- Mufaddal Diwan

ML

Assignment 1

Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# sns.set(rc = {'figure.figsize':(8,8)})
```

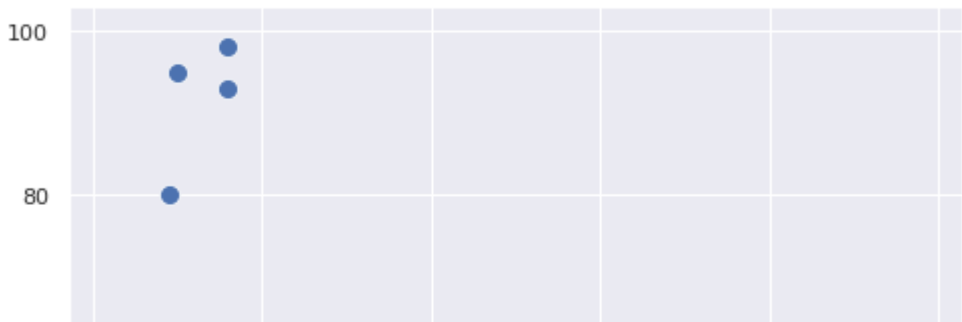
```
data = [
        (10, 95),
        (9, 80),
        (2, 10),
        (15, 50),
        (10, 45),
        (16, 98),
        (11, 38),
        (16, 93),
]
```

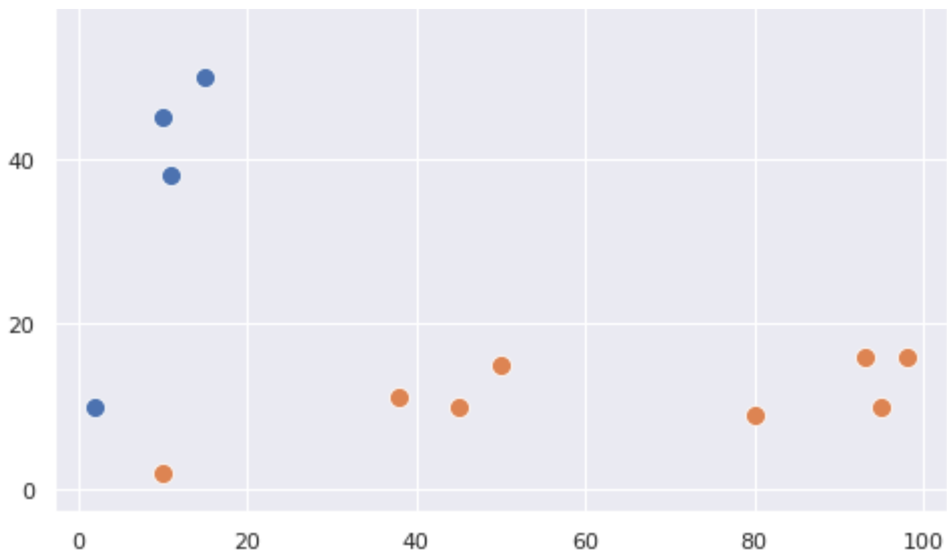
```
x = [pt[0] for pt in data]
y = [pt[1] for pt in data]
```

ORIGINAL PLOT

```
sns.scatterplot(x=x, y=y, s=100)
sns.scatterplot(x=y, y=x, s=100)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4b7c934490>
```





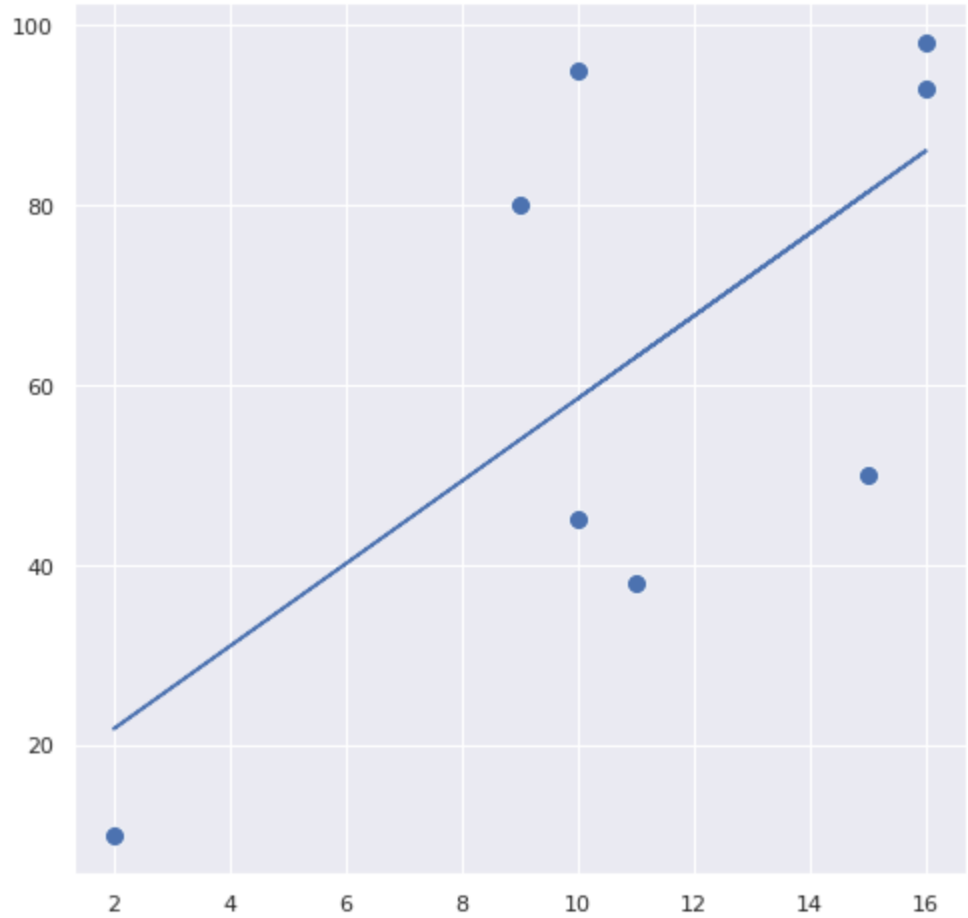
LINE PARAMETER GENERATION

