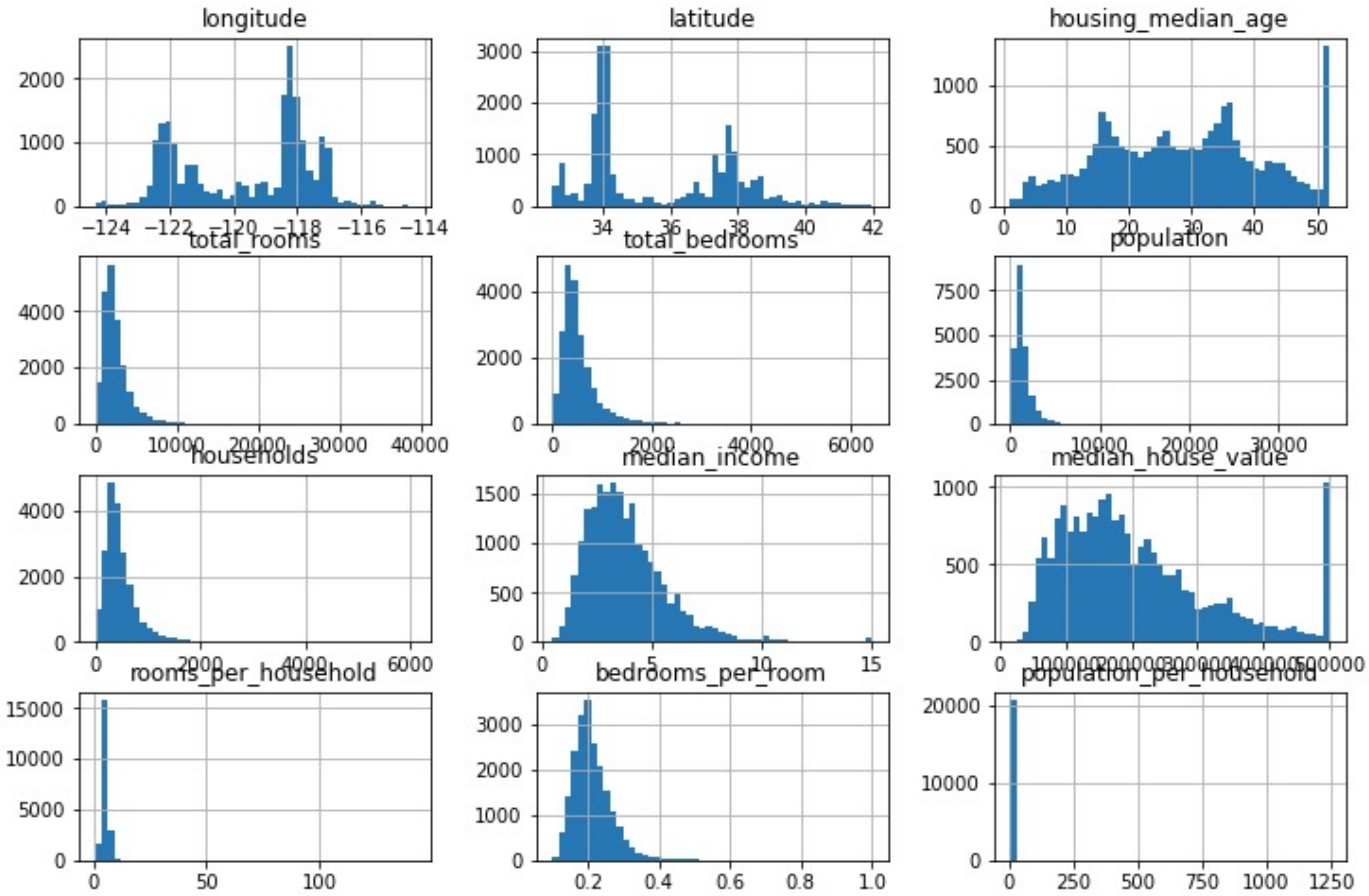
Numpy array

```
data = pd.DataFrame(X , columns = data.columns)
```

ML :

```
data.hist( bins = 50 , figsize = (12,8))
plt.show()
```

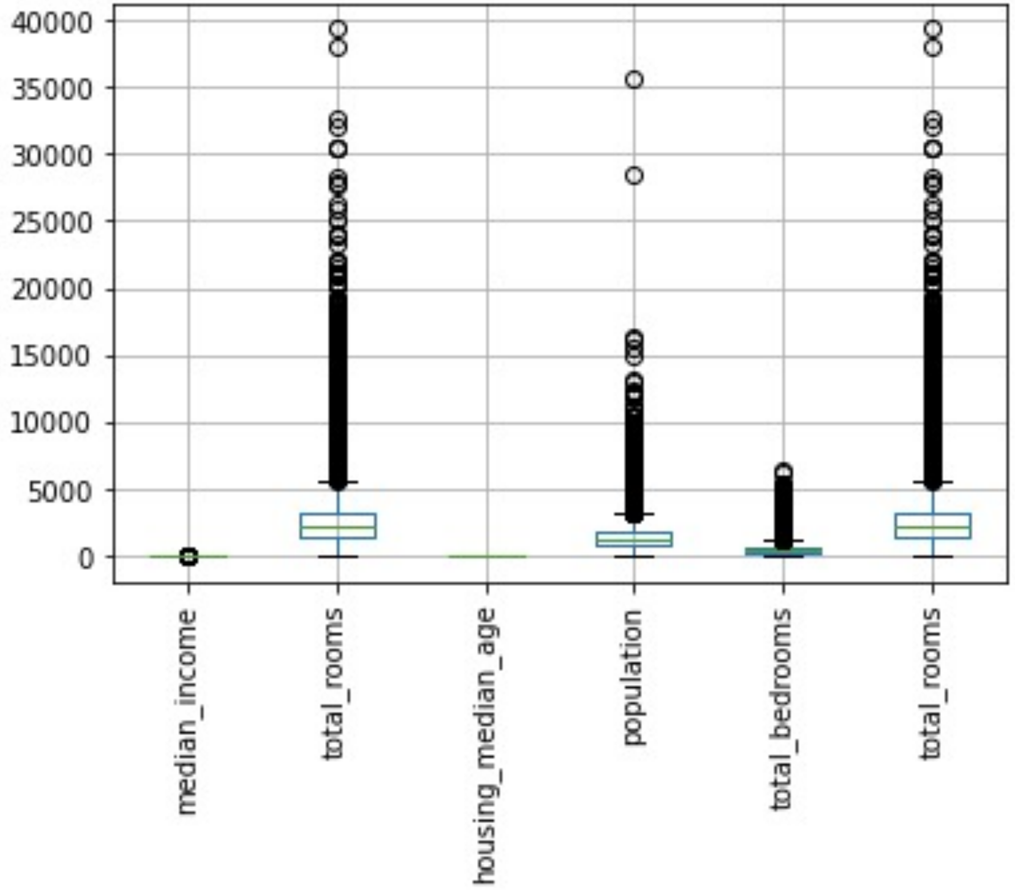
Distribution of data



ML :

