0.1 Basic Method

1 K-Nearest Neightbour Algorithm

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.

1.1 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

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

1.2 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.

1.3 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 = {}, {}
```

```python
    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)
```

1.4 Application: Handwriting Recognition

We use 32×32 martices to represent the handwritten image, where 1 stands for occupied pixel, and 0 stands for blanket. We have the dataset's dimension deduced to a 1×1024 vector and be compared with the trained data.

```
00000000000001111000000000000000 00000000011111111111100000000000 00000000000000000000000000000000
00000000000011111000000000000000 00000000011111111111110000000000 00000000000011110000000000000000
00000000000001111100000000000000 00000000111111111111111000000000 00000000001111110000000000000000
00000000000011111100000000000000 00000000111111111111111000000000 00000000011111111000000000000000
00000000000111111100000000000000 00000000111111111111111100000000 00000000111111111110000000000000
00000000000111111100000000000000 00000000011111111111111111000000 00000000111111111111100000000000
00000000000111111000000000000000 00000000000000000000111111000000 00000000111111111111110000000000
00000000000111110000000000000000 00000000000000000000111110000000 00000000111111111111111000000000
00000000000111110000000000000000 00000000000000000000111110000000 00000000111111101111111000000000
00000000001111110000000000000000 00000000000000000001111110000000 00000000111111100011111100000000
00000000000111110000000000000000 00000000000000000001111110000000 00000000111111100001111100000000
00000000001111100000000000000000 00000000000000000001111100000000 00000000011111100011111000000000
00000000001111110000000000000000 00000000000000000001111100000000 00000000011111101111110000000000
00000000011111100000000000000000 00000000000000000001111000000000 00000000011111111111000000000000
00000000011111000000000000000000 00000000011111111111100000000000 00000000000111111111100000000000
00000000011111100000000000000000 00000000111111111111000000000000 00000000000011111111000000000000
00000000011111111000000000000000 00000000111111111111110000000000 00000000000001111110000000000000
00000000011111111110000000000000 00000000111111111111111000000000 00000000000001111110000000000000
00000000011111111111000000000000 00000001111111111111111000000000 00000000000001111110000000000000
00000000011111111111100000000000 00000000111111111111111100000000 00000000000001111110000000000000
00000000011111111111110000000000 00000000000001111111111100000000 00000000001111111110000000000000
00000000011111111111111000000000 00000000000001111111111000000000 00000000111111011111100000000000
00000000011111101111111111000000 00000000000011111000111000000000 00000001111111000111110000000000
00000000011111110000000111000000 00000000000111110000000000000000 00000001111110000011100000000000
00000000011111110000000111100000 00000000000111100000000000000000 00000001111110000011110000000000
00000000000111110000000111100000 00000000000111100000000000000000 00000001111110000011111000000000
00000000000111110000011111100000 00000000011111000000000000000000 00000001111111000011111000000000
00000000000001111100111111111000 00000000011111000000000000000000 00000000011111001111111000000000
00000000000001111111111111110000 00000000011111000000000000000000 00000000001111111111110000000000
00000000000000111111111111111000 00000000011110000000000000000000 00000000001111111111100000000000
00000000000000011111111111110000 00000000011110000000000000000000 00000000000111111100000000000000
00000000000000000111111111100000 00000000011100000000000000000000 00000000000000000000000000000000
```

Listing 2: **HWR.py**

```python
from os import listdir
from K_NN import *

def imageToVec(filename):
     file =  open(filename)
    returnVec = np.zeros([1, 1024], dtype =  int)
```

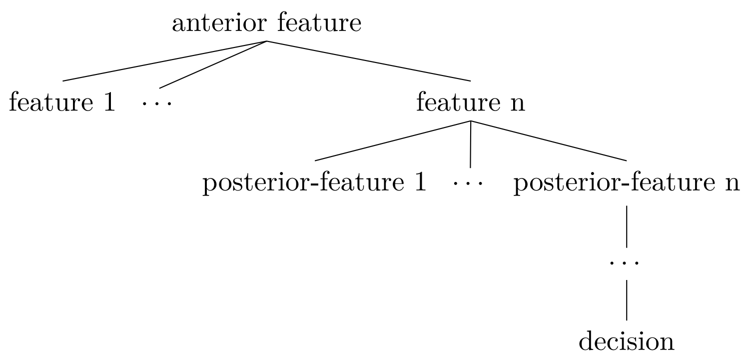
```
    for i in  range(32):
        lineStr =  file.readline()
        for j in  range(32):
            returnVec[0, i * 32 + j] =  int(lineStr[j])
    return returnVec

def handWritingRecognition(filename, dataDir): #The training dataset is stored in ./trainingDigits
    sampleVec = imageToVec(filename)
    trainingFileList = listdir(dataDir)
    listLength =  len(trainingFileList)
    trainingMat = np.zeros([listLength, 1024])
    labelVec = []
    for i in  range(listLength):
        fileName = trainingFileList[i]
        fileStr = fileName.split('.')[0]
        fileClass =  int(fileStr.split('_')[0])
        fileVec = imageToVec('trainingDigits/{}'. format(fileName))
        trainingMat[i,...] = fileVec
        labelVec.append(fileClass)
    return K_NN(sampleVec,  int( input()), trainingMat, labelVec)
```

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

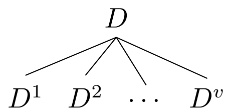
$$\mathrm{Ent}(D) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{4}$$

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

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

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 $\forall D_i D_j \in D, i \neq j,\ D_i = D_j$ **then**
  $node = leafnode,\ node \leftarrow D.y$
  return
end if
if $A = \emptyset$ **or** $\forall a_i, a_j \in A, i \neq j,\ \mathrm{Gain}(D, a_i) = \mathrm{Gain}(D, a_j)$ **then**
  $node = leafnode,\ node \leftarrow \arg\max_y(|N|,\ N = \{y|(\boldsymbol{x}, y) \in D\})$
end if
$a_* \leftarrow \arg\max_{a\in A} \mathrm{Gain}(D, a)$
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
    $node.branch^v = \mathrm{treeGenerate}(D_v,\ A - \{a_*\})$
  end if
end for

4 Code

Listing 3: **decisionTree.py**

```python
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
    for label in labelCount:
        probability = labelCount[label] / entriesNum
        shannonEntropy -= probability * log(probability, 2)
    return shannonEntropy

def splitDataset(dataset, index, expectValue): # Search the dataset with specified index and return the reduced dataset
    retDataset = []
    for lineVec in dataset:
        if lineVec[index] == expectValue:
            reducedVec = lineVec[:index]
            reducedVec += lineVec[index+1:]
            retDataset.append(reducedVec)
    return retDataset

def optimalPartition(dataset): # ID3 Algotihm
    featureNums =  len(dataset[0]) - 1
    originalEntropy = shannonEntropy(dataset)
    bestFeature = -1
    maxInfoGain = 0.00
    for featureIndex in  range(featureNums):
        labelVec =  set([dataset[i][featureIndex] for i in  range( len(dataset))])
        extraEntropy = 0.00
        for label in labelVec:
            reducedSet = splitDataset(dataset, featureIndex, label)
            extraEntropy +=  len(reducedSet) /  float( len(dataset)) * shannonEntropy(reducedSet)
        if originalEntropy - extraEntropy > maxInfoGain:
            maxInfoGain = originalEntropy - extraEntropy
            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()}
```

```python
    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
```

5 Naive Bayes Algorithm

5.1 Basic 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

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

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