600.335/435 Artificial Intelligence

Homework 4 Writeup

Huizhan Lu, Bo Lei

Decision Tree

The accuracy, precision and recall of method are shown as following. We can simply get the results by typing each of the command below. For each test, it takes less than 1 seconds to get the results.

```
====================================================================
Dataset    =  congress
Classifier =  decision_tree
Option     =  IG
Labels are:  [0 1]
Accuracy =  0.954022988506
Precision =  [0.91176470588235292, 0.98113207547169812]
Recall =  [0.96875, 0.94545454545454544]
====================================================================
```

Python train_and_test.py decision_tree IG congress 0.8

```
====================================================================
Dataset    =  congress
Classifier =  decision_tree
Option     =  IGR
Labels are:  [0 1]
Accuracy =  0.954022988506
Precision =  [0.91176470588235292, 0.98113207547169812]
Recall =  [0.96875, 0.94545454545454544]
====================================================================
```

Python train_and_test.py decision_tree IGR congress 0.8

```
====================================================================
Dataset    =  monks
Classifier =  decision_tree
Option     =  IG
Labels are:  [0 1]
Accuracy =  0.712962962963
Precision =  [0.66666666666666663, 0.79487179487179482]
Recall =  [0.85185185185185186, 0.57407407407407407]
====================================================================
```

Python train_and_test.py decision_tree IG monks 1

```
Dataset    =  monks
Classifier =  decision_tree
Option     =  IGR
Labels are:  [0 1]
Accuracy =  0.701388888889
Precision =  [0.66666666666666663, 0.754385964912280 72]
Recall =  [0.80555555555555558, 0.59722222222222221]
```

Python train_and_test.py decision_tree IGR monks 1

```
Dataset    =  monks
Classifier =  decision_tree
Option     =  IG
Labels are:  [0 1]
Accuracy =  0.648148148148
Precision =  [0.72115384615384615, 0.458333333333333 31]
Recall =  [0.77586206896551724, 0.38732394366197181]
```

Python train_and_test.py decision_tree IG monks 2

```
Dataset    =  monks
Classifier =  decision_tree
Option     =  IGR
Labels are:  [0 1]
Accuracy =  0.627314814815
Precision =  [0.73118279569892475, 0.437908496732026 14]
Recall =  [0.70344827586206893, 0.47183098591549294]
```

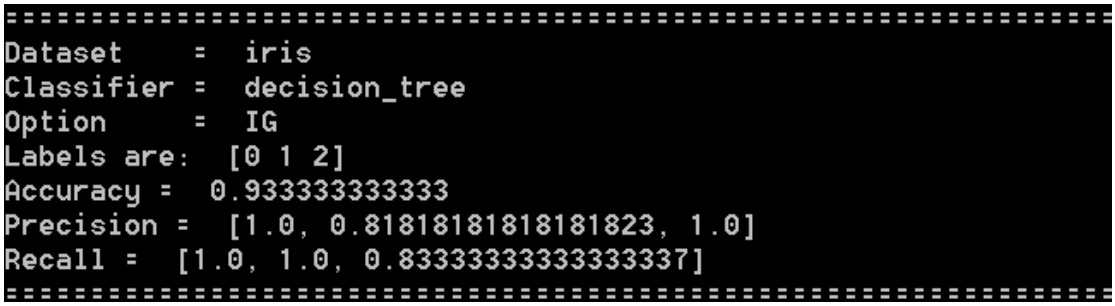
Python train_and_test.py decision_tree IGR monks 2

```
Dataset    =  monks
Classifier =  decision_tree
Option     =  IG
Labels are:  [0 1]
Accuracy =  0.944444444444
Precision =  [0.89473684210526316, 1.0]
Recall =  [1.0, 0.89473684210526316]
```

