Project 4
Learning

Hao Huang (NetID: hhuang40)

April 26, 2018

1. Decision Tree Learning

In this section, we will present the main idea of *decision tree learning*....

1.1 Main Idea of Decision Tree

A decision tree represents a function that takes as input a vector of attribute values and returns a "decision" — a single output value.

A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the input attributes, A_i, and the branches from the node are labeled with the possible values of the attribute, $A_i = v_{ik}$. Each leaf node in the tree specifies a value to be returned by the function. Our goal is to find a small tree that is consistent with all the data.

However, this is an intractable problem to find the smallest consistent tree. For n boolean type attributes, we have to search through the 2^{2^n} trees. The decision-tree-learning algorithm adopts a greedy divide-and-conquer strategy: always test the most important attribute first. This is a recursive algorithm, and each test divides the problem into several smaller subproblems. The algorithm is shown in Figure 1.

function DECISION-TREE-LEARNING($examples$, $attributes$, $parent\_examples$) **returns** a tree

  if $examples$ is empty **then return** PLURALITY-VALUE($parent\_examples$)
  else if all $examples$ have the same classification **then return** the classification
  else if $attributes$ is empty **then return** PLURALITY-VALUE($examples$)
  else
    $A \leftarrow \text{argmax}_{a \in attributes}$ IMPORTANCE(a, $examples$)
    $tree \leftarrow$ a new decision tree with root test A
    for each value v_k of A **do**
      $exs \leftarrow \{e : e \in examples$ **and** $e.A = v_k\}$
      $subtree \leftarrow$ DECISION-TREE-LEARNING(exs, $attributes - A$, $examples$)
      add a branch to $tree$ with label $(A = v_k)$ and subtree $subtree$
    return $tree$

Figure 1: Pseudocode of decision tree learning algorithm

The first three lines in this pseudocode list conditions in which a decision tree can be returned: all examples are classified or all attributes are tested. Otherwise, we choose the most important attribute, and for each value of this attribute, we build a smaller subtree recursively. The function `IMPORTANCE()` is responsible for choosing the most important attribute, which we will discuss later. The function `PLURALITY-VALUE()` elects the most common output value among a set of examples. If there are ties, choose one randomly.

1.2 Information Gain

The key issue in building a decision tree is to decide which attribute to choose at each step. Intuitively, we hope to choose the attribute that can classify all data as correctly as possible. One good measurement is information gain which is computed based on entropy. Here we just give a brief introduction of entropy and information gain. If the reader is interested in more detailed information, we recommend to refer to any textbook of *Information Theory*.

Let's define $B(q)$ as the entropy of a boolean random variable that is true with probability q:

$$B(q) = -(q \log_2 q + (1-q)\log_2(1-q)) \tag{1.1}$$

If a training set contains p positive examples and n negative examples, then the entropy of the goal attribute on the whole set is:

$$H(Goal) = B(\frac{p}{p+n}) \tag{1.2}$$

An attribute A with d distinct values divides the training set E into subsets E_1, ..., E_d. Each subset E_k has p_k positive examples and n_k negative examples. The expected entropy remaining after testing attribute A is:

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p+n} B(\frac{p_k}{p_k + n_k}) \tag{1.3}$$

Therefore, the information gain from the attribute test on A is the expected reduction in entropy:

$$Gain(A) = B(\frac{p}{p+n}) - Remainder(A) \tag{1.4}$$

At each internal node during building a decision tree, we will always choose the attribute with the maximum information gain.

1.3 Kep Points in Implementation

To implement this algorithm, we write code of the `IMPORTANCE()` function, although it is provided by the framework. This function computes the remainder value and compare it with the max remainder recursively. If the current value is larger than the max remainder, the max remainder will be substituted with the current one. In this way, the attribute of the most information gain can be found. The Java code is shown in Figure 2

```
private Variable maxImportance(List <Variable> attributes, Set<Example> examples) {
    double maxReminder = 0.0;
    Variable res = null;
    boolean init = false;
    int size = examples.size();
    for(Variable attr : attributes) {
        double reminder = 0.0;
        for(String a : attr.domain) {
            if(countExamplesWithValueForAttribute(examples, attr, a)!=0)
                reminder += (double)this.countExamplesWithValueForAttribute(examples, attr, a)/size
                *this.reminder(this.examplesWithValueForAttribute(examples, attr, a), attr, a);
        }
        if(reminder==reminder) {
            if(init == true) {
                if( reminder-maxReminder>0.0) {
                    maxReminder=reminder;
                    res = attr;
                }
            }else{
                maxReminder=reminder;
                res = attr;
                init=true;
            }
        }
    }
    return res;
}
```

Figure 2: Java snippet in decision tree learning algorithm

2. Linear Classifiers

In this section, we will present the main idea of two algorithms:`perceptron classifier` algorithm and `logistic classifier` algorithm. A linear function is the basis of both of these two classifiers. The first uses a "hard" threshold, while the second adopts a "soft" version of threshold.

2.1 Perceptron Classifier

A decision boundary is a line (or a surface, in higher dimensions) that separates the two classes. A linear decision boundary is called a linear separator and data that admit such a separator are called linearly separable.

2.1.1 Main Idea of Perceptron Classifier

For perceptron classifier, we adopt a soft version of threshold as shown in Figure 3:

$$h_{\mathbf{w}}(\mathbf{x}) = \mathit{Threshold}(\mathbf{w} \cdot \mathbf{x}) \text{ where } \mathit{Threshold}(z) = 1 \text{ if } z \geq 0 \text{ and } 0 \text{ otherwise}$$

Figure 3: Pseudocode of perceptron classifier algorithm

However, as implied in Figure 4 gradient is zero almost everywhere in weight space except at those points where $\mathbf{w} \cdot \mathbf{x} = 0$, and at those points the gradient is undefined.

