1 K-Nearest Neightbour Algorithm

1.1 Basic Method

The K-Nearest Neightbour Algorithm is an intuitive algorithm. Given an unknown sample, we compute its 'distance'

$$\boldsymbol{d}(\boldsymbol{v_i}, \boldsymbol{v_j}) = \left(\sum_{k=1}^{n} (v_i^{(k)} - v_j^{(k)})^p \right)^{\frac{1}{p}} \tag{1}$$

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

1.2 Implementation

Algorithm 1 K-NN Algorithm (S, Vec)

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

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

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

1.3 Performance Analysis

Assume that $\boldsymbol{x}$ is our testing vector, and the nearest training vector is $\boldsymbol{z}$. Then the generalizetion error is

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

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

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

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

1.4 Code (with Python)

Listing 1: **K-NN.py**

```
import numpy as np
from numpy import *

def fileToMatrix(filename):
     file =  open(filename)
    arrayOfLines =  file.readlines()
    numOfLines =  len(arrayOfLines)
    returnMat = np.zeros([numOfLines, 3], dtype = double)
    labelVector = []
    index = 0
    for line in arrayOfLines:
        line = line.strip()
        lineList = line.split('\t')
        returnMat[index,...] = lineList[0:3]
        labelVector.append( int(lineList[-1]))
        index += 1
    return returnMat, labelVector

def normalize(dataMat): #normalize the dataset
    colMinVal = dataMat. min(0)
    colMaxVal = dataMat. max(0)
    interval = colMaxVal - colMinVal
    normDataMat = np.zeros(shape(dataMat))
    colLength = shape(dataMat)[0]
    normDataMat = dataMat - np.tile(colMinVal, (colLength, 1))
    np.seterr(invalid = 'ignore')
    normDataMat = np.divide(normDataMat, np.tile(interval, (colLength, 1)))
    return normDataMat

def classify(sampleVec, dataSet, labelVec, K):
    dataSetSize = dataSet.shape[0]
    diffMat = np.tile(sampleVec, (dataSetSize, 1)) - dataSet
    sqDiffMat = diffMat ** 2
    sqDistances = sqDiffMat. sum(axis = 1) #calculate the distance to each element in the dataset
    distances = sqDistances ** (1/2)
    disIndices = distances.argsort() #replace the distances with their ranks
    totalDistance = 0
    dis, vote = {}, {}
```

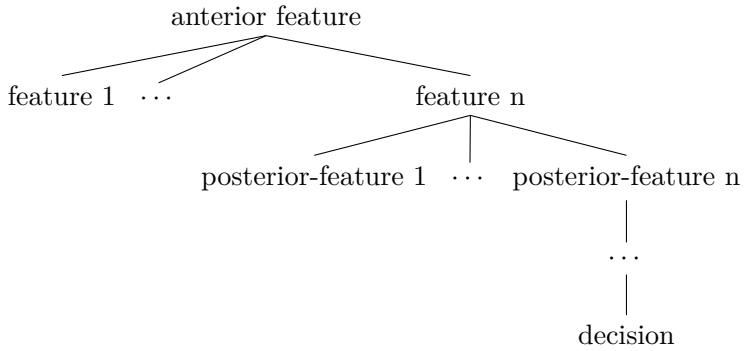
```
    for i in  range(dataSetSize):
        if disIndices[i] < K:
            dis[disIndices[i]] = (distances[i], i)
    for d in dis:
        totalDistance += dis[d][0]
    for i in dis:
        weight = dis[i][0] / totalDistance
        if labelVec[dis[i][-1]] in vote:
            vote[labelVec[dis[i][-1]]] += weight
        else:
            vote[labelVec[dis[i][-1]]] = weight
    sortedVoteCount =  sorted({v : k for k, v in vote.items()}.items(), reverse = True)
    return sortedVoteCount[0][1]

def K_NN(sampleVec, K, dataSet, labelVec):
    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 $\boldsymbol{D} = (D, p)$ be a discrete probability space, where $D = \{D_1, D_2, \ldots, D_n\}$ is a finite set, with D_i corresponds to probability p_i under definite

discrete characteristic. Then the **Shannon entropy** of $\boldsymbol{D}$ is

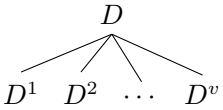
$$\mathrm{Ent}(D) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{4}$$

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

$$-\log_2(\sum_{i=1}^{n} \frac{1}{p_i} p_i) \leq \sum_{i=1}^{n} p_i(-\log_2 \frac{1}{p_i}) = -\mathrm{Ent}(D) \longrightarrow \mathrm{Ent}(D) \leq \log_2 n \tag{5}$$

Information Gain

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



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

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

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

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

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

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

3 Implementation

Assume that training sample set $D = \{(\boldsymbol{x_1}, y_1), (\boldsymbol{x_2}, y_2), \ldots, (\boldsymbol{x_n}, y_n)\}$, where $\boldsymbol{x_i}$ is the class characteristics, and y_i is the decision, eventually treated as leaftnode. Possible partition criterion for D is $A = \{a_1, a_2, \ldots, a_d\}$, where a_i denotes a possible partition.

Algorithm 2 treeGenerate(D, A)

Require: Training set D, Partition criterion set A