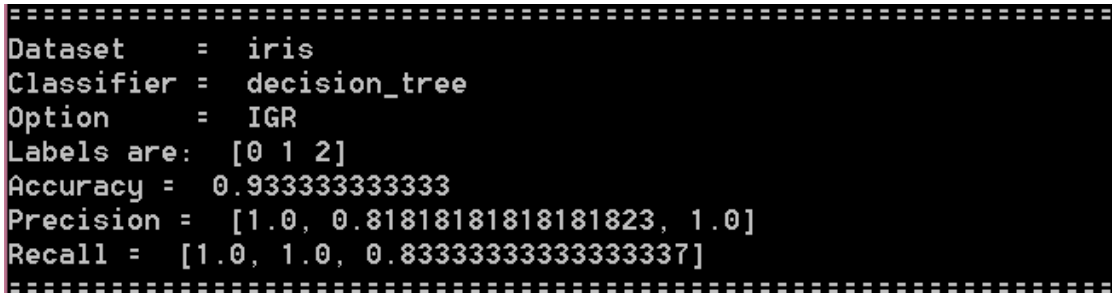
Python train_and_test.py decision_tree IG monks 3

```
Dataset    =  monks
Classifier =  decision_tree
Option     =  IGR
Labels are:  [0 1]
Accuracy =  0.888888888889
Precision =  [0.86111111111111116, 0.916666666666666 63]
Recall =  [0.91176470588235292, 0.868421052631578 98]
```

Python train_and_test.py decision_tree IGR monks 3



Python train_and_test.py decision_tree IG iris 0.8



Python train_and_test.py decision_tree IGR iris 0.8

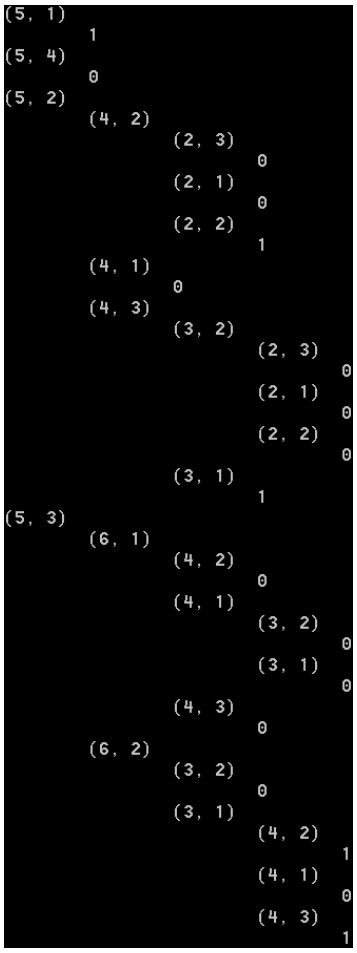
Differences between information gain & information gain ratio:

Information gain is used to show how much the entropy of the information could be reduced. As a result, we always want the attribute with the largest information gain to be the root node of a decision tree so the decision tree could be really shallow. However, for some attributes with a large number of values (also a very big information gain), even though these attributes can greatly reduce the entropy of the information, they are not very useful since they make the training set 'too well' so that a test object may not have the exact values for these attributes. For example, for ten people, the attribute, people's names has a high information gain, but we cannot use it to test a new person since it is not very possible the name of that person could fit. To solve this problem, we use information gain ratio. We simply divide the information gain by the intrinsic information. In this way, gain ratio takes number and size of branches into account when choosing an attribute, and corrects the information gain by taking the intrinsic information of a split into account. In other words, it is a compensation for information gain. For attributes with too many branches, they have a big intrinsic information. Thus the gain ration will be smaller than gain. So we can avoid those unexpected attributes in some degree.

Difference between using pruning & not using pruning:

If we use pruning, we can avoid over fitting. In other words, if we use pruning, the

decision tree will be much shallower than the one without pruning. We can draw the tree for monk-1 problem as following. The left one has been applied with pruning and the right one has not been applied with pruning. For each of the image below, the root is on the left side and the leaves are on the right side. (i,j) means the i^{th} feature's value is j. And the number on the rightmost is the label. From the comparison we can clearly see that using pruning can makes the depth of the decision tree relatively shallow.



