Homework 1 Clustering and Regression

Instructions

Answer the questions and upload your answers to courseville. Answers can be in Thai or English. Answers can be either typed or handwritten and scanned. the assignment is divided into several small tasks. Each task is weighted equally (marked with **T**). For this assignment, each task is awarded 1 points. There are also optional tasks (marked with **OT**) counts for 0.5 points each.

Metrics

In a population where the amount of cats is equal to the amount of dogs. Considering the following classification results from a classifier.

Model A	Predicted dog	Predicted cat
Actual dog	30	20
Actual cat	10	40

T1. What is the accuracy of Model A? 0.7000

T2. Consider cats as 'class 1' (positive) and dogs as 'class 0' (negative), calculate the prediction, recall, and F1. Precision 0.6667 Recall 0.800 F1 0.7273

T3. Consider class cat as 'class 0' and class dog as 'class 1', calculate the prediction, recall, and F1. Precision 0.75 Recall 0.60 F1 0.6667

It is important to specify the 'positive' class when you calculate precision, recall, and F1. If there are more than two classes, it is usually done in a one-versus-all setting where one class is considered positive and the rest of the classes are considered negative.

T4. Now consider a lopsided population where there are 80% cats. What is the accuracy of Model A? Using dog as the positive class, what is the precision, recall, and F1? Explain how and why these numbers change (or does not change) from the previous questions. Not change since model doesn't indicate how imbalanced data is.

OT1. Consider the equations for accuracy and F1

$$\begin{aligned} Accuracy &= \frac{TP+TN}{TP+TN+FP+FN} \\ F1 &= \frac{2TP}{2TP+FP+FN} \end{aligned} \tag{1}$$

When will accuracy be equal, greater, or less than F1?

Given $FP+FN=C$

$\lim_{C\to 0} \frac{TP+TN}{TP+TN+C} = 1$

$\lim_{C\to 0} \frac{2TP}{2TP+C} = 1$

So when $TP+TN=0$ accuracy & F1 will be equal

$\lim_{C\to \infty} \frac{TP+TN}{TP+TN+C} = \frac{TP+TN}{\infty}$

$\lim_{C\to \infty} \frac{2TP}{2TP+C} = \frac{2TP}{\infty}$

$2TP = TP+TP$

So F1 will be lesserthan acc when $TN>TP$ and will greater than acc when $TP>TN$

Hello Clustering

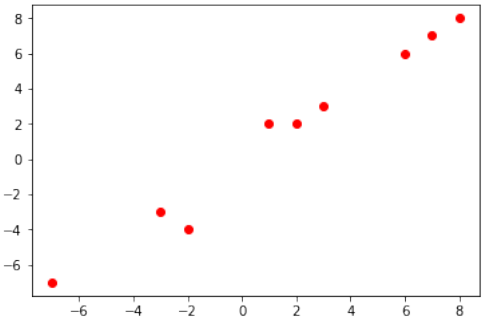
Recall from lecture that K-means has two main steps: the points assignment step, and the mean update step. After the initialization of the centroids, we assign each data point to a centroid. Then, each centroids are updated by re-estimating the means.

Concretely, if we are given N data points, x_1, x_2, ..., x_N, and we would like to form K clusters. We do the following;

1. **Initialization**: Pick K random data points as K centroid locations c_1, c_2, ..., c_K.
2. **Assign**: For each data point k, find the closest centroid. Assign that data point to the centroid. The distance used is typically Euclidean distance.
3. **Update**: For each centroid, calculate the mean from the data points assigned to it.
4. **Repeat**: repeat step 2 and 3 until the centroids stop changing (convergence).

Given the following data points in x-y coordinates (2 dimensional)

x	y
1	2
3	3
2	2
8	8
6	6
7	7
-3	-3
-2	-4
-7	-7



T5. If the starting points are (3,3), (2,2), and (-3,-3). Describe each assign and update step. What are the points assigned? What are the updated centroids? You may do this calculation by hand or write a program to do it.

T6. If the starting points are (-3,-3), (2,2), and (-7,-7), what happens?

T7. Between the two starting set of points in the previous two questions, which one do you think is better? How would you measure the 'goodness' quality of a set of starting points?

In general, it is important to try different sets of starting points when doing k-means.

OT2. What would be the best K for this question? Describe your reasoning.

My heart will go on[1]

In this part of the exercise we will work on the Titanic dataset provided by Kaggle. The Titanic dataset contains information of the passengers boarding the Titanic on its final voyage. We will work on predicting whether a given passenger will survive the trip.

Let's launch Jupyter and start coding!

We start by importing the data using Pandas

```
train_url = "http://s3.amazonaws.com/assets.datacamp.com/course/Kaggle/train.csv"
train = pd.read_csv(train_url) #training set

test_url = "http://s3.amazonaws.com/assets.datacamp.com/course/Kaggle/test.csv"
test = pd.read_csv(test_url) #test set
```

Both `train` and `test` are dataframes. Use the function `train.head()` and `train.tail()` to explore the data. What do you see?

Use the function `describe()` to get a better understanding of the data. You can read the meaning of the data fields at `https://www.kaggle.com/c/titanic/data`

[1] Many parts of this exercise are adapted from Kaggle Python Tutorial on Machine Learning

Looking at the data, you will notice a lot of missing values. For example, some age is NaN. This is normal for real world data to have some missing values. There are several ways to handle missing values. The simplest is to throw away any rows that have missing values. However, this usually reduce the amount of training data you have. Another method is to guess what the missing value should be. The simplest guess is to use the Median or Mode of the data. For this exercise we will proceed with this.

T8. What is the median age of the training set? You can easily modify the age in the dataframe by

```
train["Age"] = train["Age"].fillna(train["Age"].median())
```

Note that you need to modify the code above a bit to fill with mode() because mode() returns a series rather than a single value.

T9. Some fields like 'Embarked' are categorical. They need to be converted to numbers first. We will represent S with 0, C with 1, and Q with 2. What is the mode of Embarked? Fill the missing values with the mode. You can set the value of Embarked easily with the following command.

```
train.loc[train["Embarked"] == "S", "Embarked"] = 0
```

Do the same for Sex.

T10. Write a logistic regression classifier using gradient descent as learned in class. Use PClass, Sex, Age, and Embarked as input features. You can extract the features from Pandas to Numpy by

```
data = np.array(train[["PClass","Sex","Age","Embarked"]].values)
```

Check the datatype of each values in data, does it make sense? You can force the data to be of any datatype by using the command

```
data = np.array(train[["PClass","Sex","Age","Embarked"]].values, dtype = float)
```

When you evaluate the trained model on the test set, you will need to make a final decision. Since logistic regression outputs a score between 0 and 1, you will need to decide whether a score of 0.3 (or any other number) means the passenger survive or not. For now, we will say if the score is greater than or equal to 0.5, the passenger survives. If the score is lower than 0.5 the passenger will be dead. This process is often called 'Thresholding.' We will talk more about this process later in class.

To evaluate your results, we will use Kaggle. Kaggle is a website that hosts many machine learning competitions. Many companies put up their data as a problem for anyone to participate. If you are looking for a task for your course project, Kaggle might be a good place to start. You will need to make sure that your output is in line with the submission requirements of Kaggle: a csv file with exactly 418 entries and two columns: PassengerId and Survived. Then,

use the code provided to make a new data frame using `DataFrame()`, and create a csv file using `to_csv()` method from Pandas.

To submit your prediction, you must first sign-up for an account on `Kaggle.com`. Click participate to the competition at `https://www.kaggle.com/c/titanic/` then submit your csv file for the score.

The output file should have two columns: the passengerId and a 0,1 decision (0 for dead, 1 for survive). As shown below:

```
PassengerId,Survived
892,0
893,1
894,0
```

T11. Submit a screenshot of your submission (with the scores). Upload your code to courseville.

T12. Try adding some higher order features to your training (x_1^2, x_1x_2,...). Does this model has better **accuracy on the training set**? How does it perform on the **test set**?

T13. What happens if you reduce the amount of features to just Sex and Age?

OT3. We want to show that matrix inversion yields the same answer as the gradient descent method. However, there is no closed form solution for logistic regression. Thus, we will use normal linear regression instead. Re-do the Titanic task as a regression problem by using linear regression. Use the gradient descent method.

