Permutation codes

```
initial_permutation = [2, 6, 3, 1, 4, 8, 5, 7]
expansion_permutation = [4, 1, 2, 3, 2, 3, 4, 1]
P4 = [2, 4, 3, 1]
inv_initial_permutation = [4, 1, 3, 5, 7, 2, 8, 6]
P10 = [3, 5, 2, 7, 4, 10, 1, 9, 8, 6]
P8 = [6, 3, 7, 4, 8, 5, 10, 9]
```

```
S0 = [[1,0,3,2],
      [3,2,1,0],
      [0,2,1,3],
      [3,1,3,2]]

S1=  [[0,1,2,3],
      [2,0,1,3],
      [3,0,1,0],
      [2,1,0,3]]
```

Utility functions

Permutation

```
def permutation(x, p):
  res = []
  for ind in p:
    res.append(x[ind-1])
  return res
```

To undo cell deletion use ⌘/Ctrl+M Z or the 'Undo' option in the 'Edit' menu ×

```
def left_shift(x, pos = 1):
  pos = pos % len(x)
  return x[pos:] + x[:pos]
```

XOR

```
def xor(x, y):
  res = []
  for i in range(len(x)):
```

```
    res.append(x[i]^y[i])
  return res
```

sbox

```
def sbox_op(x, s):
  r = int(f'{x[0]}{x[3]}', 2)
  c = int(f'{x[1]}{x[2]}', 2)
  val = s[r][c]

  if val == 0: return [0, 0]
  elif val == 1: return [0, 1]
  elif val == 2: return [1, 0]
  else: return [1, 1]
```

fk

- split into l and r (4bits each)
- expansion permutation on r (r_ep = 8 bits)
- xor with key1
- split into r_ep_l, e_ep_r and s box them
- concatenate into r_ep_mod (4bits)
- P4 permutation r_ep_mod' (4 bits)
- res = xor l and r_ep_mod'
- pass res and r to next stage(swap)

```
def f_k(init, key):
  l = init[:4]
  r = init[4:]
  r_ep = permutation(r, expansion_permutation)
  r_ep_xor = xor(r_ep, key)
                                                      [4:], S1)
```

To undo cell deletion use ⌘/Ctrl+M Z or the 'Undo' option in the 'Edit' menu

```
  res = xor(l, r_ep_mod2)
  return res + r
```

Key Generation

- Permutation of K10
- Split K10
- Left Shift(1) K10_1 and K10_2
- Combine to form K10'

- Key1 = P8 permutation of K10'
- Split K10'
- Left shift(2) K10'_1 and K10'_2
- Combine to form K10"
- Key2 = P8 permutation of K10"

```
def key_generation(key):
  K10 = permutation(key, P10)
  l = K10[:5]
  r = K10[5:]
  l = left_shift(l)
  r = left_shift(r)
  K10_2 = l+r

  key1 = permutation(K10_2, P8)

  l = left_shift(l, 2)
  r = left_shift(r, 2)
  K10_3 = l+r

  key2 = permutation(K10_3, P8)

  return key1, key2
```

```
key = [1, 1, 0, 0, 0, 1, 1, 1, 1, 0]
```

```
key1, key2 = key_generation(key)
print("Key1: ", key1)
print("Key2: ", key2)

```

```
Key1:  [1, 1, 1, 0, 1, 0, 0, 1]
Key2:  [1, 0, 1, 0, 0, 1, 1, 1]
```

Encryption

To undo cell deletion use ⌘/Ctrl+M Z or the 'Undo' option in the 'Edit' menu ×

- fk
- swap
- fk
- initial permutation (inverse)

```
def encryption(plain_text, key1, key2):
  IP = permutation(plain_text, initial_permutation)
  pt_1 = f_k(IP, key1)
  pt_swap = pt_1[4:] + pt_1[:4]
  pt_2 = f_k(pt_swap, key2)
```

```
  IP_inv = permutation(pt_2, inv_initial_permutation)

  return IP_inv
```

```
plain_text = [0, 0, 1, 0, 1, 0, 0, 0]
```

```
sdes = encryption(plain_text, key1, key2)
```

```
print("Output ciphertext: ", sdes)
```

```
Output ciphertext:  [1, 0, 0, 0, 1, 0, 1, 0]
```

