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 **Minkowski 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**

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

$H[s_i] = d_i$

end for

sort H with key

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

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

1.3 Performance Analysis

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

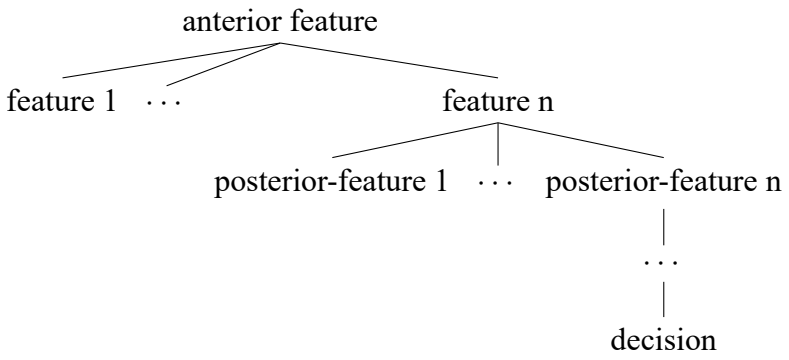
```
import numpy as np
from numpy import *

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')
```

```
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 $\boldsymbol{D} = (D, p)$ 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

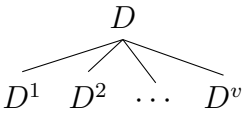
$$\text{Ent}(D) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{4}$$

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

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

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 obtain from this partition is

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

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

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

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

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

2.3 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 $\forall D_i D_j \in D, i \neq j,\ D_i = D_j$ **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**
$node = leafnode,\ node \leftarrow \arg\max_y(|N|,\ N = \{y|(\boldsymbol{x}, y) \in D\})$
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 split 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
$node.branch^v$ = treeGenerate(D_v, $A - \{a_*\}$)
end if
end for

2.4 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):
```

```
11          newNode.assignDecision(self.decision)
12          self.branches.append(newNode)
13      def visualize(self, treeNode, layer): # Visualize manipulation displays the layer an label belongs
            to and its anterior choice
14          print("({})({}: {}". format(layer, treeNode.decision, treeNode.label))
15          for s in treeNode.branches:
16              self.visualize(s, layer + 1)
17
18  def shannonEntropy(dataset):
19      entriesNum =  len(dataset)
20      labelCount = {}
21      for dataVec in dataset:
22          dataLabel = dataVec[-1] # The last element in the vector is our decision
23          if dataLabel not in labelCount:
24              labelCount[dataLabel] = 0
25          labelCount[dataLabel] += 1
26      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
```

```
    return bestFeature

def majorityCount(classList):
    classNum = {}
    for member in classList:
        if member not in classNum:
            classNum[member] = 0
        classNum[member] += 1
    reverseDict = {v:k for k, v in classNum.items()}
    orderList =  sorted(reverseDict)
    return reverseDict[ max(orderList)]


def createTree(dataset, labels):
    classList = [data[-1] for data in dataset]
    if classList.count(classList[0]) ==  len(classList):
        return decisionNode(classList[0])
    if  len(dataset[0]) == 1:
        return decisionNode(majorityCount(classList))
    bestPartition = optimalPartition(dataset)
    bestPartitionLabel = labels[bestPartition]
    newTree = decisionNode(bestPartitionLabel)
    uniqueVal =  set([data[bestPartition] for data in dataset])
    del(labels[bestPartition])
    for value in uniqueVal:
        subLabels = labels[:]
        branchNode = createTree(splitDataset(dataset, bestPartition, value), subLabels)
        branchNode.assignDecision(value)
        newTree.addBranch(branchNode)
    return newTree
```

3 Naive Bayes Algorithm

3.1 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 independent, 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

