Code implementation –

This code compares the accuracies of two code algorithms, namely traditional algorithm only KNN and KNN with Genetic algorithm (WKNNGA).

Code –

```
#Now this code clearly mentions the output of two algorithms

import csv
import random
import math
import operator
import numpy
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

#read the dataset
def loadDataset(filename, split, trainingSet=[] , testSet=[]):

    with open(filename, 'rt') as csvfile:
        lines = csv.reader(csvfile)
        dataset = list(lines)[1:]
        dataset2 = dataset
        dataset = [[float(x) for x in row] for row in dataset]
        scaler = MinMaxScaler()
        scaler.fit(dataset)
        MinMaxScaler(copy=True, feature_range=(0, 1))
        dataset = scaler.transform(dataset)
        for x in range(0,len(dataset)):
            for y in range(9):
                dataset[x][y] = float(dataset[x][y])
            if random.random() < split:
                trainingSet.append(dataset[x])
            else:
                testSet.append(dataset[x])

def euclideanDistance(instance1, instance2, length, chrom2):
    distance = 0
    for x in range(1,length):
        if chrom2[x]==1:
```

```
            distance += pow((float(instance1[x])-
float(instance2[x])), 2)
    return math.sqrt(distance)

def getNeighbors(trainingSet, testInstance, k, chrom2):
    distances = []
    length = len(testInstance)-1
    for x in range(len(trainingSet)):
        dist = euclideanDistance(testInstance, trainingSet[x], length,
chrom2)
        distances.append((trainingSet[x], dist))
    distances.sort(key=operator.itemgetter(1))
    neighbors = []
    for x in range(k):
        neighbors.append(distances[x][0])
    return neighbors

def getResponse(neighbors):
  classVotes = {}
  for x in range(len(neighbors)):
    response = neighbors[x][-1]
    if response in classVotes:
      classVotes[response] += 1
    else:
      classVotes[response] = 1
  sortedVotes = sorted(classVotes.items(), key=operator.itemgetter(1),
reverse=True)
  return sortedVotes[0][0]

def getAccuracy(testSet, predictions):
  correct = 0
  for x in range(len(testSet)):
    if testSet[x][-1] == predictions[x]:
      correct += 1
  return (correct/float(len(testSet))) * 100.0

def fitnessValue(chrom):
    random.seed(2)
    trainingSet=[]
    testSet=[]
    split = 0.8
    loadDataset('Prostate_Cancer_dataset.csv', split, trainingSet, test
Set)
    predictions=[]
    k = 3
    selected_features = [idx for idx, val in enumerate(chrom) if val==1
]
    trainingSetReduced = []
```

```
    testSetReduced = []
    for instance in trainingSet:
        reduced_instance = [instance[i] for i in selected_features]
        reduced_instance.append(instance[-1])
        trainingSetReduced.append(reduced_instance)
    for instance in testSet:
        reduced_instance = [instance[i] for i in selected_features]
        reduced_instance.append(instance[-1])
        testSetReduced.append(reduced_instance)
    for x in range(len(testSetReduced)):
        neighbors = getNeighbors(trainingSetReduced, testSetReduced[x],
 k, chrom)
        result = getResponse(neighbors)
        predictions.append(result)
    accuracy = getAccuracy(testSetReduced, predictions)
    return accuracy

# from knn import fitnessValue


def generatePop(n):
  chromAsli=[]
  for x in range(n):
    temp=[]
    temp2=[]
    for x in range(10):
      temp.append(numpy.random.randint(1))
    temp2.append(temp)
    fitness=fitnessValue(temp)
    temp2.append(fitness)
    chromAsli.append(temp2)
  return chromAsli

def eliteChild(chrom2,n):
  a=sorted(chrom2,key=lambda l:l[1], reverse=True)
  bestVal=a[0][1]
  bestFt2=a[0][0]
  next2=[]
  chrom3=[]
  for x in range(2):
    next2.append(a[x][0])
  for x in range(2,n):
    chrom3.append(a[x])
  return chrom3,next2,bestVal,bestFt2

def tournament(chrom2):
    best=[]
    for x in range(2):
        a=numpy.random.randint(0,11)
```

```
        best.append(chrom2[a])
    bestOne=best[0]
    if(best[0][1]<best[1][1]):
        bestOne=best[1]
    return bestOne[0]

def getMutation(chrom2,pm,next2,n): #pm = mutation probability
    rng=n-len(next2)
    for i in range(rng):
        a=tournament(chrom2)
        rd=[]
        for x in range(10):
            ru=numpy.random.uniform(0.0, 1.0)
            if ru<=pm:
                rd.append(1)
            else:
                rd.append(0)
        if a==rd:
            a=tournament(chrom2)
        result=toXor(a,rd)
        next2.append(result)
    return next2

def toXor(x1,x2):
  xorRes=[]
  for x in range(len(x1)):
    if x1[x]==x2[x]:
      xorRes.append(0)
    else:
      xorRes.append(1)
  return xorRes

def getCrossOver(chrom2,pr,next2): #pr = cross-over probability
  rng=int(pr*len(chrom2))
  for x in range(rng):
    a=tournament(chrom2)
    b=tournament(chrom2)
    if a==b:
      result=a
    else:
      result=toXor(a,b)
    next2.append(result)
  return next2

def getGeneration(chrom2,idx):
  chrom2,nextGen,bestFitness,bestFt2=eliteChild(chrom2,genNum)
  nextGen=getCrossOver(chrom2,0.8,nextGen)
  nextGen=getMutation(chrom2,0.3,nextGen,genNum)
```

```
  print('Generation {} {:.3f}'.format(idx+1,float(bestFitness)),end='%\
n')
  return nextGen,bestFitness,bestFt2

genNum=15
knn_chrom=[0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
knn_acc=fitnessValue(knn_chrom)
limit=100
stall=100
chrom=generatePop(genNum)
counter=0
newFit=0.0
oldFit=0.0
for x in range(limit):
    oldFit=newFit
    newChrom,newFit,bestFt=getGeneration(chrom,x)
    newFit=float(newFit)
    temp2=[]
    for y in range(genNum):
        temp=[]
        temp.append(newChrom[y])
        f=fitnessValue(newChrom[y])
        temp.append(f)
        temp2.append(temp)
    chrom=temp2
    diff=newFit - oldFit
    if diff<=0.0000001:
        counter+=1
    else:
      counter=0
    if counter==stall:
      break

#This will give us the output for KNN only
print("\nAccuracy with using K-NN without GA: ",end='')
print('{:.3f}'.format(float(knn_acc)),end='')

#This will give us the output for KNN and GA
print("\nAccuracy using K-NN with GA:",end='')
print('{:.3f}'.format(float(newFit)),end='%')

#Used features from our dataset -
print("\nUsed features:", bestFt)
```