```
n = len(x)
xx = [a * a for a in x]
xy = [x[i] * y[i] for i in range(n)]


sum_x = np.sum(x)
sum_y = np.sum(y)
sum_xx = np.sum(xx)
sum_xy = np.sum(xy)


m = (n * sum_xy - sum_x * sum_y) / (n * sum_xx - sum_x * sum_x)


b = (sum_y - m * sum_x) / n


print(f'LINE EQUATION: y = {round(m,2)} * x + {round(b,2)}')
```

```
LINE EQUATION: y = 4.59 * x + 12.58
```

PLOT WITH GIVEN LINE

```
def plot_graph(x, y, slope, intercept):
    axes = sns.scatterplot(x=x, y=y, s=100)
    x_vals = np.array(x)
    y_vals = intercept + slope * x_vals
    plt.plot(x_vals, y_vals)


plot_graph(x, y, m, b)
```



Assignment 2

Decision tree

[Reference]

old

```
import pandas as pd
import numpy as np
from itertools import chain, combinations


class Node:
  def __init__(self, col, dtype, values = None):
    self.col = col
    self.dtype = dtype
    if self.dtype == 'categorical':
      assert values is not None, 'Mention values for categorical feature.'
      self.values = values
    else: self.values = None
    self.yes = True
    self.no = False

  def __str__(self):
```

```
    return f'COLUMN: {self.col}, VALUES: {self.values},'


class DecisionTree:
  def __init__(self):
    self.tree = None

  def __gini(self, cnt):
    total = np.sum(cnt)
    if total == 0: return 0
    return 1 - (cnt[0] / total) ** 2 - (cnt[1] / total) **2

  def __powerset(self, iterable):
    s = list(iterable)
    if len(s) == 1: return [tuple(s)]
    return list(chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

  def __total_imp(self, true_count, false_count):
    true_total = np.sum(true_count)
    false_total = np.sum(false_count)
    total = true_total + false_total
    return (self.__gini(true_count) * true_total / total + self.__gini(false_cou
            self.__gini(true_count), self.__gini(false_count))

  def __get_imp(self, feature, val, data, label_name):
    if self.col_type[feature] == 'numerical':
      pass
    else:
      true_count = [0, 0]
      false_count = [0, 0]
      for i in range(len(data[feature])):
        if data[feature].iloc[i] in val:
          if data[label_name].iloc[i]: true_count[1] += 1
          else: true_count[0] += 1
        else:
          if data[label_name].iloc[i]: false_count[1] += 1
          else: false_count[0] += 1
      return self.__total_imp(true_count, false_count)

  def __feature_impurity(self, feature, data, label_name):
    if self.col_type[feature] == 'numerical':
      pass
    else:
      values = self.__powerset(data[feature].unique())
      val_imp = set()
      for val in values:
        imp = self.__get_imp(feature, val, data, label_name)
        val_imp.add((imp, val))
        # print(f'Feature: {feature}, Values: {val}, Impurity: {imp[0]}')
      return val_imp.pop()

  def __build_tree(self, data, label_name, cols, par_imp = 10):
    if len(cols) == 1: return None
    col_imp = set()
    for col in cols:
```

```
      for col in cols:
        if self.col_type[col] == 'label': continue
        col_imp.add((self.__feature_impurity(col, data, label_name), col))
      best = col_imp.pop()
      col = best[1]
      if best[0][0][0] < par_imp:
        node = Node(col, self.col_type[col], best[0][1])
        data_yes = data[data[col].isin(list(best[0][1])) == True].drop(col, axis=1
        data_no = data[data[col].isin(list(best[0][1])) == False].drop(col, axis=1
        new_cols = list(data_yes.columns)
        node.yes = self.__build_tree(data_yes, label_name, new_cols.copy(), best[0
        node.no = self.__build_tree(data_no, label_name, new_cols.copy(), best[0][
        if node.yes is None: node.yes = True
        if node.no is None: node.no = False
        return node

    def fit(self, df, label_name):
      self.col_type = {}
      self.cols = list(df.columns)
      for col in self.cols:
        if col == label_name: self.col_type[col] = 'label'
        elif type(df[col][0]) == str:
          self.col_type[col] = 'categorical'
        else: self.col_type[col] = 'numerical'
      self.tree = self.__build_tree(df, label_name, self.cols.copy())

    def __predict(self, data, node):
      if type(node) == bool: return node

      val = data[node.col]
      if val in node.values: node = node.yes
      else: node = node.no

      return self.__predict(data, node)

    def predict(self, df):
      preds = []
      for i in range(len(df)):
        preds.append(self.__predict(df.iloc[i], self.tree))
      return preds


  df = pd.read_csv('dataset.csv').drop('ID', axis=1)
  df.head()
```

	Age	Income	Gender	MaritalStatus	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

```
train_df = df[:-1].copy()
train_df['Buys'] = train_df['Buys'] == 'Yes'
```

```
test_df = df[-1:].copy().drop('Buys', axis = 1)
```

```
clf = DecisionTree()
```

```
clf.fit(train_df, 'Buys')
```

```
print(f'Root: {clf.tree.col}')
```

```
Root: Age
```

```
clf.predict(test_df)
```

```
[True]
```

new

```
class Node:
  def __init__(self, feature, values):
    self.feature = feature
    self.values = values
    self.yes = None
    self.no = None

  def __str__(self):
    return f'Feature: {self.feature}, Values: {self.values}'

class DecisionTree:
  def __gini(self, yes_count, no_count):
    yes_total = yes_count[0] + yes_count[1]
    no_total = no_count[0] + no_count[1]
    gini_yes = 1 - (yes_count[0] / yes_total) ** 2 - (yes_count[1] / yes_total)
    gini_no = 1 - (no_count[0] / no_total) ** 2 - (no_count[1] / no_total) ** 2
    return (yes_total * gini_yes  + no_total * gini_no) / (yes_total + no_total)

  def __get_impurity(self, X, y, values):
    yes_count = [0, 0]
    no_count = [0, 0]
    for i in range(len(X)):
      if X[i] in values:
        if y[i]: yes_count[1] += 1
        else: yes_count[0] += 1
      else:
        if y[i]: no_count[1] += 1
        else: no_count[0] += 1
```