```
initialize node
if ∀D_i D_j ∈ D, i ≠ j, D_i = D_j then
    node = leafnode, node ← D.y
    return
end if
if A = ∅ or ∀a_i, a_j ∈ A, i ≠ j, Gain(D, a_i) = Gain(D, a_j) then
```

$node = leafnode,\ node \leftarrow \arg\max_y(|N|,\ N = \{y|(\boldsymbol{x}, y) \in D\})$

end if

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

for $a_*^v \in a_*$ **do**

initialize $node.branch^v$, D_v be the subset splited with a_*^v

if $D_v = \emptyset$ **then**

$node.branch^v = leafnode$, $node.branch^v \leftarrow \arg\max_y(|N|,\ N = \{y|(\boldsymbol{x}, y) \in D_v\})$

else

$node.branch^v = \text{treeGenerate}(D_v,\ A - \{a_*\})$

end if

end for

4 Code

Listing 2: **decisionTree.py**

```
from math import log

class decisionNode( object): # tree organized by decision tree data structure
    def __init__(self, label):
        self.label = label
        self.branches = []
        self.decision = ""
    def assignDecision(self, decision):
        self.decision += decision
    def addBranch(self, newNode):
        newNode.assignDecision(self.decision)
        self.branches.append(newNode)
    def visualize(self, treeNode, layer): # Visualize manipulation displays the layer an label
        belongs to and its anterior choice
        print("({}){}: {}". format(layer, treeNode.decision, treeNode.label))
        for s in treeNode.branches:
            self.visualize(s, layer + 1)

def shannonEntropy(dataset):
    entriesNum =  len(dataset)
    labelCount = {}
    for dataVec in dataset:
        dataLabel = dataVec[-1] # The last element in the vector is our decision
        if dataLabel not in labelCount:
            labelCount[dataLabel] = 0
        labelCount[dataLabel] += 1
    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
        the reduced dataset
33      retDataset = []
34      for lineVec in dataset:
35          if lineVec[index] == expectValue:
36              reducedVec = lineVec[:index]
37              reducedVec += lineVec[index+1:]
38              retDataset.append(reducedVec)
39      return retDataset
40
41  def optimalPartition(dataset): # ID3 Algotihm
42      featureNums =  len(dataset[0]) - 1
43      originalEntropy = shannonEntropy(dataset)
44      bestFeature = -1
45      maxInfoGain = 0.00
46      for featureIndex in  range(featureNums):
47          labelVec =  set([dataset[i][featureIndex] for i in  range( len(dataset))])
48          extraEntropy = 0.00
49          for label in labelVec:
50              reducedSet = splitDataset(dataset, featureIndex, label)
51              extraEntropy +=  len(reducedSet) /  float( len(dataset)) * shannonEntropy(reducedSet)
52          if originalEntropy - extraEntropy > maxInfoGain:
53              maxInfoGain = originalEntropy - extraEntropy
54              bestFeature = featureIndex
55      return bestFeature
56
57  def majorityCount(classList):
58      classNum = {}
59      for member in classList:
60          if member not in classNum:
61              classNum[member] = 0
62          classNum[member] += 1
63      reverseDict = {v:k for k, v in classNum.items()}
64      orderList =  sorted(reverseDict)
65      return reverseDict[ max(orderList)]
66
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 $X = \begin{bmatrix} X^{(1)} & X^{(2)} & \cdots & X^{(n)} \end{bmatrix}^T$, and the relevant output class label set $Y \in \{c_1, c_2, \ldots, c_K\}$. The joint distribution $P(X, Y)$ generates the data outcome

