

0.1 Basic Method

1 K-Nearest Neighbour Algorithm

The K-Nearest Neighbour Algorithm is an intuitive algorithm. Given an unknown sample, we compute its ‘distance’

$$d(\mathbf{v}_i, \mathbf{v}_j) = \left(\sum_{k=1}^n (v_i^{(k)} - v_j^{(k)})^p \right)^{\frac{1}{p}} \quad (1)$$

to elements in our training set, and use the average features of the K nearest elements picked out to predict the unknown one.

1.1 Implementation

Algorithm 1 K-NN Algorithm (S, Vec)

Require: training set: S , sample vector $Vec = \{v_1, v_2, \dots, v_n\}$, hashmap H

for $s_i \in S$ **do**

$$d_i \leftarrow \sqrt{\sum_{j=1}^n ((s_i)_j - v_j)^2}$$

$$H[s_i] = d_i$$

end for

sort H with key

$array \leftarrow s_i | H[s_i] \text{ are K nearest}$

$$Vec.label = \sum_{s_i \in array} \epsilon_i \cdot s_i.label, \quad \epsilon_i = H[s_i] / \sum_{s_j \in array} H[s_j]$$

1.2 Performance Analysis

Assume that \mathbf{x} is our testing vector, and the nearest training vector is \mathbf{z} . Then the generalization error is

$$P_{\text{err}} = 1 - \sum_{c \in \mathcal{Y}} P(c|\mathbf{x})P(c|\mathbf{z}) \quad (2)$$

where \mathcal{Y} is the label set. Then assume that the training set is dense enough, such that $\forall \mathbf{x}, \exists \delta, \mathbf{z} \in \mathbf{x} + \delta$. The condition gives

$$\begin{aligned} P_{\text{err}} &= 1 - \sum_{c \in \mathcal{Y}} P(c|\mathbf{x})P(c|\mathbf{z}) \\ &\approx 1 - \sum_{c \in \mathcal{Y}} P^2(c|\mathbf{x}) \\ &= 1 - \left(\arg \max_{c \in \mathcal{Y}} P(c|\mathbf{x}) \right)^2 \\ &= \left(1 - \arg \max_{c \in \mathcal{Y}} P(c|\mathbf{x}) \right) \left(1 + \arg \max_{c \in \mathcal{Y}} P(c|\mathbf{x}) \right) \leq 2 - 2 \arg \max_{c \in \mathcal{Y}} P(c|\mathbf{x}) \end{aligned} \quad (3)$$

This manifests that the error rate of K-NN will not exceed the double of the one of the Bayes optimal classifier.