```
      return self.__gini(yes_count, no_count)

    def __parse(self, x):
      val = list(bin(x)[2:])
      return [i for i in range(len(val)) if val[i] == '1']

    def __get_feature_impurity(self, X, y):
      values = np.unique(X)
      n = 2 ** len(values) - 1
      best_impurity = 100
      for i in range(1, n):
        idx = self.__parse(i)
        val_subset = values[idx].copy()
        impurity = self.__get_impurity(X, y, val_subset)
        if impurity < best_impurity:
          best_impurity = impurity
          best_values = val_subset
      return val_subset, impurity

    def __select_best_feature(self, X, y):
      best_impurity = 100
      for feature in X.columns:
        values, impurity = self.__get_feature_impurity(list(X[feature]), y)
        if impurity < best_impurity:
          best_impurity = impurity
          best_feature = feature
          best_values = values

      return best_feature, best_values, best_impurity

    def __filter_data(self, X, y, feature, values, flag):
      X_filtered = X[X[feature].isin(values)].copy()
      idx = list(X_filtered.index)
      X_filtered = X_filtered.reset_index().drop([feature, 'index'], axis = 1)
      y_filtered = y[idx].copy()
      return X_filtered, y_filtered

    def __build_tree(self, X, y, parent_impurity = 100):
      best_feature, best_values, impurity = self.__select_best_feature(X, y)
      if impurity >= parent_impurity: return None

      node = Node(best_feature, best_values)

      X_yes, y_yes = self.__filter_data(X, y, best_feature, best_values, True)
      X_no, y_no = self.__filter_data(X, y, best_feature, best_values, False)

      node.yes = self.__build_tree(X_yes, y_yes, impurity)
      node.no = self.__build_tree(X_no, y_no, impurity)

      if node.yes is None: node.yes = True
      if node.no is None: node.no = False
      return node

    def fit(self, X, y):
```

```
    self.tree = self.__build_tree(X, y)

  def __make_prediction(self, x, node):
    if type(node) == bool: return node

    value = x[node.feature]
    if value in node.values: node = node.yes
    else: node = node.no

    return self.__make_prediction(x, node)

  def predict(self, X):
    preds = []
    for i in range(len(X)):
      preds.append(self.__make_prediction(X.iloc[i], self.tree))
    return np.array(preds)
```

```
df = pd.read_csv('dataset.csv').drop('ID', axis=1)
df
```

	Age	Income	Gender	MaritalStatus	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
5	>35	Low	Female	Married	No
6	21-35	Low	Female	Married	Yes
7	<21	Medium	Male	Single	No
8	<21	Low	Female	Married	Yes
9	>35	Medium	Female	Single	Yes
10	<21	Medium	Female	Married	Yes
11	21-35	Medium	Male	Married	Yes
12	21-35	High	Female	Single	Yes
13	>35	Medium	Male	Married	No
14	<21	Low	Female	Married	?

```
train_df = df.iloc[:-1].copy()
test_df = df.iloc[-1:].copy()
```

```
X_train, y_train = train_df.drop('Buys', axis = 1), np.array(train_df['Buys']) =
```

```
X_test = test_df.drop('Buys', axis = 1)


clf = DecisionTree()


clf.fit(X_train, y_train)


clf.predict(test_df)
```

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

Muf

```
import pandas as pd
import numpy as np


class Node:
  def __init__(self, feature_values, feature_name, impurity):
    self.true = True
    self.false = False
    self.feature_values = feature_values
    self.feature_name = feature_name
    self.impurity = impurity

  def get_feature_values(self):
    return self.feature_values

  def get_impurity(self):
    return self.impurity

  def get_true(self):
    return self.true

  def get_false(self):
    return self.false

  def get_feature_name(self):
    return self.feature_name

  def set_true(self, node):
    if node is None:
      node = True
    self.true = node

  def set_false(self, node):
    if node is None:
      node = False
    self.false = node
```

```
  def __str__(self):
    return f"{self.get_feature_name()}"


class DecisionTree:
  def __init__(self):
    self.tree = None

  def __weighted_values(self, v1, v2):
    if (v1 + v2) == 0:
      return 0, 0
    return v1/(v1 + v2), v2/(v1 + v2)

  def __gini(self, true_count, false_count):
    t1, t2 = self.__weighted_values(true_count, false_count)
    return 1.0 - t1*t1 - t2*t2

  def __get_node_impurity(self, mat):
    g_false = self.__gini(mat[0][1], mat[0][0])
    g_true = self.__gini(mat[1][1], mat[1][0])
    w_false, w_true = self.__weighted_values(mat[0][0]+mat[0][1], mat[1][0] + ma
    return w_false * g_false + w_true * g_true

  def __get_power_set(self, lis):
    n = len(lis)
    ps = []
    for i in range(1, pow(2, n) - 1, 1):
      bi = (bin(i).replace('0b','')).rjust(n, "0")
      ret = []
      for j in range(len(bi)):
        if bi[j] == '1':
          ret.append(lis[j])
      ps.append(ret)
    return ps

  def __get_feature_atomic_impurity(self, X, y, feature):
    best_impurity = 100
    best_node = None
    unique_values = X[feature].unique()

    for uniques in self.__get_power_set(unique_values):
      mat = [
          # True False
          [0, 0], # True wrt feature
          [0, 0] # False ,,   ,,
      ]
      for i in range(X.shape[0]):
        r = int(X.iloc[i][feature] in uniques)
        c = int(y[i])
        mat[r][c] += 1
      impurity = self.__get_node_impurity(mat)
      if impurity < best_impurity:
        best_impurity = impurity
        best_node = Node(uniques, feature, impurity)
    return best_node, best_impurity
```