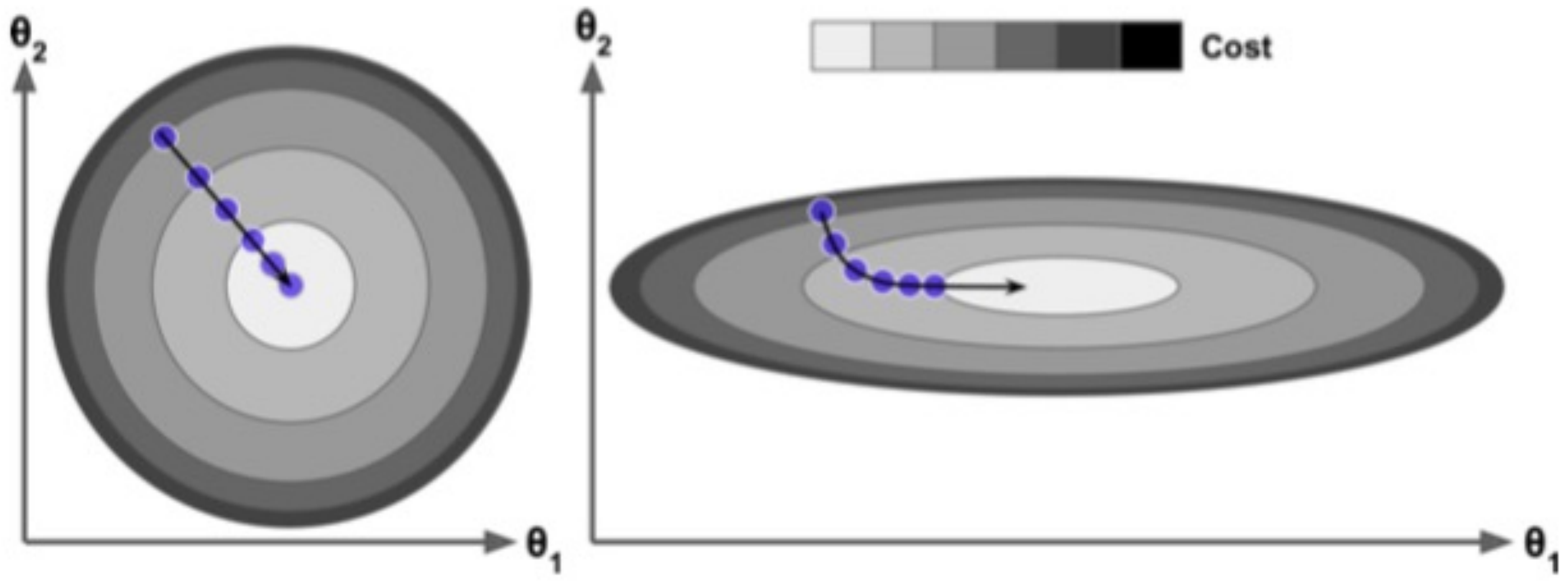
```
attributes = ["median_income", "total_rooms",
              "housing_median_age",'population','total_bedrooms','total_rooms']
data[attributes].boxplot()

plt.xticks(rotation = 90)
plt.show()
```

Scale of Data



ML :

Why scaling the data?



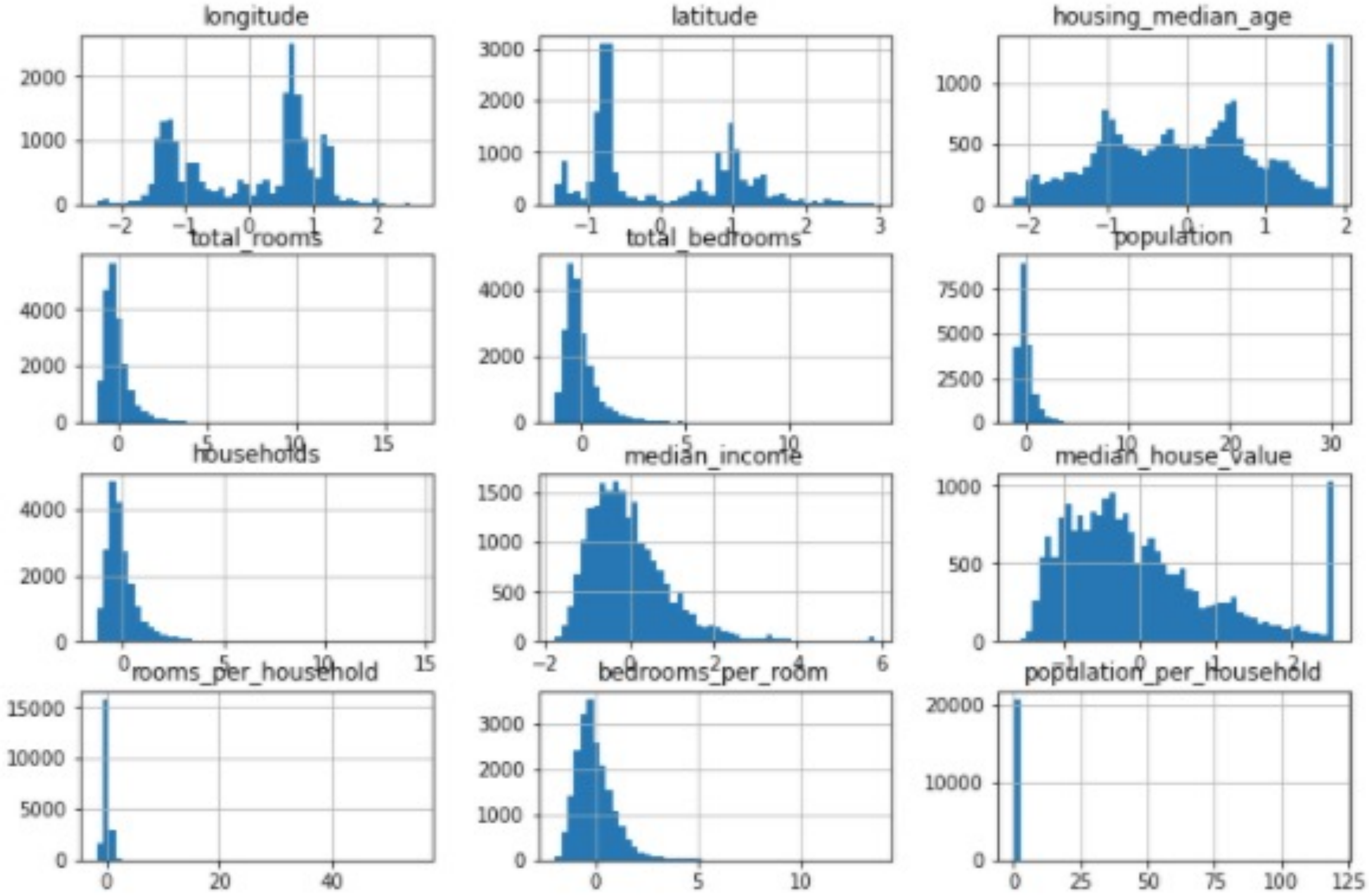
1. Faster to train the data,
2. More stable model (not too much sensitive to new samples).

ML :

```python
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled = scaler.fit_transform(data)
scaled_data = pd.DataFrame(scaled , columns = data.columns)
```

```python
scaled_data.hist(bins = 50 , figsize = (12,8))
plt.show()
```