1.3 Code (with Python)

Listing 1: K-NN.py

```
1 import numpy as np
2 from numpy import *
3
4 def fileToMatrix(filename):
5     file = open(filename)
6     arrayOfLines = file.readlines()
7     numOfLines = len(arrayOfLines)
8     returnMat = np.zeros([numOfLines, 3], dtype = double)
9     labelVector = []
10    index = 0
11    for line in arrayOfLines:
12        line = line.strip()
13        lineList = line.split('\t')
14        returnMat[index,...] = lineList[0:3]
15        labelVector.append( int(lineList[-1]))
16        index += 1
17    return returnMat, labelVector
18
19 def normalize(dataMat): #normalize the dataset
20     colMinVal = dataMat. min(0)
21     colMaxVal = dataMat. max(0)
22     interval = colMaxVal - colMinVal
23     normDataMat = np.zeros(shape(dataMat))
24     colLength = shape(dataMat)[0]
25     normDataMat = dataMat - np.tile(colMinVal, (colLength, 1))
26     np.seterr(invalid = 'ignore')
27     normDataMat = np.divide(normDataMat, np.tile(interval, (colLength, 1)))
28     return normDataMat
29
30 def classify(sampleVec, dataSet, labelVec, K):
31     dataSetSize = dataSet.shape[0]
32     diffMat = np.tile(sampleVec, (dataSetSize, 1)) - dataSet
33     sqDiffMat = diffMat ** 2
34     sqDistances = sqDiffMat. sum(axis = 1) #calculate the distance to each element in the dataset
35     distances = sqDistances ** (1/2)
36     disIndices = distances.argsort() #replace the distances with their ranks
37     totalDistance = 0
38     dis, vote = {}, {}
```

```

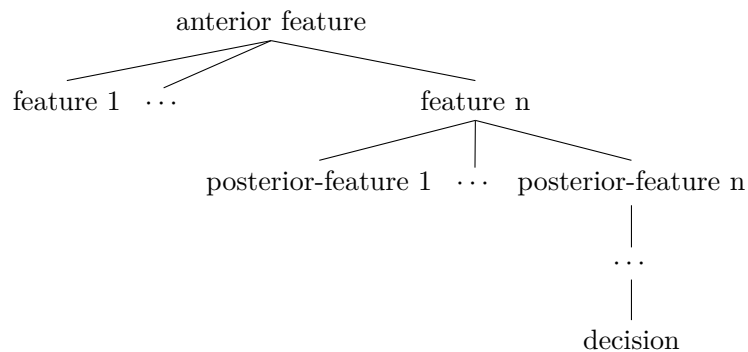
39     for i in range(dataSetSize):
40         if disIndices[i] < K:
41             dis[disIndices[i]] = (distances[i], i)
42     for d in dis:
43         totalDistance += dis[d][0]
44     for i in dis:
45         weight = dis[i][0] / totalDistance
46         if labelVec[dis[i][-1]] in vote:
47             vote[labelVec[dis[i][-1]]] += weight
48         else:
49             vote[labelVec[dis[i][-1]]] = weight
50     sortedVoteCount = sorted({v : k for k, v in vote.items()}.items(), reverse = True)
51     return sortedVoteCount[0][1]
52
53 def K_NN(sampleVec, K, dataSet, labelVec):
54     return classify(normalize(sampleVec), normalize(dataSet), labelVec, K)

```

2 Decision Tree

2.1 Basic Model

We tend to enable machines to do decision-making like humans. The data structure we use is the decision tree, where each internal node corresponds to a characteristic testing a_i , and each leaf node denotes a final decision y_i . The core manipulation is to build up optimal classification at each node. Every top-down process on analyzing a sample corresponds to a testing sequence.



2.2 Partition Scenario

Shannon Entropy

For a decision set D , the **information size** $H_0(D)$ which denotes the number of bits needed to encode elements in D is $H_0(D) = \log_2 |D|$. Let $\mathbf{D} = (D, p)$ be a discrete probability space, where $D = \{D_1, D_2, \dots, D_n\}$ is a finite set, with D_i corresponds to probability p_i under definite

discrete characteristic. Then the **Shannon entropy** of D is

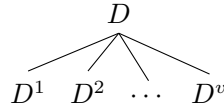
$$\text{Ent}(D) = - \sum_{i=1}^n p_i \log_2 p_i \quad (4)$$

Since $-\log_2 x$ is convex, we give an upper-bound for the entropy where

$$-\log_2 \left(\sum_{i=1}^n \frac{1}{p_i} p_i \right) \leq \sum_{i=1}^n p_i \left(-\log_2 \frac{1}{p_i} \right) = -\text{Ent}(D) \longrightarrow \text{Ent}(D) \leq \log_2 n \quad (5)$$

Information Gain

Given a partition criterion $a = \{a^1, a^2, \dots, a^v\}$ where $a^i \in a$ is the possible value. For sample set D , we split it into subsets D^1, D^2, \dots, D^v where D^i is the subset determined by criterion a^i .



Then the **information gain** we obtain from this partition is

$$\text{Gain}(D, a) = \text{Ent}(D) - \sum_{i=1}^v \frac{|D^i|}{|D|} \text{Ent}(D^i) \quad (6)$$

In ID3 Algorithm, the **optimal class partition** a_* for A with sample D is defined as

$$a_* = \arg \max_{a \in A} \text{Gain}(D, a) \quad (7)$$

while for C4.5 Algorithm, the optimal one is defined as

$$a_* = \arg \max_{a \in A} \text{GainRatio}(D, a), \quad \text{GainRatio}(D, a) = \text{Gain}(D, a) / \text{Ent}(D) \quad (8)$$

3 Implementation

Assume that training sample set $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$, where \mathbf{x}_i is the class characteristics, and y_i is the decision, eventually treated as leafnode. Possible partition criterion for D is $A = \{a_1, a_2, \dots, a_d\}$, where a_i denotes a possible partition.

