

**SCHOOL OF ENGINEERING AND COMPUTER SCIENCE**

**Assignment Cover Sheet**

**Full Name: \_\_\_\_\_\_\_\_\_\_\_\_\_Tao Shi\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_**

**Student ID: \_\_300409943\_\_ \_\_Course: \_\_\_\_\_ COMP307 \_ \_\_\_**

**Assignment No.: \_\_\_\_\_1\_\_\_ \_ Due Date: 07 April at 11:59pm**

**Part 1: Nearest Neighbour Method**

1. When k = 1, the running result of program is as follow, the classification accuracy is 90.67%.

Test instance: ([5.0, 3.0, 1.6, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.0, 3.4, 1.6, 0.4], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.2, 3.5, 1.5, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.2, 3.4, 1.4, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([4.7, 3.2, 1.6, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([4.8, 3.1, 1.6, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.4, 3.4, 1.5, 0.4], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.2, 4.1, 1.5, 0.1], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.5, 4.2, 1.4, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([4.9, 3.1, 1.5, 0.1], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.0, 3.2, 1.2, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.5, 3.5, 1.3, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([4.9, 3.1, 1.5, 0.1], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([4.4, 3.0, 1.3, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.1, 3.4, 1.5, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.0, 3.5, 1.3, 0.3], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([4.5, 2.3, 1.3, 0.3], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([4.4, 3.2, 1.3, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.0, 3.5, 1.6, 0.6], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.1, 3.8, 1.9, 0.4], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([4.8, 3.0, 1.4, 0.3], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.1, 3.8, 1.6, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([4.6, 3.2, 1.4, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.3, 3.7, 1.5, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([5.0, 3.3, 1.4, 0.2], 'Iris-setosa'); Predicted class: Iris-setosa; Result: correct

Test instance: ([6.6, 3.0, 4.4, 1.4], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([6.8, 2.8, 4.8, 1.4], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([6.7, 3.0, 5.0, 1.7], 'Iris-versicolor'); Predicted class: Iris-virginica; Result: incorrect

Test instance: ([6.0, 2.9, 4.5, 1.5], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.7, 2.6, 3.5, 1.0], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.5, 2.4, 3.8, 1.1], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.5, 2.4, 3.7, 1.0], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.8, 2.7, 3.9, 1.2], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([6.0, 2.7, 5.1, 1.6], 'Iris-versicolor'); Predicted class: Iris-virginica; Result: incorrect

Test instance: ([5.4, 3.0, 4.5, 1.5], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([6.0, 3.4, 4.5, 1.6], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([6.7, 3.1, 4.7, 1.5], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([6.3, 2.3, 4.4, 1.3], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.6, 3.0, 4.1, 1.3], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.5, 2.5, 4.0, 1.3], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.5, 2.6, 4.4, 1.2], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([6.1, 3.0, 4.6, 1.4], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.8, 2.6, 4.0, 1.2], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.0, 2.3, 3.3, 1.0], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.6, 2.7, 4.2, 1.3], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.7, 3.0, 4.2, 1.2], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.7, 2.9, 4.2, 1.3], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([6.2, 2.9, 4.3, 1.3], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.1, 2.5, 3.0, 1.1], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([5.7, 2.8, 4.1, 1.3], 'Iris-versicolor'); Predicted class: Iris-versicolor; Result: correct

Test instance: ([7.2, 3.2, 6.0, 1.8], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.2, 2.8, 4.8, 1.8], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.1, 3.0, 4.9, 1.8], 'Iris-virginica'); Predicted class: Iris-versicolor; Result: incorrect

Test instance: ([6.4, 2.8, 5.6, 2.1], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([7.2, 3.0, 5.8, 1.6], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([7.4, 2.8, 6.1, 1.9], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([7.9, 3.8, 6.4, 2.0], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.4, 2.8, 5.6, 2.2], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.3, 2.8, 5.1, 1.5], 'Iris-virginica'); Predicted class: Iris-versicolor; Result: incorrect

Test instance: ([6.1, 2.6, 5.6, 1.4], 'Iris-virginica'); Predicted class: Iris-versicolor; Result: incorrect

Test instance: ([7.7, 3.0, 6.1, 2.3], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.3, 3.4, 5.6, 2.4], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.4, 3.1, 5.5, 1.8], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.0, 3.0, 4.8, 1.8], 'Iris-virginica'); Predicted class: Iris-versicolor; Result: incorrect