OT4. Now try using matrix inversion instead. However Are the weights learned from the two methods similar? Report the Mean Squared Errors (MSE) of the difference between the two weights.

[Optional] Fun with matrix algebra

Prove the following statements. All of them can be solved by first expanding out the matrix notation as a combination of their elements, and then use the definitions of trace and matrix derivatives to help finish the proof. For example, the (i, j) element of $Y = AB$ is $Y_{i,j} = \Sigma_m A_{i,m} B_{m,j}$.

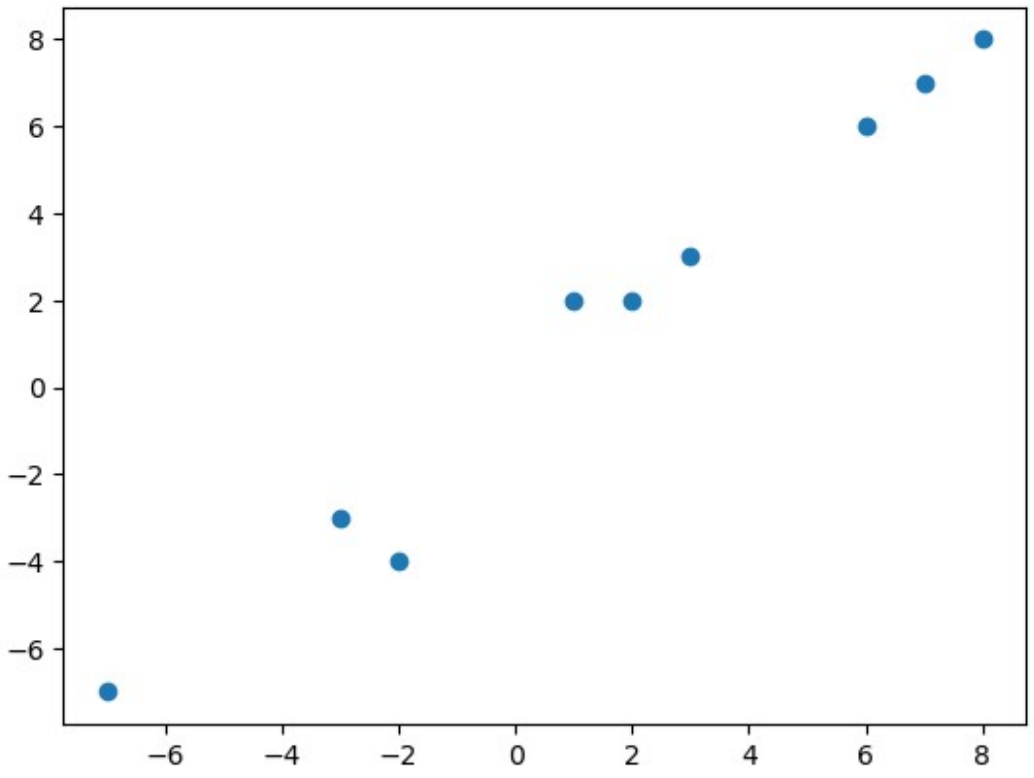
OT5. $\nabla_A trAB = B^T$

OT6. $\nabla_{A^T} f(A) = (\nabla_A f(A))^T$

OT7. $\nabla_A trABA^TC = CAB + C^TAB^T$

Hint: Try first solving the easier equation of $\nabla_A trBAC = (CB)^T = B^TC^T$

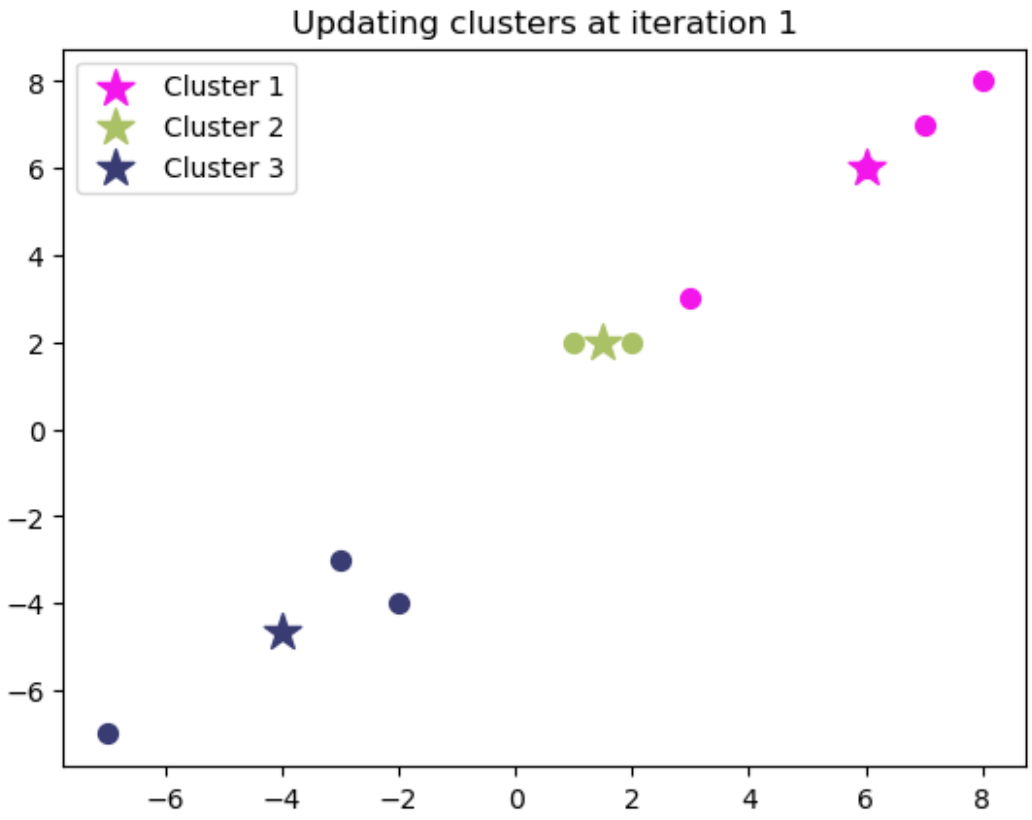
```python
import matplotlib.pyplot as plt
import numpy as np

data_point = np.array([
  [1,2],
  [3,3],
  [2,2],
  [8,8],
  [6,6],
  [7,7],
  [-3,-3],
  [-2,-4],
  [-7,-7],
])

x = data_point[:,0]
y = data_point[:,1]

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




```python
class KNN:
  def __init__(self,clusters,data_points) -> None:
    self.clusters = clusters
    self.data_points = data_points
```

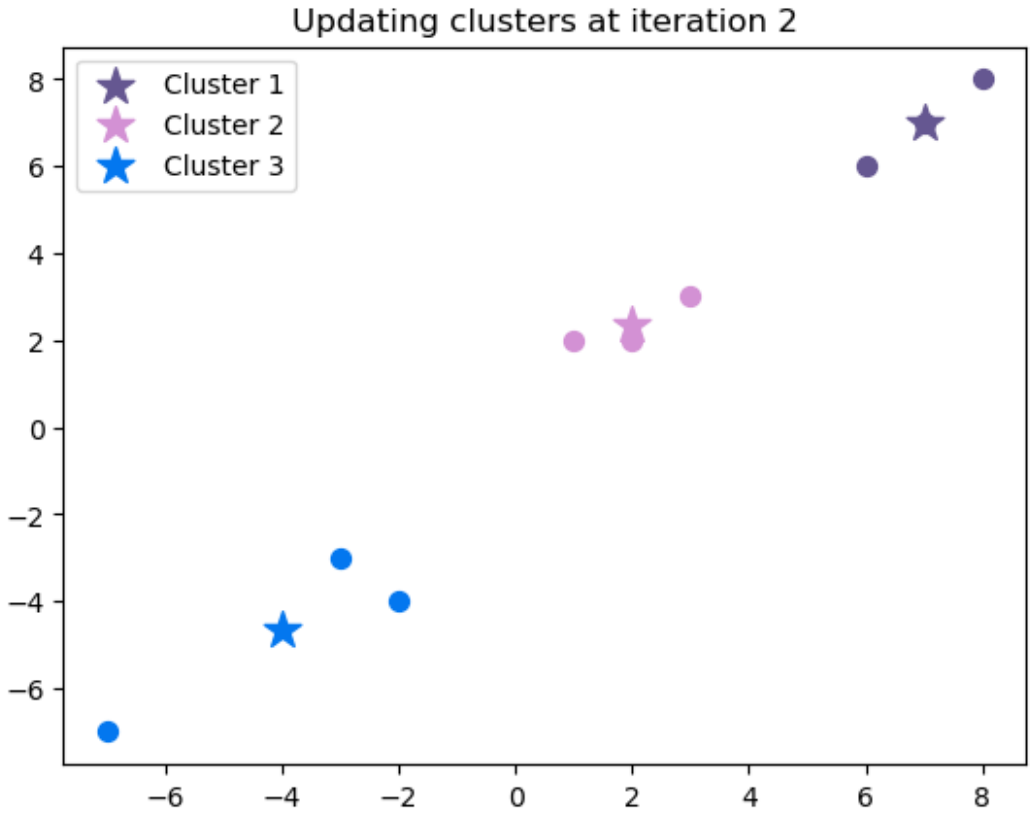
```
    self.plot_colors = ['r', 'g', 'b', 'y', 'c', 'm']