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

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

3.2 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

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

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

3.3 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]
```

4 Logistic Regression

4.1 Logistic Distribution

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

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

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

4.2 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 separating hyperplane** for classes $Y \in \{1, 2, \ldots, K\}$. Then we expand x from $\mathbb{R}^n$ to $\mathbb{R}^{n+1}$,

$$\boldsymbol{w}_k = \begin{bmatrix} w_k^{(0)} \\ w_k^{(1)} \\ \vdots \\ w_k^{(n)} \end{bmatrix} = \begin{bmatrix} b \\ w_k^{(1)} \\ \vdots \\ w_k^{(n)} \end{bmatrix}, \quad \boldsymbol{x}_k = \begin{bmatrix} x_k^{(0)} \\ x_k^{(1)} \\ \vdots \\ x_k^{(n)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_k^{(1)} \\ \vdots \\ x_k^{(n)} \end{bmatrix} \tag{16}$$

where b is the bias.

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 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

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

4.3 Gradient Descent Algorithm

4.3.1 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 $\boldsymbol{w}_T$

4.3.2 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

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

4.3.3 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

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

4.4 Code

Listing 4: **logisticRegression.py**

```
import numpy as np
import matplotlib.pyplot as plt

def loadDataset(filename):
    dataMat = []; labelMat = []
     file =  open(filename)
    fileString =  file.readlines()
    for line in fileString:
        lineArray = line.strip().split()
        dataMat.append([1.0] + [ float(i) for i in lineArray[:-1]])
        labelMat.append([ int( float(i)) for i in lineArray[-1:]])
    dataMat = np.asarray(dataMat)
    labelMat = np.asarray(labelMat)
    return dataMat, labelMat

def sigmoid(X):
    return 1.00/(1+np.exp(-X))

