PG :03

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

def kernel(point, xmat, k):
    m, n = np.shape(xmat)
    weights = np.mat(np.eye((m)))
    for j in range(m):
        diff = point - X[j]
        weights[j, j] = np.exp(diff * diff.T / (-2.0 * k ** 2))
    return weights

def localWeight(point, xmat, ymat, k):
    wei = kernel(point, xmat, k)
    W = (X.T * (wei * X)).I * (X.T * (wei * ymat.T))
    return W

def localWeightRegression(xmat, ymat, k):
    m, n = np.shape(xmat)
    ypred = np.zeros(m)
    for i in range(m):
        ypred[i] = xmat[i] * localWeight(xmat[i], xmat, ymat, k)
    return ypred

# load data points
data = pd.read_csv('dataset_3.csv')
bill = np.array(data.total_bill)
tip = np.array(data.tip)

# preparing and add 1 in bill
mbill = np.mat(bill)
mtip = np.mat(tip)

m = np.shape(mbill)[1]
one = np.mat(np.ones(m))  # Corrected line
X = np.hstack((one.T, mbill.T))

# set k here
ypred = localWeightRegression(X, mtip, 0.5)
SortIndex = X[:, 1].argsort(0)
xsort = X[SortIndex][:, 0]

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.scatter(bill, tip, color='green')
ax.plot(xsort[:, 1], ypred[SortIndex], color='red', linewidth=5)
plt.xlabel('Total bill')
plt.ylabel('Tip')
plt.show()
```

PG:01

```
import numpy as np
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics

# Read dataset to pandas dataframe
dataset = pd.read_csv("iris.csv")
X = dataset.iloc[:, :-1]
y = dataset.iloc[:, -1]
print(X.head())
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.10)

classifier = KNeighborsClassifier(n_neighbors=5).fit(Xtrain, ytrain)

ypred = classifier.predict(Xtest)

i = 0
print ("\n-------------------------------------------------------------------------")
print ('%-25s %-25s %-25s' % ('Original Label', 'Predicted Label', 'Correct/Wrong'))
print ("-------------------------------------------------------------------------")
for label in ytest:
    print ('%-25s %-25s' % (label, ypred[i]), end="")
    if (label == ypred[i]):
        print (' %-25s' % ('Correct'))
    else:
        print (' %-25s' % ('Wrong'))
    i = i + 1
print ("-------------------------------------------------------------------------")
print("\nConfusion Matrix:\n",metrics.confusion_matrix(ytest, ypred))
print ("-------------------------------------------------------------------------")
print("\nClassification Report:\n",metrics.classification_report(ytest, ypred))
print ("-------------------------------------------------------------------------")
print('Accuracy of the classifer is %0.2f' % metrics.accuracy_score(ytest,ypred))
print ("-------------------------------------------------------------------------")
```

PG:02

```
from sklearn.cluster import KMeans
from sklearn import preprocessing
from sklearn.mixture import GaussianMixture
from sklearn.datasets import load_iris
import sklearn.metrics as sm
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Load the iris dataset
dataset = load_iris()
X = pd.DataFrame(dataset.data)
X.columns = ['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width']
y = pd.DataFrame(dataset.target)
y.columns = ['Targets']

plt.figure(figsize=(14, 7))
colormap = np.array(['red', 'lime', 'black'])

# Plot the Original Classifications using Petal features
plt.subplot(1, 3, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Clusters')

plt.subplot(1, 3, 2)
model = KMeans(n_clusters=3)
model.fit(X)
predy = np.choose(model.labels_, [0, 1, 2]).astype(np.int64)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('KMeans Clustering')

deviation of 1.
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns=X.columns)
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
y['cluster_gmm'] = gmm.predict(xs)

plt.subplot(1, 3, 3)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.cluster_gmm], s=40)
plt.title('GMM Clustering')
plt.show()
```

PG:04

```
import numpy as np
import pandas as pd
x = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = x / np.amax(x, axis=0)
Y = y / 100
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def derivatives_sigmoid(x):
    return x * (1 - x)
epoch = 5000
learning_rate = 0.1
inputlayer_neurons = 2
hiddenlayer_neurons = 3
output_neurons = 1

wh = np.random.uniform(size=(inputlayer_neurons, hiddenlayer_neurons))
bh = np.random.uniform(size=(1, hiddenlayer_neurons))
wout = np.random.uniform(size=(hiddenlayer_neurons, output_neurons))
bout = np.random.uniform(size=(1, output_neurons))
for i in range(epoch):
    hinpl = np.dot(X, wh)
    hinp = hinpl + bh
    hlayer_act = sigmoid(hinp)
    outinpl = np.dot(hlayer_act, wout)
    outinp = outinpl + bout
    output = sigmoid(outinp)
    EO = y - output
    outgrad = derivatives_sigmoid(output)
    d_output = EO * outgrad
    EH = d_output.dot(wout.T)
    hiddengrad = derivatives_sigmoid(hlayer_act)
    d_hiddenlayer = EH * hiddengrad
    wout += hlayer_act.T.dot(d_output) * learning_rate
    wh += X.T.dot(d_hiddenlayer) * learning_rate
    print("Input: \n" + str(X))
    print("Actual Output: \n" + str(Y))
    print("Predicted Output:\n", output)
```

PG:05

```
import random
POPULATION_SIZE = 100
GENES = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890, .-;:_!\"#%&/
()=?@${[]}"
```

```python
TARGET = "I Love Ironman"

def random_num(start, end):
    return random.randint(start, end)

def mutated_genes():
    return random.choice(GENES)

def create_gnome():
    return ''.join(mutated_genes() for _ in range(len(TARGET)))

class Individual:
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    def mate(self, par2):
        child_chromosome = ''
        for i in range(len(self.chromosome)):
            p = random.random()
            if p < 0.45:
                child_chromosome += self.chromosome[i]
            elif p < 0.90:
                child_chromosome += par2.chromosome[i]
            else:
                child_chromosome += mutated_genes()
        return Individual(child_chromosome)

    def cal_fitness(self):
        return sum(1 for c1, c2 in zip(self.chromosome, TARGET) if c1 != c2)

def main():
    generation = 0
    population = [Individual(create_gnome()) for _ in range(POPULATION_SIZE)]
    found = False

    while not found:
        population.sort(key=lambda x: x.fitness)

        if population[0].fitness <= 0:
            found = True
            break

        new_generation = population[:10]

        for _ in range(90):
            parent1 = random.choice(population[:50])
            parent2 = random.choice(population[:50])
            offspring = parent1.mate(parent2)
            new_generation.append(offspring)

        population = new_generation

        print(f"Generation: {generation}\tString: {population[0].chromosome}\tFitness: {population[0].fitness}")
        generation += 1

    print(f"Generation: {generation}\tString: {population[0].chromosome}\tFitness: {population[0].fitness}")

if __name__ == "__main__":
    main()
```