  def __plot(self,data, set_cluster, clusters, colors,title="KNN"):
    labels = [f"Cluster {i+1}" for i in range(len(colors))]
    data_x = data[:, 0]
    data_y = data[:, 1]

    for i in range(len(data)):
        plt.scatter(data_x[i], data_y[i], marker="o",
color=colors[set_cluster[i]], s=50)

    for k in range(len(clusters)):
        kx = clusters[k][0]
        ky = clusters[k][1]
        plt.scatter(kx, ky, marker="*", color=colors[k], s=200, label
= labels[k])
    plt.legend()
    plt.title(title)
    plt.show()

  def __euclidian_distance(self,p1,p2):
    return np.sqrt(np.sum(np.square((p1-p2)),axis=1))

  def assign(self):
    cluster_distance = []
    for centroid in self.clusters:
      dist = self.__euclidian_distance(centroid,self.data_points)
      cluster_distance.append(dist)
    set_cluster = np.argmin(cluster_distance,axis=0)
    return set_cluster

  def update(self,set_cluster):
    new_clusters = []
    for i in range(len(self.clusters)):

new_clusters.append(np.mean(self.data_points[set_cluster==i],axis=0))
    self.clusters = new_clusters
    return self.clusters

  def cost(self):
    set_cluster = self.assign()
    cost = 0
    for i in range(len(self.clusters)):
      x  = np.squeeze(self.data_points[np.argwhere(set_cluster==i)])
      dist = np.sqrt(np.sum(np.square((x-self.clusters[i]))))
      cost += dist
    return cost

  def plot(self):
```

```
self.__plot(self.data_points,self.assign(),self.clusters,np.random.ran
d(len(self.clusters),3),"KNN")

  def run(self):
    old_cost = float('inf')
    for i in range(1,100):
      set_cluster = self.assign()

self.__plot(self.data_points,set_cluster,self.clusters,np.random.rand(
len(self.clusters),3),f"Assigning clusters at iteration {i}")
      self.update(set_cluster)

self.__plot(self.data_points,set_cluster,self.clusters,np.random.rand(
len(self.clusters),3),f"Updating clusters at iteration {i}")
      print("Cost function =",self.cost())
      if old_cost == self.cost():
        print(f"Converged with {i} iterations")
        break
      old_cost = self.cost()

clusters = np.array([[3,3],[2,2],[-3,-3]])
knn = KNN(clusters,data_point)

knn.run()
```

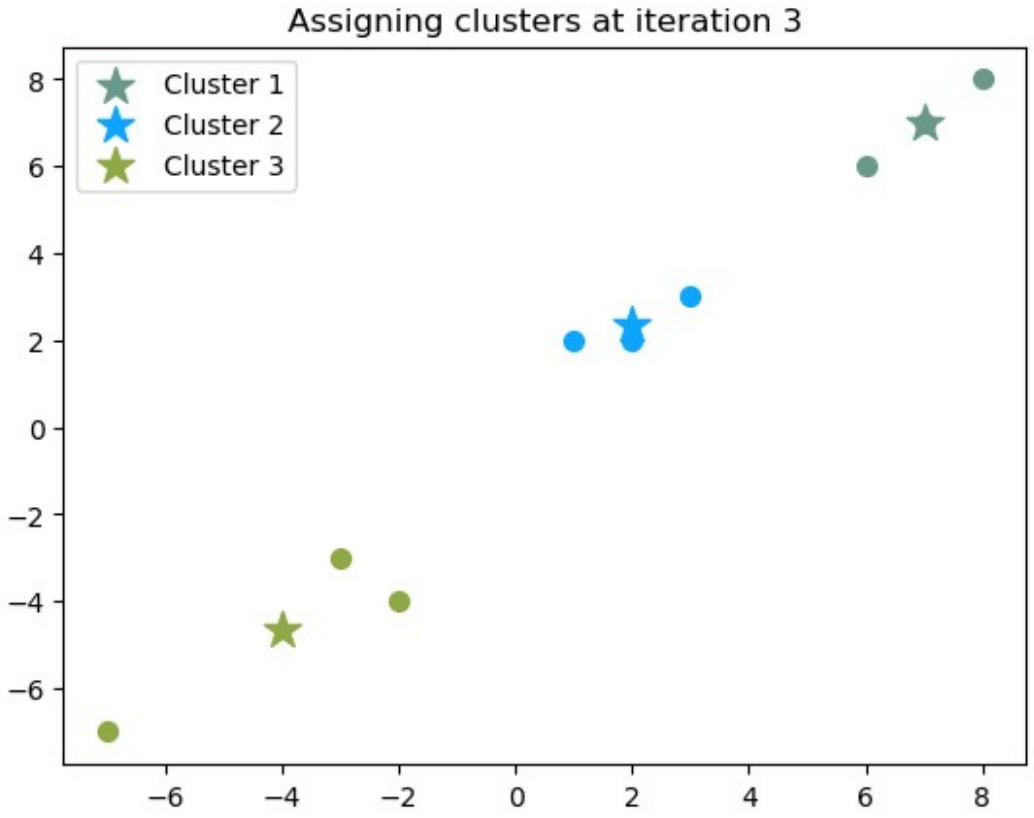


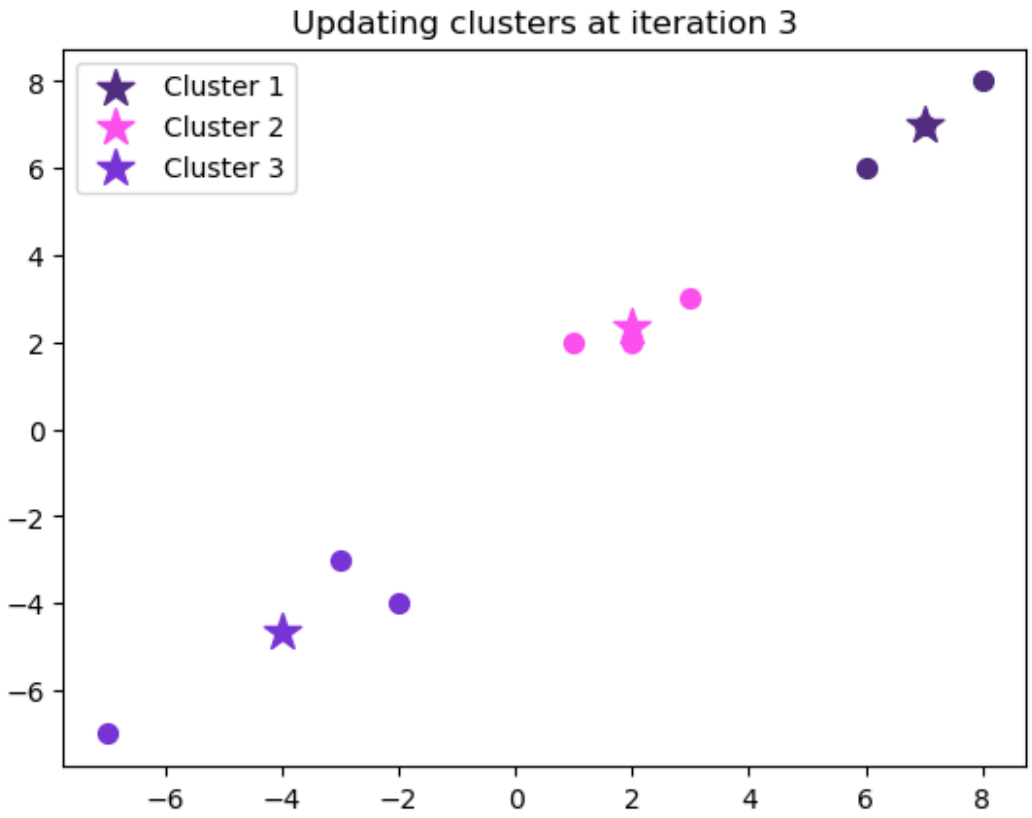