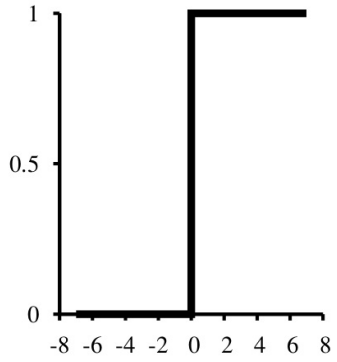

Figure 4: Gradient is undefined at zero point in a "hard" threshold

However, a simple weight update rule that converges to a solution: For a single example (x, y), we have such update rule:

$$w_i \leftarrow w_i + \alpha(y - h_{\mathbf{w}(\mathbf{x})}) \times x_i \tag{2.5}$$

This is called the perceptron learning rule. In general, the perceptron rule may not converge to a stable solution for fixed learning rate α, but if α decays as $O(1/t)$ where t is the iteration number, then the rule can be shown to converge to a minimum-error solution when examples are presented in a random sequence. In our experiment, we choose:

$$\alpha(t) = 1000/(1000 + t) \tag{2.6}$$

2.1.2 Key Points in Implementation

Once we have pseudocode of perceptron classifier, turn it into Java code is relatively easy, as shown in Figure 5.

```
public void update(double[] x, double y, double alpha) {
    // Must be implemented by you
    double hw = threshold(VectorOps.dot(this.weights, x));
    for(int i=0;i<this.weights.length;i++) {
        this.weights[i]+=alpha*(y-hw)*x[i];
    }
}
```

Figure 5: Java snippet of perceptron classifier

2.2 Logistic Classifer

The hypothesis $h_{\mathbf{w}}(\mathbf{x})$ is not differentiable. Furthermore, the linear classifier always announces a completely confident prediction of 1 or 0, even for examples that are very close

to the boundary, which means that this is not suitable for predicting linear-inseparable data. This problem can be solved by introducing a soft version of threshold.

2.2.1 Main Idea of Logistic Classifer

We can replace the hard version of threshold with a continuous, differentiable function:

$$Logistic(z) = \frac{1}{1+e^{-z}} \tag{2.7}$$

The plot of this function is shown in Figure 6. It is obvious that this sigmoid function is differentiable everywhere.

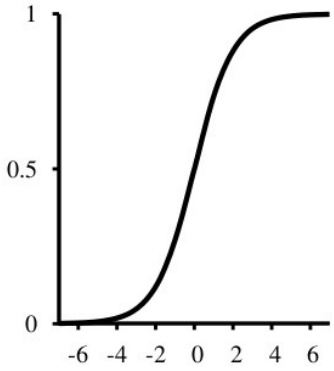

Figure 6: Gradient is defined everywhere in a "soft" threshold

With the logistic function replacing the threshold function, the prediction function is defined as:

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w}\cdot\mathbf{x} = \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}}) \tag{2.8}$$

The output is a number between 0 and 1, which can be interpreted as a probability of belonging to the class labeled 1. The hypothesis forms a soft boundary in the input space and gives a probability of 0.5 for any input at the center of the boundary region, and approaches 0 or 1 as we move away from the boundary. This is the form of logistic regression.

To compute the derivative of loss with respect to weights, we use L_2 norm and the chain rule:

$$\frac{\partial}{\partial w_i} Loss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w}\cdot\mathbf{x}) \times x_i \tag{2.9}$$

The derivative $g'(z) = g(z)(1-g(z))$, so we can get:

$$g'(\mathbf{w}\cdot\mathbf{x}) = g(\mathbf{w}\cdot\mathbf{x})(1-g(\mathbf{w}\cdot\mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})(1-h_{\mathbf{w}}(\mathbf{x})) \tag{2.10}$$

Plugging Equation 2.11 into Equation 2.9, we have the update rule for weights:

$$w_i \leftarrow w_i + \alpha(y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i \tag{2.11}$$

When the data are noisy and nonseparable, logistic regression converges far more quickly and reliably.

2.2.2 Key Points in Implementation

The Java code for logistic classifier is very similar to that of perceptron classifier, and it is shown in Figure 7. The only difference is the update rule.

```
public void update(double[] x, double y, double alpha) {
    // Must be implemented by you
    for(int i=0;i<this.weights.length;i++) {
        double hw = threshold(VectorOps.dot(this.weights, x));
        this.weights[i]+=alpha*(y-hw)*x[i]*hw*(1-hw);
    }
}
```

Figure 7: Java snippet of logistic classifier

3. Test Cases

For each model described above, we provide three dataset as test cases. For each test case, users will just follow the prompt hints to run it. For perceptron and logistic classifiers, users can specify the following parameters: the training iterations (steps), alpha policy, running mode, initial weights, the ratio between training and testing set.

Note that:

1. If alpha is set to 0, it will adopt alpha decay as described in Equation 2.6. If alpha is set to a positive number, it will be fixed to this number during the whole training process.

2. If the mode is set to "train", the model will use all data samples as training set and report their accuracy. If the mode is set to "test", the model will split all data samples into a training set and a testing set. The accuracy of both training set and testing set will be reported. In addition, before split, all data samples have been shuffled.

3.1 Decision Tree Learning Cases

3.1.1 WillWait Data

The model can replicate the textbook results for the restaurant *WillWait* example, as shown in Figure 8 and Figure 9. In this case, users do not need to specify any parameter.

```
Patrons
  None:
    No
  Some:
    Yes
  Full:
    Hungry
      No:
        No
      Yes:
        Type
          French:
            Yes
          Italian:
            No
          Thai:
            Fri/Sat
              No:
                No
              Yes:
                Yes
          Burger:
            Yes
```

Figure 8: Decision tree generated by *WillWait* data

```
[Yes, Some, No, $$$, No, Yes, 0-10, Yes, No, French] -> Yes     Yes
[Yes, Full, No, $, No, Yes, 30-60, No, No, Thai] -> No  No
[No, Some, No, $, Yes, No, 0-10, No, No, Burger] -> Yes Yes
[Yes, Full, Yes, $, No, Yes, 10-30, No, Yes, Thai] -> Yes       Yes
[No, Full, No, $$$, No, Yes, >60, Yes, Yes, French] -> No       No
[Yes, Some, Yes, $$, Yes, No, 0-10, Yes, No, Italian] -> Yes    Yes
[No, None, Yes, $, Yes, No, 0-10, No, No, Burger] -> No No
[Yes, Some, Yes, $$, No, No, 0-10, Yes, No, Thai] -> Yes        Yes
[No, Full, Yes, $, Yes, No, >60, No, Yes, Burger] -> No No
[Yes, Full, No, $$$, Yes, Yes, 10-30, Yes, Yes, Italian] -> No  No
[No, None, No, $, No, No, 0-10, No, No, Thai] -> No     No
[Yes, Full, No, $, Yes, Yes, 30-60, No, Yes, Burger] -> Yes     Yes
correct: 12/12 (100.00)%finish
```

Figure 9: Accuracy of *WillWait* data

3.1.2 House Vote Data

For this dataset, users can specify which mode to run: training mode and testing mode.