Algorithm 2 treeGenerate(D, A)

Require: Training set D , Partition criterion set A

initialize *node*

if $\forall D_i D_j \in D, i \neq j, D_i = D_j$ **then**

node = *leafnode*, *node* $\leftarrow D.y$

return

end if

if $A = \emptyset$ **or** $\forall a_i, a_j \in A, i \neq j, \text{Gain}(D, a_i) = \text{Gain}(D, a_j)$ **then**

```

    node = leafnode, node  $\leftarrow \arg \max_y (|N|, N = \{y | (\mathbf{x}, y) \in D\})$ 
end if
 $a_* \leftarrow \arg \max_{a \in A} \text{Gain}(D, a)$ 
for  $a_*^v \in a_*$  do
    initialize node.branchv,  $D_v$  be the subset splitted with  $a_*^v$ 
    if  $D_v = \emptyset$  then
        node.branchv = leafnode, node.branchv  $\leftarrow \arg \max_y (|N|, N = \{y | (\mathbf{x}, y) \in D_v\})$ 
    else
        node.branchv = treeGenerate( $D_v, A - \{a_*\}$ )
    end if
end for
end for

```

4 Code

Listing 2: decisionTree.py

```

1 from math import log
2
3 class decisionNode( object): # tree organized by decision tree data structure
4     def __init__(self, label):
5         self.label = label
6         self.branches = []
7         self.decision = ""
8     def assignDecision(self, decision):
9         self.decision += decision
10    def addBranch(self, newNode):
11        newNode.assignDecision(self.decision)
12        self.branches.append(newNode)
13    def visualize(self, treeNode, layer): # Visualize manipulation displays the layer an label
14        belongs to and its anterior choice
15        print("{}({}): {}".format(layer, treeNode.decision, treeNode.label))
16        for s in treeNode.branches:
17            self.visualize(s, layer + 1)
18
19 def shannonEntropy(dataset):
20     entriesNum = len(dataset)
21     labelCount = {}
22     for dataVec in dataset:
23         dataLabel = dataVec[-1] # The last element in the vector is our decision
24         if dataLabel not in labelCount:
25             labelCount[dataLabel] = 0
26             labelCount[dataLabel] += 1
27     shannonEntropy = 0.000

```

```
27     for label in labelCount:
28         probability = labelCount[label] / entriesNum
29         shannonEntropy -= probability * log(probability, 2)
30     return shannonEntropy
31
32 def splitDataset(dataset, index, expectValue): # Search the dataset with specified index and return
33     the reduced dataset
34     retDataset = []
35     for lineVec in dataset:
36         if lineVec[index] == expectValue:
37             reducedVec = lineVec[:index]
38             reducedVec += lineVec[index+1:]
39             retDataset.append(reducedVec)
40     return retDataset
41
42 def optimalPartition(dataset): # ID3 Algotihm
43     featureNums = len(dataset[0]) - 1
44     originalEntropy = shannonEntropy(dataset)
45     bestFeature = -1
46     maxInfoGain = 0.00
47     for featureIndex in range(featureNums):
48         labelVec = set([dataset[i][featureIndex] for i in range(len(dataset))])
49         extraEntropy = 0.00
50         for label in labelVec:
51             reducedSet = splitDataset(dataset, featureIndex, label)
52             extraEntropy += len(reducedSet) / float(len(dataset)) * shannonEntropy(reducedSet)
53         if originalEntropy - extraEntropy > maxInfoGain:
54             maxInfoGain = originalEntropy - extraEntropy
55             bestFeature = featureIndex
56     return bestFeature
57
58 def majorityCount(classList):
59     classNum = {}
60     for member in classList:
61         if member not in classNum:
62             classNum[member] = 0
63         classNum[member] += 1
64     reverseDict = {v:k for k, v in classNum.items()}
65     orderList = sorted(reverseDict)
66     return reverseDict[ max(orderList)]
67
68 def createTree(dataset, labels):
69     classList = [data[-1] for data in dataset]
70     if classList.count(classList[0]) == len(classList):
```

```

71     return decisionNode(classList[0])
72 if len(dataset[0]) == 1:
73     return decisionNode(majorityCount(classList))
74 bestPartition = optimalPartition(dataset)
75 bestPartitionLabel = labels[bestPartition]
76 newTree = decisionNode(bestPartitionLabel)
77 uniqueVal = set([data[bestPartition] for data in dataset])
78 del(labels[bestPartition])
79 for value in uniqueVal:
80     subLabels = labels[:]
81     branchNode = createTree(splitDataset(dataset, bestPartition, value), subLabels)
82     branchNode.assignDecision(value)
83     newTree.addBranch(branchNode)
84 return newTree