```
Cost function = 9.85972161896732
```


Assigning clusters at iteration 2
Cluster 1
Cluster 2
Cluster 3
8
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6
8

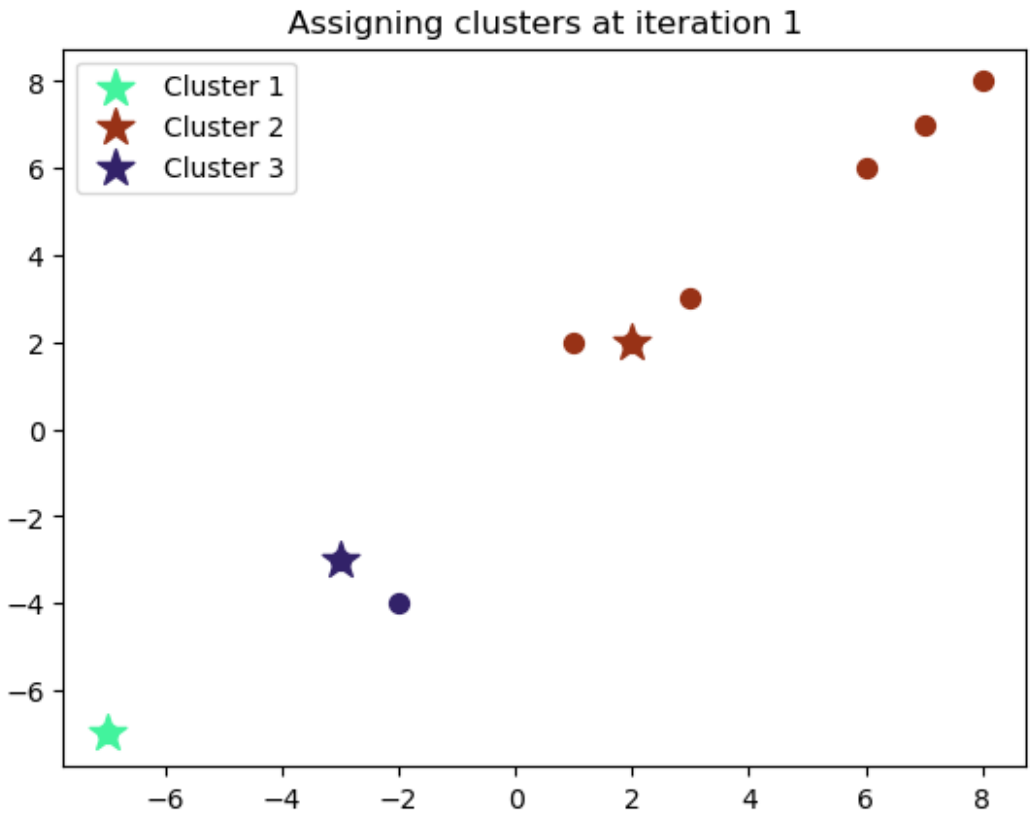

Updating clusters at iteration 2
Cluster 1
Cluster 2
Cluster 3
8
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6
8

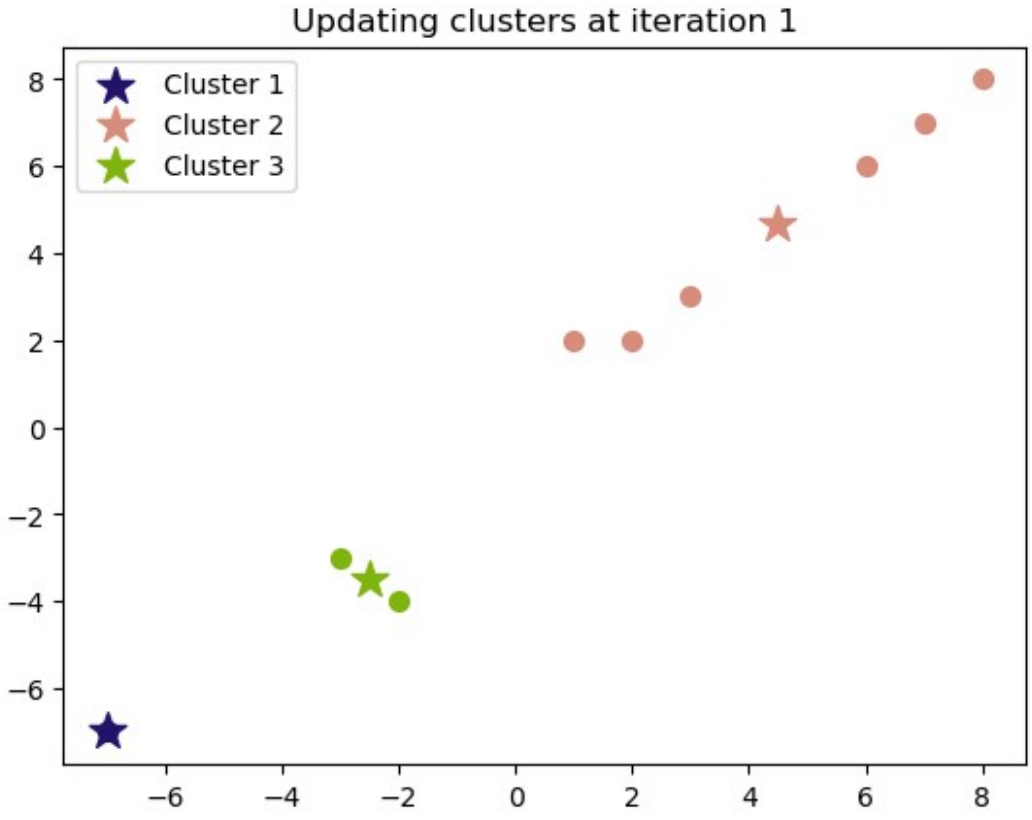
```
Cost function = 8.393945447550685
```



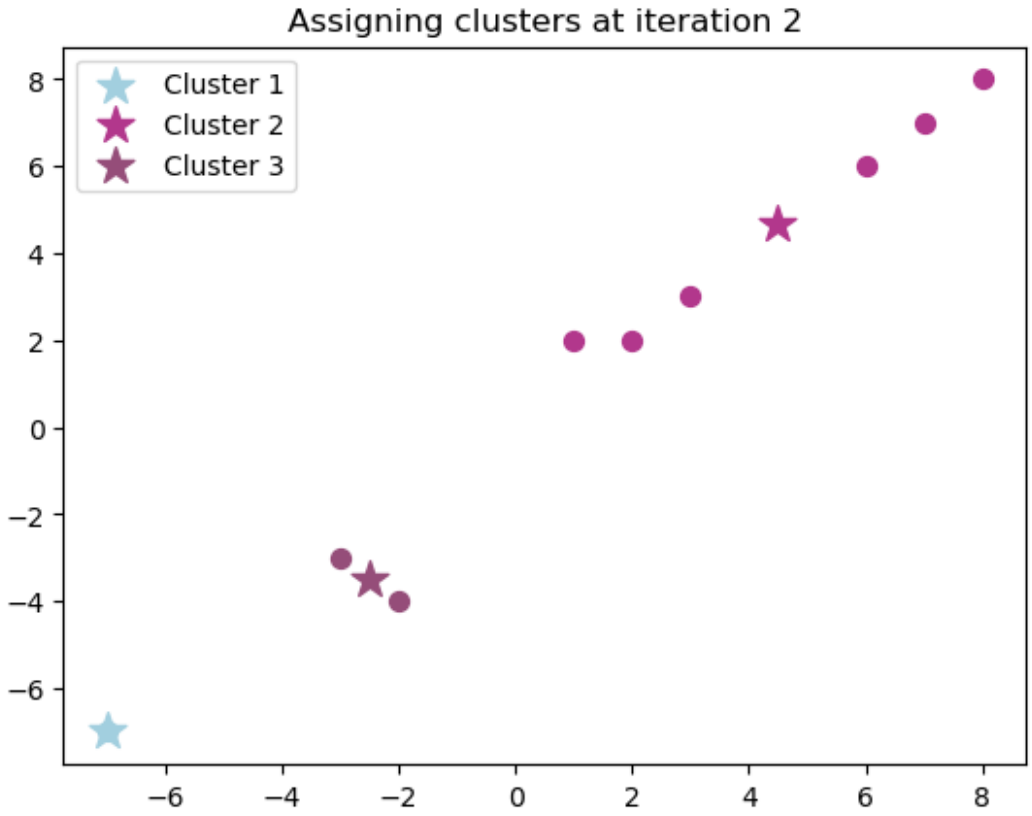


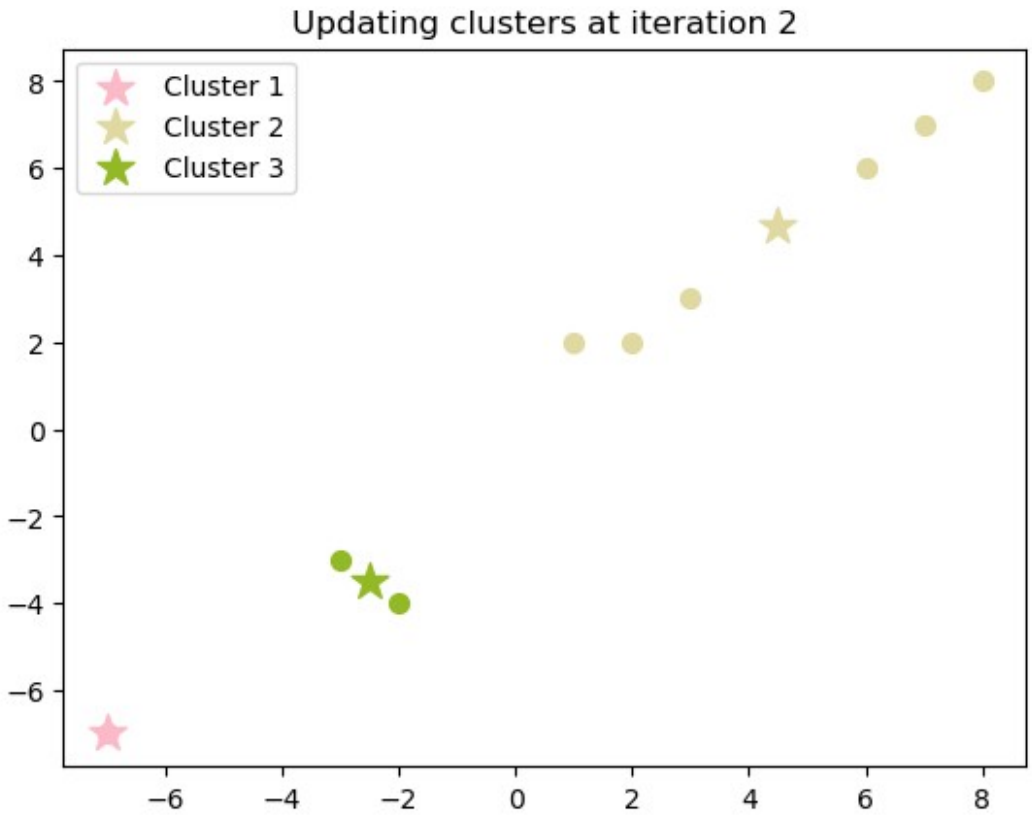