Decryption

```
def decryption(plain_text, key1, key2):
  IP = permutation(plain_text, initial_permutation)
  pt_1 = f_k(IP, key2)
  pt_swap = pt_1[4:] + pt_1[:4]
  pt_2 = f_k(pt_swap, key1)
  IP_inv = permutation(pt_2, inv_initial_permutation)

  return IP_inv
```

```
decryption(sdes, key1, key2)
```

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

```python
import numpy as np
```

Utility functions

```python
def nibble_to_hex(n):
  assert len(n) <= 4, 'Invalid nibble provided.'
  if len(n) < 4: n = (4-len(n))*'0' + n
  return hex(int(n, 2))[2:]
```

```python
def hex_to_nibble(h):
  assert len(h) == 1, 'Invalid hex digit.'
  n = bin(int(h,16))[2:]
  return (4-len(n))*'0' + n
```

```python
def block_to_state(b):
  return [
          [b[0], b[2]],
          [b[1], b[3]]
  ]
```

```python
def state_to_block(s):
  return [s[0][0], s[1][0], s[0][1], s[1][1]]
```

```python
def sub_nibbles(s):
  S = [
       ['9', '4', 'a', 'b'],
       ['d', '1', '8', '5'],
       ['6', '2', '0', '3'],
       ['c', 'e', 'f', '7']
  ]
  b = state_to_block(s)
  b_new = []
  for h in b:
    n = hex_to_nibble(h)
    n_new = S[int(n[:2], 2)][int(n[2:], 2)]
    b_new.append(n_new)
  return block_to_state(b_new)
```

```python
def shift_rows(s):
  return [
          [s[0][0], s[0][1]],
          [s[1][1], s[1][0]]
  ]
```

```python
def mul(x, y):
  p1 = [int(c) for c in hex_to_nibble(x)]
```

```
  p2 = [int(c) for c in hex_to_nibble(y)]
  return np.polymul(p1, p2)

def add(x, y):
  p = list(np.polyadd(x, y))
  p = [c%2 for c in p]
  _, r = np.polydiv(p, [1, 0, 0, 1, 1])
  r = [str(int(c%2)) for c in r]
  return nibble_to_hex(''.join(r))

def mix_columns(s):
  C = [
       ['1', '4'],
       ['4', '1']
  ]
  s_new = [
           [None, None],
           [None, None]
  ]
  for i in range(2):
    for j in range(2):
      s_new[i][j] = add(mul(C[i][0], s[0][j]), mul(C[i][1], s[1][j]))
  return s_new
```

```
def rot_word(w):
  return [w[1], w[0]]

def sub_word(w):
  S = [
       ['9', '4', 'a', 'b'],
       ['d', '1', '8', '5'],
       ['6', '2', '0', '3'],
       ['c', 'e', 'f', '7']
  ]
  w_new = []
  for h in w:
    n = hex_to_nibble(h)
    n_new = S[int(n[:2], 2)][int(n[2:], 2)]
    w_new.append(n_new)
  return w_new

def xor(w1, w2):
  w = []
  for i in range(2):
    x = int(hex_to_nibble(w1[i]), 2)
    y = int(hex_to_nibble(w2[i]), 2)
    w.append(nibble_to_hex(bin(x^y)[2:]))
  return w

def key_expansion(k):
  w0, w1 = k[:2], k[2:]
  r1 = ['8', '0']
  t2 = xor(sub_word(rot_word(w1)), r1)
  w2 = xor(w0, t2)
```

```
  w3 = xor(w1, w2)
  r2 = ['3', '0']
  t4 = xor(sub_word(rot_word(w3)), r2)
  w4 = xor(w2, t4)
  w5 = xor(w3, w4)
  return w0 + w1, w2 + w3, w4 + w5
```

```
def add_round_key(k, s):
  k_state = block_to_state(k)
  w1 = xor([k_state[0][0], k_state[1][0]], [s[0][0], s[1][0]])
  w2 = xor([k_state[0][1], k_state[1][1]], [s[0][1], s[1][1]])
  return [
          [w1[0], w2[0]],
          [w1[1], w2[1]]
  ]
```