Test instance: ([6.9, 3.1, 5.4, 2.1], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.7, 3.1, 5.6, 2.4], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.9, 3.1, 5.1, 2.3], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([5.8, 2.7, 5.1, 1.9], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.8, 3.2, 5.9, 2.3], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.7, 3.3, 5.7, 2.5], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.7, 3.0, 5.2, 2.3], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.3, 2.5, 5.0, 1.9], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.5, 3.0, 5.2, 2.0], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([6.2, 3.4, 5.4, 2.3], 'Iris-virginica'); Predicted class: Iris-virginica; Result: correct

Test instance: ([5.9, 3.0, 5.1, 1.8], 'Iris-virginica'); Predicted class: Iris-versicolor; Result: incorrect

When k = 1, the accuracy is 90.67%

2. When k = 3, the running result of program is as follow, the classification accuracy is 96.00%.

When k = 3, the accuracy is higher than when k = 1. This is because Nearest Neighbor Method is subject to underfitting and overfitting. In this case 1-nearest neighbors is overfitting, while the 3-nearest-neighbors decision boundary is better; higher k might cause underfitting, for example, when k = 5, the accuracy is 94.67%. By the way, during the debugging, I found the value of range indeed has influence on the result, though only refer to one or two test instances.

3. The main advantages of Nearest Neighbor Method are simple, easy to use, and can achieve good results in many cases, especially in low-dimensional spaces with plenty of data. But as the number of dimensions rises we will encounter a problem: the nearest neighbors in high-dimensional spaces are usually not very near. This problem has been called the curse of dimensionality. In addition, the efficiency of Nearest Neighbor Method is not ideal, in particular when dataset are very huge and k is high.

4. Step 1: chop the iris data (150 instances totally) into 5 equal chunks, each chunk has 30 instances;

Step 2: For each chunk in turn: Treat it as the test data set, and treat the rest 4 chunks as the training data set. Use Nearest Neighbor Method to achieve the class of every instance in test set, and the result is the accuracy of classifier learned from the training set;

Step 3: The process is then repeated 5 times, with each of the 5 chunks used exactly once as the test data set.

Step 4: The final result could be the mean of 5 accuracies.

5. We can use K-Means Clustering Method to group the instances, and the major steps are as follows:

Step 1: Set 3 initial “means” randomly from the data set.

Step 2: Create 3 clusters by associating every instance with the nearest mean based on a distance measure.

Step 3: Replace the old means with the centroid of each of the 3 clusters (as the new means).

Step 4: Repeat the above two steps until convergence (no change in each cluster center).

6. Please find the code in KM.py which implements the clustering method above, the running result of program is as follow, the results after 4 times and after 5 times are same, means convergence.

After 1 time:

setosa\_center: [5.005660377358491, 3.360377358490567, 1.562264150943396, 0.2886792452830188]

versicolor\_center: [6.179104477611939, 2.8149253731343293, 4.680597014925374, 1.4731343283582088]

versicolor\_center: [6.573333333333334, 3.0466666666666664, 5.579999999999999, 2.1933333333333334]

After 2 time:

setosa\_center: [5.005999999999999, 3.4180000000000006, 1.464, 0.2439999999999999]

versicolor\_center: [5.934375000000001, 2.7546875, 4.418750000000001, 1.4234375000000001]

versicolor\_center: [6.844444444444445, 3.0805555555555544, 5.772222222222221, 2.1249999999999996]

After 3 time:

setosa\_center: [5.005999999999999, 3.4180000000000006, 1.464, 0.2439999999999999]

versicolor\_center: [5.9064516129032265, 2.745161290322581, 4.401612903225807, 1.417741935483871]

versicolor\_center: [6.842105263157895, 3.0789473684210518, 5.728947368421052, 2.0973684210526313]

After 4 time:

setosa\_center: [5.005999999999999, 3.4180000000000006, 1.464, 0.2439999999999999]

versicolor\_center: [5.888524590163935, 2.737704918032787, 4.39672131147541, 1.4180327868852458]

versicolor\_center: [6.846153846153845, 3.0820512820512818, 5.702564102564101, 2.0794871794871788]

After 5 time:

setosa\_center: [5.005999999999999, 3.4180000000000006, 1.464, 0.2439999999999999]

versicolor\_center: [5.888524590163935, 2.737704918032787, 4.39672131147541, 1.4180327868852458]

versicolor\_center: [6.846153846153845, 3.0820512820512818, 5.702564102564101, 2.0794871794871788]

The results after 4 times and after 5 times are same, so the final center is:

setosa\_center: [5.005999999999999, 3.4180000000000006, 1.464, 0.2439999999999999]

versicolor\_center: [5.888524590163935, 2.737704918032787, 4.39672131147541, 1.4180327868852458]

versicolor\_center: [6.846153846153845, 3.0820512820512818, 5.702564102564101, 2.0794871794871788]

**Part 2: Decision Tree Learning Method**

1. My program prints out the tree classifier as follows (the leaf node includes the classification and the number of subset). By applying my program to the hepatitis-training.dat and hepatitis-test.dat files, the classification accuracy in terms of the fraction of the test instances is only 77.78%. While using the baseline classifier, because the majority classifier of training data is ‘live’, if classify all the test instance as ‘live’, the accuracy is 85.19%, which is higher than the result of my decision tree classifier. This problem is called overfitting.