```
Cost function = 8.393945447550685
Converged with 3 iterations

# T5: Updated centroids
[tuple(centroid) for centroid in knn.clusters]

[(7.0, 7.0), (2.0, 2.3333333333333335), (-4.0, -4.666666666666667)]

clusters = np.array([[-7,-7],[2,2],[-3,-3]])
knn2 = KNN(clusters,data_point)
knn2.run()
```


Assigning clusters at iteration 1
Cluster 1
Cluster 2
Cluster 3
8
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6
8


Updating clusters at iteration 1
Cluster 1
Cluster 2
Cluster 3
8
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6
8

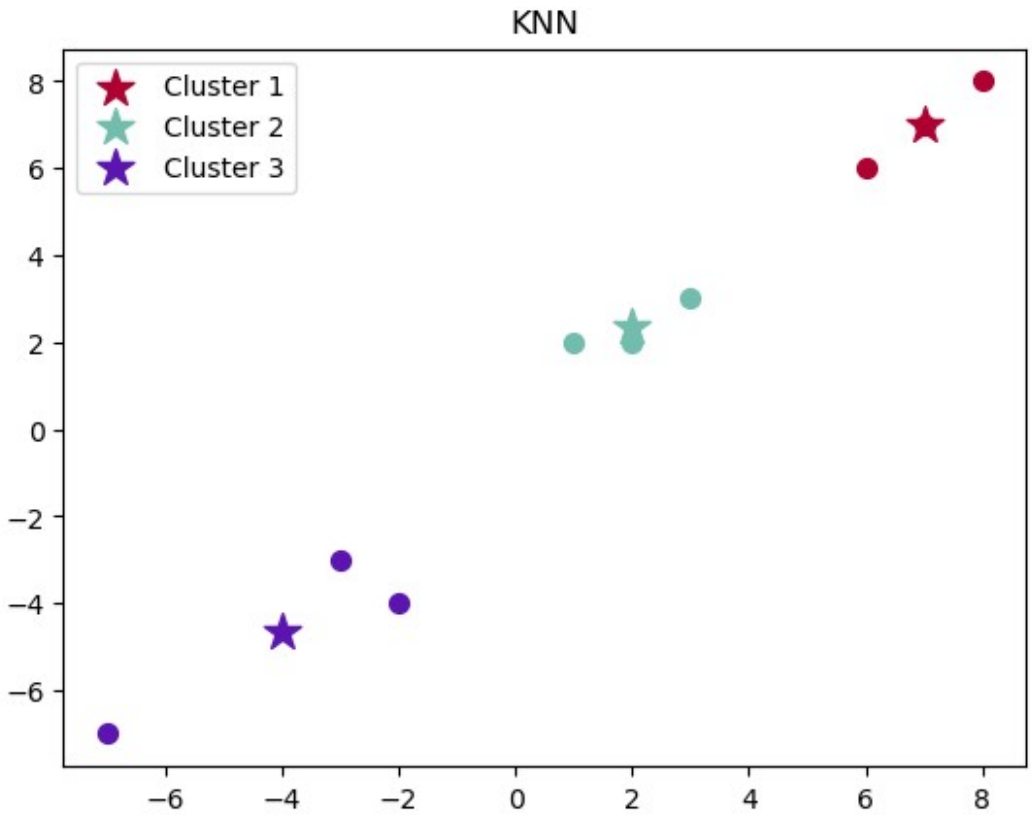
```
Cost function = 9.765462528203138
```

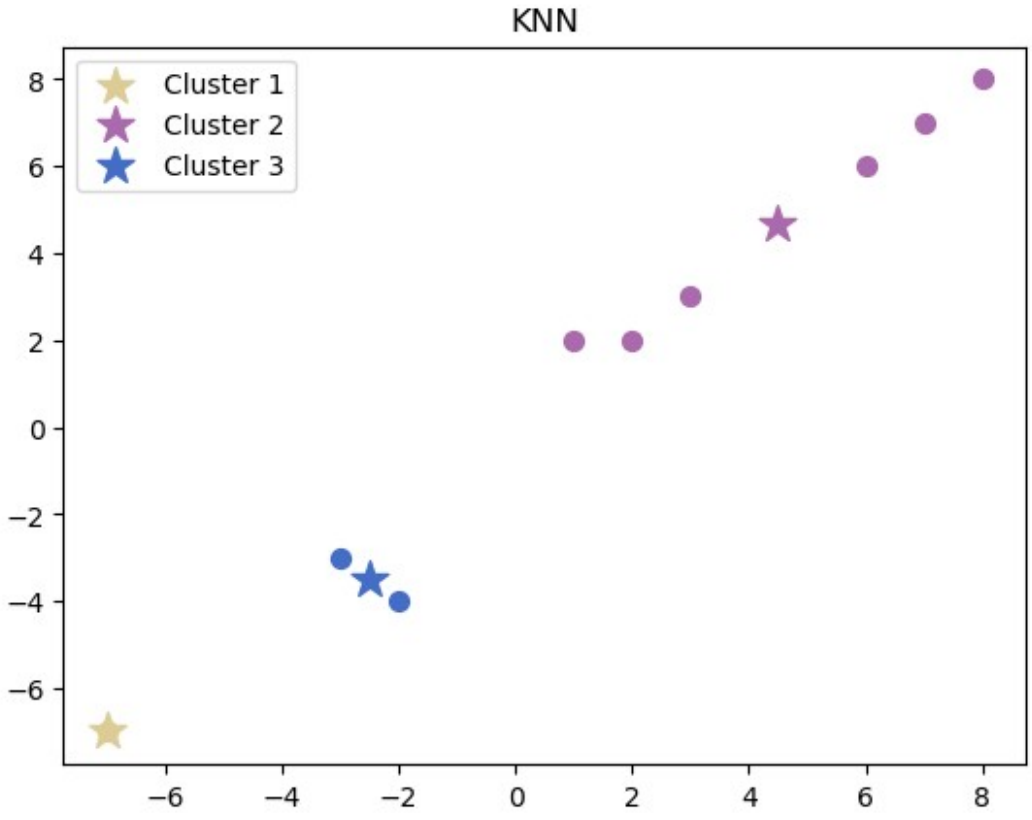