The decision tree for this data set is too depth to be shown in the report. Users can run and view the tree by themselves. The result of training mode is shown in Figure 10:

```
[y, n, y, y, n, y, y, y, n, n, n, y, y, y, n, y] -> republican  republican
[n, n, y, y, ?, y, y, y, ?, y, y, n, n, n, ?, n] -> democrat    democrat
[?, n, y, n, y, y, y, y, y, y, y, ?, n, n, n, y] -> democrat    democrat
[y, n, n, y, n, y, y, n, y, y, n, y, y, y, n, y] -> republican  republican
[n, n, y, n, n, y, y, y, y, y, n, n, n, n, n, n] -> democrat    democrat
[y, y, n, y, n, y, n, n, n, n, n, y, y, y, ?, y] -> republican  republican
[y, n, ?, y, n, y, ?, ?, n, ?, n, y, y, y, n, y] -> republican  republican
[y, n, n, y, n, n, y, n, n, n, ?, y, y, y, y, y] -> republican  republican
correct: 435/435 (100.00)%finish
```

Figure 10: Accuracy of house vote data

As we can see, the accuracy on training set is 100%. If users choose to run testing mode, the results are shown in Figure 11 and 12. (Here we use 70% of all data for training, and the rest for testing.)

```
[n, y, y, n, y, y, n, y, y, y, y, n, n, n, ?, n] -> democrat    democrat
[n, n, y, y, y, y, y, y, y, y, y, n, n, n, y, n] -> democrat    democrat
[n, n, y, n, y, y, y, y, n, y, n, y, n, n, n, n] -> republican  republican
[y, y, n, y, y, ?, y, y, y, n, n, y, n, y, y, n] -> democrat    democrat
[y, n, n, y, n, n, y, n, n, n, n, y, y, y, n, y] -> republican  republican
[y, n, n, y, n, y, n, n, n, n, n, y, y, y, n, y] -> republican  republican
[n, n, y, n, y, y, n, y, y, y, y, n, n, n, y, n] -> democrat    democrat
correct: 304/304 (100.00)%training set
```

Figure 11: Accuracy of house vote training data

```
[n, y, y, n, y, ?, n, y, y, y, y, n, n, n, n, n] -> democrat    democrat
[y, y, n, ?, n, n, n, n, y, n, n, y, y, y, y, y] -> democrat    republican
[y, ?, ?, y, n, ?, n, ?, n, n, ?, ?, y, y, y, ?] -> republican  republican
[y, n, ?, ?, n, ?, y, ?, n, n, ?, ?, y, y, y, y] -> republican  republican
[n, n, y, n, y, ?, n, y, y, y, y, n, n, n, n, n] -> democrat    democrat
[y, n, n, ?, n, ?, n, n, n, n, n, ?, y, y, y, y] -> republican  republican
correct: 118/131 (90.08)%testing set
```

Figure 12: Accuracy of house vote testing data

The accuracy of training data is 100%, while that of testing data is about 90%. It is reasonable, since testing data is different from training data.

3.1.3 Iris Data

For iris dataset, we choose the discrete version, so we do not need to spend time in discreting continuous data value. If the mode is set to "train", the decision tree is shown in Figure 13, and the accuracy is shown in Figure 14.

```
Petal width
  S:
    Iris-setosa
  MS:
    Iris-versicolor
  L:
    Iris-virginica
  ML:
    Petal length
      S:
        Iris-versicolor
      MS:
        Iris-versicolor
      L:
        Iris-virginica
      ML:
        Sepal width
          S:
            Sepal length
              S:
                Iris-virginica
              MS:
                Iris-versicolor
              L:
                Iris-versicolor
              ML:
                Iris-versicolor
          MS:
            Sepal length
              S:
                Iris-versicolor
              MS:
                Iris-versicolor
              L:
                Iris-versicolor
              ML:
                Iris-versicolor
          L:
            Iris-versicolor
          ML:
            Iris-versicolor
```

Figure 13: Decision tree generated by all samples from discrete Iris dataset

```
[L, ML, ML, L] -> Iris-virginica        Iris-virginica
[L, ML, ML, L] -> Iris-virginica        Iris-virginica
[L, MS, ML, ML] -> Iris-virginica       Iris-virginica
[L, S, ML, ML] -> Iris-virginica        Iris-virginica
[L, MS, ML, ML] -> Iris-virginica       Iris-virginica
[L, ML, ML, ML] -> Iris-virginica       Iris-virginica
[ML, MS, MS, ML] -> Iris-virginica      Iris-versicolor
correct: 143/150 (95.33)%finish
```

Figure 14: Accuracy of discrete Iris data

For this time, the accuracy of training set is not 100% any more. It means that the only four attributes are not enough to separate all samples. We need more attributes or more discriminative attributes.

If we split all samples into training set (about 70%) and testing set (about 30%), the results are shown in Figure 15 and 16.

```
[S, MS, S, S] -> Iris-setosa    Iris-setosa
[ML, MS, MS, MS] -> Iris-versicolor     Iris-versicolor
[ML, MS, ML, ML] -> Iris-versicolor     Iris-versicolor
[MS, S, S, MS] -> Iris-versicolor       Iris-versicolor
[ML, MS, ML, L] -> Iris-virginica       Iris-virginica
[L, MS, L, L] -> Iris-virginica Iris-virginica
correct: 100/105 (95.24)%training set
```

Figure 15: Accuracy of discrete Iris training data

```
[L, MS, MS, ML] -> Iris-virginica       Iris-virginica
[S, ML, S, S] -> Iris-setosa    Iris-setosa
[S, MS, S, S] -> Iris-setosa    Iris-setosa
[ML, MS, ML, ML] -> Iris-virginica      Iris-versicolor
[ML, MS, L, L] -> Iris-virginica        Iris-virginica
[L, MS, MS, ML] -> Iris-virginica       Iris-virginica
correct: 42/45 (93.33)%testing set
```

Figure 16: Accuracy of discrete Iris testing data

3.2 Perceptron Classifier Cases

3.2.1 Earthquake Clean Data

In this case we have at least 5 parameters to be specified, including: alpha policy, number of iterations (steps), running mode, and weight initialization. To avoid too many figures, we fixed alpha policy to 0, which means we use learning rate decay as described in Equation 2.6; and we also set the number of iterations to 10000. We choose "test " mode as an example to report, since it contains both training and testing statistics. In addition, we randomly select 70% samples for training, and the rest for testing.