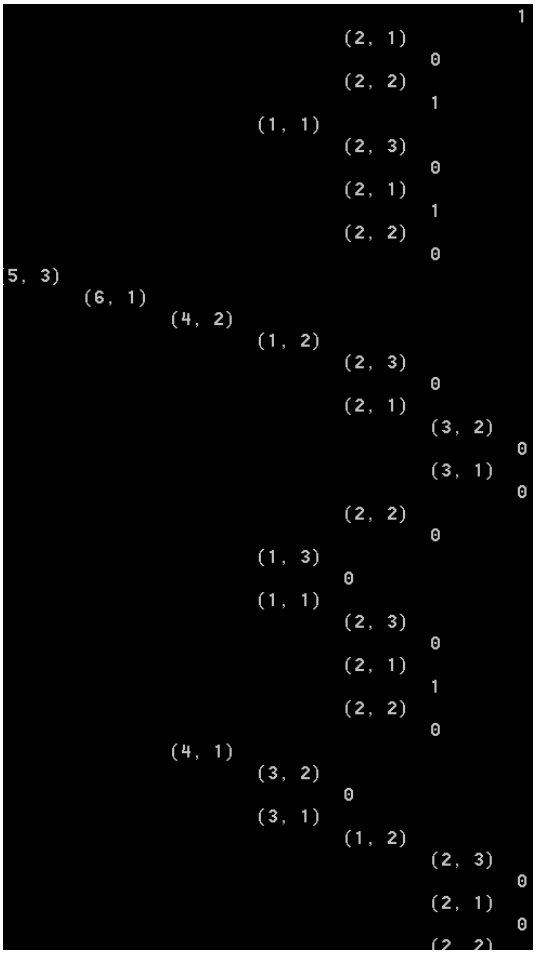


Figure1 Decision tree with pruning *Figure2 Desition tree without pruning*

One point worth mentioning is that when storing the attribute value, we have convert float numbers to integrals. For example, we convert 1.2 to 1, 3.8 to 4, and so on. This is used to avoid that for some specific value (float value) , the test data cannot fit very well so we will not get a very reliable results.

Na ïve Bayes

The accuracy, precision and recall of method are shown as following. We can simply get the results by typing each of the command below. For each test, it takes less than 1 seconds to get the results.

```
==========================================================================
Dataset    =  congress
Classifier =  naive_bayes
Option     =  normal
Labels are:  [0, 1]
Accuracy =  0.931034482759
Precision =  [0.88235294117647056, 0.96226415094339623]
Recall =  [0.9375, 0.92727272727272725]
--------------------------------------------------------------------------
```

Python train_and_test.py na ïve_bayes normal congress 0.8

```
==========================================================================
Dataset    =  monks
Classifier =  naive_bayes
Option     =  normal
Labels are:  [0, 1]
Accuracy =  0.712962962963
Precision =  [0.69327731092436973, 0.73711340206185572]
Recall =  [0.76388888888888884, 0.66203703703703709]
==========================================================================
```

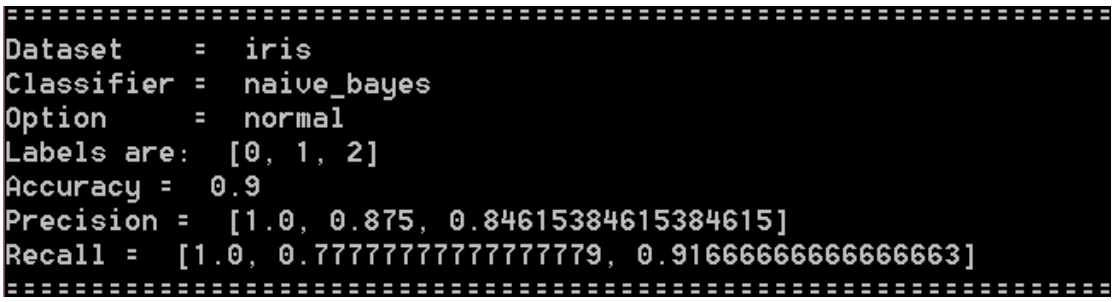
Python train_and_test.py na ïve_bayes normal monks 1

```
==========================================================================
Dataset    =  monks
Classifier =  naive_bayes
Option     =  normal
Labels are:  [0, 1]
Accuracy =  0.615740740741
Precision =  [0.67514124293785316, 0.34615384615384615]
Recall =  [0.82413793103448274, 0.19014084507042253]
--------------------------------------------------------------------------
```

Python train_and_test.py na ïve_bayes normal monks 2

```
==========================================================================
Dataset    =  monks
Classifier =  naive_bayes
Option     =  normal
Labels are:  [0, 1]
Accuracy =  0.972222222222
Precision =  [0.94444444444444442, 1.0]
Recall =  [1.0, 0.94736842105263153]
==========================================================================
```

Python train_and_test.py na ïve_bayes normal monks 3



Python train_and_test.py naïve_bayes normal iris 0.8

Neural Networks

In this part, we implemented three types of neural networks. The “shallow” neural network that contains only one hidden layer, the “medium” that contains two and the “momentum_medium” that uses momentum to accelerate convergence based on “medium”. The neural net-works take at most 2 minutes to run. This is because we used *list* and *dict* instead of *ndarray* for computation. The training and testing results are shown below.