```
    return best_node, best_impurity


  def __get_best_node(self, X, y):
    features = list(X.columns)

    best_impurity = 100
    best_node = None

    for feature in features:
      node, impurity = self.__get_feature_atomic_impurity(X, y, feature)
      if impurity < best_impurity:
        best_impurity = impurity
        best_node = node
    return best_node, best_impurity

  def __get_child_data(self, X, y, parent_node):
    feature_name = parent_node.get_feature_name()
    feature_values = parent_node.get_feature_values()

    X_false = X.copy()
    X_true = X.copy()

    false_lis, true_lis = [], []

    for i in range(X.shape[0]):
      if X.iloc[i][feature_name] in feature_values:
        true_lis.append(i)
      else:
        false_lis.append(i)

    X_false = X_false.drop(labels = true_lis, axis = 0).drop(feature_name, axis
    X_true = X_true.drop(labels = false_lis, axis = 0).drop(feature_name, axis =
    y_false = np.delete(y, true_lis, axis = 0)
    y_true = np.delete(y, false_lis, axis = 0)

    return (X_false.reset_index().drop("index", axis = 1), y_false,
        X_true.reset_index().drop("index", axis=1), y_true)

  def __build_tree(self, X, y, parent_impurity = 100):
    if X.empty:
      return None
    feature_node, impurity = self.__get_best_node(X, y)
    if impurity >= parent_impurity:
      return None

    X_false, y_false, X_true, y_true = self.__get_child_data(X, y, feature_node)

    feature_node.set_true(self.__build_tree(X_true, y_true, impurity))
    feature_node.set_false(self.__build_tree(X_false, y_false, impurity))
    return feature_node

  def fit(self, X, y):
    self.tree = self.__build_tree(X, y)
```

```
  def __predict_single(self, x, parent):
    if type(parent) == bool:
      return parent

    feature_name = parent.get_feature_name()
    feature_values = parent.get_feature_values()

    val = x[feature_name]

    if val in feature_values:
      return self.__predict_single(x, parent.get_true())
    return self.__predict_single(x, parent.get_false())

  def predict(self, X):
    preds = []
    for i in range(X.shape[0]):
      pred = self.__predict_single(X.iloc[i], self.tree)
      preds.append(pred)
    return preds
```

```
df = pd.read_csv('dataset.csv').drop("ID", axis = 1)
X_train = df.copy().drop("Buys", axis = 1).drop([df.shape[0]-1], axis=0)
X_test = df.iloc[df.shape[0]-1:].drop("Buys", axis = 1)
y_train = np.array(df["Buys"].drop([df.shape[0]-1], axis = 0)) == "Yes"
print(X_train)
print(X_test)
```

```
      Age  Income  Gender MaritalStatus
0     <21    High    Male        Single
1     <21    High    Male       Married
2   21-35    High    Male        Single
3     >35  Medium    Male        Single
4     >35     Low  Female        Single
5     >35     Low  Female       Married
6   21-35     Low  Female       Married
7     <21  Medium    Male        Single
8     <21     Low  Female       Married
9     >35  Medium  Female        Single
10    <21  Medium  Female       Married
11  21-35  Medium    Male       Married
12  21-35    High  Female        Single
13    >35  Medium    Male       Married
    Age Income  Gender MaritalStatus
14  <21    Low  Female       Married
```

```
dt = DecisionTree()
```

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

```
print(dt.tree.get_feature_name())
print(dt.predict(X_test))
```

```
Age
[False]
```

Assignment 3

K-NN Classifier

```
import numpy as np


class kNN:
  def __init__(self, k):
    self.k = k
    self.X = []
    self.y = []

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

  def __distance(self, x, y):
    return (x[0] - y[0]) ** 2 + (x[1] - y[1]) ** 2

  def __get_class(self, X):
    distances = []
    for i in range(len(self.X)):
      distances.append((self.__distance(X, self.X[i]), self.y[i]))
    distances.sort()
    distances = distances[:self.k]
    counts = {}
    for d in distances:
      try: counts[d[1]] += 1
      except: counts[d[1]] = 1
    return max(counts, key = lambda i: counts[i])

  def predict(self, X):
    preds = []
    for x in X:
      preds.append(self.__get_class(x))
    return preds

  def __get_weighted_class(self, X):
    distances = []
    for i in range(len(self.X)):
      distances.append((self.__distance(X, self.X[i]), self.y[i]))
    distances.sort()
    distances = distances[:self.k]
    counts = {}
    for d in distances:
```

```
      try: counts[d[1]] += 1 / d[0]
      except: counts[d[1]] = 1 / d[0]
    return max(counts, key = lambda i: counts[i])

  def predict_weighted(self, X):
    preds = []
    for x in X:
      preds.append(self.__get_weighted_class(x))
    return preds

  def __get_locally_weighted_average_class(self, X):
    distances = []
    for i in range(len(self.X)):
      distances.append((self.__distance(X, self.X[i]), self.y[i]))
    distances.sort()
    distances = distances[:self.k]
    counts = {}
    for d in distances:
      try: counts[d[1]].append(1 / d[0])
      except: counts[d[1]] = [1 / d[0]]
    for c in counts:
      counts[c] = np.mean(counts[c])
    return max(counts, key = lambda i: counts[i])

  def predict_locally_weighted_average(self, X):
    preds = []
    for x in X:
      preds.append(self.__get_weighted_class(x))
    return preds
```