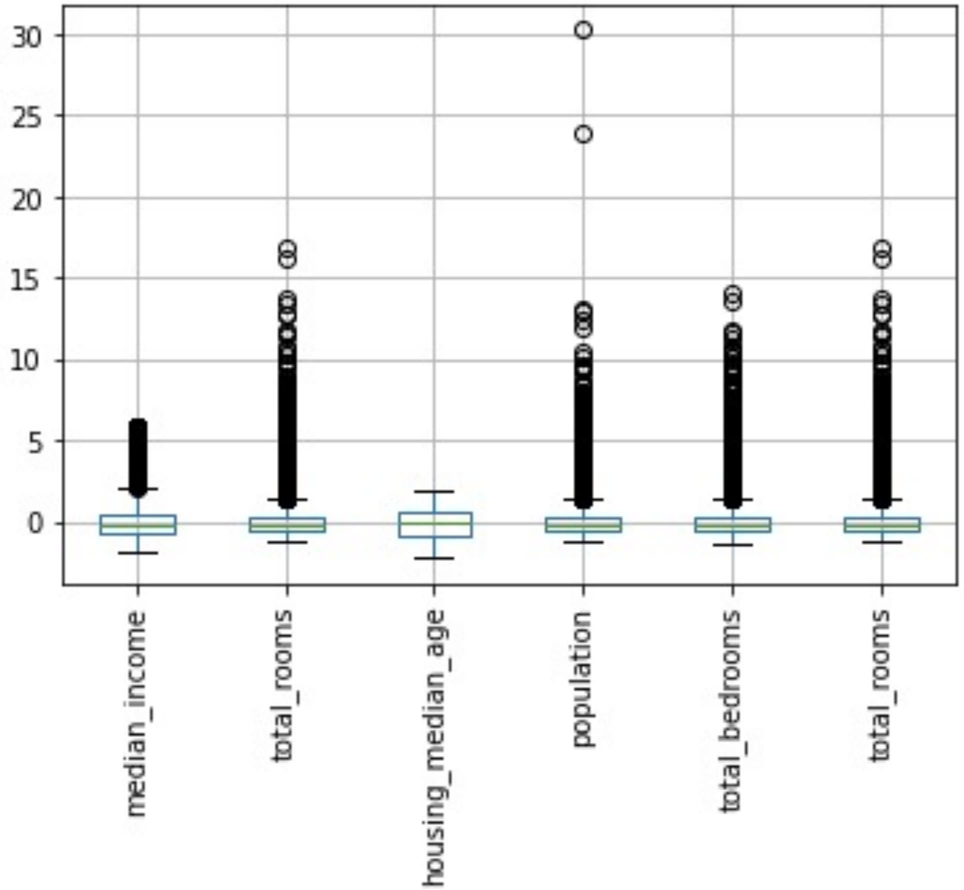
Standardization



ML :

```python
attributes = ["median_income", "total_rooms",
              "housing_median_age",'population','total_bedrooms','total_rooms']
scaled_data[attributes].boxplot()

plt.xticks(rotation = 90)
plt.show()
```

Does it helpful?



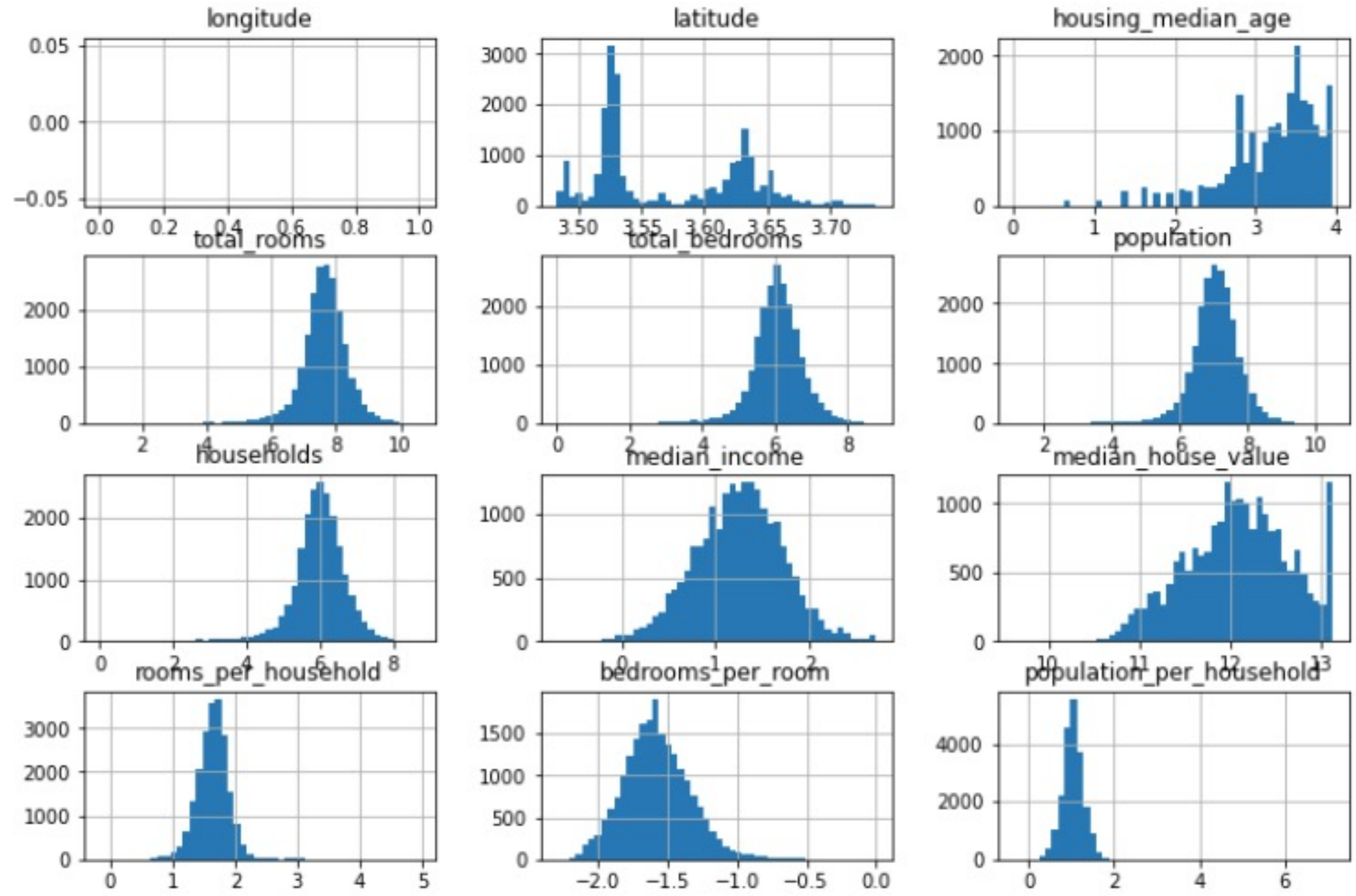
The problem with outliers…

ML :

```python
import numpy as np
data_log = np.log(data)
```

```python
data_log.hist(bins = 50 , figsize = (12,8))
plt.show()
```

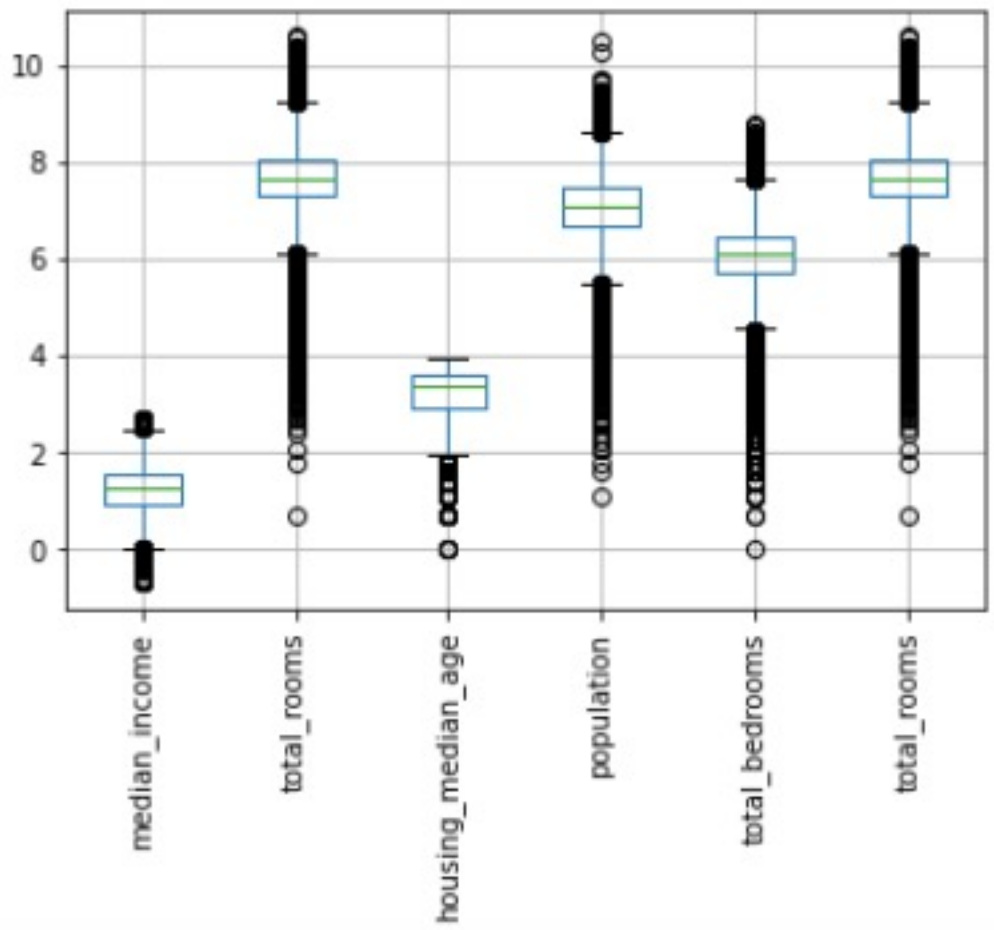
Log Transformation



ML :

```
attributes = ["median_income", "total_rooms",
              "housing_median_age",'population','total_bedrooms','total_rooms']
data_log[attributes].boxplot()
plt.xticks(rotation = 90)
plt.show()
```