We implemented 3 different initialization methods for weights. The default weight initialization is to randomly select a number from

(-0.01, 0.01)

The alternate methods considers the nodes for current layer and selects from

$$(-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}})$$

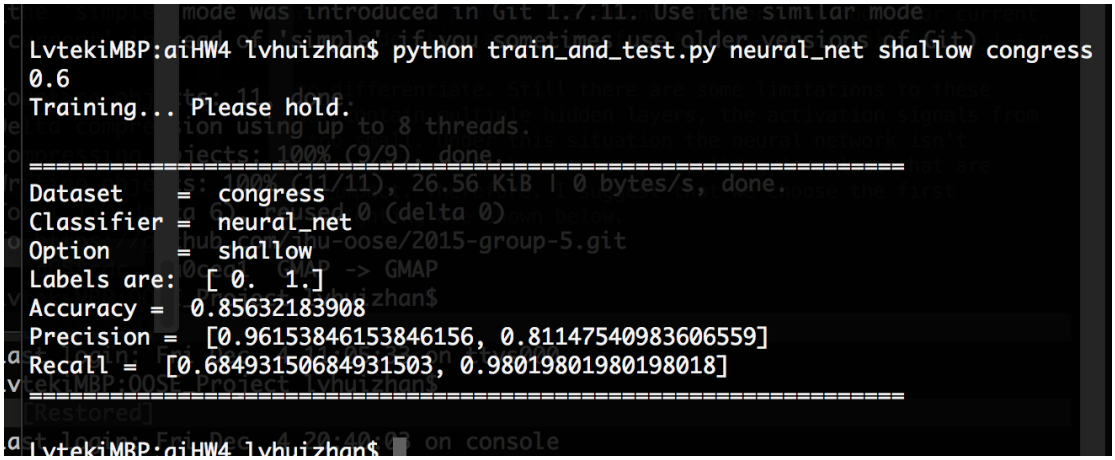
These initialization methods serve to break the symmetry of our neural networks such that the nodes can start to differentiate. Still there are some limitations to these methods. In our experiments, the first initialization works under shallow networks, but usually fails under the other networks, but usually fails under the other networks.

For medium networks that contain multiple hidden layers, the activation signals from input to output layer, and the error signals from output back to input will die out if the initial weights are too small. Under this situation the neural network isn't capable of learning anything. On the other hand, if the weights are set to inappropriately big values then the activations will be saturate, resulting in values that are asymptotic. Also due to the derivatives of activations being too small the error messages wouldn't have too much impact. Thus initial weights should not be too large or too small.

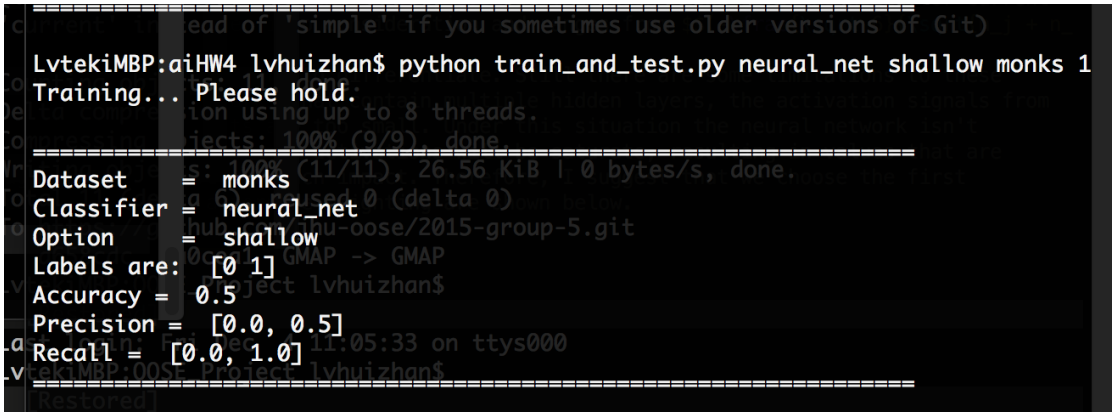
Therefore, I suggest that we choose the first initialization method for shallow network and the second for the others. The statistics for not using alternate weighting are shown below.