Output snip –

```
Generation 1 16.000%
Generation 2 16.000%
Generation 3 84.000%
Generation 4 84.000%
Generation 5 84.000%
Generation 6 84.000%
Generation 7 84.000%
Generation 8 84.000%
Generation 9 84.000%
Generation 10 84.000%
Generation 11 84.000%
Generation 12 84.000%
Generation 13 84.000%
Generation 14 84.000%
Generation 15 84.000%
Generation 16 84.000%
Generation 17 84.000%
Generation 18 84.000%
Generation 19 84.000%
Generation 20 84.000%
Generation 21 84.000%
Generation 22 84.000%
Generation 23 84.000%
Generation 24 84.000%
Generation 25 84.000%
Generation 26 84.000%
Generation 27 84.000%
Generation 28 84.000%
Generation 29 84.000%
Generation 30 84.000%
Generation 31 84.000%
Generation 32 84.000%
Generation 33 84.000%
Generation 34 84.000%
Generation 35 84.000%
Generation 36 84.000%
Generation 37 84.000%
Generation 38 84.000%
Generation 39 84.000%
Generation 40 84.000%
Generation 41 84.000%
Generation 42 84.000%
Generation 43 84.000%
Generation 44 84.000%
Generation 45 84.000%
Generation 46 84.000%
Generation 47 84.000%
Generation 48 84.000%
Generation 49 84.000%
Generation 50 84.000%
Generation 51 84.000%
Generation 52 84.000%
Generation 53 84.000%
Generation 54 84.000%
Generation 55 84.000%
Generation 56 84.000%
Generation 57 84.000%
Generation 58 84.000%
Generation 59 84.000%
Generation 60 84.000%
Generation 61 84.000%
Generation 62 84.000%
Generation 63 84.000%
Generation 64 84.000%
Generation 65 84.000%
Generation 66 84.000%
Generation 67 84.000%
Generation 68 84.000%
Generation 69 84.000%
Generation 70 84.000%
Generation 71 84.000%
Generation 72 84.000%
Generation 73 84.000%
Generation 74 84.000%
Generation 75 84.000%
Generation 76 84.000%
```

```
Generation 77 84.000%
Generation 78 84.000%
Generation 79 84.000%
Generation 80 84.000%
Generation 81 84.000%
Generation 82 84.000%
Generation 83 84.000%
Generation 84 84.000%
Generation 85 84.000%
Generation 86 84.000%
Generation 87 84.000%
Generation 88 84.000%
Generation 89 84.000%
Generation 90 84.000%
Generation 91 84.000%
Generation 92 84.000%
Generation 93 84.000%
Generation 94 84.000%
Generation 95 84.000%
Generation 96 84.000%
Generation 97 84.000%
Generation 98 84.000%
Generation 99 84.000%
Generation 100 84.000%

Accuracy with using K-NN without GA: 8.000
Accuracy using K-NN with GA:84.000%
Used features: [0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
```

