1 K-Nearest Neightbour Algorithm

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

$array \leftarrow s_i | H[s_i]$ 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

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

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

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

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)

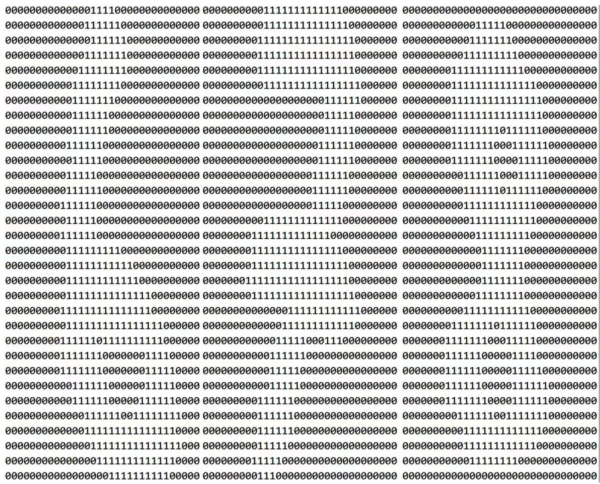
Listing 1: **K-NN.py**

```
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 = {}, {}
    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:
```

```python
        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.



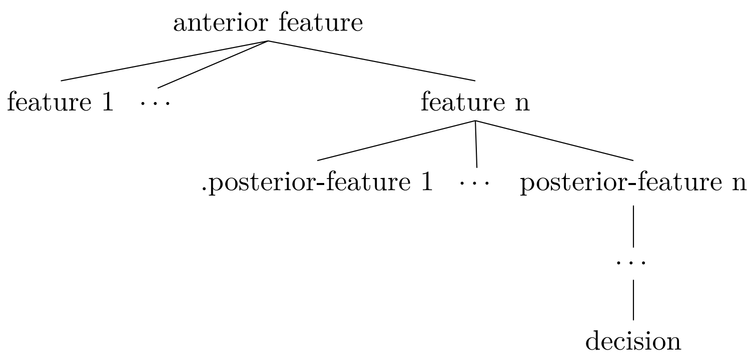
Listing 2: **HWR.py**

```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

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, and each leaf node denotes a final decision. 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.



2.1 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

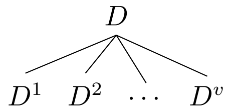
$$\mathrm{Ent}(\boldsymbol{D}) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{3}$$

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

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

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

$$\mathrm{Gain}(D, a) = \mathrm{Ent}(D) - \sum_{i=1}^{v}\frac{|D^i|}{|D|}\mathrm{Ent}(D^i) \tag{5}$$

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

$$a_* = \arg\max_{a\in A} \mathrm{Gain}(D, a) \tag{6}$$

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

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

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

```

```python
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()}
    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
```