Encryption

```
def encrypt(plaintext, k):
  k1, k2, k3 = key_expansion(k)
  state = block_to_state(plaintext)
  state = add_round_key(k1, state)

  ### ROUND 1
  state = sub_nibbles(state)
  state = shift_rows(state)
  state = mix_columns(state)
  state = add_round_key(k2, state)

  ### ROUND 2
  state = sub_nibbles(state)
  state = shift_rows(state)
  #state = [['6', '4'],['7', 'b']]
  state = add_round_key(k3, state)

  ciphertext = state_to_block(state)
  return ciphertext
```

```
plaintext = ['1', 'a', '2', '3']
key = ['2', '4', '7', '5']
```

```
encrypt(plaintext, key)
```

```
['d', 'a', '4', '2']
```

0s completed at 19:42

```
import math
```

```
class User:
  def __init__(self, g, p, private_key):
    self.private_key = private_key
    self.g = g
    self.p = p
    self.public_key = math.pow(self.g, self.private_key) % self.p

  def get_shared_key(self, public_key2):
    shared_key = math.pow(public_key2, self.private_key) % self.p
    print("Shared Key = ", shared_key)
    return shared_key

  def get_public_key(self):
    return self.public_key

```

```
A = User(17, 23, 13)
B = User(17, 23, 9)
```

```
pk_A = A.get_public_key()
pk_B = B.get_public_key()
print("public key for A:", pk_A)
print("public key for B:", pk_B)
```

```
public key for A: 9.0
public key for B: 7.0
```

```
shared_key_A_B = A.get_shared_key(pk_B)
```

```
Shared Key =  20.0
```

✓ 0s completed at 15:52

```
class RSA:
  def __init__(self, p, q):
    self.p = p
    self.q = q
    self.n = p * q
    self.e = self.generate_e(p, q)
    self.d = self.generate_d(p, q)
    self.public_key = [self.n, self.e]
    self.private_key = [self.n, self.d]

  def __gcd(self, a, b):
      if (a == 0 or b == 0): return 0
      if (a == b): return a
      if (a > b): return self.__gcd(a - b, b)
      return self.__gcd(a, b - a)

  def is_coprime(self, x, y):
    if self.__gcd(x, y) == 1:
      return 1
    else:
      return 0

  def generate_e(self, p, q):
    x = (p-1) * (q-1)
    e = 0
    for i in range(2, x):
      if self.is_coprime(i, x):
        return i

  def generate_d(self, p, q):
    m = (p-1) * (q-1)
    for x in range(1, m):
      if (((self.e % m) * (x % m)) % m == 1):
        return x

  def encrypt(self, M):
    return (M ** self.e) % self.n

  def decrypt(self, C):
    return (C ** self.d) % self.n

  def show_keys(self):
    print("Private key:", self.private_key)
    print("Public key:", self.public_key)

```

```
rsa = RSA(7, 17)
```

```
rsa.show_keys()
```

```
Private key: [119, 77]
Public key: [119, 5]
```

```
plain_text = 19
cipher_text = rsa.encrypt(19)
print("Plain text {} is encrypted as {}".format(plain_text, cipher_text))

decrypted_text = rsa.decrypt(cipher_text)
print("Cipher text {} is decrypted as {}".format(cipher_text, decrypted_text))
```

```
Plain text 19 is encrypted as 66
Cipher text 66 is decrypted as 19
```

0s completed at 16:25

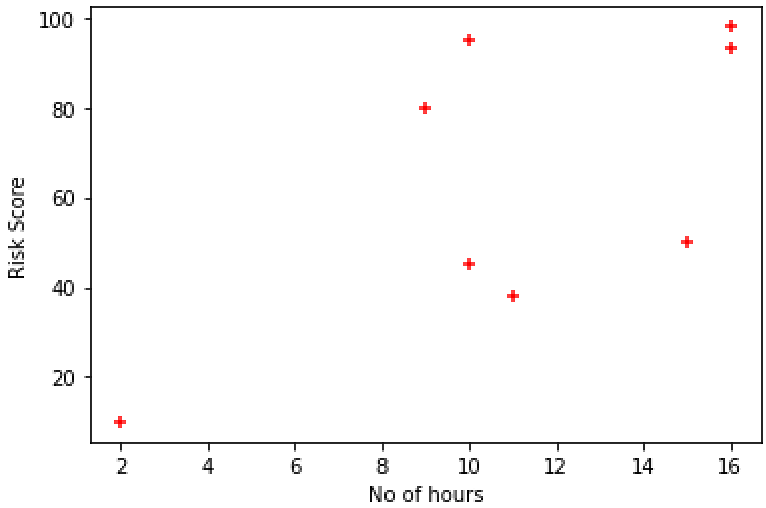
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import mpl_toolkits
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import datetime as dt
%matplotlib inline
```