def sigmoidClassify(X):
    if 1 / (1 +  float(np.exp(-X))) > 0.5:
        return 1
    else:
        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()
```

5 Support Vector Machine

5.1 Constraint Condition for Linear SVM

Consider **binary label** sample $T = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)\}$, where $\boldsymbol{x} \in \mathcal{X} = \mathbb{R}^n$, $y \in \{-1, +1\}$. We assume that the optimal hyperplane separating data pairs into binary classes is

$$\boldsymbol{w}^T \boldsymbol{x} + b = 0 \tag{29}$$

Relevant decision function is the **Heaviside step function**

$$f(\boldsymbol{x}) = \text{sign}(\boldsymbol{w}^T \boldsymbol{x} + b) \tag{30}$$

For each data pair $(\boldsymbol{x}_i, y_i) \in T$, the **geometric margin** is defined as

$$\gamma_i = y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \;/\; ||\boldsymbol{w}|| \tag{31}$$

Since the confidence we have in the separate hyperplane is positive proportional to the geometric margin, then we define the geometric margin between the hyperplane $(\boldsymbol{w}, b)$ and dataset T as

$$\gamma = \min_{1\le i\le N} \gamma_i = \min_{1\le i\le N} \frac{y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)}{||\boldsymbol{w}||} \tag{32}$$

The core implementation of SVM is to maximize the geometric margin, i.e. to find

$$\arg\max_{\boldsymbol{w},b} \left(\min_{1\le i\le N} \frac{y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)}{||\boldsymbol{w}||} \right) \tag{33}$$

If the minimum function margin ($= \min_{1\le i\le N} y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)$) is fixed to 1, then the preposition is equivalent to

$$\arg\max_{\boldsymbol{w},b} \left(\min_{1\le i\le N} \frac{y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)}{||\boldsymbol{w}||} \right) \Bigg|_{\gamma\equiv 1} \sim \arg\max_{\boldsymbol{w},b} \frac{1}{2}||\boldsymbol{w}||^2, \;\; \text{s.t. } \gamma_i = y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \ge \min_{1\le i\le N} \gamma_i = 1$$

Take Lagrange multiplier matrix $\boldsymbol{\alpha} = \begin{bmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_n \end{bmatrix}^T$,then the Lagrange function is

$$L(\boldsymbol{w}, \boldsymbol{\alpha}, b) = \frac{1}{2}||\boldsymbol{w}||^2 - \sum_{i=1}^{N} \alpha_i y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) + \sum_{i=1}^{N} \boldsymbol{x}_i \tag{34}$$

The Lagrange dual problem gives

$$\arg\max_{\alpha} \min_{\boldsymbol{w},b} L(\boldsymbol{w}, \boldsymbol{\alpha}, b) \tag{35}$$

We firstly consider internal minimization, where

$$\begin{aligned} \nabla_{\boldsymbol{w}} L(\boldsymbol{w}, \boldsymbol{\alpha}, b) &= \sum_{j=1}^{n} \frac{\partial}{\partial w^{(j)}} \left(\frac{1}{2}||\boldsymbol{w}||^2 - \sum_{i=1}^{N} \alpha_i y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) + \sum_{i=1}^{N} \boldsymbol{x}_i \right) \\ &= \sum_{j=1}^{n} \left(w^{(j)} - \sum_{i=1}^{N} \alpha_i y_i x_i^{(j)} \right) = \boldsymbol{w} - \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i = 0 \end{aligned} \tag{36}$$

$$\nabla_b L(\boldsymbol{w}, \boldsymbol{\alpha}, b) = -\sum_{i=1}^{N} \alpha_i y_i = 0 \tag{37}$$

Then the original function becomes

$$\begin{aligned} L(\boldsymbol{w}, \boldsymbol{\alpha}, b) &= \frac{1}{2} \left(\sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i \right)^T \left(\sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i \right) - \sum_{i=1}^{N} \alpha_i y_i \left(\left(\sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i \right)^T \boldsymbol{x}_i + b \right) + \sum_{i=1}^{N} \boldsymbol{x}_i \\ &= -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle + \sum_{i=1}^{N} \alpha_i \end{aligned} \tag{38}$$

The terminal target function is

$$\boxed{\arg\min_{\boldsymbol{\alpha}} \left(\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle - \sum_{i=1}^{N} \alpha_i \right) = \arg\min_{\boldsymbol{\alpha}} W(\boldsymbol{\alpha}) \quad \text{s.t.} \sum_{i=1}^{N} \alpha_i y_i = 0} \tag{39}$$

