**Implementing Huffman coding and K-means Algorithm**

NAME :MIZANUR RAHMAN PRANTO

**Software Engineering**

**JUNE 30 ,2019**

**SICHUAN UNIVERSITY**

**Contents**

1. **Acknowledgement**
2. **Introduction**
3. **K-means Clustering Algorithm**
4. **Huffman Coding**
5. **Implementing Huffman tree**
6. **Experimental result**
7. **Conclusion**
8. **Teacher Evaluation**

**Acknowledgment**

Before producing this project, I take this opportunity to express my sincere gratitude to everyone who offered me a great patronage for the accomplishment of this Project. This Project would not have been imaginable without the support of many people. Thank you for everyone who supported me to accomplish this project.

**Introduction**

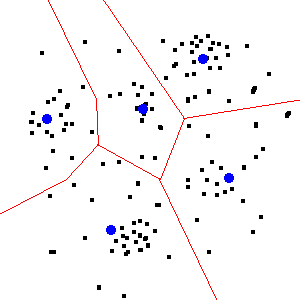
I have made this project in C++ programming language using some elementary concepts.

* Read 300 input data
* Divide this input data into 8 clusters by K-means Algorithm
* Compress this group centers by Huffman coding
* Replace step 2 by Principal Component Algorithm

**K-means Clustering Algorithm**

K-means clustering is a method of classifying/grouping items into k groups (where k is the number of pre-chosen groups). The grouping is done by minimizing the sum of squared distances (Euclidean distances) between items and the corresponding centroid.

A centroid is "The centre of mass of a geometric object of uniform density", though here; we'll consider mean vectors as centroids.



A clustered scatter plot. The black dots are data points. The red lines illustrate the partitions created by the k-means algorithm. The blue dots represent the centroids which define the partitions.

* For each point place it in the cluster whose current centroid it is in nearest
* After all points are assigned, update the locations of centroids of the k clusters
* Reassign all points to their closest centroid sometimes moves points between clusters
* Repeat 2 and 3 until convergence

**Convergence**: points don’t move between clusters and centroids stabilize

* In each round we have to examine each input point exactly once to find closest centroid
* Each round is o(kN) for N points, k clusters
* But the number of rounds to convergence can be very large!

**Huffman coding**

 A Huffman code is a particular type of optimal [prefix code](https://en.wikipedia.org/wiki/Prefix_code" \o "Prefix code) that is commonly used for [lossless data compression](https://en.wikipedia.org/wiki/Lossless_data_compression" \o "Lossless data compression).

Huffman coding is based on the frequency of occurrence of a pixel in images. The principle is to use a lower number of bits to encode the data that occurs more frequently.

The Huffman algorithm is now briefly summarised:

* A bottom-up approach

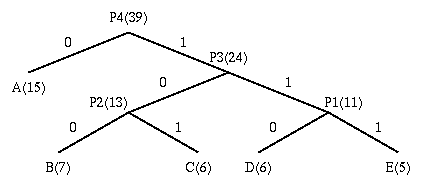
1. Initialization: Put all nodes in an OPEN list, keep it sorted at all times (e.g., ABCDE).

2. Repeat until the OPEN list has only one node left:

(a) From OPEN pick two nodes having the lowest frequencies/probabilities, create a parent node of them.

(b) Assign the sum of the children's frequencies/probabilities to the parent node and insert it into OPEN.

(c) Assign code 0, 1 to the two branches of the tree, and delete the children from OPEN.



Symbol Count log (1/p) Code Subtotal (# of bits)

------ ----- ------- ------ ---------------

A 15 1.38 0 15

B 7 2.48 100 21

C 6 2.70 101 18

D 6 2.70 110 18

E 5 2.96 111 15

TOTAL (# of bits): 87

The following points are worth noting about the above algorithm:

* Decoding for the above two algorithms is trivial as long as the coding table (the statistics) is sent before the data. (There is a bit overhead for sending this, negligible if the data file is big.)
* **Unique Prefix Property**: no code is a prefix to any other code (all symbols are at the leaf nodes) > great for decoder, unambiguous.
* If prior statistics are available and accurate, then Huffman coding is very good.