```
data = [[10, 95], [9, 80], [2, 10], [15, 50], [10, 45], [16, 58], [11, 38], [16,
df = pd.DataFrame(data, columns = ['Hours', 'Risk'])
```

```
x = np.array([10, 9, 2, 15, 10, 16, 11, 16])
y = np.array([95, 80, 10, 50, 45, 98, 38, 93])
```

```
plt.xlabel('No of hours')
plt.ylabel('Risk Score')
plt.scatter(x,y,color='red',marker='+')
```

```
<matplotlib.collections.PathCollection at 0x7f445d88e2d0>
```




```
#principle of least squares
def getCoef(x,y):
    mean_x = np.mean(x)
    mean_y = np.mean(y)

    n = len(x)

    numer = 0
    denom = 0
    for i in range(n):
        numer += (x[i] - mean_x) * (y[i] - mean_y)
        denom += (x[i] - mean_x) ** 2
    b1 = numer / denom
    b0 = mean_y - (b1 * mean_x)
```

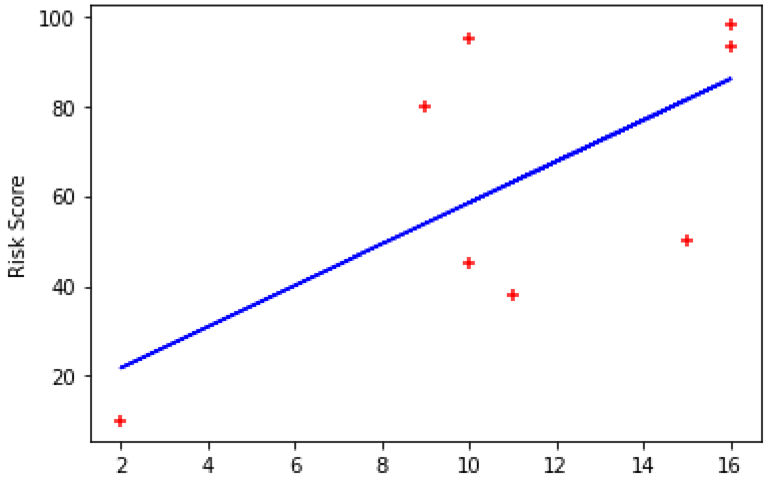
```
    return(b0, b1)
```

```
#y = b0 + b1 * x
coefs_ = getCoef(x,y)
print("Coefficients")
print(coefs_[0])
print(coefs_[1])
```

```
Coefficients
12.584627964022893
4.58789860997547
```

```
plt.xlabel('No of hours')
plt.ylabel('Risk Score')
plt.scatter(x,y,color='red',marker='+')
y_pred = coefs_[0] + coefs_[1]*x
plt.plot(x, y_pred, color = "b")
```

```
[<matplotlib.lines.Line2D at 0x7f445d3769d0>]
```



```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
from matplotlib import pyplot as plt
```

```
data = pd.read_csv('/content/gdrive/MyDrive/Colab Notebooks/LP3/ML/ML_2_data.cs
data.head()
```

	Age	Income	Gender	MaritialStatus	Buys
0	<21	High	Male	Single	No
1	<21	High	Male	Married	No
2	21-35	High	Male	Single	Yes
3	>35	Medium	Male	Single	Yes
4	>35	Low	Female	Single	Yes

```
le = LabelEncoder()
data = data.apply(le.fit_transform)
x = data.iloc[:, :-1]
#x = x.apply(le.fit_transform)
x.head()
```

	Age	Income	Gender	MaritialStatus
0	1	0	1	1
1	1	0	1	0
[illegible]	[illegible]	[illegible]	[illegible]	1
3	2	2	1	1
4	2	1	0	1

A Google Drive error has occurred. ×