```
Cost function = 9.765462528203138
Converged with 2 iterations
```

```
# T6: Changed Centroids
knn.plot()
knn2.plot()
print("Old centroids (T5) :",[tuple(centroid) for centroid in
knn.clusters])
print("New centroids :",[tuple(centroid) for centroid in
knn2.clusters])

# Explaination :
### The algorithm is not converging to the same centroids because the
initial centroids are different.
### That means initial centroids can affect the final result of the
algorithm.
```


KNN
Cluster 1
Cluster 2
Cluster 3
8
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6
8


KNN
Cluster 1
Cluster 2
Cluster 3
8
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6
8

```
Old centroids (T5) : [(7.0, 7.0), (2.0, 2.3333333333333335), (-4.0, -4.666666666666667)]
New centroids : [(-7.0, -7.0), (4.5, 4.666666666666667), (-2.5, -3.5)]
```

```
# T6: Changed Centroids
print("Old centroids (T5) :",[tuple(centroid) for centroid in knn.clusters])
print("New centroids :",[tuple(centroid) for centroid in knn2.clusters])

# Explaination :
### The algorithm is not converging to the same centroids because the initial centroids are different.
### That means initial centroids can affect the final result of the algorithm.
```

```
Old centroids (T5) : [(7.0, 7.0), (2.0, 2.3333333333333335), (-4.0, -4.666666666666667)]
New centroids : [(-7.0, -7.0), (4.5, 4.666666666666667), (-2.5, -3.5)]
```

```
#T7: Which is better?
knn.cost(),knn2.cost()

# Explaination :
### The cost function of the first algorithm is better because it is lower than the second one.
### That means the first algorithm is better than the second one.
```

```
(8.393945447550685, 9.765462528203138)
```

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
train_url = "http://s3.amazonaws.com/assets.datacamp.com/course/Kaggle/train.csv"
train_df = pd.read_csv(train_url) #training set
test_url = "http://s3.amazonaws.com/assets.datacamp.com/course/Kaggle/test.csv"
test_df = pd.read_csv(test_url) #test set
```

In [3]:
```
train_df.isna().sum()
# So Age, Cabin and Embarked have missing values
```

Out[3]:
```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

In [4]:
```
test_df.isna().sum()
```

Out[4]:
```
PassengerId      0
Pclass           0
Name             0
Sex              0
Age             86
SibSp            0
Parch            0
Ticket           0
Fare             1
Cabin          327
Embarked         0
dtype: int64
```

In [5]:
```
train_df.info()
test_df.info()
# So we will have to impute the missing values
# Seem like Cabin missing TOO MUCH values, so we will drop it later
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object 
 4   Sex          891 non-null    object 
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object 
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object 
 11  Embarked     889 non-null    object 
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 418 entries, 0 to 417
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  418 non-null    int64  
 1   Pclass       418 non-null    int64  
 2   Name         418 non-null    object 
 3   Sex          418 non-null    object 
 4   Age          332 non-null    float64
 5   SibSp        418 non-null    int64  
 6   Parch        418 non-null    int64  
 7   Ticket       418 non-null    object 
 8   Fare         417 non-null    float64
 9   Cabin        91 non-null     object 
 10  Embarked     418 non-null    object 
dtypes: float64(2), int64(4), object(5)
memory usage: 36.1+ KB
```

In [6]:
```
# Fill values with median !!FROM TRAINING DATA!!
train_df["Age"] = train_df["Age"].fillna(train_df["Age"].median())
test_df["Age"] = test_df["Age"].fillna(train_df["Age"].median())

# T8: Median age of training data is
train_df["Age"].median()
```

Out[6]: 28.0

In [7]:
```
# Fill values with most common value !!FROM TRAIN_dfING DATA!!
train_df["Embarked"] = train_df["Embarked"].fillna(train_df["Embarked"].value_count
test_df["Embarked"] = test_df["Embarked"].fillna(train_df["Embarked"].value_counts(

# T9: Most common port of embarked is
train_df["Embarked"].value_counts().idxmax()
```

Out[7]: 'S'

In [8]:
```
# Fare and PClass seems to be correlated, so we will use PClass to impute Fare
test_df["Fare"] = test_df["Fare"].fillna(train_df.groupby("Pclass")["Fare"].transfo
```

In [9]:
```
embarked_categories = dict([(k,i) for i,k in enumerate(train_df["Embarked"].astype(
train_df["EmbarkedClass"] = train_df["Embarked"].map(embarked_categories)
test_df["EmbarkedClass"] = test_df["Embarked"].map(embarked_categories)
```

In [10]:
```
sex_categories = dict([(k,i) for i,k in enumerate(train_df["Sex"].astype('category'
train_df["SexClass"] = train_df["Sex"].map(sex_categories)
test_df["SexClass"] = test_df["Sex"].map(sex_categories)
```

In [11]:
```
train_df.isna().sum(),test_df.isna().sum()