```

5 Naive Bayes Algorithm

5.1 Principle and Method

Given that the input vector $X \subseteq \mathbb{R}^n$, where $X = \begin{bmatrix} X^{(1)} & X^{(2)} & \dots & X^{(n)} \end{bmatrix}^T$, and the relevant output class label set $Y \in \{c_1, c_2, \dots, c_K\}$. The joint distribution $P(X, Y)$ generates the data outcome

$$T = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\} \quad (9)$$

Applying Bayes rule we construct the algorithm for determining the label for an arbitrary new input.

Assume that the input vector is $x = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(n)} \end{bmatrix}$, the probable label set is $\mathbf{c} = \{c_1, c_2, \dots, c_K\}$, then for $c_i \in \mathbf{c}$

$$P(X = x|Y = c_i) = P(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, \dots, X^{(n)} = x^{(n)}|Y = c_i) \quad (10)$$

Assume that the variables in X are mutually independant, then the posteriori distribution is

$$P(Y = c_i|X = x) = \frac{P(X = x|Y = c_i)P(Y = c_i)}{\sum_i^k P(X = x|Y = c_i)P(Y = c_i)} = \frac{\prod_j P(X^{(j)} = x^{(j)}|Y = c_i)P(Y = c_i)}{\sum_i \prod_j P(X^{(j)} = x^{(j)}|Y = c_i)P(Y = c_i)}$$

The optimal choice for c_i is

$$\begin{aligned}
 y = f(x) = \arg \max_{c_i} P(Y = c_i|X = x) &= \arg \max_{c_i} \frac{\prod_j P(X^{(j)} = x^{(j)}|Y = c_i)P(Y = c_i)}{\sum_i \prod_j P(X^{(j)} = x^{(j)}|Y = c_i)P(Y = c_i)} \\
 &= \arg \max_{c_i} \prod_j P(X^{(j)} = x^{(j)}|Y = c_i)P(Y = c_i)
 \end{aligned} \quad (11)$$

Assume that $x^{(j)} \in \mathbf{a}_j = \{a_{j,1}, a_{j,2}, \dots, a_{j,S_j}\}$, where \mathbf{a}_j is the probable value set, then the apriori probability is

$$P(Y = c_i) = \frac{\sum_{k=1}^N I(y_k = c_i)}{N}$$

$$P(X^{(j)} = a_{j,l} | Y = c_i) = \frac{\sum_{k=1}^N I(x_k^{(j)} = a_{j,l}, y_k = c_i)}{\sum_{k=1}^N I(y_k = c_i)} \quad 1 \leq l \leq S_j, \quad 1 \leq j \leq n, \quad 1 \leq i \leq K$$

5.2 Laplace Smoothing

When calculating the conditional probability, some special cases might annihilate the indicator random variable, where

$$I(x_k^{(j)} = a_{j,l}, y_k = c_i) = 0 \quad (12)$$

Then we use a coefficient λ to prevent. Redefine the probabilities as

$$P(Y = c_i) = \frac{\sum_{k=1}^N I(y_k = c_i) + \lambda}{N + K\lambda}$$

$$P(X^{(j)} = a_{j,l} | Y = c_i) = \frac{\sum_{k=1}^N I(x_k^{(j)} = a_{j,l}, y_k = c_i) + \lambda}{\sum_{k=1}^N I(y_k = c_i) + S_j\lambda} \quad (13)$$

When $\lambda = 1$, it is called **Laplace Smoothing**.