Log Transformation



ML :

Pre-processing

All the steps including data acuisition and data preparation like handling null values, data transformation, standardization, encoding, … are called pre-processing,

ML :

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
features = scaled_data.drop('median_house_value' ,axis = 1)
target = scaled_data[['median_house_value']]

lin_model = LinearRegression()
```

Training loss:

```
from sklearn.metrics import mean_squared_error
lin_model.fit(features,target)
predictions = lin_model.predict(train)
mse = mean_squared_error(target, predictions )
rmse = np.sqrt(mse)
rmse
```

```
0.5945904829245694
```

ML :

Validation loss:

```
scores = cross_val_score(lin_model,features, target,
                         scoring = "neg_mean_squared_error", cv = 10)
```

```
rmse = np.sqrt(-scores)
rmse
```

```
array([0.56986624, 0.53786391, 0.74194455, 0.49912052, 0.69925988,
       0.6093961 , 0.46481939, 0.73728758, 0.67160372, 0.48182607])
```

```
rmse.mean()
```

```
0.6012987948287257
```

ML :

Validation loss:

```
scores = cross_val_score(lin_model,features, target,
                         scoring = "neg_mean_squared_error", cv = 10)
```

```
rmse = np.sqrt(-scores)
rmse
```

```
array([0.56986624, 0.53786391, 0.74194455, 0.49912052, 0.69925988,
       0.6093961 , 0.46481939, 0.73728758, 0.67160372, 0.48182607])
```

```
rmse.mean()
```

```
0.6012987948287257
```

What is the meaning of low amount training loss and relatively high value of validation loss?

ML :

Example: learning life expectancy based on gdp per capita by linear regression.

Underfitting Example

	gdp per capita	life expectancy	population	color
0	974.580338	43.828	31.889923	red
1	5937.029526	76.423	3.600523	green
2	6223.367465	72.301	33.333216	blue
3	4797.231267	42.731	12.420476	blue
4	12779.379640	75.320	40.301927	yellow

Nb samples = 142

ML :

Underfitting Example

Example: learning life expectancy based on gdp per capita by linear regression.

```
features = data[['gdp per capita']]
target = data[['life expectancy']]
```

```
model = LinearRegression()
model.fit(features , target)
predicts = model.predict(features)
```

```
mse = mean_squared_error(target, predicts)
rmse = np.sqrt(mse)
rmse
```

```
8.835757281743057
```

```
scores = cross_val_score(model, features, target, scoring = "neg_mean_squared_error" , cv = 10)
rmse = np.sqrt(-scores)
rmse.mean()
```

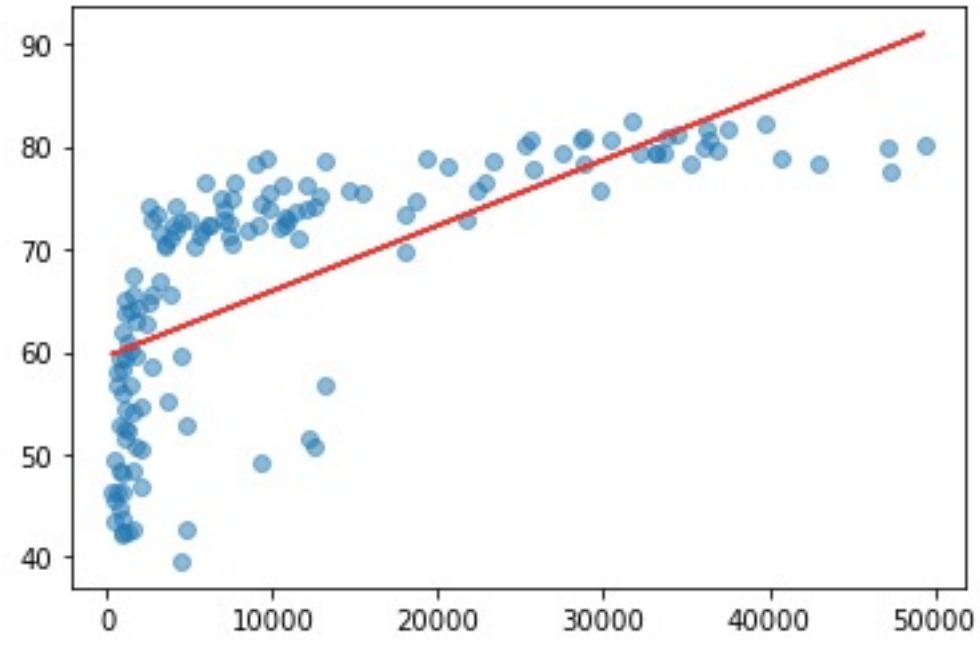
```
8.79033632355108
```

High error values of training and loss!

ML :

Example: learning life expectancy based on gdp per capita by linear regression.

```
plt.scatter(data[['gdp per capita']] , data[['life expectancy']] , alpha = 0.5)
tetha_0 = model.intercept_[0]
tetha_1 = model.coef_[0]
y = tetha_0 + tetha_1 * np.array(data['gdp per capita'])
plt.plot(data['gdp per capita'] , y , c = 'r')
plt.show()
```

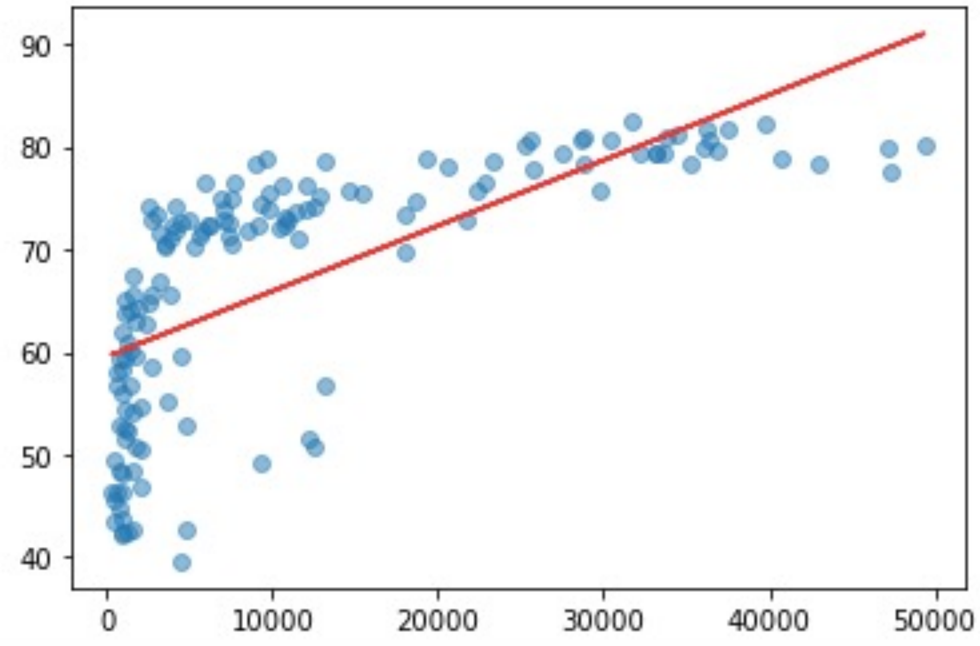


The model is too simple to be trained for the dataset.

ML :

Example: learning life expectancy based on gdp per capita by linear regression.