# Data is quite cleaned!!
```

Out[11]:
```
(PassengerId        0
 Survived           0
 Pclass             0
 Name               0
 Sex                0
 Age                0
 SibSp              0
 Parch              0
 Ticket             0
 Fare               0
 Cabin            687
 Embarked           0
 EmbarkedClass      0
 SexClass           0
 dtype: int64,
 PassengerId        0
 Pclass             0
 Name               0
 Sex                0
 Age                0
 SibSp              0
 Parch              0
 Ticket             0
 Fare               0
 Cabin            327
 Embarked           0
 EmbarkedClass      0
 SexClass           0
 dtype: int64)
```

In [12]:
```
train_data = np.array(train_df[["Pclass","SexClass","Age","EmbarkedClass"]].values,
train_label = np.array(train_df["Survived"].values,dtype=np.int8).reshape(-1,1)
train_data
```

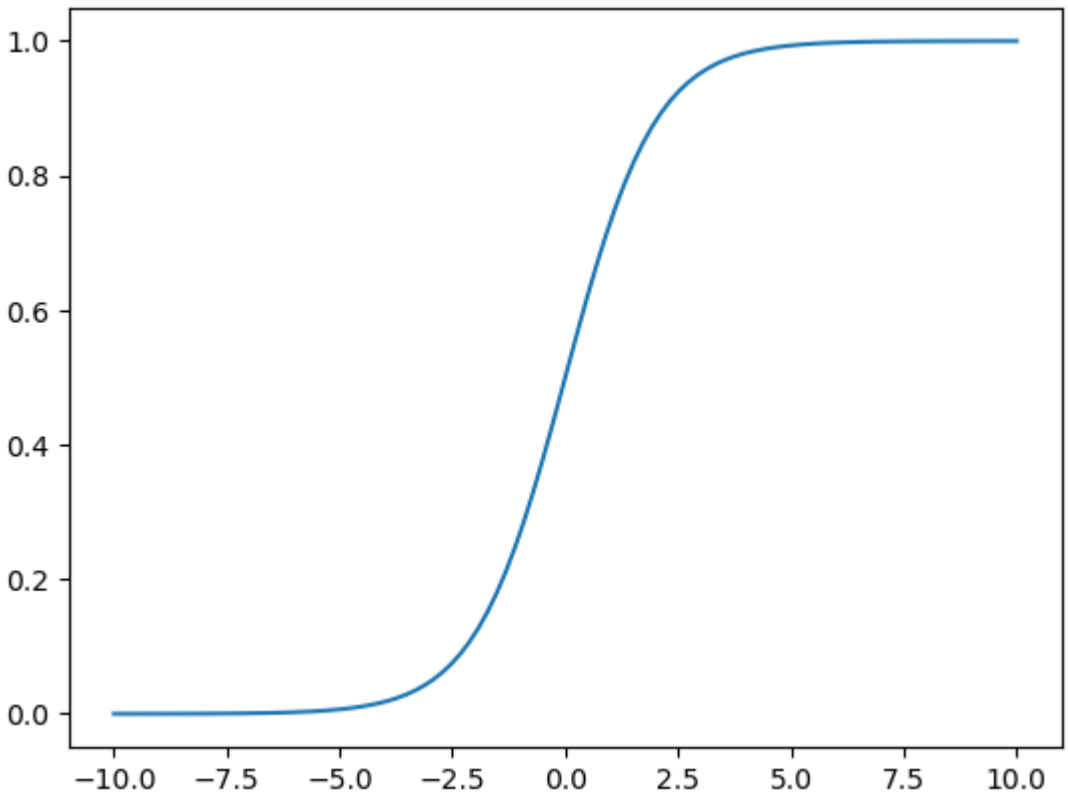
Out[12]:
```
array([[ 3.,  1., 22.,  2.],
       [ 1.,  0., 38.,  0.],
       [ 3.,  0., 26.,  2.],
       ...,
       [ 3.,  0., 28.,  2.],
       [ 1.,  1., 26.,  0.],
       [ 3.,  1., 32.,  1.]], dtype=float32)
```

In [13]:
```
test_data = np.array(test_df[["Pclass","SexClass","Age","EmbarkedClass"]].values,dt
test_data
```

Out[13]:
```
array([[ 3. ,  1. , 34.5,  1. ],
       [ 3. ,  0. , 47. ,  2. ],
       [ 2. ,  1. , 62. ,  1. ],
       ...,
       [ 3. ,  1. , 38.5,  2. ],
       [ 3. ,  1. , 28. ,  2. ],
       [ 3. ,  1. , 28. ,  0. ]], dtype=float32)
```

In [14]:
```
def sigmoid(z): return 1/(1+np.exp(-z))

x = np.linspace(-10,10,100)
z = sigmoid(x)
plt.plot(x,z)
```

Out[14]:
```
[<matplotlib.lines.Line2D at 0x7f93fad7f3e0>]
```



In [15]:
```
class LogisticRegression:
  def  __init__(self,train_x,train_y,test_x,learning_rate=1e-3) -> None:
    self.train_x = train_x
    self.train_y = train_y
```

```
    self.pred = np.zeros_like(train_y)
    self.pred_class = np.zeros_like(train_y)
    self.test_x = test_x
    self.cost,self.grad = self.cost_function()
    self.learning_rate = learning_rate
    self.W = self.__random_init_param()
    self.test_accuracies = []

  def __random_init_param(self):
    #size of X is [m, n] where m=sample, n=features
    W = np.random.randn(len(self.train_x[0]), 1) # +1 for Bias term
    return W

  def __sigmoid(self,z): return 1/(1+np.exp(-z))

  def __add_bias(self,X):
    Bias = np.ones((len(X), 1))
    res = np.concatenate((Bias, X), axis=1)
    return res

  def cost_function(self):
    if type(self.pred) == "None" : return None
    m = len(self.train_y)
    loss = np.dot(-self.train_y.T, np.log(self.__sigmoid(self.pred)+1e-10))-np.dot(
    cost = (1/m) * loss
    grad = (1/m) * (np.dot(self.train_x.T, (self.__sigmoid(self.pred)-self.train_y)
    return cost.astype('float64'), grad.astype('float64')

  def predict(self, X=None,sigmoid=True):
    if X is None : X = self.train_x
    h = np.dot(X, self.W)
    return self.__sigmoid(h) if sigmoid else h

  def train_accuracy(self):
    return np.squeeze(np.squeeze((sum(self.train_y == self.pred_class)/len(self.tra


  def step(self):
    #update parameters
    self.W = self.W - self.learning_rate * self.grad
    self.grad = 0

  def train(self,epoch=int(1e+8),interuption_step=10,logging_step=None):
    if logging_step is None : logging_step = epoch**0.5
    loss_step = 0
    best_W = self.W
    for i in range(1,epoch+1):
      self.pred = self.predict(None,False)
      self.pred_class = np.where(self.__sigmoid(self.pred) >= 0.5, 1, 0)
      cost, grad = self.cost_function()
      self.grad = grad
      self.step()
      if i%logging_step == 0: print(f"Epoch : {i}/{epoch}, Train Accuracy :{self.tr
      if cost.item() < self.cost.item() :
        loss_step = 0
        best_W = self.W
```

```
      else :
        loss_step += 1
        if loss_step == interuption_step :
          print(f"Loss is increasing, stop training at epoch {i}")
          self.W = best_W
          break
      self.cost = cost

  def test(self):
    self.test_pred = self.predict(self.test_x)
    return self.test_pred
```

In [16]:
```
model = LogisticRegression(train_data,train_label,test_data,0.001)
model.train()
print("Cost :",model.cost_function()[0],"\tTrain accuracy :",model.train_accuracy()
```

```
Epoch : 10000/100000000, Train Accuracy :71.49%, Cost : 0.56660
Epoch : 20000/100000000, Train Accuracy :77.33%, Cost : 0.54443
Epoch : 30000/100000000, Train Accuracy :77.67%, Cost : 0.53633
Epoch : 40000/100000000, Train Accuracy :77.67%, Cost : 0.53302
Epoch : 50000/100000000, Train Accuracy :78.00%, Cost : 0.53162
Epoch : 60000/100000000, Train Accuracy :77.67%, Cost : 0.53100
Epoch : 70000/100000000, Train Accuracy :77.55%, Cost : 0.53073
Epoch : 80000/100000000, Train Accuracy :77.55%, Cost : 0.53061
Epoch : 90000/100000000, Train Accuracy :77.67%, Cost : 0.53056
Epoch : 100000/100000000, Train Accuracy :77.89%, Cost : 0.53053
Epoch : 110000/100000000, Train Accuracy :77.89%, Cost : 0.53052
Epoch : 120000/100000000, Train Accuracy :77.89%, Cost : 0.53052
Epoch : 130000/100000000, Train Accuracy :77.89%, Cost : 0.53051
Epoch : 140000/100000000, Train Accuracy :77.89%, Cost : 0.53051
Epoch : 150000/100000000, Train Accuracy :77.78%, Cost : 0.53051
Epoch : 160000/100000000, Train Accuracy :77.78%, Cost : 0.53051
Epoch : 170000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 180000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 190000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 200000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 210000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 220000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 230000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 240000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 250000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 260000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 270000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 280000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 290000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 300000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 310000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 320000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Epoch : 330000/100000000, Train Accuracy :77.67%, Cost : 0.53051
Loss is increasing, stop training at epoch 335870
Cost : [[0.53051089]]   Train accuracy : 77.665544332211
```

In [17]:
```
res = model.test()
pred_class = np.where(res>=0.5,1,0)
test_df["Survived"] = pred_class
```