If we initialize all weights to 0, the results is shown in Figure 17, Figure 18 and Figure 19.

```
train:  9997    0.9772727272727273
test:   9997    0.95
train:  9998    0.9772727272727273
test:   9998    0.95
train:  9999    0.9772727272727273
test:   9999    0.95
train:  10000   0.9772727272727273
test:   10000   0.95
```

Figure 17: Accuracy of both training and testing samples of earthquake clean data

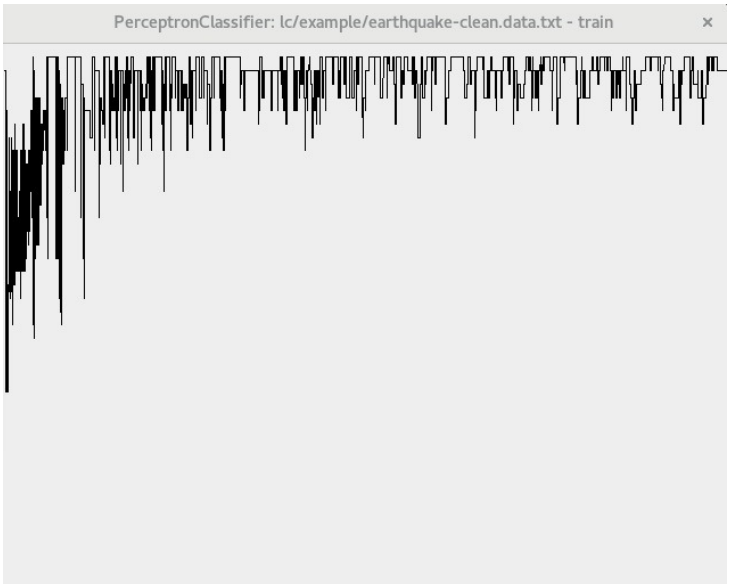

Figure 18: Plot of accuracy of training samples of earthquake clean data

Figure 19: Plot of accuracy of testing samples of earthquake clean data

The accuracy of training samples is higher than that of testing samples, which is consistent with our expectation.

If we initialize weights between -1.0 and 1.0 randomly instead of all zeros, the results if shown in Figure 20, Figure 21 and Figure 22.

```
train:  9996    0.9545454545454546
test:   9996    1.0
train:  9997    0.9545454545454546
test:   9997    1.0
train:  9998    0.9545454545454546
test:   9998    1.0
train:  9999    0.9545454545454546
test:   9999    1.0
train:  10000   0.9545454545454546
test:   10000   1.0
```

Figure 20: Accuracy of both training and testing samples of earthquake clean data

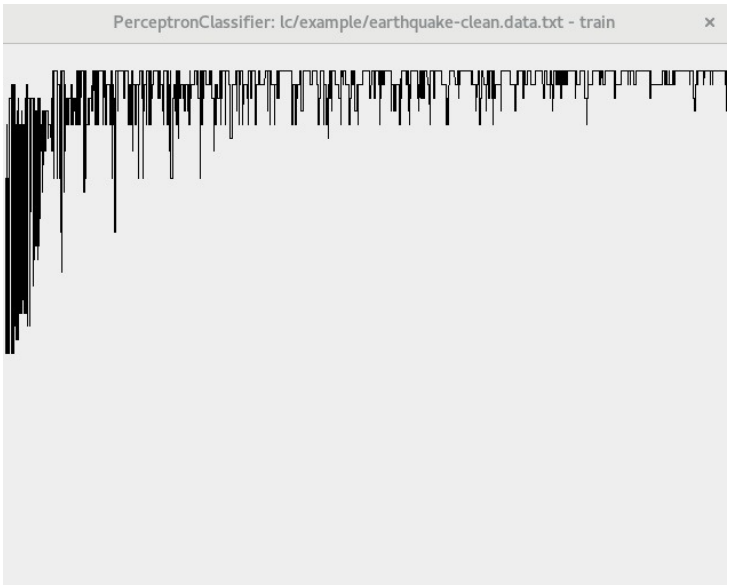

Figure 21: Plot of accuracy of training samples of earthquake clean data

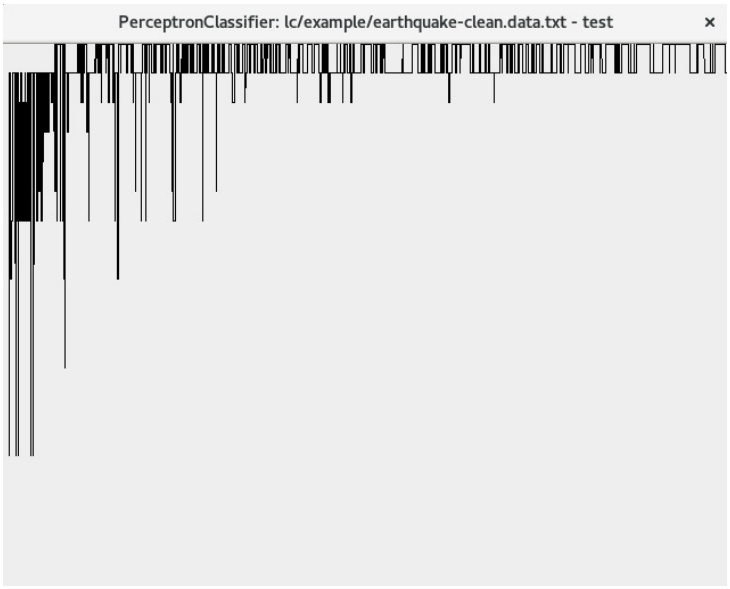

Figure 22: Plot of accuracy of testing samples of earthquake clean data

As we can see, random initialization of weights increase the accuracy of testing data, though decrease the accuracy of training data. It implies that this technique can make a model more robust and less over-fitting.

3.2.2 Earthquake Noisy Data

In this case, we set alpha to 0, the number of iterations (steps) to 10000, and initialize all weights to be 0. In test mode, we test two split ratio: one is 20% for testing, and another is 50% for testing. We expect the former one will have a higher testing accuracy. The results of the first configuration are shown in Figure 23, Figure 24 and Figure 25.

```
train:  9998    0.9298245614035088
test:   9998    0.9333333333333333
train:  9999    0.9298245614035088
test:   9999    0.9333333333333333
train:  10000   0.9298245614035088
test:   10000   0.9333333333333333
```

Figure 23: Accuracy of both training and testing samples of earthquake noisy data