In the above example:

Number of bits needed for Huffman Coding is: 87 / 39 = 2.23

**Implementing Huffman tree**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **E 2** | **D 3** | **B 4** | **A 5** | **C 6** |

|  |  |
| --- | --- |
| **Character** | **Frequency** |
| A | 5 |
| B | 4 |
| C | 6 |
| D | 3 |
| E | 2 |

**2 3 4 5 5 6 9 11 20**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **E** | **D** | **B** | **A** | **E/D** | **C** | **B/A** | **ED/C** | **B/A/E/D/C** |

**A B C D E E/D B/A E/D/C B/A/E/D/C**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **6 -1 -1** | **6 -1 -1** | **7 -1 -1** | **5 -1 -1** | **5 -1 -1** | **7 4 3** | **8 1 0** | **8 5 2** | **-1 6 7** |

**0 1 2 3 4 5 6 7 8**

**5 9 11 20**

B/A/E/D/C/

E/D/C

B/A

E/D

E/D/C

B/A

C

E/D

A

B

D

E

**B/A/E/D/C**

**0 1**

**B/A**

**E/D/C**

**0 1 0 1**

**A**

**B**

**E/D**

**C**

**0 1**

**D**

**E**

|  |  |
| --- | --- |
| **Character** | **Binary code** |
| **A** | 11 |
| **B** | 10 |
| **C** | 00 |
| **D** | 011 |
| **E** | 010 |

**Experimental Result**

Data from the text file “UniformData.txt” and put that data into a vector.

Each pair of coordinates is separated by a space in the file, these two coordinate strings are read as float values and added to the vector as a pair.

void readData(std::vector<std::pair<float, float>>& voData)

{

FILE \*fp = fopen(g\_FileName.c\_str(),"r");

int pairNum;

fscanf(fp,"%d\n", &pairNum);

for(int i=0;i<pairNum;i++)

{

float weight,height;

fscanf(fp,"%f %f\n",&weight, &height);

voData.push\_back(std::make\_pair(weight,height));

}

}

K-Means algorithm.

1. Initialize Cluster Centers
2. Update Cluster Indexes
3. Update Cluster Centers

Step 2 and 3 is repeated until the error difference is lower than the threshold.

void executeKMeans(std::vector<std::pair<float, float>>& vInputData, std::vector<std::pair<float, float>>& voClusterCenter, std::vector<int>& voClusterIndex)

{

initClusterCenter(vInputData, voClusterCenter);

voClusterIndex.resize(vInputData.size());

float LastError = FLT\_MAX;

float CurrentError;

int NumIteration = 0;

while (NumIteration < 100)

{

updateClusterIndex(vInputData, voClusterCenter, voClusterIndex);

updateClusterCenter(vInputData, voClusterIndex, voClusterCenter);

CurrentError = computeError(vInputData, voClusterIndex, voClusterCenter);

if (fabs(CurrentError - LastError) < 0.0001) break;

LastError = CurrentError;

NumIteration++;

}

}

Huffman coding algorithm.

1. Generate Frequency Table
2. Generate Huffman Tree
3. Generate Huffman codes and Generate the output bit stream

Generate Frequency Table

Frequency of each cluster center is calculated by iterating through all the input data. This frequency is then added to the code table along with the original cluster center coordinates.

void generateFrequencyTable(const std::vector<int>& vInput, std::vector<CodeTableElement>& voCodeTable)

{

for(int i = 0; i < vInput.size(); i++)

{

int j;

for(j = 0; j < voCodeTable.size(); j++)

{

if(voCodeTable[j].Data == vInput[i])

{

voCodeTable[j].Frequency++;

break;

}

}

if (j == voCodeTable.size())

{

CodeTableElement t;

t.Data = vInput[i];

t.Frequency = 1;

voCodeTable.push\_back(t);

}

}

\_ASSERT(voCodeTable.size() == g\_NumCluster);

int Total = 0;

for (int i=0; i<voCodeTable.size(); i++) Total += voCodeTable[i].Frequency;

\_ASSERT(Total == vInput.size());

}

Initializing the priority queue

A priority queue holding the tree nodes is generated here. Each tree node contains its index and the frequency along with the references to its children and parent. These sorted nodes then passed to the Huffman tree vector.

void initPriorityQueueAndTree(std::vector<CodeTableElement>& vCodeTable, std::priority\_queue<QueueElement>& voPriorityQueue, std::vector<HuffmanTreeNode>& voHuffmanTree)

{

QueueElement q;

HuffmanTreeNode t;

for (int i=0; i<vCodeTable.size(); i++)