ASCITES= True:

SPIDERS= True:

VARICES= True:

FIRMLIVER= True:

('live', 49)

FIRMLIVER= False:

BIGLIVER= True:

STEROID= True:

('live', 5)

STEROID= False:

FEMALE= True:

('live', 2)

FEMALE= False:

ANTIVIRALS= True:

FATIGUE= True:

('die', 1)

FATIGUE= False:

('live', 4)

ANTIVIRALS= False:

('die', 1)

BIGLIVER= False:

('live', 7)

VARICES= False:

('die', 1)

SPIDERS= False:

FIRMLIVER= True:

AGE= True:

('live', 1)

AGE= False:

SGOT= True:

('live', 1)

SGOT= False:

ANTIVIRALS= True:

('die', 4)

ANTIVIRALS= False:

STEROID= True:

('live', 1)

STEROID= False:

('die', 1)

FIRMLIVER= False:

SGOT= True:

BIGLIVER= True:

SPLEENPALPABLE= True:

('live', 4)

SPLEENPALPABLE= False:

ANOREXIA= True:

('die', 2)

ANOREXIA= False:

('live', 1)

BIGLIVER= False:

('die', 3)

SGOT= False:

('live', 10)

ASCITES= False:

BIGLIVER= True:

STEROID= True:

('die', 7)

STEROID= False:

ANOREXIA= True:

('die', 2)

ANOREXIA= False:

('live', 2)

BIGLIVER= False:

('live', 1)

2. The average accuracy of the classifiers over the 10 trials is 77.30%, the accuracies of every trial and the average one are printed out as follows.

run01

Accuracy: 83.78%

run02

Accuracy: 83.78%

run03

Accuracy: 81.08%

run04

Accuracy: 75.68%

run05

Accuracy: 75.68%

run06

Accuracy: 67.57%

run07

Accuracy: 83.78%

run08

Accuracy: 67.57%

run09

Accuracy: 75.68%

run10

Accuracy: 78.38%

Average accuracy: 77.30%

3. The technique called decision tree pruning can eliminate overfitting by deleting the nodes that are not clearly relevant.

(a) We should start with a full tree generated by DT Learning, then look at a test node that has only leaf nodes as descendants. If the test appears to be irrelevant—detecting only noise in the data—then we eliminate the test, replacing it with a leaf node. We repeat this process, considering each test with only leaf descendants, until each one has either been pruned or accepted as is.

(b) Although the irrelevant attributes which are eliminated by pruning technique have not information gain, they can split the examples into accurate subsets. That is why pruning technique would cause a lower accuracy on the training set.

(c) The cost of higher accuracy on the training set is overfitting, the technique of pruning eliminates overfitting, which might improve accuracy on the test set.

4. If there are two possible classes, the result from impurity measure is smaller the better, the best situation is 0, which means the subset is pure. However, the result from impurity measure cannot demonstrate the purity well if there are three or more possible classes that the decision tree must distinguish. For example, there are three possible classes. One dataset includes the instances which belong to only one class, and another dataset includes the instances which belong to two classes, which means no the third class included. The results of above two datasets both are 0 if using impurity measure, but the actual purities are different obviously.

**Part 3**

1. According to the running result of my program as follows, after 154 times learning, all the training images achieve the right class, and I can find a correct set of weights referring to the training data as follows. But the perceptron need more test data to evaluate its performance.

Initial weight set is:

[0.20543438387748614, 0.5552908227535004, 0.3113359593586684, 0.6129900301185299, 0.052093068583833446, 0.5141174997951445, 0.4651586975221542, 0.15447208553816993, 0.35021018233738954, 0.0185754102975918, 0.7492731556824024, 0.8246595927552173, 0.03678063363026551, 0.08719052164260199, 0.28712634021807903, 0.8026930629720304, 0.21221595312257835, 0.5169003665538904, 0.34991963647200197, 0.21061948223389215, 0.6770196948441439, 0.837545259514719, 0.7597173507707454, 0.11052708322547822, 0.8141349817079127, 0.6691098864674611, 0.34845097858400553, 0.8374039935693941, 0.7927362884697448, 0.7534147151656199, 0.8880876449789827, 0.09169135084541313, 0.8553067674866526, 0.3426697317935201, 0.6564181698297581, 0.46282411547339586, 0.2635062383479835, 0.7251360518942618, 0.6146223371461653, 0.7146186493068156, 0.9261488346184448, 0.9358240193360249, 0.5466852247212777, 0.5302369998802394, 0.2571246047369703, 0.6913727269066513, 0.8699642718775498, 0.006255189672447203, 0.9907210468688957, 0.9808359570597113, 0.8467279200673162]