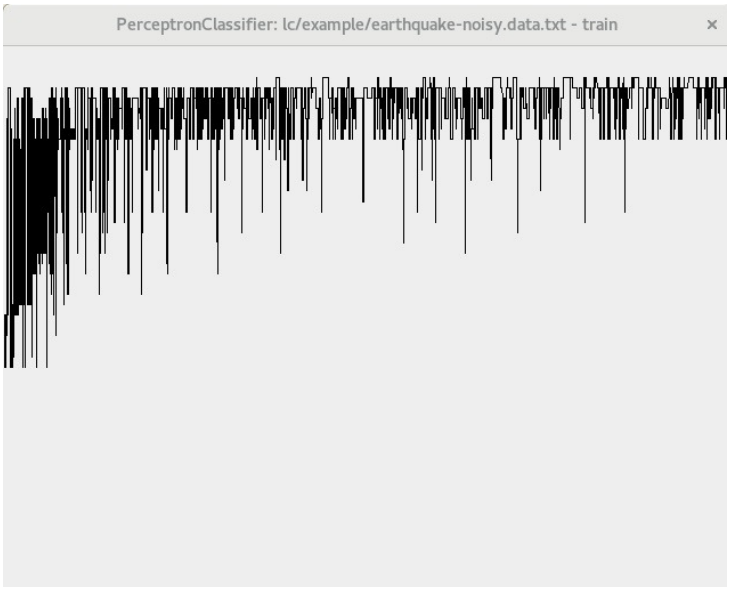

Figure 24: Accuracy of training samples of earthquake noisy data

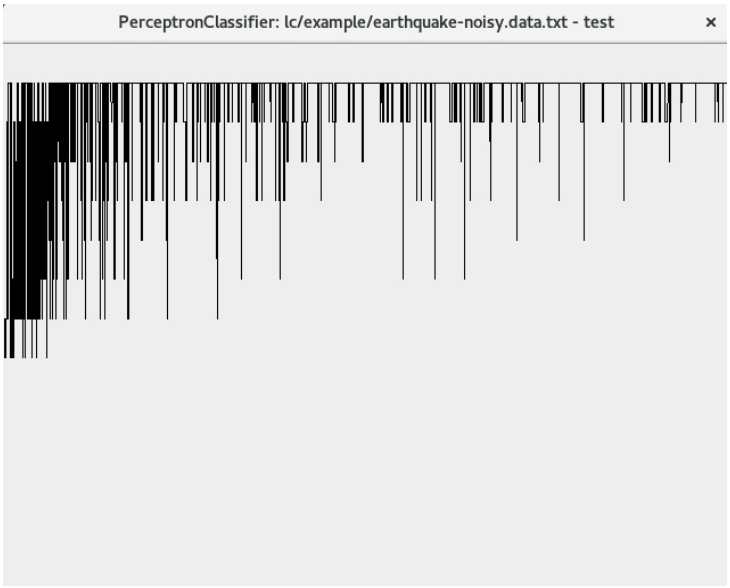

Figure 25: Accuracy of testing samples of earthquake noisy data

Then we increase the number of testing samples while decreasing that of training samples. The results are shown in Figure 26, Figure 27 and Figure 28.

```
train: 9998     1.0
test:  9998     0.8611111111111112
train: 9999     1.0
test:  9999     0.8611111111111112
train: 10000    1.0
test:  10000    0.8611111111111112
```

Figure 26: Accuracy of both training and testing samples of earthquake noisy data

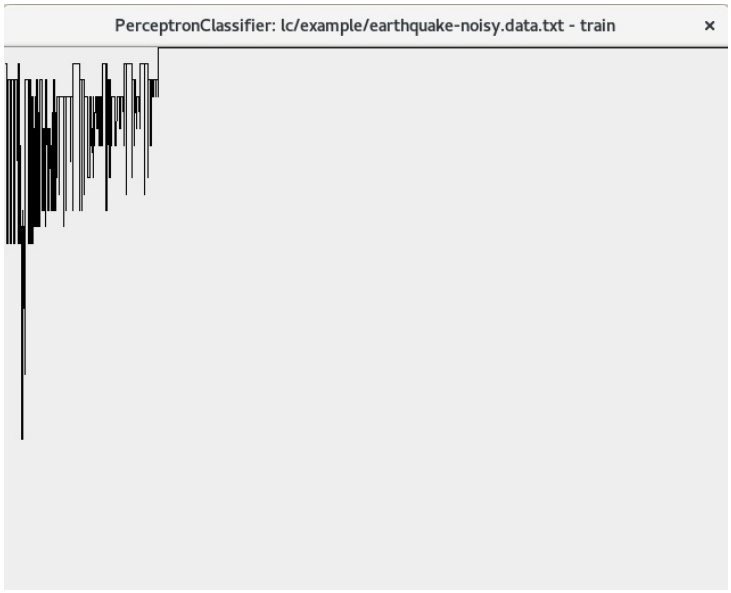

Figure 27: Accuracy of training samples of earthquake noisy data

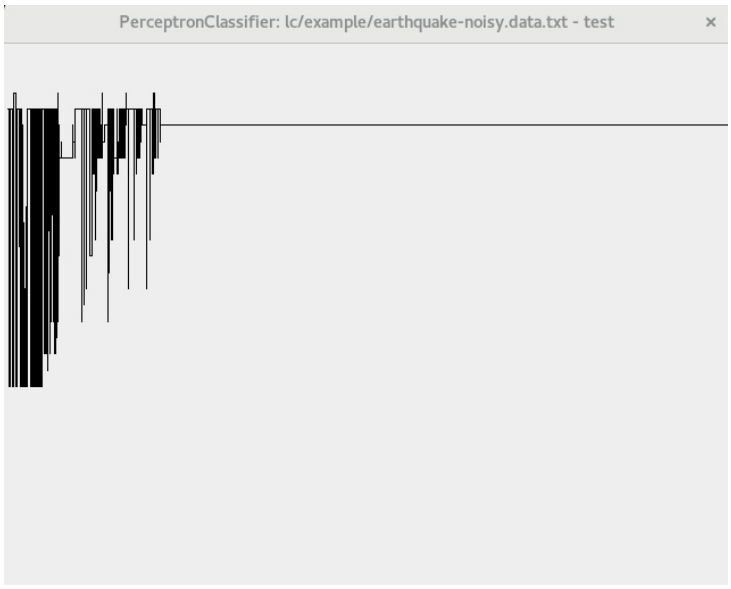

Figure 28: Accuracy of testing samples of earthquake noisy data

We've noticed that if the number of testing samples increases and that of training samples decreases, the accuracy of training increases, but the accuracy of testing decreases. That's is over-fitting. The model overfits on a smaller number of training set, but loss the

generalization power on testing set. This is not a good choice during learning process.