Once the iteration has reached equilibrium state, denoted as $\boldsymbol{\alpha}^*$, then we can solve the corresponding parameters $\boldsymbol{w}^*$ and b^*, where

$$\boldsymbol{w}^* = \sum_{i=1}^{n} \alpha_i^* y_i \boldsymbol{x}_i \tag{40}$$

$$y_j(\langle \boldsymbol{w}^*, \boldsymbol{x}_j \rangle + b^*) = 1 \rightarrow b^* = y_j - \sum_{i=1}^{n} \alpha_i^* y_i \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle \tag{41}$$

5.2 Constraint Condition under Soft Margin Assumption

The above constraint condition is derived under the condition that dataset T is linearly separable. For dataset that is not linearly separable, i.e. **the convex hulls determined by two classes of data points have intersecting edges**.

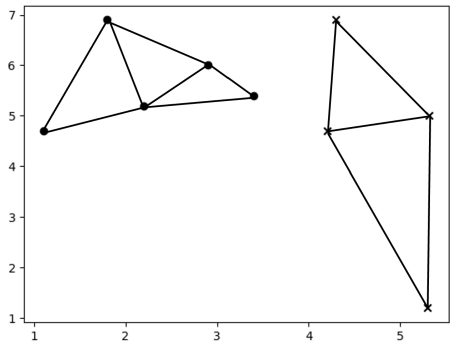


Fig. 1: Separable

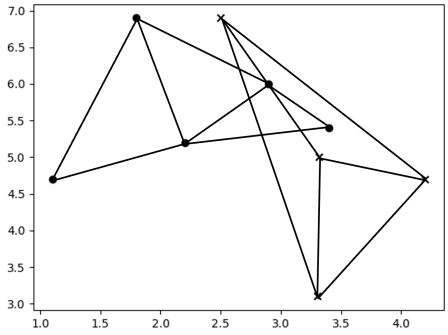


Fig. 2: Not Separable

To cancel out the effect of some specific data points, we introduce **slack variable** ξ_i for each data pair $(\boldsymbol{x}_i, y_i)$, and define the **soft geometric margin** as

$$\gamma = \frac{y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) + \xi_i}{||\boldsymbol{w}||} \tag{42}$$

Then for each slack variable, the target function gains a relevant penalty $C\xi_i$. Then the equivalent problem is

$$\begin{aligned} &\arg\max_{\boldsymbol{\alpha}} \min_{\boldsymbol{w},\boldsymbol{\xi},b} \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{N} \xi_i \\ &\text{s.t. } y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1 - \xi_i, \;\; \xi_i \geq 0, \end{aligned} \tag{43}$$

Lagrange function with double multipliers $(\boldsymbol{\alpha}, \boldsymbol{\mu})$ is

$$L(\boldsymbol{\alpha}, \boldsymbol{w}, \boldsymbol{\xi}, b) = \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\alpha_i\left(y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) - 1 + \xi_i\right) - \sum_{i=1}^{N}\mu_i\xi_i \tag{44}$$

with the dual problem

$$\arg\max_{\boldsymbol{\alpha}}\min_{\boldsymbol{w},\boldsymbol{\xi},b} L(\boldsymbol{\alpha}, \boldsymbol{w}, \boldsymbol{\xi}, b) \quad \text{s.t. } \alpha_i \geq 0,\ \mu_i \geq 0 \tag{45}$$

then

$$\frac{\partial}{\partial \boldsymbol{w}}L = \frac{\partial}{\partial \boldsymbol{x}_i}L = \frac{\partial}{\partial b}L = 0 \longrightarrow \begin{cases} \boldsymbol{w} = \sum_{i=1}^{N}\alpha_i y_i \boldsymbol{x}_i \\ \sum_{i=1}^{N}\alpha_i y_i = 0 \\ C - \alpha_i - \mu_i = 0,\ \ \alpha_i \geq 0,\ \mu_i \geq 0 \rightarrow 0 \leq \alpha_i \leq C \end{cases} \tag{46}$$

Likewise, we obtain a new **quadratic programming** target

$$\begin{aligned} \arg\min_{\boldsymbol{\alpha}} \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{y}_i\rangle - \sum_{i=1}^{N}\alpha_i \\ \text{s.t. } \sum_{i=1}^{N}\alpha_i y_i = 0,\ 0 \leq \alpha_i \leq C \end{aligned} \tag{47}$$

5.3 Hinge Loss Function

We can give an equivalent description of the original optimizing problem. The minimizing part gives

$$\arg\min_{\boldsymbol{w},\boldsymbol{\xi},b} \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{N}\xi_i, \ \ \text{s.t. } y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1 - \xi_i,\ \xi_i \geq 0 \tag{48}$$

If we take

$$\left[1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\right]_+ = \xi_i \quad \left(f(x)_+ = \begin{cases} f(x) & f(x) > 0 \\ 0 & x \leq 0 \end{cases}\right) \tag{49}$$

Then if $1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) > 0$, then $y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) = 1 - \xi_i$. If $1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \leq 0$, then $\xi_i = 0$. It meets the condition that $y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1 - \xi_i$. Hence, the minimization can be rewritten as