```
y = data.iloc[:, -1]
y.head()
```

```
0    0
1    0
2    1
3    1
```

```
4    1
```

```
dt = DecisionTreeClassifier()
dt.fit(x.values, y.values)
```

```
DecisionTreeClassifier()
```

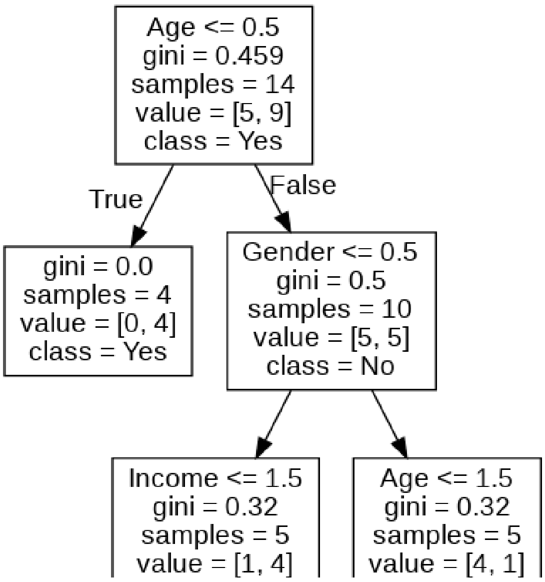
```
pred = dt.predict([[0, 0, 1, 1]])
pred[0]
```

```
1
```

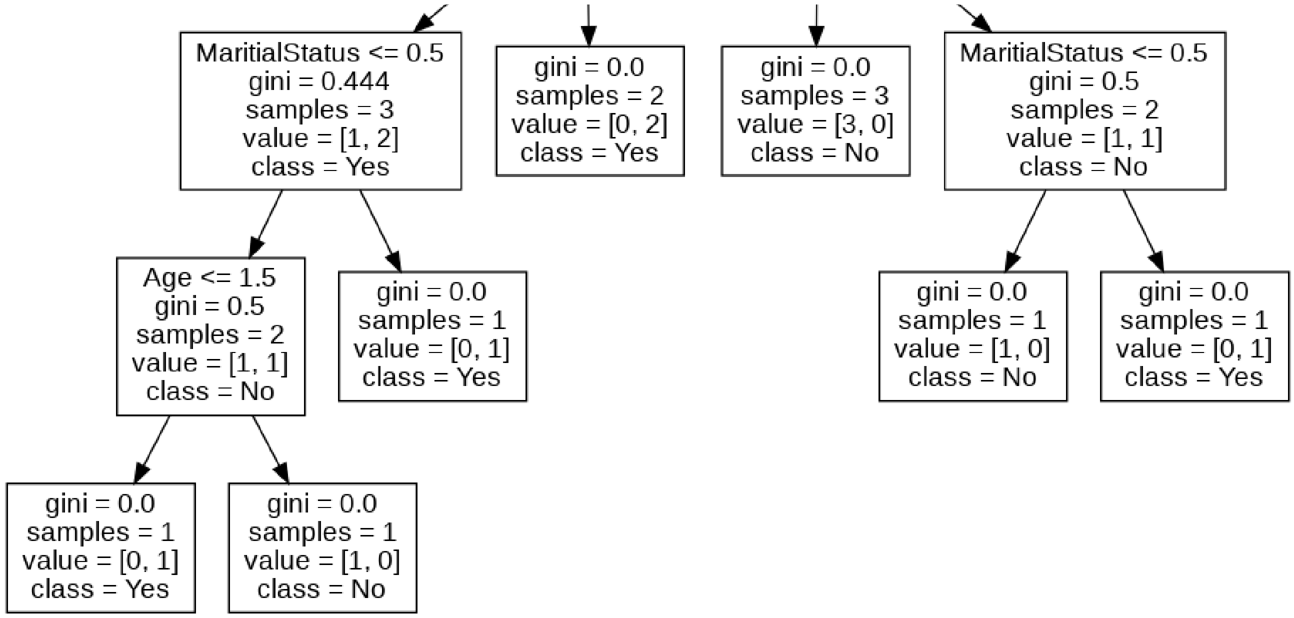
```
export_graphviz(dt,out_file="data.dot",feature_names=x.columns,class_names=["No'
!dot -Tpng data.dot -o tree.png

```