```
X = [
     (2, 4),
     (4, 6),
     (4, 4),
     (4, 2),
     (6, 4),
     (6 ,2)
]
y = ['Y', 'Y', 'B', 'Y', 'Y', 'B']
```

```
model = kNN(3)
```

```
model.fit(X, y)
```

```
print(f'Standard k-NN: {model.predict([(6, 6)])}')
```

```
Standard k-NN: ['Y']
```

```
print(f'Distance Weighted k-NN: {model.predict_weighted([(6, 6)])}')
```

```
Distance Weighted k-NN: ['Y']
```

```
print(f'Locally Weighted Average k-NN: {model.predict_locally_weighted_average([
```

```
Locally Weighted Average k-NN: ['Y']
```

Assignment 4

K-means Clustering

```
import pandas as pd
import seaborn as sns
# sns.set(rc={'figure.figsize':(7, 7)})


class KMeans:
  def __init__(self, k):
    self.k = k

  def __distance(self, x, y):
    return (x[0] - y[0]) ** 2 + (x[1] - y[1]) ** 2

  def fit(self, points, centroids):
    prev_clusters = None
    clusters = [set() for _ in range(self.k)]

    while prev_clusters != clusters:
      prev_clusters = clusters
      for p in points:
        idx = 0
        for i in range(1, self.k):
          if self.__distance(p, centroids[i]) < self.__distance(p, centroids[idx
            idx = i
        clusters[idx].add(p)
      for i in range(self.k):
        centroids[i] = np.mean(list(clusters[i]), axis = 0)

    return clusters, centroids


points = [
          (0.1, 0.6),
          (0.15, 0.71),
          (0.08,0.9),
          (0.16, 0.85),
          (0.2,0.3),
          (0.25,0.5),
          (0.24,0.1),
          (0.3,0.2)
]
```

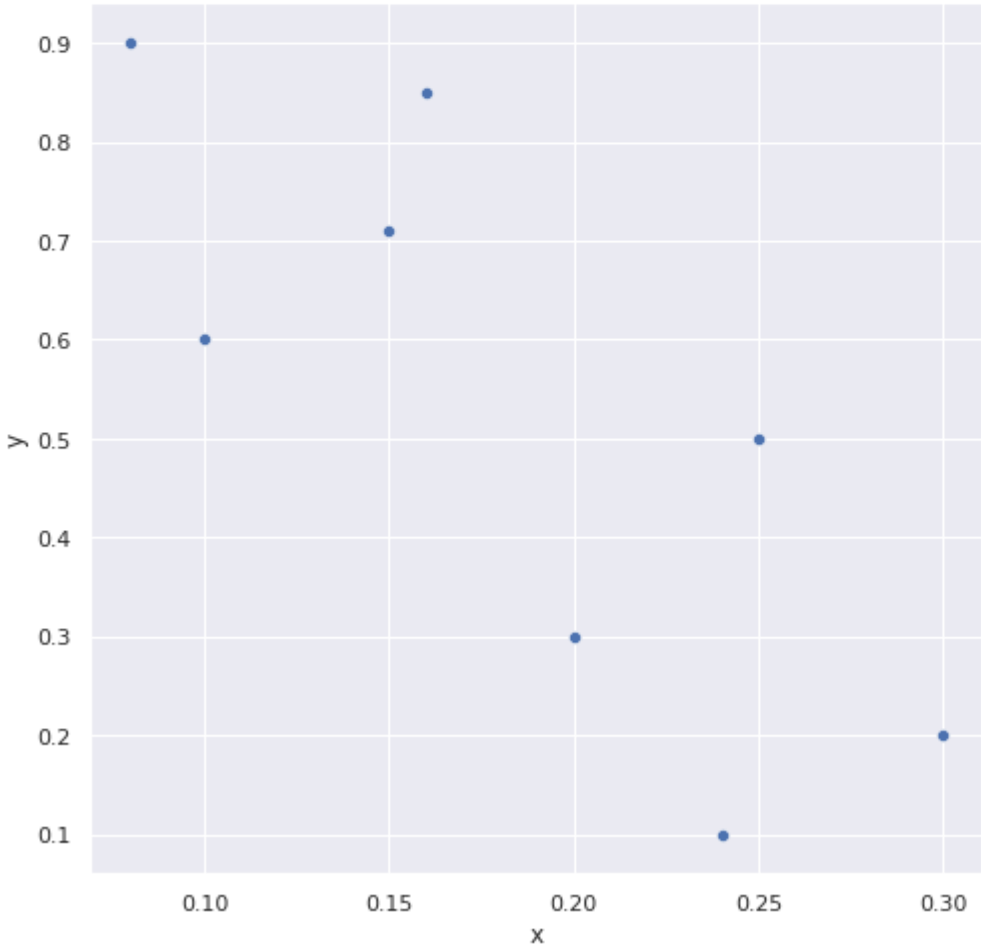
BEFORE CLUSTERING

BEFORE CLUSTERING

```
raw_df = pd.DataFrame()
x = [p[0] for p in points]
y = [p[1] for p in points]
raw_df['x'] = x
raw_df['y'] = y
raw_df
```

```
sns.scatterplot(data = raw_df, x = 'x', y = 'y')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4b7caa49d0>
```



AFTER CLUSTERING