```
plt.scatter(data[['gdp per capita']] , data[['life expectancy']] , alpha = 0.5)
tetha_0 = model.intercept_[0]
tetha_1 = model.coef_[0]
y = tetha_0 + tetha_1 * np.array(data['gdp per capita'])
plt.plot(data['gdp per capita'] , y , c = 'r')
plt.show()
```



The model is too simple to be trained for the dataset.

Solutions to underfitting:

1. Choose a more complex learning model,
2. Use more features,

ML (Train-Validation-Test) :

- **We train a model to do predictions,**
- **A model performs well if it do the write predictions on unseen data,**
- **Therefore, Prevent Data Leakage during training (How?)**
- **Split data to train (80%) and test (20%).**

1. Given a data set, split the data to representative train set and test set,
2. Do data cleaning and pre-processing on features (on both train and test)
3. Train different models on the train set and find the best one by evaluating their training and validation errors,
4. Do prediction using the best model on the test set,
5. Evaluate the performance of final model on test data (if possible)

ML (Train-Validation-Test) :

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY

Housing dataset:

```
from sklearn.model_selection import train_test_split
train_set ,test_set = train_test_split(data ,test_size = 0.2 , random_state = 42)
```

```
data.shape
```

```
(20640, 10)
```

```
train_set.shape
```

```
(16512, 10)
```

```
test_set.shape
```

```
(4128, 10)
```

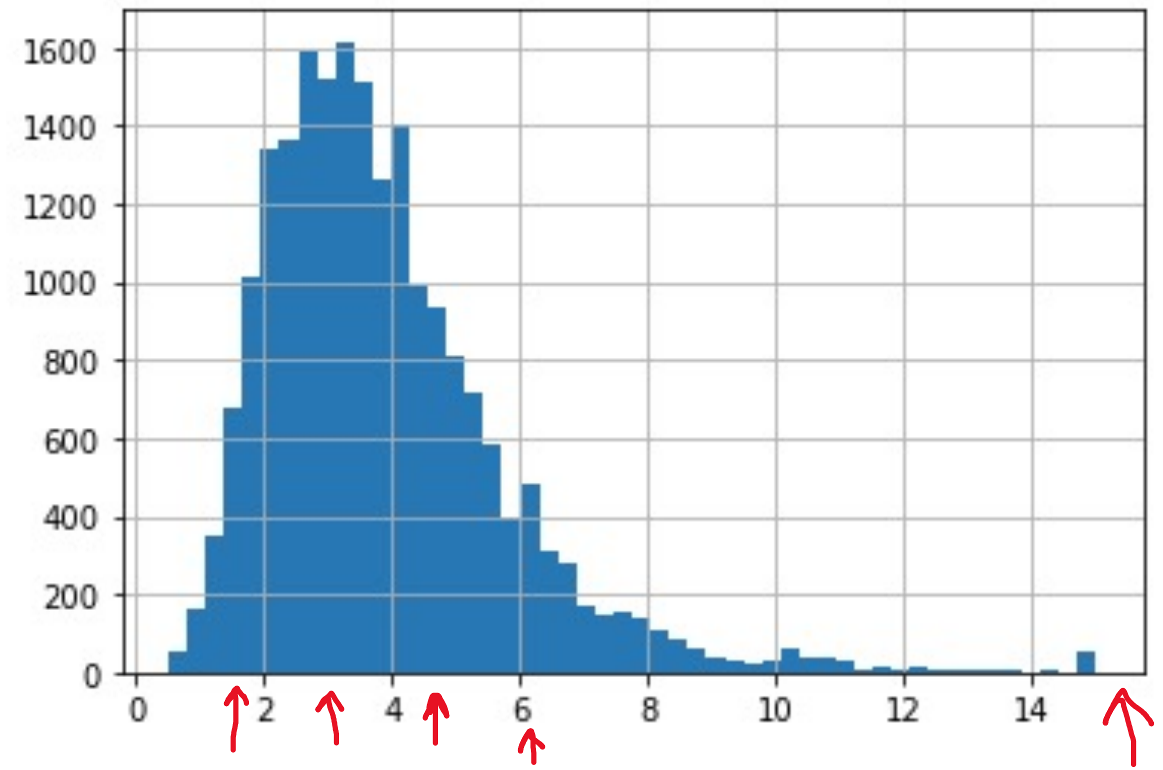
ML (Train-Validation-Test) :

➢ Based on correlation values we know that median income is strongly related to the median house value,

```
data['median_income'].hist(bins = 50)
```

<AxesSubplot:>

Test Set Being Representative:



ML (Train-Validation-Test) :

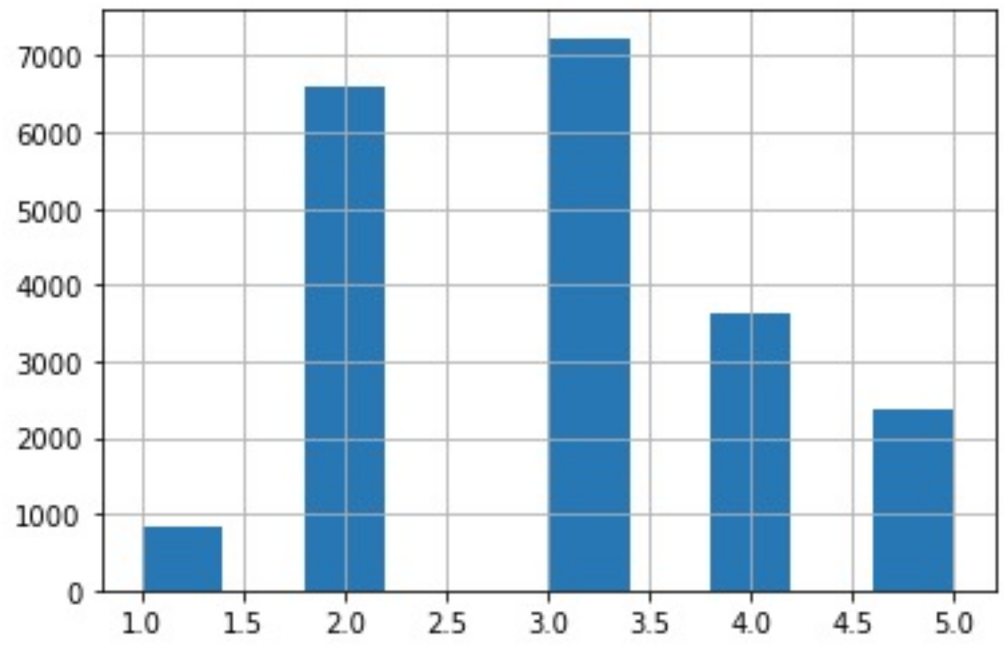
➢ Based on correlation values we know that median income is strongly related to the median house value,

Test Set Being Representative:

```
data['income_cat'] = pd.cut( data['median_income'],
                            bins = [0,1.5,3.0,4.5,6, np.inf], labels = [1,2,3,4,5])
```

```
data['income_cat'].hist()
```

<AxesSubplot:>



ML (Train-Validation-Test) :

```
from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits = 1, test_size = 0.2 , random_state = 42)

for (train_index, test_index) in split.split(data , data['income_cat']):
    strat_train_set = data.loc[train_index]
    strat_test_set = data.loc[test_index]
```

Test Set Being Representative:

```
strat_train_set['income_cat'].value_counts() / len(strat_train_set)
```

```
3    0.350594
2    0.318859
4    0.176296
5    0.114402
1    0.039850
Name: income_cat, dtype: float64
```

```
strat_test_set['income_cat'].value_counts()/len(strat_test_set)
```

```
3    0.350533
2    0.318798
4    0.176357
5    0.114583
1    0.039729
Name: income_cat, dtype: float64
```

ML (Train-Validation-Test) :

Some notes

1. If you have have filled the null values in training set with statistic measures(median, mean, mode), fill the null values in the test set with the corresponding values in the training set,

2. Use the same Transformation technique on both train and test,

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

num_pipeline = Pipeline([('imputer' , SimpleImputer(strategy = 'median'))
                        ,('std_scalar', StandardScaler())])
```

```
train_np = num_pipeline.fit_transform(strat_train_set)
train_set = pd.DataFrame(train_np , columns = strat_train_set.columns)
```

```
test_np = num_pipeline.fit_transform(strat_test_set)
test_set = pd.DataFrame(test_np,columns = strat_test_set.columns)
```

ML (Binary Classification problem) :

- The target values are dochotomous (They only have two values 0 and 1 or -1 and 1)

- Multi-classification: The target values are finite discrete (0,1,2,...,9)

- Difference weth regression problems: we need some extra part to convert the output of regression to a specified discrete range.