{

q.Frequency = vCodeTable[i].Frequency;

q.NodeIndex = i;

voPriorityQueue.push(q);

t.Left = t.Right = t.Parent = -1;

t.NodeIndex = i;

voHuffmanTree.push\_back(t);

}

}

Generate Huffman Tree

The least two elements are popped out from the priority queue. The frequencies of these are added and put into the Huffman tree and the priority queue as a new node. This node is referenced as the parent of the first two elements and those are referenced as children of the new node.

void generateHuffmanTree(std::priority\_queue<QueueElement>& vPriorityQueue, std::vector<HuffmanTreeNode>& voHuffmanTree)

{

QueueElement Left, Right, t;

HuffmanTreeNode Node;

while (vPriorityQueue.size() > 1)

{

Left=vPriorityQueue.top();

vPriorityQueue.pop();

Right=vPriorityQueue.top();

vPriorityQueue.pop();

t.Frequency= Left.Frequency+Right.Frequency;

t.NodeIndex=voHuffmanTree.size();

vPriorityQueue.push(t);

Node.Left = Left.NodeIndex;

Node.Right = Right.NodeIndex;

Node.Parent = -1;

Node.NodeIndex = t.NodeIndex;

voHuffmanTree[Left.NodeIndex].Parent = Node.NodeIndex;

voHuffmanTree[Right.NodeIndex].Parent = Node.NodeIndex;

voHuffmanTree.push\_back(Node);

}

}

Generate Huffman Code

The path to the root of the Huffman tree from each leaf is generated, so that each left child is represented as 0 and right child as 1. These codes are then reversed so that they can be followed from the root to the leaf later.

void generateHuffmanCode(std::vector<HuffmanTreeNode>& vHuffmanTree, std::vector<CodeTableElement>& voCodeTable)

{

for (int i = 0; i < 8; i++)

{

std::vector<bool> ReverseCode;

HuffmanTreeNode Node = vHuffmanTree[i];

HuffmanTreeNode ParentNode;

while (Node.Parent != -1)

{

ParentNode = vHuffmanTree[Node.Parent];

ReverseCode.push\_back(ParentNode.Left == Node.NodeIndex);

Node = ParentNode;

}

for (int j = ReverseCode.size() - 1; j >= 0; j--)

{

voCodeTable[i].Code.push\_back(ReverseCode[j]);

}

}

}

**Clustering coding**

void generateEncodedBitStream(const std::vector<int>& vInputData, std::vector<CodeTableElement>& vCodeTable, std::vector<bool>& voOutput)

{

for (int i = 0; i < vInputData.size(); i++)

{

for (int j = 0; j < 8; j++)

if (vInputData[i] == vCodeTable[j].Data)

for (bool b : vCodeTable[j].Code)

voOutput.push\_back(b);

}

}

User defined structures

Huffman Tree Node

struct HuffmanTreeNode

{

int NodeIndex, Left, Right, Parent;

};

queue Element – THis is needed to compare two elements in the QUEUE

struct QueueElement

{

int Frequency;

int NodeIndex;

friend bool operator<(const QueueElement& vLeft, const QueueElement& vRight) { return vLeft.Frequency > vRight.Frequency; }

};

code table element

struct CodeTableElement

{

int Data;

int Frequency;

std::vector<bool> Code;

};

<vector>, <string> and <queue> libraries are imported at the beginning of the code.

The final output :   
[100010010011111000110100111111000000011001100111101001111010000110010101100111011110  
0110011111101011001101111011101011000110101010110011111111000101011000001100011011111101  
00011100111000110101000001011110111110000101000101001111101010011011001101011111100111  
001101101001111011111111111001010111111101000011000111011110111011000001011011101001011111  
110010100110000011010100000011010111111000111000001101110110110101101000000000101  
00000101101101000110110001011010111110010010100111101111011101110101110101001100111011  
01101111110110100011010110111111000111100000111111100011111111011111111101101111011011100010  
1111101011011010010111110010101101110100000001110011110100100100111011111011111100111101  
0000011110010111000010011101111001110001110000110101100100110111001000011000010100  
011110100010010111110111111101100001101101100111001101101000111100011110100011010110010  
00001001]

**Conclusion**

* K-means Clustering and Huffman coding has been implemented successfully and successfully divided 300 input data into 8 clusters and compressed this clusters by using K-means Algorithm and Huffman coding.

**Teacher Evaluation**

* Grade :
* Teacher’s Signature :