$$\arg\min_{\boldsymbol{w},\boldsymbol{\xi},b} \frac{1}{2C}||\boldsymbol{w}||^2 + \sum_{i=1}^{N}\xi_i \overset{\lambda=1/2C}{\longrightarrow} \arg\min_{\boldsymbol{w},b} \lambda||\boldsymbol{w}||^2 + \sum_{i=1}^{N}\left[1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\right]_+ \tag{50}$$

Then loss function $\left[1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\right]_+$ on the RHS is defined as the **hinge loss function**, which requires a higher standard for sample studying.

5.4 Kernel Trick for Non-linear SVM

The linear separable and linear inseparable sample can be split using a hyperplane. Whereas non-linear samples require a hypersurface for partition.

5.4.1 Kernel Function

Assume input space is $\mathcal{X}$ and characteristic space $\mathcal{H}$, there exists a mapping $\phi(x)$, where

$$\phi(x) : \mathcal{X} \mapsto \mathcal{H} \tag{51}$$

such that $\forall x, y \in \mathcal{X}$,

$$K(x, y) = \langle \phi(x), \phi(y) \rangle \tag{52}$$

then $K(x, y)$ is the kernel function, and $\phi(x)$ is the mapping function. For a given kernel $K(x, y)$, the selection for characteristic space and mapping function is not unique.

In non-linear SVM, we take the place of inner function in the optimization by the kernel function, i.e.

$$\begin{aligned} \arg\min_{\boldsymbol{\alpha}} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(\boldsymbol{x}_i, \boldsymbol{y}_i) - \sum_{i=1}^{N} \alpha_i \\ \text{s.t. } \sum_{i=1}^{N} \alpha_i y_i = 0,\ 0 \le \alpha_i \le C \end{aligned} \tag{53}$$

5.4.2 Positive Definite Kernel Function

The following will give the necessary and sufficient condition for $K(x, y)$ to be **positive definite kernel**. We first consider mapping function

$$\phi : x \mapsto K(\ \cdot\ , x) \tag{54}$$

Then $\forall x_i \in \mathcal{X}, \alpha_i \in \mathbb{R}$, define linear mapping

$$f(\ \cdot\ ) = \sum_{i=1}^{N} \alpha_i K(\ \cdot\ , x_i) \tag{55}$$

the space $\mathcal{S}$ is determined by mapping ϕ. Define operation $\langle f, g, \rangle$ in space $\mathcal{S}$ for f, g, where

$$f(\ \cdot\ ) = \sum_{i=1}^{N} \alpha_i K(\ \cdot\ , x_i),\ \ g(\ \cdot\ ) = \sum_{j=1}^{N} \beta_j K(\ \cdot\ , y_j) \tag{56}$$

$$\langle f, g \rangle = \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \beta_j K(x_i, y_j) \tag{57}$$

Lemma The operation is a inner product of $\mathcal{S}$.

$$\sum_{i=1}^{N}\sum_{j=1}^{N} C\alpha_i\beta_j K(x_i,y_j) = C\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\beta_j K(x_i,y_j) \longrightarrow \langle Cf,g\rangle = C\langle f,g\rangle \tag{58}$$

$$\langle f,g\rangle = \langle g,f\rangle \ \text{(trivial)} \tag{59}$$

$$\sum_{i=1}^{N}\sum_{j=1}^{N}((\alpha_i+\beta_i)\gamma_j K(y_i,z_j)) = \sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\gamma_j K(x_i,z_j) + \sum_{i=1}^{N}\sum_{j=1}^{N}\beta_i\gamma_j K(y_i,z_j)$$

$$\rightarrow \langle f+g,h\rangle = \langle f,h\rangle + \langle g,h\rangle \tag{60}$$

$$\langle f,f\rangle = \sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j K(x_i,x_j) = A^T G_{\boldsymbol{x}} A \geq 0 \tag{61}$$

The following will give the proof that $\langle f,f\rangle = 0$ if and only if $f(\ \cdot\ ) = 0$. It is evident that $\mathcal{S}$ is a closure, then $\forall f,g\in\mathcal{S}, \lambda\in\mathbb{R}$, $f+\lambda g\in\mathcal{S}$. Then

$$\langle f+\lambda g, f+\lambda g\rangle = \langle f,f\rangle + 2\lambda\langle f,g\rangle + \lambda^2\langle g,g\rangle \geq 0$$

$$\Delta = 4\left(||\langle f,f\rangle||^2 - \langle f,f\rangle\langle g,g\rangle\right) \leq 0 \rightarrow ||\langle f,f\rangle||^2 - \langle f,f\rangle\langle g,g\rangle \leq 0 \tag{62}$$

then

$$\forall x\in\mathcal{X}, \langle K(\ \cdot\ ,x), f(\ \cdot\ )\rangle = f(x) \tag{63}$$

with the inequality above,

$$||\,\langle K(\ \cdot\ ,x), f(\ \cdot\ )\rangle\,||^2 \leq \langle K(\ \cdot\ ,x), K(\ \cdot\ ,x)\rangle\langle f(\ \cdot\ ), f(\ \cdot\ )\rangle \tag{64}$$