ML (Classification-MNIST) :

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
```

```
mnist.keys()
```

```
dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_names', 'DESCR', 'details', 'url'])
```

```
X, y = mnist["data"], mnist["target"]
```

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

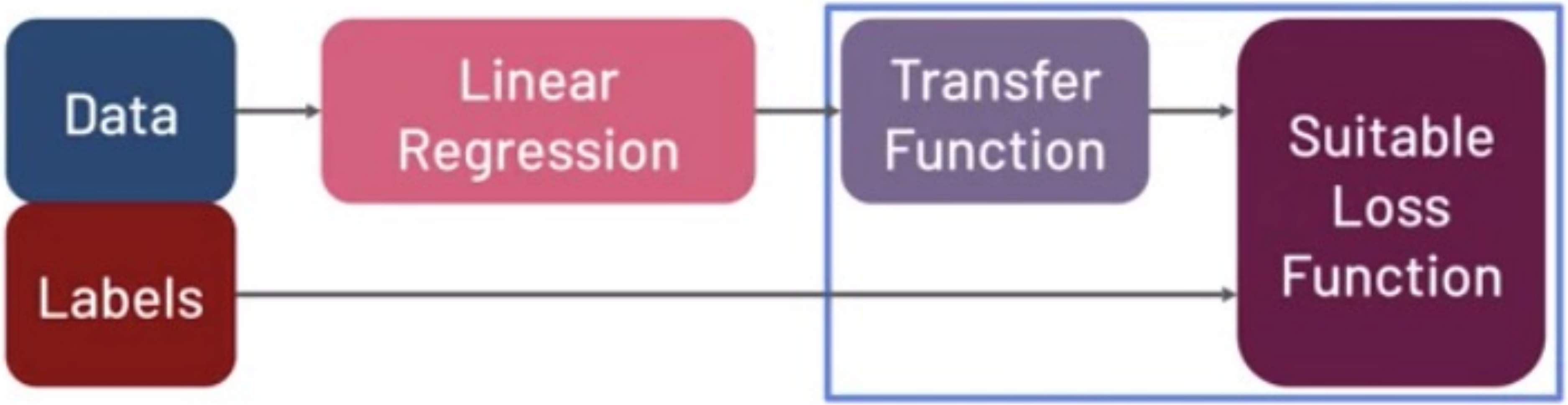
ML (Classification-MNIST) :

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

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(some_digit_image, cmap = mpl.cm.binary, interpolation="nearest")
plt.axis("off")
plt.show()
```

ML (Binary-Classification-MNIST) :

```
y_train_5 = (y_train == 5) # True for all 5s, False for all other digits.
y_test_5 = (y_test == 5)
```

Sign function(more appropriately logistic function)



Mean square error is not always the best

ML (Binary-Classification-MNIST) :

MSE : $(\theta^T X - y)^2$

{1 , 0}

A total mis-classification might minimize mean-squared-error.

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train_5)
```

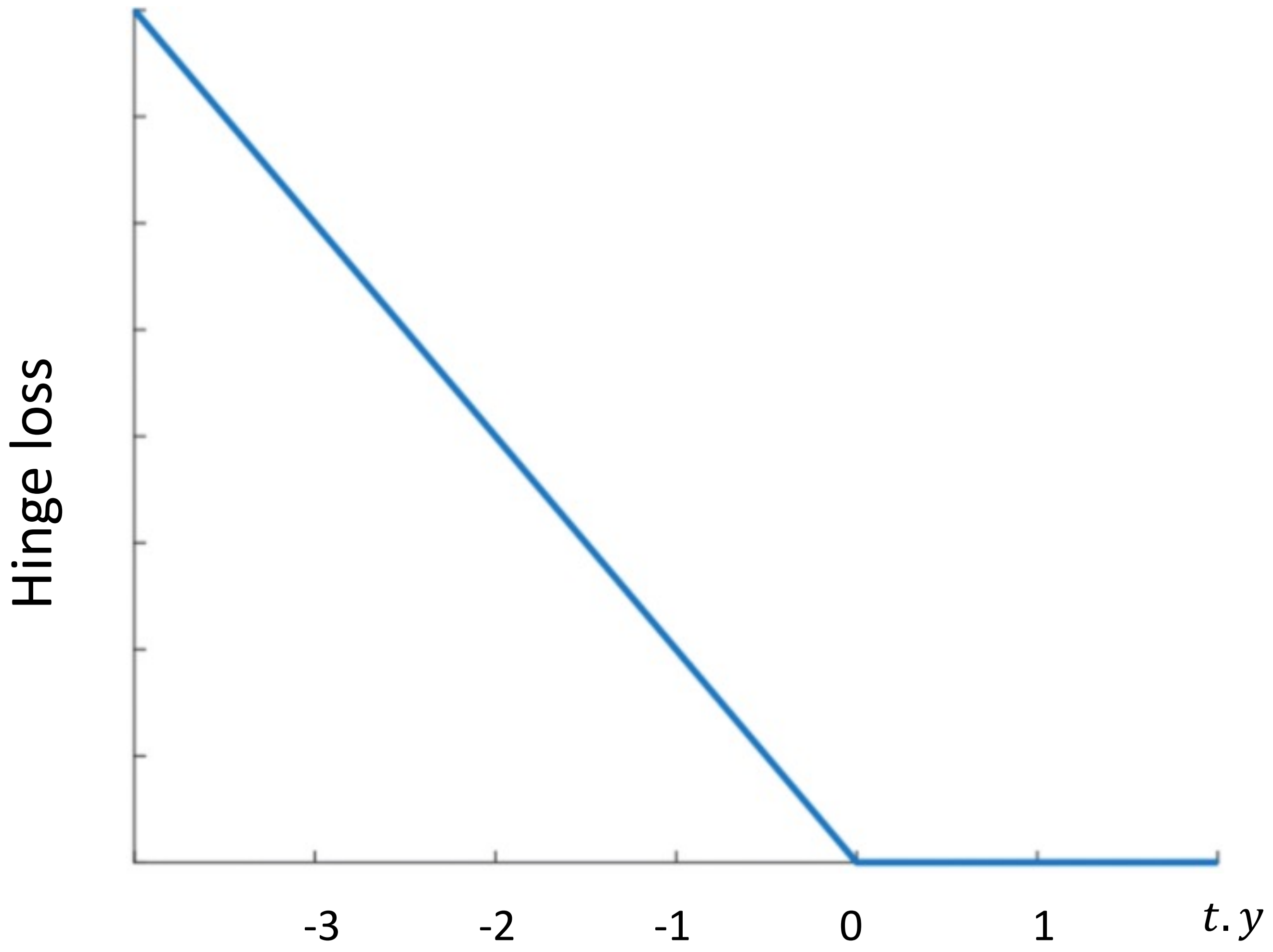
```
/Users/hosseinkhani/miniconda3/envs/dataenv/lib/python3.9/site-packages/sklearn/linear_model/_logistic.py:763: Conver
genceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

ML (Binary-Classification-MNIST) :

Stochastic Gradient Descent Classifier:

1. Minimize the Hinge loss ($\max(0, 1 - t.y)$)
2. Do the minimization using stochastic gradient descent algorithm

ML (Binary-Classification-MNIST) :