The results for momentums are shown below. The momentums accelerate the convergence and helps avoid local minima. In our results, the convergence for dataset congress is greatly accelerated by momentums. The accuracies of some datasets, such as monks3, are increased.

Because the labels are not always binary for these datasets, we need to use arrays to represent the precision and recall for each label.



Python train_and_test.py neural_net shallow congress 0.6



Python train_and_test.py neural_net shallow monks 1

Python train_and_test.py neural_net shallow monks 2

```
Error: Cannot find dataset name.
LvtekiMBP:aiHW4 lvhuizhan$ python train_and_test.py neural_net shallow monks 3
Training... Please hold.

========================================================================
Dataset    =  monks
Classifier =  neural_net
Option     =  shallow
Labels are:  [0 1]
Accuracy =  0.472222222222
Precision =  [0.47222222222222221, 0.0]
Recall =  [1.0, 0.0]
========================================================================
```

Python train_and_test.py neural_net shallow monks 3

```
LvtekiMBP:aiHW4 lvhuizhan$ python train_and_test.py neural_net shallow iris 0.6
Training... Please hold.

========================================================================
Dataset    =  iris
Classifier =  neural_net
Option     =  shallow
Labels are:  [ 0.  1.  2.]
Accuracy =  0.65
Precision =  [0.42857142857142855, 0.0, 0.95999999999999996]
Recall =  [1.0, 0.0, 1.0]
========================================================================
```

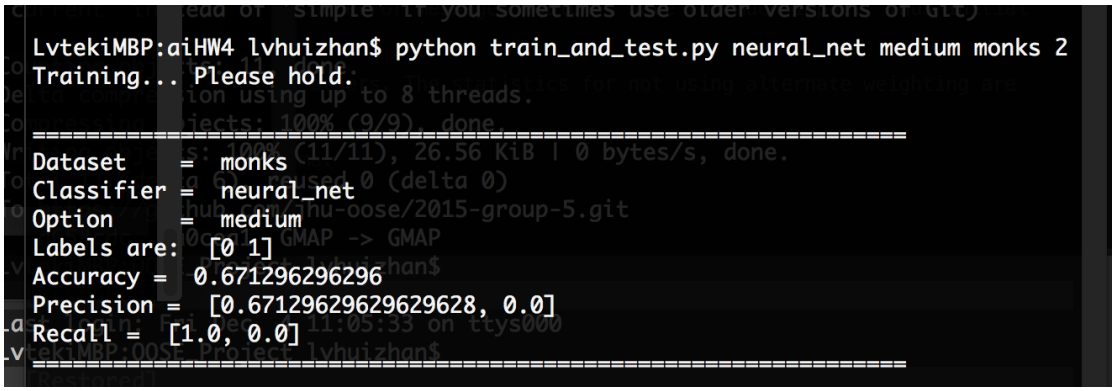
Python train_and_test.py neural_net shallow iris 0.6

```
LvtekiMBP:aiHW4 lvhuizhan$ python train_and_test.py neural_net medium congress 0
.6
Training... Please hold.

========================================================================
Dataset    =  congress
Classifier =  neural_net
Option     =  medium
Labels are:  [ 0.  1.]
Accuracy =  0.890804597701
Precision =  [0.82926829268292679, 0.94565217391304346]
Recall =  [0.93150684931506844, 0.86138613861386137]
========================================================================
```