```
model = KMeans(2)
```

```
clusters, centroids = model.fit(points, centroids = [(0.1, 0.6),(0.3,0.2)])
```

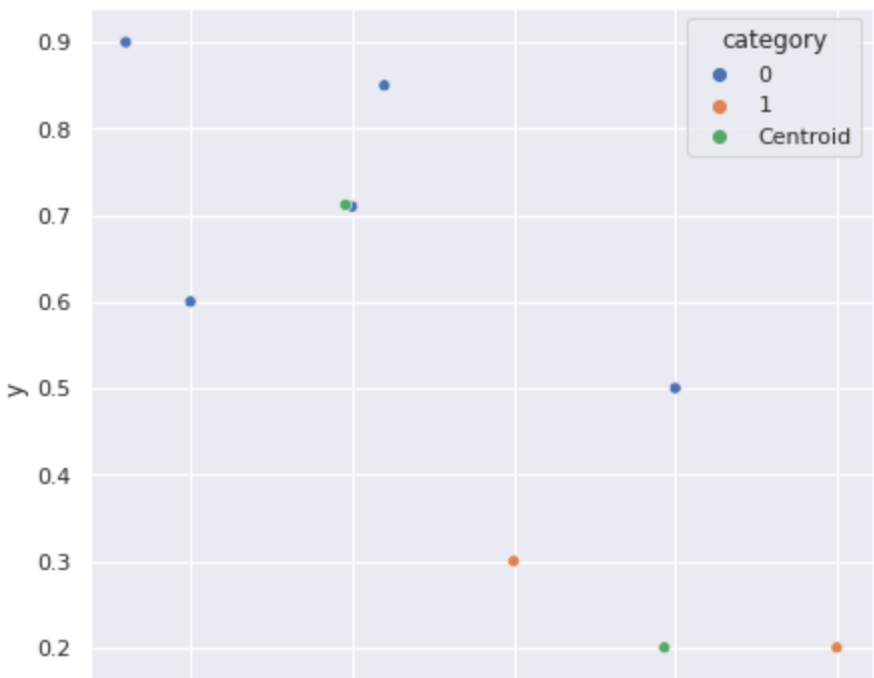
```
clustered_df = pd.DataFrame()
x = []
y = []
category = []
for i in range(len(clusters)):
```

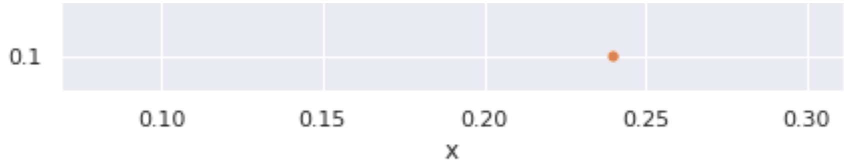
```
for i in range(len(clusters)):
  for p in clusters[i]:
    x.append(p[0])
    y.append(p[1])
    category.append(f'{i}')
for c in centroids:
  x.append(c[0])
  y.append(c[1])
  category.append('Centroid')
clustered_df['x'] = x
clustered_df['y'] = y
clustered_df['category'] = category
clustered_df
```

	x	y	category
0	0.080000	0.900	0
1	0.160000	0.850	0
2	0.100000	0.600	0
3	0.150000	0.710	0
4	0.250000	0.500	0
5	0.200000	0.300	1
6	0.240000	0.100	1
7	0.300000	0.200	1
8	0.148000	0.712	Centroid
9	0.246667	0.200	Centroid

```
sns.scatterplot(data = clustered_df, x = 'x', y = 'y', hue = 'category')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb132310d50>
```





Mini Project 1

Travelling Salesman Problem using Genetic Algorithm

[ ] ↳ *11 cells hidden*

ICS

Assignment 1

Simple Data Encryption Standard (S-DES)

[Reference]()

```
import random
```

Constants initialisation

```
P10 = [] # Random permutation of size 10
P8 = [] # Sample 8 elemets from P10
P4 = [] # Permutation of size 4
EP = [] # Shuffle two permutations of size 4 # Expansion permutation
IP = [] # # Random permutation of size 8 # Initial permutation
IP_INV = [] # To be calculated from IP IP_INV[IP[i] - 1] = i + 1
```

```
def gen_random_permutation(n):
  out = [i+1 for i in range(n)]
  random.shuffle(out)
  return out;
```

```
IP = gen_random_permutation(8)
IP_INV = [1]*8
for i in range(len(IP)):
  IP_INV[IP[i]-1] = i+1
```

```
EP = gen_random_permutation(4)*2
```

```
P10 = gen_random_permutation(10)
```

```
P10 = gen_random_permutation(10)
P4 = gen_random_permutation(4)


P8 = random.sample(gen_random_permutation(10), 8)


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]
]
```

Helper functions

```
def bin_to_dec(x):
  return int(x, 2)
def dec_to_bin(x):
  return bin(x).replace("0b","")


def left_circular_shift(x, shifts=1):
  shifts = shifts % len(x)
  return x[shifts:] + x[:shifts]


def permutate(key, perm):
  ret = ""
  for k in perm:
    ret += key[k-1]
  return ret


def split_str(key):
  half = len(key)//2
  key1 = key[:half]
  key2 = key[half:]
  return key1, key2


def xor(a, b):
  ret = ""
  for i in range(len(a)):
    if a[i] == b[i]: ret += "0"
    else: ret += "1"
  return ret
```

Algorithm necessary functions

```
def gen_subkeys(key):
  n_key = permutate(key, P10)

  left_key, right_key = split_str(n_key)

  left_key = left_circular_shift(left_key, 1)
  right_key = left_circular_shift(right_key, 1)

  k1 = permutate(left_key + right_key, P8)

  left_key = left_circular_shift(left_key, 2)
  right_key = left_circular_shift(right_key, 2)

  k2 = permutate(left_key + right_key, P8)

  return k1, k2


def s_box(text, s):
  r = text[0] + text[3]
  c = text[1] + text[2]

  r = bin_to_dec(r)
  c = bin_to_dec(c)
  out = s[r][c]
  out = dec_to_bin(out)
  while len(out) < 2:
    out = "0" + out
  return out


def function(left, right, subkey):
  text = right
  text = permutate(text, EP)
  text = xor(text, subkey)
  text_left, text_right = split_str(text)
  text = s_box(text_left, S0) + s_box(text_right, S1)
  text = permutate(text, P4)
  text = xor(text, left)
  return text, right


def encryption(plaintext, key):
  k1, k2 = gen_subkeys(key)

  ciphertext = permutate(plaintext, IP)

  left, right = split_str(ciphertext)
  left, right = function(left, right, k1)

  left, right = right, left
```

```
  left, right = function(left, right, k2)

  ciphertext = permutate(left + right, IP_INV)

  return ciphertext


def decryption(ciphertext, key):
  k1, k2 = gen_subkeys(key)

  plaintext = permutate(ciphertext, IP)

  left, right = split_str(plaintext)
  left, right = function(left, right, k2)

  left, right = right, left

  left, right = function(left, right, k1)

  plaintext = permutate(left + right, IP_INV)

  return plaintext
```

Testing

```
key = "1010001011"
plaintext = "10001010"


c = encryption(plaintext, key)
p = decryption(c, key)


assert(p==plaintext)
```

Assignment 2

Simplified Advanced Encryption Standard (S-AES)

[Reference]