ML (Binary-Classification-MNIST) :

```
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

```
SGDClassifier(random_state=42)
```

```
sgd_clf.predict([some_digit])
```

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

Hand-written 5

ML (Binary-Classification-MNIST) :

```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
array([0.95035, 0.96035, 0.9604 ])
```

Is this performance reliable?

ML (Binary-Classification-MNIST) :

```
from sklearn.base import BaseEstimator
class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

```
never_5_clf = Never5Classifier()
```

```
from sklearn.model_selection import cross_val_score
cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
array([0.91125, 0.90855, 0.90915])
```

Imbalanced Classification problem

ML (Binary-Classification-Metrics) :

```
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

```
y_train_pred
```

```
array([ True, False, False, ...,  True, False, False])
```

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_train_5, y_train_pred)
```

```
array([[53892,   687],
       [ 1891,  3530]])
```

negative class

positive class

actual class

predicted class

687 false positives

1891 false negatives

ML (Binary-Classification-Metrics) :

```
y_train_perfect_predictions = y_train_5 # pretend we reached perfection
confusion_matrix(y_train_5, y_train_perfect_predictions)
```

```
array([[54579,     0],
       [    0,  5421]])
```

ML (Binary-Classification-MNIST) :

precision = $\frac{TP}{TP+FP}$ (true positive rate)

recall = $\frac{TP}{TP+FN}$ (false negative rate)

```
from sklearn.metrics import precision_score , recall_score
precision = precision_score(y_train_5 , y_train_pred)
recall = recall_score(y_train_5 , y_train_pred)
print("The precision score is {} and the recall score is {}".format(precision , recall))
```

```
The precision score is 0.8370879772350012 and the recall score is 0.6511713705958311
```

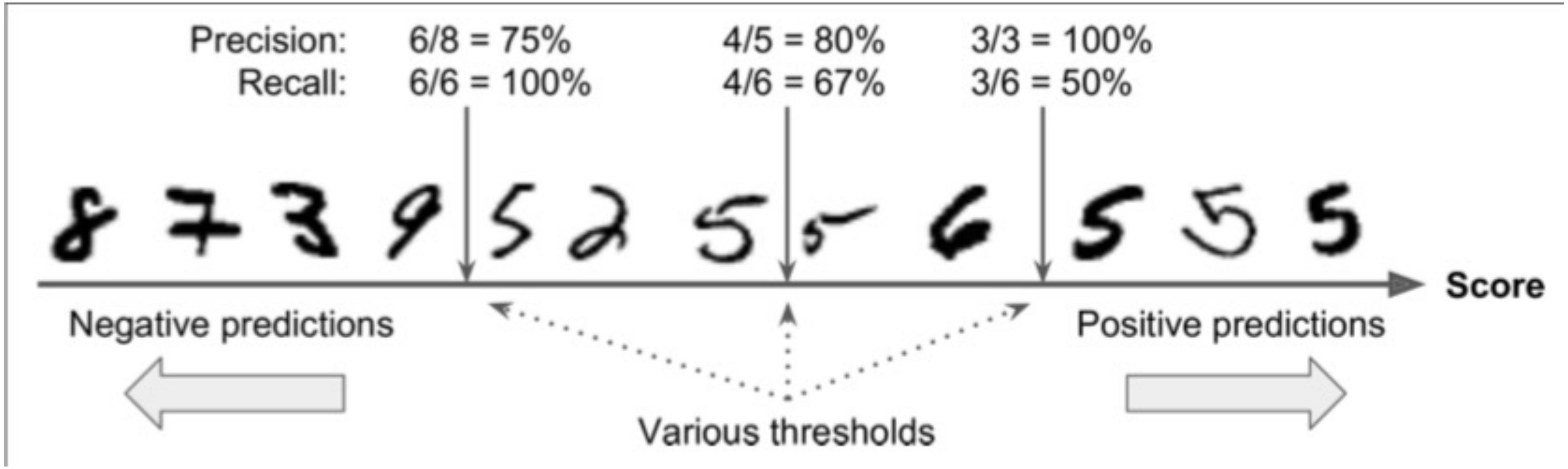
ML (Binary-Classification-MNIST) :

$$F1_{score} = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2 \times \frac{precision \times recall}{precision + recall}$$

```
from sklearn.metrics import f1_score
f1_score(y_train_5, y_train_pred)
```

```
0.7325171197343846
```

ML (Binary-Classification-Metrics) :




```
y_scores = sgd_clf.decision_function([some_digit])
y_scores
```

```
array([2164.22030239])
```

default threshold is 0

```
prediction = (y_scores > 0)
prediction
```

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

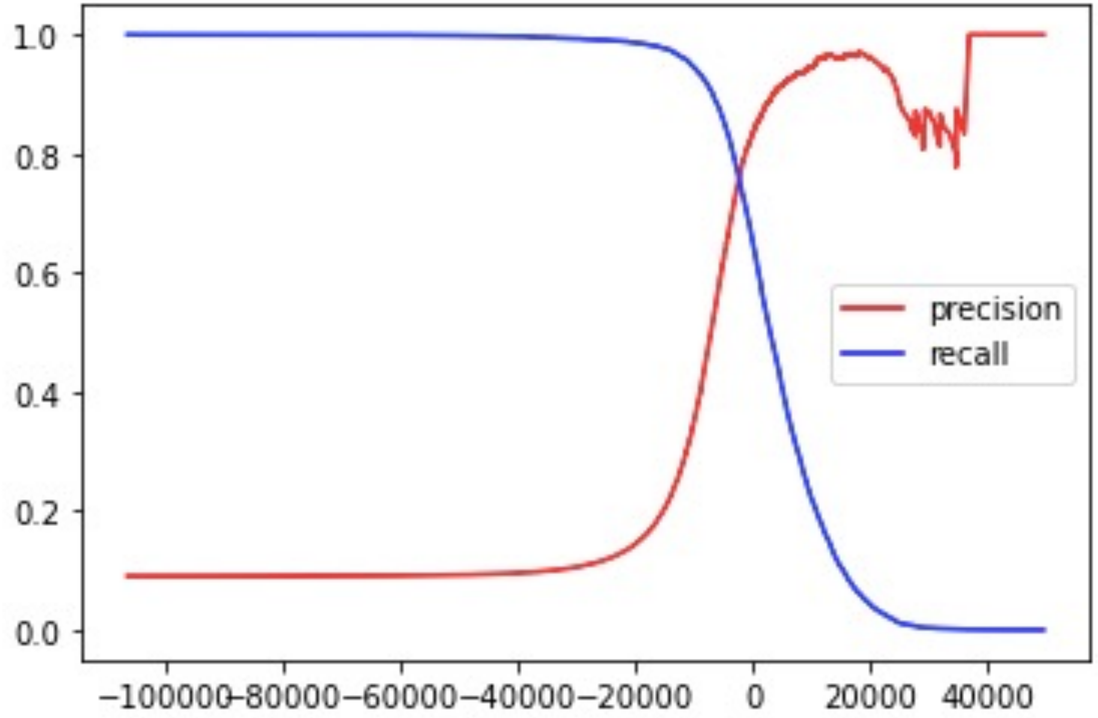
```
threshold = 8000
prediction = (y_scores > threshold)
prediction
```

```
array([False])
```

ML (Binary-Classification-Metrics) :

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

```
import matplotlib.pyplot as plt
plt.plot(thresholds, precisions[:-1] , c='r', label = 'precision')
plt.plot(thresholds, recalls[:-1] , c = 'b' , label = 'recall')
plt.legend()
plt.show()
```



ML (Binary-Classification - ROC) :