```
from google.colab.patches import cv2_imshow
import cv2
fig = cv2.imread('tree.png')
cv2_imshow(fig)
```

A Google Drive error has occurred.



1



A Google Drive error has occurred.

completed at 23:06

```
train_data_X = [[2, 4], [4, 4], [4, 6], [4, 2], [6, 2], [6, 4]]
train_data_y = ["Orange", "Blue", "Orange", "Orange", "Blue", "Orange"]
test_data = [[6, 6]]

```

```
import math
```

```
class kNN:
  def __init__(self, k=2, algorithm = 'auto'):
    self.k = k
    self.X = []
    self.y = []
    self.algorithm = algorithm

  def get_distance(self, pt1, pt2):
    return math.sqrt((pt1[0]-pt2[0])*(pt1[0]-pt2[0]) + (pt1[1]-pt2[1])*(pt1[1]-p

  def fit(self, X, y):
    self.X = X
    self.y = y

  def auto_knn(self, test, distances):
    prediction = max((distances), key = lambda tup: tup[1])
    return prediction[1]

  def distance_weighted_knn(self, test, distances):
    weights = {}
    for d in distances:
      try:
        weights[d[1]]+=float(1/d[0])
      except:
        weights[d[1]] = float(1/d[0])
    prediction = max((weights), key = lambda x: weights[x])
    return prediction

  def locally_weighted_averaging_knn(self, test, distances):
    frequencies = {}
    weights = {}
    for d in distances:
      try:
        weights[d[1]]+=float(1/d[0])
        frequencies[d[1]]+=1
      except:
        weights[d[1]] = float(1/d[0])
        frequencies[d[1]]=1

    for w in weights:
      weights[w]/= frequencies[w]

    prediction = max((weights), key = lambda x: weights[x])
    return prediction

```

```
  def predict(self, tests):
    results = []
    for test in tests:
      distances = []

      for i in range(len(self.X)):
        distances.append([self.get_distance(self.X[i], test), self.y[i]])
      distances.sort(key=lambda tup: tup[0])
      distances = distances[:self.k]
      print("Nearest Neighbours:", distances)

      if self.algorithm == 'auto':
        result = self.auto_knn(test, distances)
        results.append(result)
      elif self.algorithm == 'distance-weighted':
        result = self.distance_weighted_knn(test, distances)
        results.append(result)
      elif self.algorithm == 'locally-weighted-averaging':
        result = self.locally_weighted_averaging_knn(test, distances)
        results.append(result)
    return results

```

Basic kNN

```
n = kNN(3)
n.fit(train_data_X, train_data_y)
y_test = n.predict(test_data)
print(y_test)
```

```
Nearest Neighbours: [[2.0, 'Orange'], [2.0, 'Orange'], [2.8284271247461903, '
['Orange']
```

Distance Weighted kNN

```
n = kNN(3, 'distance-weighted')
n.fit(train_data_X, train_data_y)
y_test = n.predict(test_data)
print(y_test)
```

```
Nearest Neighbours: [[2.0, 'Orange'], [2.0, 'Orange'], [2.8284271247461903, '
['Orange']
```

Locally Weighted Averaging kNN

```
n = kNN(3, 'locally-weighted-averaging')
n.fit(train_data_X, train_data_y)
```

```
y_test = n.predict(test_data)
print(y_test)
```

```
Nearest Neighbours: [[2.0, 'Orange'], [2.0, 'Orange'], [2.8284271247461903, '
['Orange']
```

0s completed at 17:20

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
from copy import deepcopy
import numpy as np
import pandas as pd
from·matplotlib·import·pyplot·as·plt
import math
import seaborn as sns
```

```
data = pd.read_csv('/content/gdrive/MyDrive/Colab Notebooks/LP3/ML/ML_4_data.csv
data.head()
```

	X	Y
0	0.10	0.60
1	0.15	0.71
2	0.08	0.90
3	0.16	0.85
4	0.20	0.30

```
X = np.array(data)
```

```
c_x = np.array([0.1,0.3])
c_y = np.array([0.6,0.2])

centroids = np.array(list(zip(c_x,c_y)))
centroids
```

```
array([[0.1, 0.6],
       [0.3, 0.2]])
```

```
class K_Means:
  def __init__(self, k=2, tol=0.01, max_iter = 300):
    self.k = k
    self.tol = tol
    self.max_iter = max_iter
    self.cur_centroid = []

  def get_cluster(self, data):
    distances = []
    x1 = data[0]
    y1 = data[1]

    for centroid in self.cur_centroid:
```