```
import numpy as np
```

Helper functions

```
def bin_to_dec(x):
  return int(x, 2)
```

```
def dec_to_bin(x):
  return bin(x).replace("0b","")
def hex_to_bin(x):
  ret = dec_to_bin(int(x, 16))
  ret = assert_value_size(ret, len(x)*4)
  return ret
def bin_to_hex(x):
  return hex(bin_to_dec(x))


def assert_value_size(x, s):
  while len(x) < s:
    x = "0" + x
  return x


def xor(a, b):
  ret = ""
  for i in range(len(a)):
    if a[i] == b[i]: ret += "0"
    else: ret += "1"
  return ret


def split_str(val):
  half = len(val)//2
  return val[:half], val[half:]


def get_indices(nib):
  r = bin_to_dec(nib[:2])
  c = bin_to_dec(nib[2:])
  return r, c


def nibble_list(x):
  x = assert_value_size(x, 16)
  ret = [x[i:i+4] for i in range(0, len(x), 4)]
  return ret

def list_to_mat(l):
  return [
      [l[0], l[2]],
      [l[1], l[3]]
  ]

def mat_to_list(m):
  return [m[0][0], m[1][0], m[0][1], m[1][1]]


def rot_nib(val):
  half = len(val)//2
  return val[half:] + val[:half]


def mul_nib(nib1, nib2):
  n1 = [int(c) for c in nib1]
```

```
  p1 = [int(c) for c in nib1]
  p2 = [int(c) for c in nib2]
  ret = np.polymul(p1, p2)
  ret = [str(c) for c in ret]
  return "".join(ret)

def add_nib(nib1, nib2):
  p1 = [int(c) for c in nib1]
  p2 = [int(c) for c in nib2]
  ret = np.polyadd(p1, p2)
  ret = [c % 2 for c in ret]
  _, r = np.polydiv(ret, [1, 0, 0, 1, 1])
  nib = [str(int(c%2)) for c in r]
  nib = "".join(nib)
  while len(nib) > 4:
    nib = nib[1:]
  nib = assert_value_size(nib, 4)
  return nib
```

Constants

```
def gen_inv_s_box(s):
  ret = [r[:] for r in s]
  for i in range(4):
    for j in range(4):
      r, c = get_indices(hex_to_bin(s[i][j]))
      ret[r][c] = bin_to_hex(assert_value_size(dec_to_bin(i), 2) + assert_value_
  return ret


S = [
    ["1", "2", "3", "4"],
     ["5", "6", "7", "8"],
     ["9", "A", "B", "C"],
     ["D", "E", "F", "0"]
]
INV_S = gen_inv_s_box(S)
M = [
     ["1", "4"],
     ["4", "1"]
]
INV_M = [
    ["9", "2"],
    ["2", "9"]
]
print(INV_S)
```

```
[['f', '0', '1', '2'], ['3', '4', '5', '6'], ['7', '8', '9', 'a'], ['b', 'c
```

Algorithm necessary functions

```
def sub_nib(x, s):
  ret = ""
  for i in range(0, len(x), 4):
    nib = x[i:i+4]
    r, c = get_indices(nib)
    ret += hex_to_bin(s[r][c])
  return ret

def sub_nibs(x, s):
  for i in range(len(x)):
    for j in range(len(x[i])):
      x[i][j] = sub_nib(x[i][j], s)
  return x


def mixcol(A, B):
  ret = [
      [None, None],
      [None, None]
  ]
  for i in [0, 1]:
    for j in [0, 1]:
      ret[i][j] = add_nib(mul_nib(A[i][0], B[0][j]), mul_nib(A[i][1], B[1][j]))
  return ret


def shift_row(state):
  state[1][0], state[1][1] = state[1][1], state[1][0]
  return state


def add_round_key(state, key):
  k_mat = list_to_mat(nibble_list(key))
  for i in range(2):
    for j in range(2):
      state[i][j] = xor(state[i][j], k_mat[i][j])
  return state


def get_subkey(prev_key, t):
  w0, w1 = split_str(prev_key)
  w2 = w0
  w2 = xor(w2, t)
  w2 = xor(w2, sub_nib(rot_nib(w1), S))
  w3 = xor(w2, w1)
  return w2 + w3


def gen_subkeys(key):
  key0 = key
  key1 = get_subkey(key0, hex_to_bin("80"))
  key2 = get_subkey(key1, hex_to_bin("60"))
  return key0, key1, key2


def encrypt(plaintext, key):
```

```
def encrypt(plaintext, key):
  key0, key1, key2 = gen_subkeys(key)
  state = list_to_mat(nibble_list(plaintext))

  # Round 0
  state = add_round_key(state, key0)

  #Round 1
  state = sub_nibs(state, S)
  state = shift_row(state)
  state = mixcol(M, state)
  state = add_round_key(state, key1)

  # Round 2
  state = sub_nibs(state, S)
  state = shift_row(state)
  state = add_round_key(state, key2)


  ciphertext = "".join(mat_to_list(state))

  return ciphertext


def decrypt(ciphertext, key):
  key0, key1, key2 = gen_subkeys(key)
  state = list_to_mat(nibble_list(ciphertext))