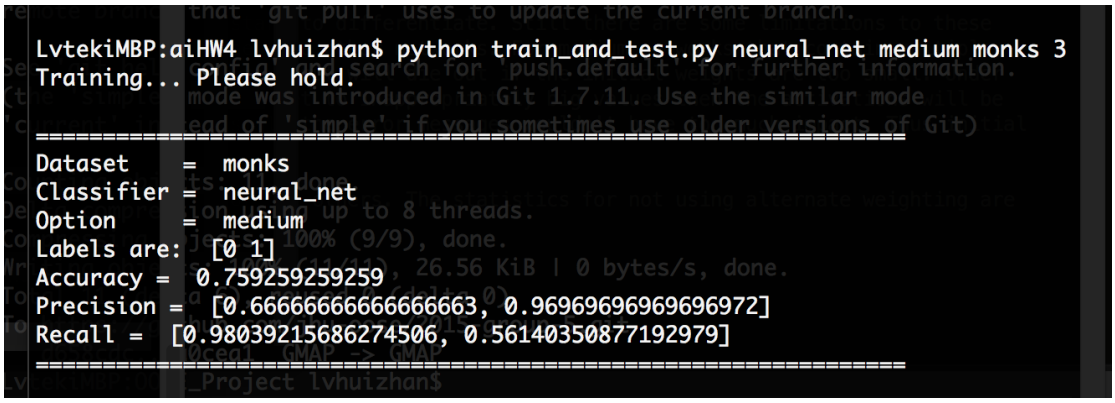
Python train_and_test.py neural_net medium congress 0.6

```
LvtekiMBP:aiHW4 lvhuizhan$ python train_and_test.py neural_net medium monks 1
Training... Please hold.

========================================================================
Dataset    =  monks
Classifier =  neural_net
Option     =  medium
Labels are:  [0 1]
Accuracy =  0.5625
Precision =  [0.75471698113207553, 0.53562005277044855]
Recall =  [0.18518518518518517, 0.93981481481481477]
========================================================================
```

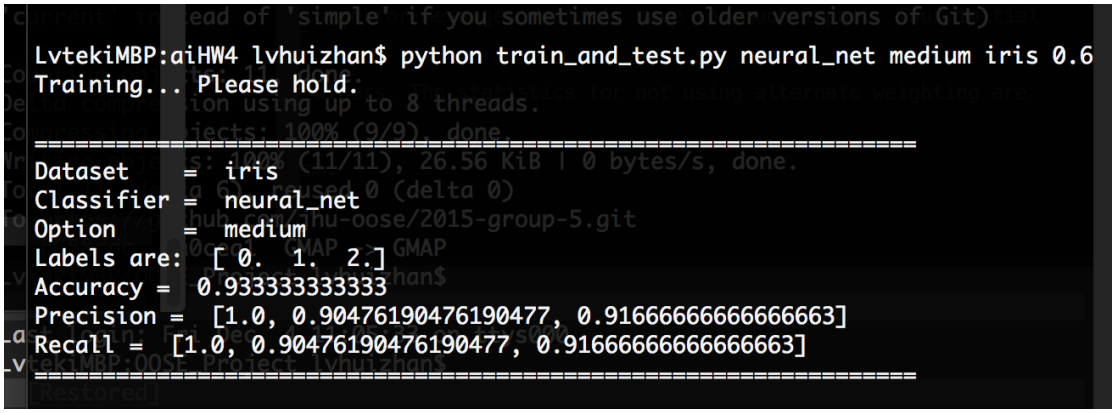
Python train_and_test.py neural_net medium monks 1



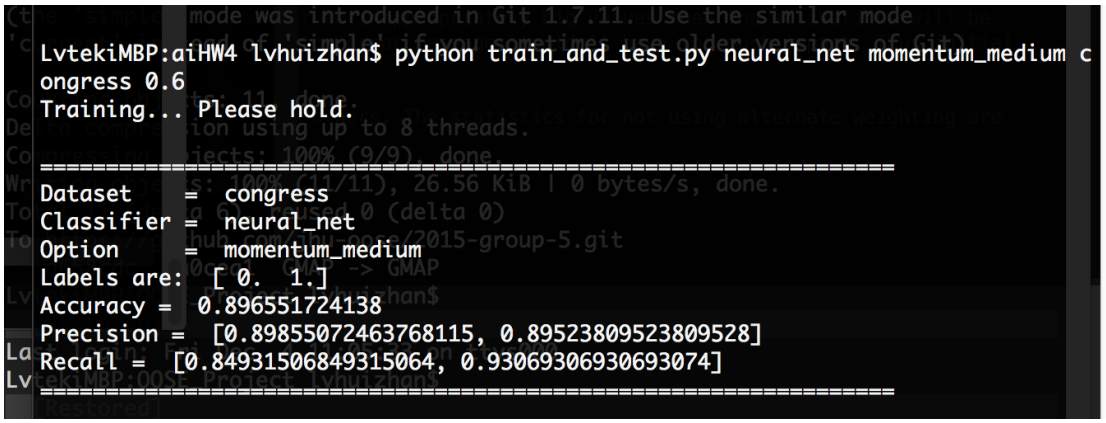
Python train_and_test.py neural_net medium monks 2