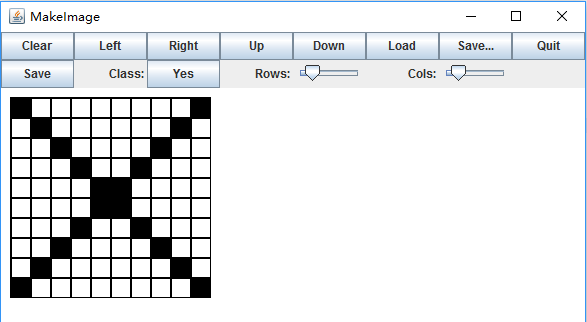
After 230 times learning, the matched number of image is 100:

Final weight set is:

[-20.79456561612251, -9.4447091772465, -2.6886640406413314, 12.61299003011853, 5.052093068583833, 11.514117499795145, -8.534841302477846, -8.84552791446183, 2.3502101823373893, -12.981424589702408, 11.749273155682403, 19.824659592755218, 36.036780633630265, 1.0871905216426025, -13.71287365978192, 5.802693062972031, 12.212215953122579, -39.48309963344611, 10.349919636472002, -17.78938051776611, -8.322980305155856, -18.16245474048528, 13.759717350770746, 2.1105270832254774, 0.8141349817079127, -28.33089011353254, -18.651549021415995, -12.162596006430604, 4.792736288469744, -8.24658528483438, 19.88808764497898, -13.908308649154588, 9.855306767486653, 12.34266973179352, -15.343581830170242, 24.462824115473396, 10.263506238347983, 27.72513605189426, -0.3853776628538341, -15.285381350693186, 14.926148834618445, -21.064175980663975, -0.4533147752787219, -32.46976300011976, 10.257124604736973, -24.30862727309335, -3.1300357281224507, -9.993744810327552, -15.009278953131105, -32.01916404294029, 20.846727920067316]

2. Because the perceptron is achieved by learning all the training data, we cannot evaluate the perceptron’s performance on the training data any more. Then I create three additional typical image data in order to measure the perceptron’s performance more effectively:

a. typical X, the class from perceptron is correct:



“test0.data” in directory part3/:

P1

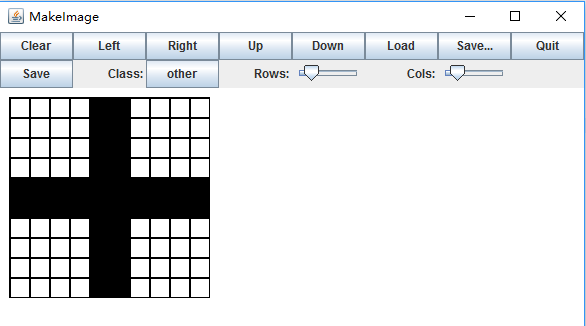
#Yes

10 10

10000000010100000010001000010000010010000000110000000011000000010010000

01000010001000000101000000001

b. typical non-X, the class from perceptron is correct:



“test1.data” in directory part3/:

P1

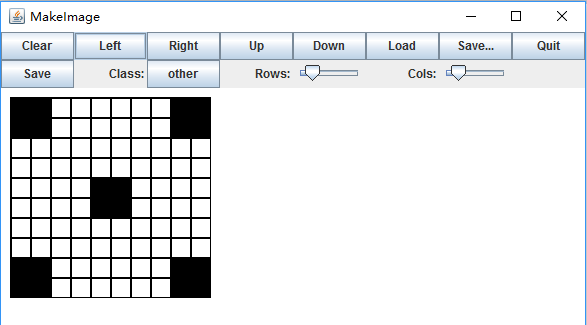
#other

10 10

00001100000000110000000011000000001100001111111111111111111100001100000

00011000000001100000000110000

c. Non-X with partial features of X, the class from perceptron is incorrect:



“test2.data” in directory part3/:

P1

#other

10 10

11000000111100000011000000000000000000000000110000000011000000000000000

00000000011000000111100000011

Consequently, the performance of my perceptron is not excellent.