  # Inv round 2
  state = add_round_key(state, key2)
  state = shift_row(state)
  state = sub_nibs(state, INV_S)

  # Inv round 1
  state = add_round_key(state, key1)
  state = mixcol(INV_M, state)
  state = shift_row(state)
  state = sub_nibs(state, INV_S)

  # Inv round 0
  state = add_round_key(state, key0)

  plaintext = "".join(mat_to_list(state))
  return plaintext
```

Testing

```
plaintext = hex_to_bin("BC78")
key = hex_to_bin("2B85")


c = encrypt(plaintext, key)
p = decrypt(c, key)
```

```
assert(p == plaintext)
```

Assignment 3

Diffie-Hellman Key Exchange

[Reference]

```
def fpow(a, b, m):
  if b == 0:
    return 1
  r = fpow(a, b//2, m)
  r = (r * r) % m
  if b % 2 == 1:
    r = (r * a) % m
  return r
```

```
# Global variables
P = 23
```

```
# Calculating G (primitive root)
G = 0
for r in range(1, P, 1):
  s = set()
  for x in range(P-1):
    s.add(fpow(r, x, P))
  if len(s) == P-1:
    G = r
    break;
print(G)
```

```
5
```

```
# Private keys
Ra = 3
Rb = 4
```

```
# Public keys
Ua = fpow(G, Ra, P)
Ub = fpow(G, Rb, P)
```

```
# Symmetric key calculated by A and B
symm_key_a = fpow(Ub, Ra, P) # A has access to B's public key and A's private ke
symm_key_b = fpow(Ua, Rb, P) # B has access to A's public key and B's private ke
print(symm_key_a)
```

```
18
```

```
assert(symm_key_a == symm_key_b)
```

Assignment 4

RSA Algrithm

[Reference]()

Helper functions

```
def gcd(a, b):
  if a == 0:
    return b
  return gcd(b % a, a)
```

```
def mod_pow(a, b, m):
  if b==0:
    return 1
  r = mod_pow(a, b//2, m)
  r = (r * r) % m
  if b % 2 == 1:
    r = (r * a) % m
  return r
```

Generating keys

```
P = 53
Q = 59
```

```
n = P * Q
phi_n = (P-1) * (Q-1)
```

```
# Generating e
e = 2
while e < phi_n:
  if gcd(e, phi_n) == 1:
    break
  e += 1
```

```
# Generating d
k = 1
while (k * phi_n + 1) % e != 0:
  k += 1
```

```
  k += 1
d = (k * phi_n + 1) // e


U = [e, n] # Public key
R = [d, n] # Private key


print("Primes:\t\t", P, ",", Q)
print("N:\t\t", n)
print("phi(N):\t\t", phi_n)
print("e:\t\t", e)
print("d:\t\t", d)
print("Public key:\t", "[e, n] =", U)
print("Private key:\t", "[d, n] =", R)
```

```
Primes:          53 , 59
N:               3127
phi(N):          3016
e:               3
d:               2011
Public key:      [e, n] = [3, 3127]
Private key:     [d, n] = [2011, 3127]
```

Testing

```
def encrypt(P, U):
  e, n = U
  c = mod_pow(P, e, n)
  return c


def decrypt(C, R):
  d, n = R
  ret = mod_pow(C, d, n)
  return ret


plaintext = 89


c = encrypt(plaintext, U)
p = decrypt(c, R)


assert(p == plaintext)
```

Assignment 5

Elliptic Curve Cryptography (ECC)

[Reference 1]

[Reference 2]

[Reference 3]

```
P = 11


def modmul(a, b, m = P):
  return ((a % m) * (b % m)) % m

def mod_pow(a, b, m = P):
  if b == 0:
    return 1
  r = mod_pow(a, b//2, m)
  r = (r*r)%m
  if b%2:
    r = (r*a)%m
  return r

def moddiv(a, b, m = P):
  return modmul(a, mod_pow(b, m-2, m), m)
```

Classes

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def __eq__(self, p2):
    return self.x == p2.x and self.y == p2.y
  def __str__(self) -> str:
      return f"({self.x}, {self.y})"


class EllipticCurve:
  def __init__(self, a, b):
    self.a = a
    self.b = b

  def add(self, p1, p2, m = P):
    l = 0
    if p1 == p2:
      num = 3 * p1.x * p1.x + self.a
      den = 2 * p1.y
    else:
      num = p2.y - p1.y
      den = p2.x - p1.x
    l = moddiv(num, den, m)
    x3 = (l*l - p1.x - p2.x) % m
    y3 = (l*(p1.x - x3) - p1.y) % m
    return Point(x3, y3)

  def mul(self, k, p): # p is always generator point G
```

```
  def mul(self, k, p): # p is always generator point G
    temp = p
    while k != 1:
      temp = self.add(temp, p)
      k -= 1
    return temp

  def sub(self, p1, p2):
    np = Point(p2.x, -p2.y)
    return self.add(p1, np)
```

Constants

```
curve = EllipticCurve(2, 4) # Points lying on this curve:{0, 2}, {0, 5}, {1, 0},
G = Point(0, 2)
```

Algorithm specific functions

```
def encrypt(p, U):
  k = 5
  c = [
       curve.mul(k, G),
       curve.add(p, curve.mul(k, U))
  ]
  return c
```

```
def decrypt(C, R):
  p = curve.sub(C[1], curve.mul(R, C[0]))
  return p
```

Testing

```
R = 5               # Private key
U = curve.mul(R, G) # Public key
```

```
plaintext = Point(3, 4)
```

```
ciphertext = encrypt(plaintext, U)
p = decrypt(ciphertext, R)
```

```
print(p)
```

```
(3, 4)
```

```
assert(p == plaintext)
```

```
        assert(p == plaintext)
```