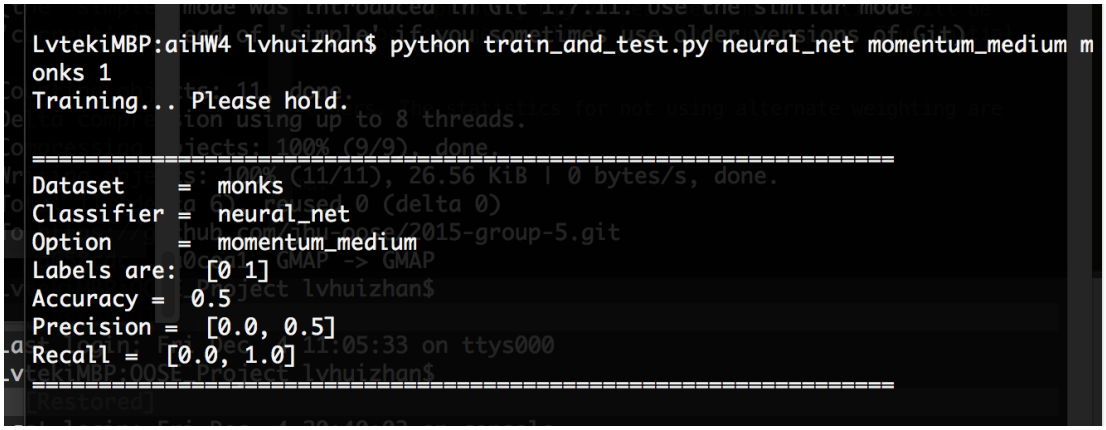
Python train_and_test.py neural_net medium monks 3



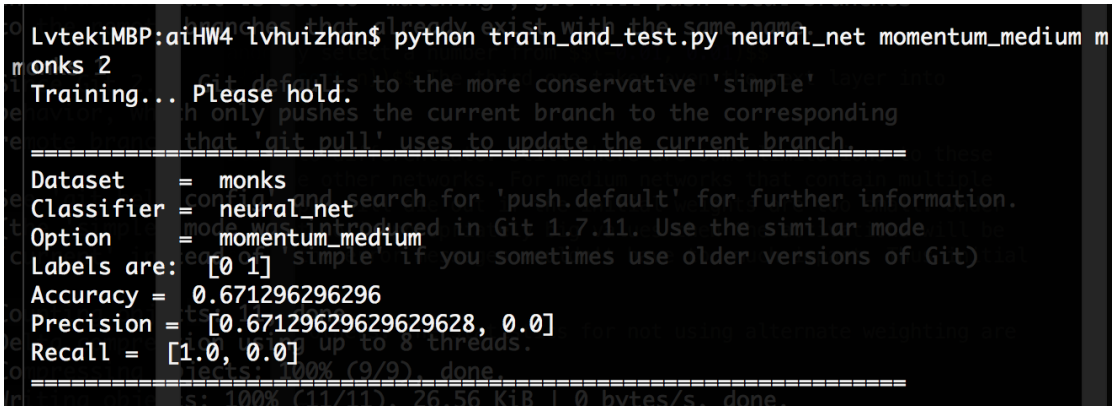
Python train_and_test.py neural_net medium iris 0.6