3.2.3 House Vote Data

In this case, we aim to test the impact of alpha. Therefore, we set the number of iterations (steps) to 10000, and initialize all weights to be 0. The split ratio is set to 7:3 (training:testing). First, we fix alpha to be 1.0, and the results are shown in Figure 29, Figure 30 and Figure 31.

```
train:  9998    0.9769736842105263
test:   9998    0.9541984732824428
train:  9999    0.9769736842105263
test:   9999    0.9541984732824428
train:  10000   0.9769736842105263
test:   10000   0.9541984732824428
```

Figure 29: Accuracy of both training and testing samples of house vote data

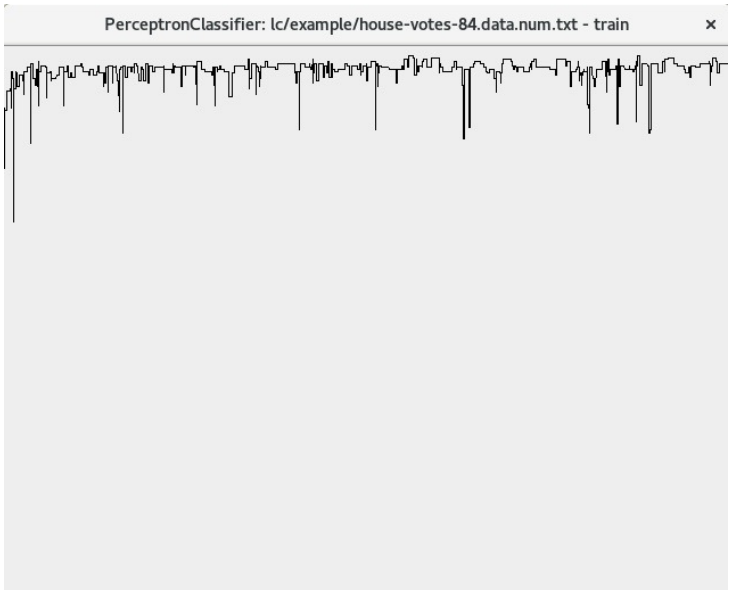

Figure 30: Accuracy of both training samples of house vote data

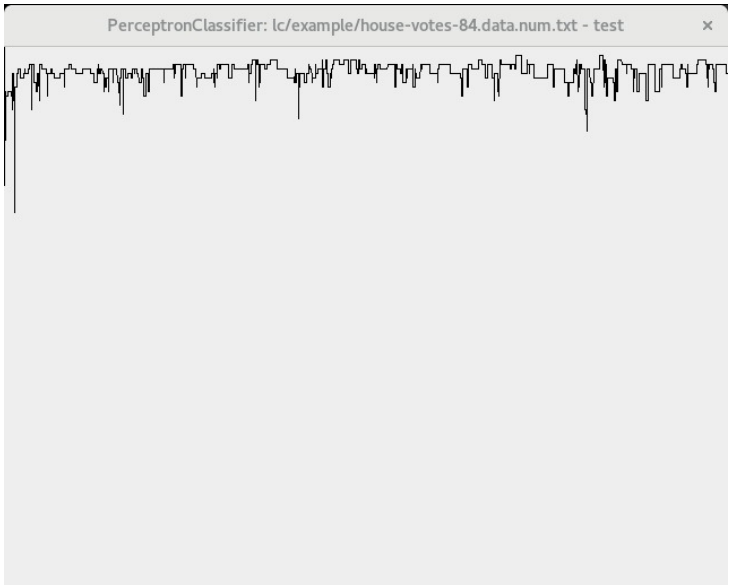

Figure 31: Accuracy of testing samples of house vote data

Then we adopt learning rate decay as describe in Equation 2.6, and the results are shown in Figure 32, Figure 33 and Figure 34.

```
train:  9998    0.9703947368421053
test:   9998    0.9618320610687023
train:  9999    0.9703947368421053
test:   9999    0.9618320610687023
train:  10000   0.9703947368421053
test:   10000   0.9618320610687023
```

Figure 32: Accuracy of both training and testing samples of house vote data

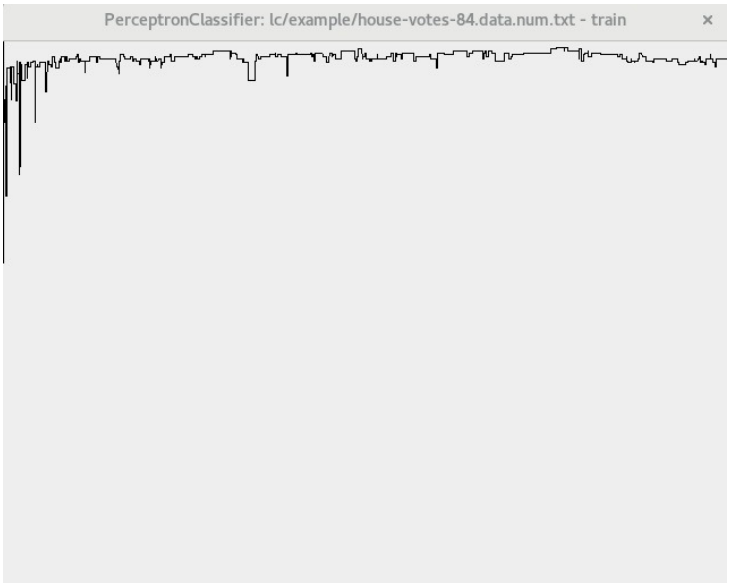

Figure 33: Accuracy of both training samples of house vote data

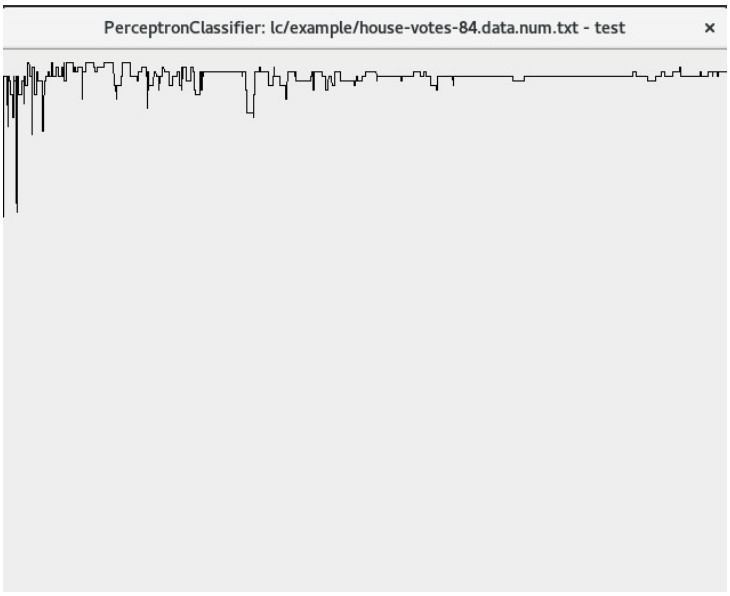

Figure 34: Accuracy of testing samples of house vote data

As shown in above figures, the change of alpha does not have much impact on prediction accuracy; at least not as much as we expect. That might be due to the simplicity of the model.

3.3 Logistic Classifier Cases

All dataset used here to test the logistic classifier model is as the same as the one used to test the previous model.

3.3.1 Earthquake Clean Data

In this case we set all parameters to the same value as described in previous section, except for the number of iterations (steps). We want to test how the number of iterations will affect the prediction accuracy.