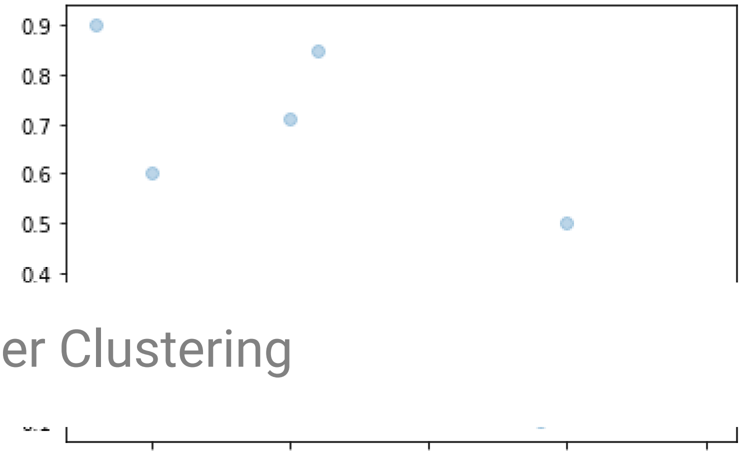
```
      distances.append((y1-centroid[1])*(y1-centroid[1]) + (x1-centroid[0])*(x1-

    classification = distances.index(min(distances))
    return classification

  def fit(self, data, centroid):
    self.cur_centroid = centroid
    new_centroids = []

    iter = 0

    while iter<self.max_iter:
      i=0
      cur_clusters = [[] for i in range(self.k)]

      for pt in data:
        clust = self.get_cluster(pt)
        cur_clusters[clust].append(pt)

      new_centroids = [np.average(cur_clusters[i], axis=0) for i in range(self.k

      optimised = True
      for i in range(len(self.cur_centroid)):
        if (np.abs(np.sum((new_centroids[i]-self.cur_centroid[i])/self.cur_centr
          optimised = False

      if optimised == True:
        break

      self.cur_centroid = new_centroids
      iter+=1

    print("Final centroids", self.cur_centroid)
    return cur_clusters, self.cur_centroid
```

```
km = K_Means(2, 0.01, 1)
clusters, centroids = km.fit(X, centroids)
```

```
Final centroids [array([0.148, 0.712]), array([0.24666667, 0.2       ])]
```

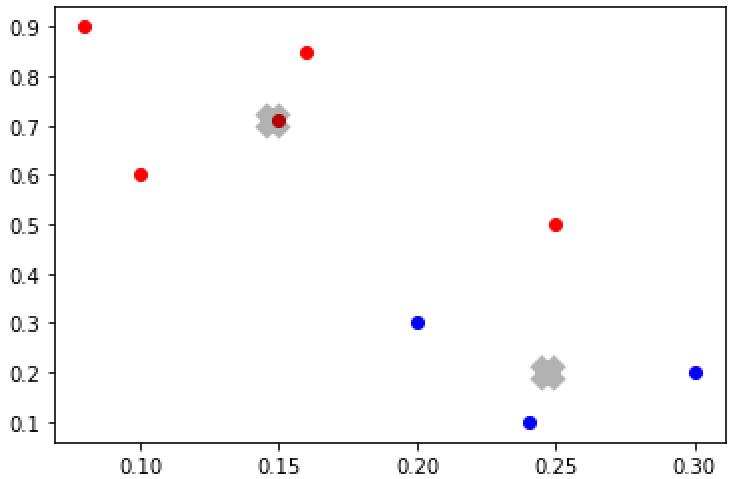
Before Clustering

```
plt.figure()
plt.scatter(X[:,0],X[:,1],alpha=0.3)
plt.show()
```



After Clustering

```
colors = ['r','b']
plt.figure()

for i in range(len(clusters)):
  for c in clusters[i]:
    plt.scatter(c[0], c[1], color = colors[i])

for i in range(len(centroids)):
    plt.scatter(centroids[i][0], centroids[i][1], marker = 'x', color = 'black',
plt.show()
```



0s completed at 22:54