It is clearly shown in the output that in the case of on KNN algorithm with no GA, the accuracy is extremely low, 8.00%, which is not a good value for any algorithm.

Reason for this low accuracy is the poor fitness function which leads to poor selection of the features.

Fitness function used for KNN only –

```
def fitnessValue(chrom):
    random.seed(2)
    trainingSet=[]
    testSet=[]
    split = 0.8
    loadDataset('Prostate_Cancer_dataset.csv', split, trainingSet, test
Set)
    predictions=[]
    k = 3
    for x in range(len(testSet)):
        neighbors = getNeighbors(trainingSet, testSet[x], k, chrom)
        result = getResponse(neighbors)
        predictions.append(result)
    accuracy = getAccuracy(testSet, predictions)
    return repr(accuracy)
```

That's why to overcome this problem we applied our KNN algorithm with GA, which improved the fitness function.

Fitness function modified –

```
def fitnessValue(chrom):
    random.seed(2)
    trainingSet=[]
    testSet=[]
    split = 0.8
    loadDataset('Prostate_Cancer_dataset.csv', split, trainingSet, test
Set)
    predictions=[]
    k = 3
```

```
    selected_features = [idx for idx, val in enumerate(chrom) if val==1
]
    trainingSetReduced = []
    testSetReduced = []
    for instance in trainingSet:
        reduced_instance = [instance[i] for i in selected_features]
        reduced_instance.append(instance[-1])
        trainingSetReduced.append(reduced_instance)
    for instance in testSet:
        reduced_instance = [instance[i] for i in selected_features]
        reduced_instance.append(instance[-1])
        testSetReduced.append(reduced_instance)
    for x in range(len(testSetReduced)):
        neighbors = getNeighbors(trainingSetReduced, testSetReduced[x],
 k, chrom)
        result = getResponse(neighbors)
        predictions.append(result)
    accuracy = getAccuracy(testSetReduced, predictions)
    return accuracy
```

This significantly improved the accuracy, up to 84.00%. Thus we have comparison base for out problem statement and implementation.

Submitted by –

Group 10