5.3 Code

Listing 3: Naive Bayes Method.py

```

1 def posterioriProb(inputVec, trainData, smoothCoef): # Laplacian smooth coefficient is used to
    prevent possibility annihilation
2     labels = trainData[-1]
3     uniqueLabels = set(labels)
4     labelClasses = len(set(labels))
5     trainData = trainData[:-1]
6     totalSample = len(trainData[0])
7     characterList = []
8     labelDict = {}
9     posterioriProbList = []
10    for i in labels:
11        if i not in labelDict:
12            labelDict[i] = 1
13        else:
14            labelDict[i] += 1
15    for character in trainData:
16        singleCharacter = len(set(character))
17        characterDict = {}
18        for j in range(totalSample):

```



```

19         if (character[j], labels[j]) not in characterDict:
20             characterDict[(character[j], labels[j])] = 1
21         else:
22             characterDict[(character[j], labels[j])] += 1
23     for j in characterDict:
24         characterDict[j] = (characterDict[j] + smoothCoef) /
25             float(labelDict[j[-1]] + singleCharacter*smoothCoef)
26     characterList.append(characterDict)
27 for label in labelDict:
28     probability = (labelDict[label] + smoothCoef) / (totalSample + smoothCoef * labelClasses)
29     for i in range( len(trainData)):
30         layerCharacter = characterList[i]
31         probability *= layerCharacter[(inputVec[i], label)]
32     posterioriProbList.append(probability)
33 posterioriIndex = posterioriProbList.index( max(posterioriProbList))
34 return list(uniqueLabels)[posterioriIndex]

```

6 Logistic Regression

6.1 Logistic Distribution

If random variable X follows logistic distribution, i.e. $X \sim \text{logistic}(\mu, \gamma)$, then

$$F_X(x) = \frac{1}{1 + e^{-(x-\mu)/\gamma}} \quad f_X(x) = \frac{e^{-(x-\mu)/\gamma}}{\gamma(1 + e^{-(x-\mu)/\gamma})^2} \quad (14)$$

where μ is denotes the translation, and γ is the scale factor.

6.2 Multi-nominal Logistic Regression Model

For sample data x , we tend to build up estimator for $f(x)$, such that

$$\hat{f}(x) = \arg \max_k \Pr(Y = k | \mathbf{x}) \quad (15)$$

The core of logistic regression is to build up **linear interface** for classes $Y \in \{1, 2, \dots, K\}$. Then we expand x from \mathbb{R}^n to \mathbb{R}^{n+1} ,

$$\mathbf{w}_k = \begin{bmatrix} w_k^{(1)} \\ w_k^{(2)} \\ \vdots \\ w_k^{(n)} \end{bmatrix}, \quad \mathbf{x}_k = \begin{bmatrix} x_k^{(1)} \\ x_k^{(2)} \\ \vdots \\ x_k^{(n)} \end{bmatrix} \quad (16)$$

Then if two classes intersect, we take the class K as the standard, then

$$\log \frac{\Pr(Y = k | \mathbf{x})}{\Pr(Y = K | \mathbf{x})} = \mathbf{w}_k^T \mathbf{x} \rightarrow \Pr(Y = k | \mathbf{x}) = \Pr(Y = K | \mathbf{x}) \exp(\mathbf{w}_k^T \mathbf{x}) \quad (17)$$

Apply the normalized characteristic of probability, we have

$$\sum_{k=1}^K \Pr(Y = k|\mathbf{x}) = \Pr(Y = K|\mathbf{x}) \sum_{k=1}^K \exp(\mathbf{w}_k^T \mathbf{x}) \longrightarrow \Pr(Y = K|\mathbf{x}) = \frac{1}{\sum_{k=1}^K \exp(\mathbf{w}_k^T \mathbf{x})} \quad (18)$$