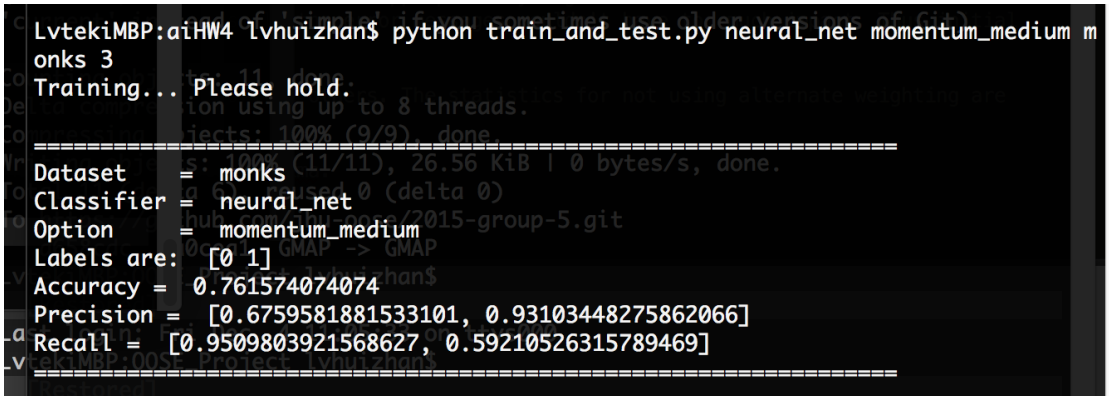
Python train_and_test.py neural_net momentum_medium congress 0.6



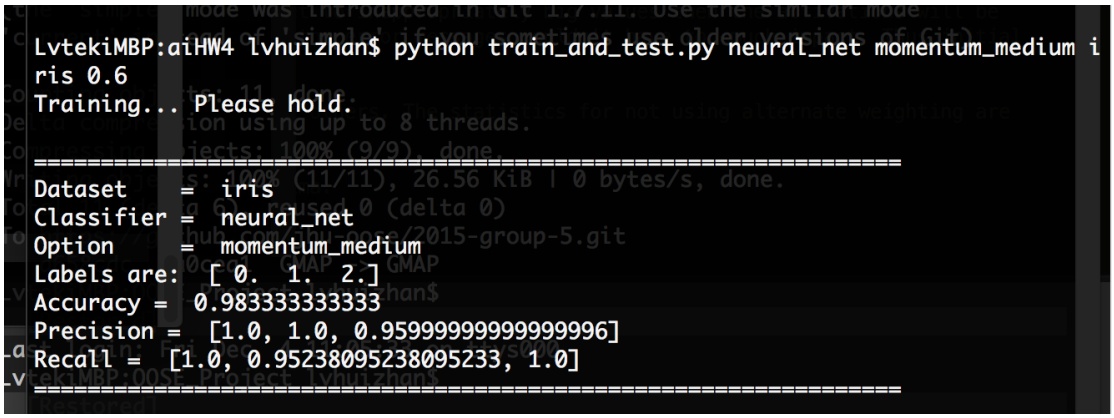
Python train_and_test.py neural_net momentum_medium monks 1



Python train_and_test.py neural_net momentum_medium monks 2



Python train_and_test.py neural_net momentum_medium monks 3



Python train_and_test.py neural_net momentum_medium iris 0.6

Discussion

ACCURACY	DT/IG	DT/IGR	Na ïve Bayes	NN/ Shalow	NN/Medium	NN/Mome
Congress	0.954	0.954	0.931	0.856	0.891	0.897
Monk1	0.712	0.701	0.713	0.500	0.563	0.500
Monk2	0.648	0.627	0.616	0.671	0.671	0.671
Monk3	0.944	0.889	0.972	0.472	0.759	0.762
Iris	0.933	0.933	0.900	0.650	0.933	0.983

Above is the statistical accuracy of each method on each data (the red data is the highest one within the dataset). From the result, we can see for Congress, Decision Tree method works the best. For Monk 1 and Monk 3, Na ïve Bayes works the best. And for monk2 and Iris, Neural Network works the best.

For Congress problem, since all of the attributes are like *yes/no* questions, and one or more attributes have very large information gain Decision Tree can quickly sort the datasets with a higher accuracy. Thus Decision Tree works better than the other two.

For Monk problem, it is probably that each of the attributes is independent of each other so that Na ïve Bayes works the best. Similarly, it is also probably that actually the attributes in other problems are not totally independent of each other, which contradicts with the independent assumption, so that Na ïve Bayes does not work that well.

For Iris problem, it is obvious that the characters of a flower are not totally independent of each other, so Na ïve Bayes does not work the best. Since the attributes are numerical, so decision tree cannot utilize all information. Neural network, which is based on numerical analysis, works the best.