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

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

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

$$P(X = x|Y = c_i) = P\left(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, \ldots, X^{(n)} = x^{(n)}|Y = c_i\right) \tag{10}$$

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

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

Assume that $x^{(j)} \in \boldsymbol{a_j} = \{a_{j,1}, a_{j,2}, \ldots, a_{j,S_j}\}$, where $\boldsymbol{a_j}$ is the probable value set, then the apriori probability is

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

5.2 Laplace Smoothing

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

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

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

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

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

5.3 Code

Listing 3: **Naive Bayes Method.py**

```
def posterioriProb(inputVec, trainData, smoothCoef): # Laplacian smooth coefficient is used to
    prevent possibility annihilation
    labels = trainData[-1]
    uniqueLabels =  set(labels)
    labelClasses =  len( set(labels))
    trainData = trainData[:-1]
    totalSample =  len(trainData[0])
    characterList = []
    labelDict = {}
    posterioriProbList = []
    for i in labels:
        if i not in labelDict:
            labelDict[i] = 1
        else:
            labelDict[i] += 1
    for character in trainData:
        singleCharacter =  len( set(character))
        characterDict = {}
        for j in  range(totalSample):
```

```
            if (character[j], labels[j]) not in characterDict:
                characterDict[(character[j], labels[j])] = 1
            else:
                characterDict[(character[j], labels[j])] += 1
        for j in characterDict:
            characterDict[j] = (characterDict[j] + smoothCoef) /
                float(labelDict[j[-1]] + singleCharacter*smoothCoef)
        characterList.append(characterDict)
    for label in labelDict:
        probability = (labelDict[label] + smoothCoef) / (totalSample + smoothCoef * labelClasses)
        for i in  range( len(trainData)):
            layerCharacter = characterList[i]
            probability *= layerCharacter[(inputVec[i], label)]
        posterioriProbList.append(probability)
    posterioriIndex = posterioriProbList.index( max(posterioriProbList))
    return  list(uniqueLabels)[posterioriIndex]
```

6 Logistic Regression

6.1 Logistic Distribution

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

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

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

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

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

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

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

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

Apply the normalized characteristic of probability, we have

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

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

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

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

$$L(\boldsymbol{w_0}, \boldsymbol{w_1}, \ldots, \boldsymbol{w_n}) = \prod_{i=1}^{K} \Pr\{Y = y_i | \boldsymbol{x_i}\} = \prod_{i=1}^{K} \frac{\exp(\boldsymbol{w}_{y_i}^T \boldsymbol{x})}{1+\sum_{k=1}^{K-1} \exp(\boldsymbol{w}_{y_i}^T \boldsymbol{x})} \tag{20}$$

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

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

Then the likelihood function is

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

Or

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

6.3 Gradient Descent Algorithm

6.3.1 Method

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

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

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 $\boldsymbol{w}$

Algorithm 3 gradientDescent($\boldsymbol{X} = [\boldsymbol{x_1}\ \boldsymbol{x_2}\ \cdots\ \boldsymbol{x_N}]$, $\boldsymbol{Y} = [y_1\ y_2\ \cdots\ y_N]$)

Require: $\boldsymbol{w}_1 = \boldsymbol{0}$
  for $t = 1$ **to** T **do**
    $\hat{\boldsymbol{Y}} = \boldsymbol{w}_t^T \boldsymbol{X}$
    evaluate $L = f(\boldsymbol{w_t}, \boldsymbol{X}, \boldsymbol{Y})$
    $\boldsymbol{w}_t := \boldsymbol{w}_t - \eta \nabla_{\boldsymbol{w}} L(\boldsymbol{w_t}, \boldsymbol{X}, \boldsymbol{Y})$
  end for
  return $\boldsymbol{w}_T$

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. $L(\boldsymbol{w}, \boldsymbol{x}, y) = 1/2 \cdot (\boldsymbol{w}^T \boldsymbol{x} - y)^2$. Then

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

Then the recurrence equation becomes

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

6.3.3 Logistic Algorithm

We have derived that

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

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

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

Or

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

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()
```