Since $\mathbf{w}_K^T \mathbf{x} = 0$, then

$$\Pr(Y = k|\mathbf{x}) = \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{1 + \sum_{k=1}^{K-1} \exp(\mathbf{w}_k^T \mathbf{x})} \quad \Pr(Y = K|\mathbf{x}) = \frac{1}{1 + \sum_{k=1}^{K-1} \exp(\mathbf{w}_k^T \mathbf{x})} \quad (19)$$

With training set $T = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)\}$, the likelihood function is

$$L(\mathbf{w}_0, \mathbf{w}_1, \dots, \mathbf{w}_N) = \prod_{i=1}^K \Pr\{Y = y_i | \mathbf{x}_i\} = \prod_{i=1}^K \frac{\exp(\mathbf{w}_{y_i}^T \mathbf{x}_i)}{1 + \sum_{k=1}^{K-1} \exp(\mathbf{w}_k^T \mathbf{x}_i)} \quad (20)$$

If $K = 2$, then the model degrades to a binomial LR model with 0 – 1 label. We take

$$\Pr(Y = 0|\mathbf{x}) = \frac{\exp(\mathbf{w}^T \mathbf{x})}{1 + \exp(\mathbf{w}^T \mathbf{x})} = \pi(\mathbf{x}) \quad \Pr(Y = 1|\mathbf{x}) = \frac{1}{1 + \exp(\mathbf{w}^T \mathbf{x})} = 1 - \pi(\mathbf{x}) \quad (21)$$

Then the likelihood function is

$$\begin{aligned} L(\mathbf{w}) &= \prod_{i=1}^N [\pi(x_i)]^{y_i} [1 - \pi(x_i)]^{1-y_i} \rightarrow \frac{\partial}{\partial \mathbf{w}^{(j)}} \log L(\mathbf{w}) = \frac{\partial}{\partial \mathbf{w}^{(j)}} \sum_{i=1}^N y_i (\mathbf{w}^T \mathbf{x}_i) - \log[1 + \exp(\mathbf{w}^T \mathbf{x}_i)] \\ &= \sum_{i=1}^N \left[y_i - \frac{\exp(\sum_{k=0}^n w^{(k)} x_i^{(k)})}{1 + \exp(\sum_{k=0}^n w^{(k)} x_i^{(k)})} \right] x_i^{(j)} \end{aligned}$$

Or

$$\frac{\partial}{\partial \mathbf{w}} \log L(\mathbf{w}) = \sum_{i=1}^N \mathbf{x}_i \left(y_i - \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x}_i)} \right) = 0 \quad (22)$$

6.3 Gradient Descent Algorithm

6.3.1 Method

If we give estimator $\hat{\mathbf{w}}$ for weight vector \mathbf{w} , then the **Loss Function** is defined as

$$L(\mathbf{w}, \mathbf{x}, y) = f(\mathbf{w}^T \mathbf{x}, y) \quad (23)$$

Where $f : \mathbb{R} \mapsto \mathbb{R}$ maps the loss outcome to quantity that corresponds to the form of y .

The Gradient Descent algorithm use iteration to approach the optimal choice with **step width** η for \mathbf{w}

Algorithm 3 gradientDescent($\mathbf{X} = [\mathbf{x}_1 \ \mathbf{x}_2 \ \dots \ \mathbf{x}_N]$, $\mathbf{Y} = [y_1 \ y_2 \ \dots \ y_N]$)

Require: $\mathbf{w}_1 = \mathbf{0}$

for $t = 1$ **to** T **do**

$\hat{\mathbf{Y}} = \mathbf{w}_t^T \mathbf{X}$

 evaluate $L = f(\mathbf{w}_t, \mathbf{X}, \mathbf{Y})$

$\mathbf{w}_t := \mathbf{w}_t - \eta \nabla_{\mathbf{w}} L(\mathbf{w}_t, \mathbf{X}, \mathbf{Y})$