First, we set iteration numbers to be 10000 and the results are listed in Figure 35, Figure 36 and Figure 37.

```
train:  9998    0.8782403062254256
test:   9998    0.8781306712981534
train:  9999    0.8711638666175291
test:   9999    0.8727146228750174
train:  10000   0.8184920857443825
test:   10000   0.8326008639789095
```

Figure 35: Accuracy of both training and testing samples of earthquake clean data

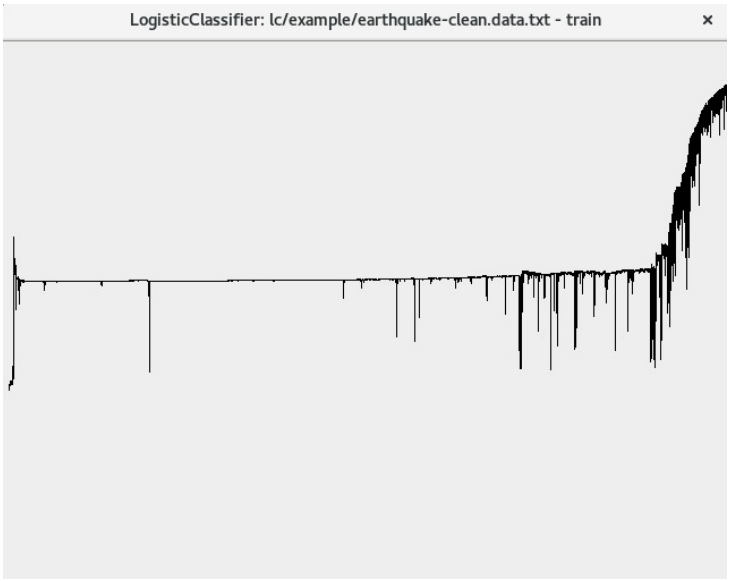

Figure 36: Accuracy of both training samples of earthquake clean data

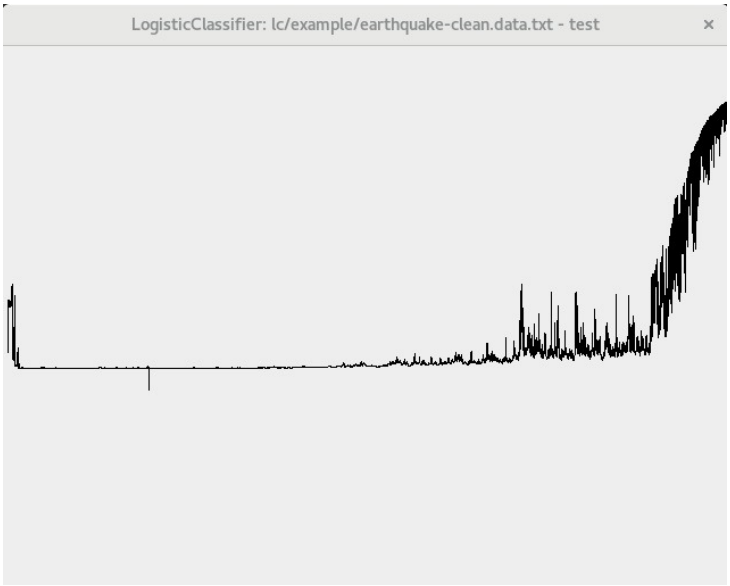

Figure 37: Accuracy of testing samples of earthquake clean data

Now we decrease the number of iterations down to 1000, and the results are shown in Figure 38, Figure 39 and Figure 40.

```
train:  998     0.5464640980131106
test:   998     0.5519945648463673
train:  999     0.5464682137398368
test:   999     0.5519502216428042
train:  1000    0.5464689721742095
test:   1000    0.5519334789487689
```

Figure 38: Accuracy of both training and testing samples of earthquake clean data

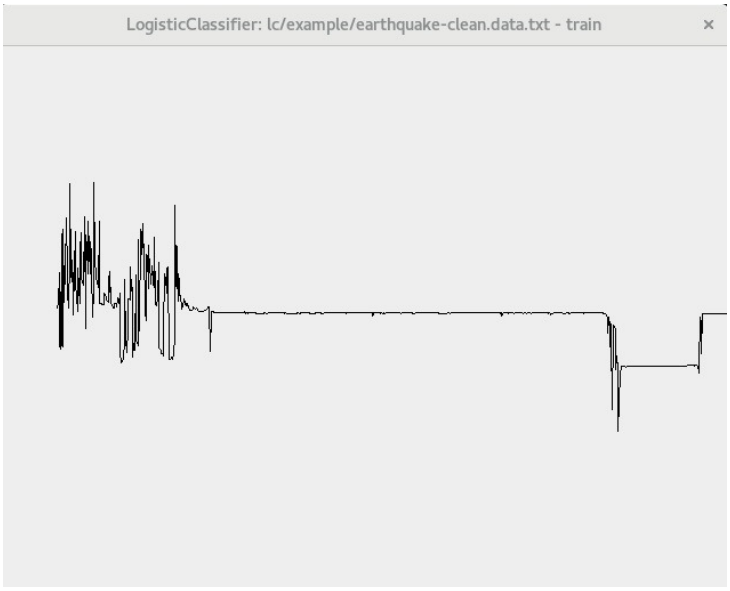

Figure 39: Accuracy of both training samples of earthquake clean data

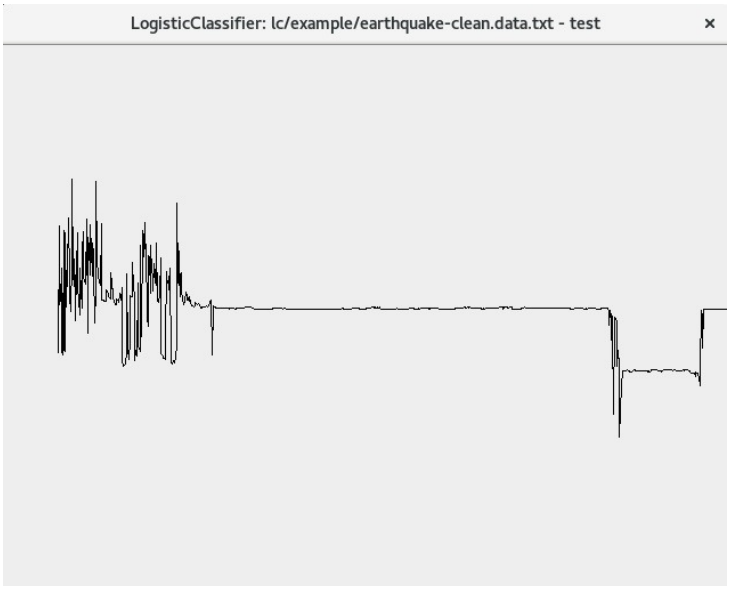

Figure 40: Accuracy of testing samples of earthquake clean data

There is a significant drop of accuracy of both training set and testing set. It demonstrate that the number of iterations is very important during learning process. If the number of iterations is too small, the model cannot learn enough information from data,

and all weights will not be adjusted to an optimal state.