```
test_df[["PassengerId","Survived"]].to_csv("submission.csv",index=False)
test_df
```

Out[17]:

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
0	892	3	Kelly, Mr. James	male	34.5	0	0	330911	7.8292
1	893	3	Wilkes, Mrs. James (Ellen Needs)	female	47.0	1	0	363272	7.0000
2	894	2	Myles, Mr. Thomas Francis	male	62.0	0	0	240276	9.6875
3	895	3	Wirz, Mr. Albert	male	27.0	0	0	315154	8.6625
4	896	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.0	1	1	3101298	12.2875
...	...	...	...	...	...	...	...	...	...
413	1305	3	Spector, Mr. Woolf	male	28.0	0	0	A.5. 3236	8.0500
414	1306	1	Oliva y Ocana, Dona. Fermina	female	39.0	0	0	PC 17758	108.9000
415	1307	3	Saether, Mr. Simon Sivertsen	male	38.5	0	0	SOTON/O.Q. 3101262	7.2500
416	1308	3	Ware, Mr. Frederick	male	28.0	0	0	359309	8.0500
417	1309	3	Peter, Master. Michael J	male	28.0	1	1	2668	22.3583

418 rows × 14 columns

T11: My submission score is 0.76076

submission.csv
Complete · 9h ago

0.76076

In [30]:
```
poly_features_train = np.array(train_df[["Pclass","SexClass","Age","EmbarkedClass"]
poly_features_test = np.array(test_df[["Pclass","SexClass","Age","EmbarkedClass"]].
for i in range(train_data.shape[1]):
  poly_features_train = np.concatenate((poly_features_train,train_data[:,i].reshape
  poly_features_test = np.concatenate((poly_features_test,test_data[:,i].reshape(-1
  for j in range(i,train_data.shape[1]):
    poly_features_train = np.concatenate((poly_features_train,train_data[:,i].resha
    poly_features_test = np.concatenate((poly_features_test,test_data[:,i].reshape(
```

In [31]:
```
poly_features_train.shape,poly_features_test.shape
```

Out[31]:
```
((891, 18), (418, 18))
```

In [45]:
```
model2 = LogisticRegression(poly_features_train,train_label,poly_features_test,3e-5
model2.train()
res2 = model2.test()
res2_class = np.where(res2>=0.5,1,0)
print("Cost :",model2.cost_function()[0],"\tTrain accuracy :",model2.train_accuracy
test_df["Survived"] = res2_class
test_df[['PassengerId','Survived']].to_csv('submission2.csv',index=False)
```

```
/tmp/ipykernel_294501/3797932207.py:18: RuntimeWarning: overflow encountered in exp
  def __sigmoid(self,z): return 1/(1+np.exp(-z))
Epoch : 10000/100000000, Train Accuracy :42.99%, Cost : 4.41231
Epoch : 20000/100000000, Train Accuracy :71.04%, Cost : 2.40492
Loss is increasing, stop training at epoch 23068
Cost : [[1.40367608]]   Train accuracy : 75.75757575757575
```

T12: My submission has acccuracy of model2 is worse than fisrt try

In [29]:
```
model3 = LogisticRegression(np.array(train_df[["SexClass","Age"]].values,dtype=np.f
model3.train(1000000)
print("Cost :",model.cost_function()[0],"\tTrain accuracy :",model.train_accuracy()
res3 = model3.test()
res3_class = np.where(res3>=0.5,1,0)
test_df["Survived"] = res3_class
test_df[['PassengerId','Survived']].to_csv('submission3.csv',index=False)
```

```
Epoch : 1000/1000000, Train Accuracy :61.62%, Cost : 0.60242
Epoch : 2000/1000000, Train Accuracy :61.62%, Cost : 0.58151
Epoch : 3000/1000000, Train Accuracy :78.68%, Cost : 0.56721
Epoch : 4000/1000000, Train Accuracy :78.68%, Cost : 0.55732
Epoch : 5000/1000000, Train Accuracy :78.68%, Cost : 0.55038
Epoch : 6000/1000000, Train Accuracy :78.68%, Cost : 0.54547
Epoch : 7000/1000000, Train Accuracy :78.68%, Cost : 0.54196
Epoch : 8000/1000000, Train Accuracy :78.68%, Cost : 0.53942
Epoch : 9000/1000000, Train Accuracy :78.68%, Cost : 0.53757
Epoch : 10000/1000000, Train Accuracy :78.68%, Cost : 0.53621
Epoch : 11000/1000000, Train Accuracy :78.68%, Cost : 0.53521
Epoch : 12000/1000000, Train Accuracy :78.68%, Cost : 0.53447
Epoch : 13000/1000000, Train Accuracy :78.68%, Cost : 0.53392
Epoch : 14000/1000000, Train Accuracy :78.68%, Cost : 0.53351
Epoch : 15000/1000000, Train Accuracy :78.68%, Cost : 0.53320
Epoch : 16000/1000000, Train Accuracy :78.68%, Cost : 0.53297
Epoch : 17000/1000000, Train Accuracy :78.68%, Cost : 0.53279
Epoch : 18000/1000000, Train Accuracy :78.68%, Cost : 0.53266
Epoch : 19000/1000000, Train Accuracy :78.68%, Cost : 0.53256
Epoch : 20000/1000000, Train Accuracy :78.68%, Cost : 0.53249
Epoch : 21000/1000000, Train Accuracy :78.68%, Cost : 0.53243
Epoch : 22000/1000000, Train Accuracy :78.68%, Cost : 0.53238
Epoch : 23000/1000000, Train Accuracy :78.68%, Cost : 0.53235
Epoch : 24000/1000000, Train Accuracy :78.68%, Cost : 0.53233
Epoch : 25000/1000000, Train Accuracy :78.68%, Cost : 0.53231
Epoch : 26000/1000000, Train Accuracy :78.68%, Cost : 0.53229
Epoch : 27000/1000000, Train Accuracy :78.68%, Cost : 0.53228
Epoch : 28000/1000000, Train Accuracy :78.68%, Cost : 0.53227
Epoch : 29000/1000000, Train Accuracy :78.68%, Cost : 0.53227
Epoch : 30000/1000000, Train Accuracy :78.68%, Cost : 0.53226
Epoch : 31000/1000000, Train Accuracy :78.68%, Cost : 0.53226
Epoch : 32000/1000000, Train Accuracy :78.68%, Cost : 0.53225
Epoch : 33000/1000000, Train Accuracy :78.68%, Cost : 0.53225
Epoch : 34000/1000000, Train Accuracy :78.68%, Cost : 0.53225
Epoch : 35000/1000000, Train Accuracy :78.68%, Cost : 0.53225
Epoch : 36000/1000000, Train Accuracy :78.68%, Cost : 0.53225
Epoch : 37000/1000000, Train Accuracy :78.68%, Cost : 0.53225
Epoch : 38000/1000000, Train Accuracy :78.68%, Cost : 0.53225
Epoch : 39000/1000000, Train Accuracy :78.68%, Cost : 0.53225
Epoch : 40000/1000000, Train Accuracy :78.68%, Cost : 0.53225
Epoch : 41000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 42000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 43000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 44000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 45000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 46000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 47000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 48000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 49000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 50000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 51000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 52000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 53000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 54000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 55000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 56000/1000000, Train Accuracy :78.68%, Cost : 0.53224
```

```
Epoch : 57000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 58000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 59000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 60000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 61000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 62000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 63000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 64000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 65000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 66000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 67000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 68000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 69000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 70000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 71000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 72000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 73000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 74000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 75000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 76000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 77000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 78000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 79000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 80000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 81000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 82000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 83000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 84000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 85000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 86000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 87000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 88000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 89000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 90000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 91000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 92000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 93000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 94000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 95000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 96000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 97000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 98000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 99000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 100000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 101000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 102000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Epoch : 103000/1000000, Train Accuracy :78.68%, Cost : 0.53224
Loss is increasing, stop training at epoch 103268
Cost : [[0.53051089]]   Train accuracy : 77.665544332211
```

T13: Seems a bit better?

12484	**Patrick Cho #2**		0.76555	10	21m

Your Best Entry!
Your submission scored 0.72488, which is not an improvement of your previous score. Keep trying!