end for

return \mathbf{w}_T

6.3.2 Widrow-Hoff Algorithm

If we reasonably choose the loss function to simplify the calculation. The Widrow-Hoff algorithm choose the euclidean distance as the loss function mapping, i.e. $L(\mathbf{w}, \mathbf{x}, y) = 1/2 \cdot (\mathbf{w}^T \mathbf{x} - y)^2$. Then

$$\begin{aligned} \nabla_{\mathbf{w}} \frac{1}{2} (\mathbf{w}^T \mathbf{x} - y)^2 &= \sum_{j=0}^n \frac{\partial}{\partial w^{(j)}} \frac{1}{2} \left(\sum_{k=0}^n w^{(k)} x^{(k)} - y \right)^2 \\ &= \sum_{j=0}^n \left(\sum_{k=0}^n w^{(k)} x^{(k)} - y \right) x^{(j)} \\ &= (\mathbf{w}^T \mathbf{x} - y) \mathbf{x} \end{aligned} \quad (24)$$

Then the recurrence equation becomes

$$\mathbf{w}_t := \mathbf{w}_t - \eta (\mathbf{w}_t^T \mathbf{X} - \mathbf{Y}) \mathbf{X} \quad (25)$$

6.3.3 Logistic Algorithm

We have derived that

$$\nabla_{\mathbf{w}} \log L(\mathbf{w}) = \sum_{i=1}^N \mathbf{x}_i \left(y_i - \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x}_i)} \right) \quad (26)$$

then substitute it into gradient descent algorithm as the loss function. We have

$$\mathbf{w}_t := \mathbf{w}_t - \eta \sum_{i=1}^N \mathbf{x}_i \left(y_i - \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x}_i)} \right) \quad (27)$$

Or

$$\mathbf{w}_t := \begin{bmatrix} w_t^{(0)} \\ w_t^{(1)} \\ \vdots \\ w_t^{(n)} \end{bmatrix} - \eta \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_N \end{bmatrix} \begin{bmatrix} y_1 - \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x}_1)} \\ y_2 - \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x}_2)} \\ \vdots \\ y_N - \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x}_N)} \end{bmatrix} \quad (28)$$

6.4 Code

Listing 4: **logisticRegression**

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def loadDataset(filename):
5     dataMat = []; labelMat = []
6     file = open(filename)
7     fileString = file.readlines()
8     for line in fileString:
9         lineArray = line.strip().split()
```

```
10     dataMat.append([1.0] + [ float(i) for i in lineArray[:-1]])
11     labelMat.append([ int( float(i)) for i in lineArray[-1:]])
12 dataMat = np.asarray(dataMat)
13 labelMat = np.asarray(labelMat)
14 return dataMat, labelMat
15
16 def sigmoid(X):
17     return 1.00/(1+np.exp(-X))
18
19 def sigmoidClassify(X):
20     if 1 / (1 + float(np.exp(-X))) > 0.5:
21         return 1
22     else:
23         return 0
24
25 def gradAscent(dataMat, labelMat, iterTimes, stepLength):
26     row, column = np.shape(dataMat)
27     weights = np.ones((column, 1))
28     for i in range(iterTimes):
29         h = sigmoid(dataMat.dot(weights))
30         error = labelMat - h
31         weights = weights + stepLength * (dataMat.transpose().dot(error))
32     return weights
33
34 def plotFit(weights, filename): # Plot the two-dimensional model
35     dataMat, labelMat = loadDataset(filename)
36     col = np.shape(dataMat)[0]
37     x1, y1, x2, y2 = [], [], [], []
38     for i in range(col):
39         if labelMat[i] == 1:
40             x1.append(dataMat[i, 1])
41             y1.append(dataMat[i, 2])
42         else:
43             x2.append(dataMat[i, 1])
44             y2.append(dataMat[i, 2])
45     figure = plt.figure()
46     plt.scatter(x1, y1, c = 'red')
47     plt.scatter(x2, y2, c = 'green')
48     x = np.arange(-4, 4, 0.1)
49     y = (-weights[0] - weights[1] * x) / weights[2]
50     plt.plot(x, y)
51     plt.xlabel('Type 1')
52     plt.ylabel('Type 2')
53     plt.show()
```