3.3.2 Earthquake Noisy Data

In this case we set all parameters to the same value as described section 3.2.2, and the split ratio is 8:2. In this way, we can compare the power between perceptron classifier and logistic classifier. The results are shown in Figure 41, Figure 42 and Figure 43.

```
train:  9998    0.9292634118415843
test:   9998    0.9923513099237955
train:  9999    0.9290502911652365
test:   9999    0.992253185608035
train:  10000   0.9287761476239125
test:   10000   0.9921042057370016
```

Figure 41: Accuracy of both training and testing samples of earthquake noisy data

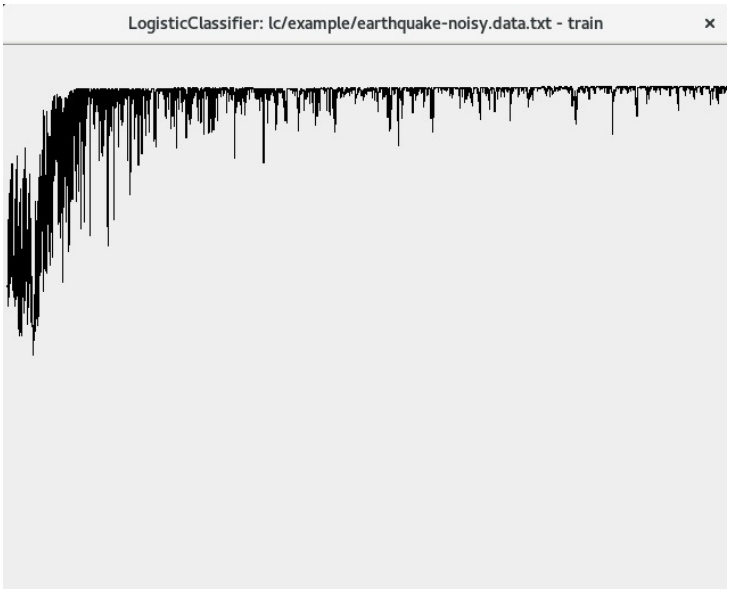

Figure 42: Accuracy of both training samples of earthquake noisy data

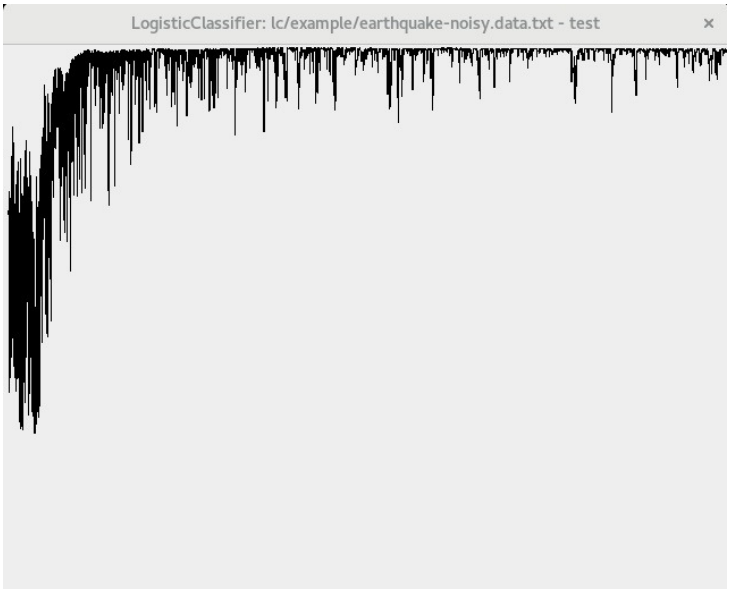

Figure 43: Accuracy of testing samples of earthquake noisy data

From the results above, we notice that the accuracy of testing data increase a little bit. It is consistent with our knowledge that logistic classifier is more capable for classifying non-linear separable data.

3.3.3 House Vote Data

For the last case, we randomly set parameters of alpha to be 0.5, the number of iterations to be 5000, all weights are initialized to be between -1.0 and 1.0, and the split ratio set to be 7:3 (training:testing). The results are shown in Figure 44, Figure 45 and Figure 46.

```
train: 4998     0.977713718097409
test:  4998     0.9784495414542252
train: 4999     0.9777129848075276
test:  4999     0.9784491307743877
train: 5000     0.9777129872117888
test:  5000     0.9784491286070774
```

Figure 44: Accuracy of both training and testing samples of house vote data

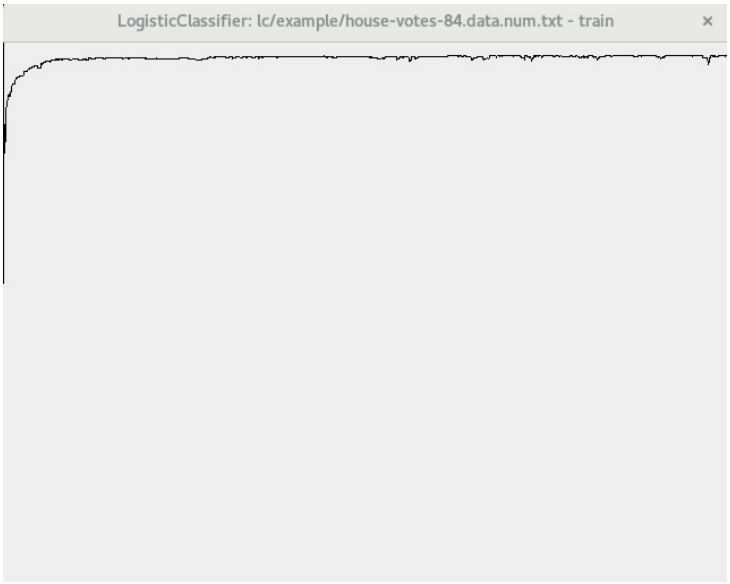

Figure 45: Accuracy of both training samples of house vote data

Figure 46: Accuracy of testing samples of house vote data

We can see that actually we do not need too many iteration times. In addition, we cannot achieve 100% accuracy, even if we train more iterations. That means the dataset is not linear separable. After all, 97% accuracy is good enough.

References

[1] Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M., & Edwards, D. D. (2003). *Artificial intelligence: a modern approach* (Vol. 2, No. 9). Upper Saddle River: Prentice hall.