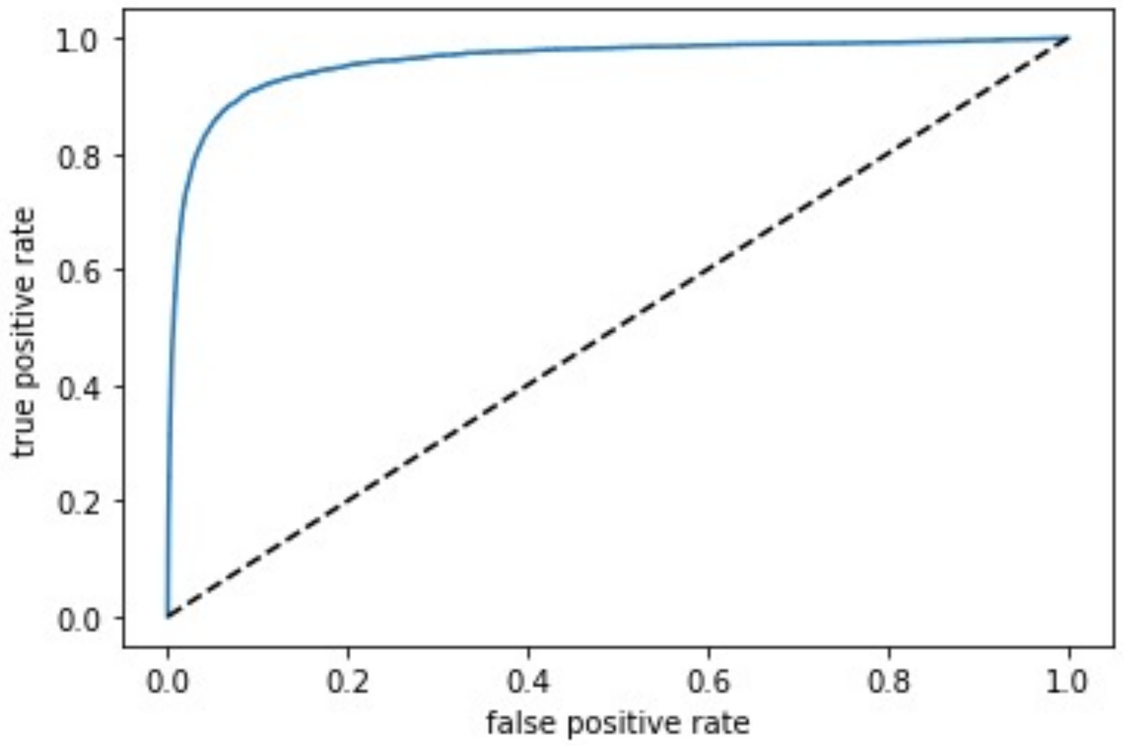
fpr : false positive rate

tpr : true positive rate

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

```
plt.plot(fpr , tpr)
plt.xlabel('false positive rate')
plt.ylabel('true positive rate')
plt.plot([0,1],[0,1],'k--')
```

```
[<matplotlib.lines.Line2D at 0x1237dc4f0>]
```



ML (Binary-Classification - AUC) :

Area Under the Curve more close to 1, better

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
```

0.9604938554008616

ML (Multi - Classification) :

Two Approach:

1. *one versus all* : train 10 classifiers,

- class 0-detector (distinguish zeros from non-zeros)
- class 1-detector (distinguish one from non-ones)
- …

➢ *prediction*: the predicted class of an image is the class with higher score.

2. *one versus one*: for any two class train a classifier

- 0's versus 1's
- 0's versus 2's
- …
- If there are n classes in general, we need $\frac{n(n-1)}{2}$ classifiers (they should be trained on smaller portions of data).

➢ *prediction*: the class assosiated to an image is one that win more duels!

ML (Multi - Classification) :

- Sklearn detects when a binary classifier is used for a multi-classification problem.

```
sgd_clf.fit(X_train, y_train)
```

```
SGDClassifier(random_state=42)
```

- In this case it automatically uses *one versus all* technique

```
sgd_clf.decision_function([some_digit])
```

Highest score

```
array([[-31893.03095419, -34419.69069632,  -9530.63950739,
          1823.73154031, -22320.14822878,  -1385.80478895,
        -26188.91070951, -16147.51323997,  -4604.35491274,
        -12050.767298  ]])
```

ML (Multi - Classification) :

Some classification problems perform purely on large datasets, in such cases it might be more convenient to use *one versus one* technique.

```
from sklearn.multiclass import OneVsOneClassifier
ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
ovo_clf.fit(X_train, y_train)
ovo_clf.predict([some_digit])
```

```
array([5], dtype=uint8)
```

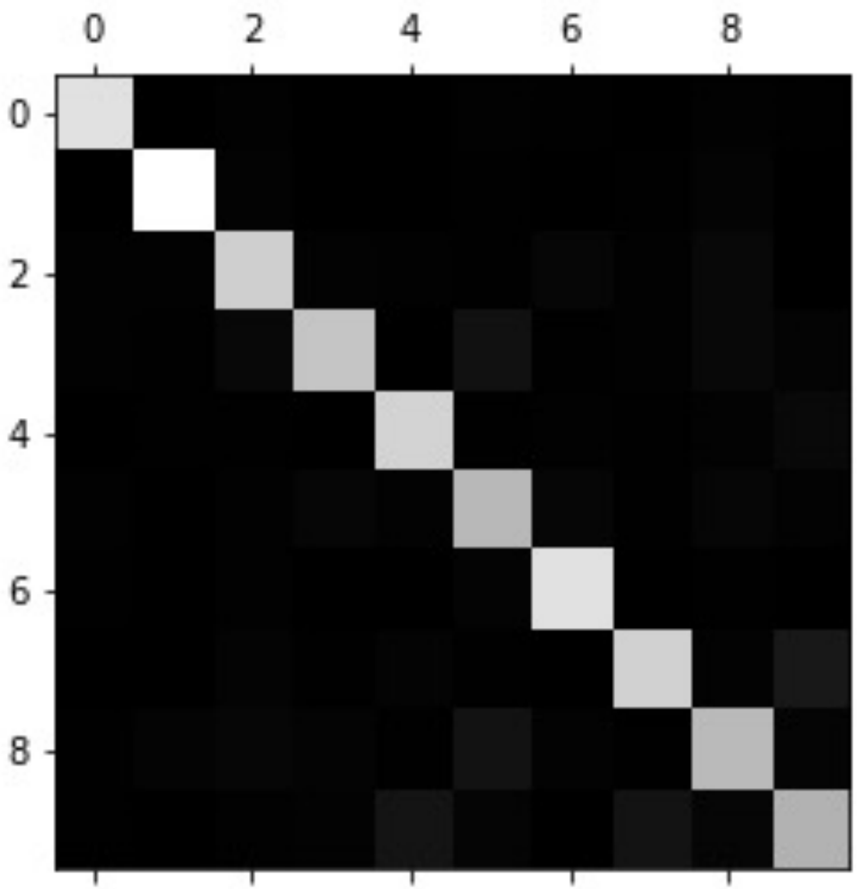
ML (Multi – Classification - Metrics) :

```
conf_mx = confusion_matrix(y_train, y_train_pred)
conf_mx
```

```
array([[5635,    0,   61,   10,   16,   50,   46,    7,   66,   32],
       [   3, 6393,   95,   21,   16,   47,   15,   27,  109,   16],
       [  72,   56, 5174,   89,   69,   39,  163,   66,  212,   18],
       [  58,   32,  217, 4941,   23,  441,   32,   56,  216,  115],
       [  11,   26,   46,    6, 5298,   26,   73,   32,   87,  237],
       [  68,   23,   58,  150,   83, 4606,  174,   26,  152,   81],
       [  40,   13,   56,    6,   22,  113, 5625,    5,   36,    2],
       [  23,   24,  103,   36,  124,   40,   10, 5228,   75,  602],
       [  40,  101,  158,  122,   49,  457,   77,   35, 4666,  146],
       [  33,   18,   66,   83,  515,  127,    4,  485,  166, 4452]])
```

ML (Multi – Classification - Metrics) :

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



ML (Multi – Classification - Metrics) :

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
np.fill_diagonal(norm_conf_mx, 0)
```

```
plt.matshow(norm_conf_mx , cmap = plt.cm.gray)
plt.show()
```