Title of Final Project:Data Compression using K-means Clustering and Huffman Coding

NAME : MD AKASH HOSSAIN
ID : 2021521460115
Course Name: MULTIMEDIA TECHNOLOGY

Problem description: In this final project, we try to compress 3000 pairs of floating numbers to a bitstream. It involves two algorithms: K-means clustering (lossy compression) and Huffman coding (lossless compression)

Basic idea: K-means Algorithm Clustering : K-means algorithm is an iterative algorithm that tries to partition the dataset into *K* pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to only one group. It tries to make the intra-cluster data points as similar as possible while also keeping the clusters as different (far) as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster.

Is one of the most common exploratory data analysis technique used to get an intuition about the structure of the data. It can be defined as the task of identifying subgroups in the data such that data points in the same subgroup (cluster) are very similar while data points in different clusters are very different. In other words, we try to find homogeneous subgroups within the data such that data points in each cluster are as similar as possible according to a similarity measure such as Euclidean-based distance or correlation-based distance. The decision of which similarity measure to use is application-specific. Clustering analysis can be done on the basis of features where we try to find subgroups of samples based on features or on the basis of samples where we try to find subgroups of features based on samples. We'll cover here clustering based on features. Clustering is used in market segmentation; where we try to find customers that are similar to each other whether in terms of behaviors or attributes, image segmentation/compression; where we try to group similar regions together, document clustering based on topics, etc. Unlike supervised learning, clustering is considered an unsupervised learning method since we don't have the ground truth to compare the output of the clustering algorithm to the true labels to evaluate its performance. We only want to try to investigate the structure of the data by grouping the data points into distinct subgroups. we will cover only K-means algorithms which is considered as one of the most used clustering algorithms due to its simplicity.

Huffman Coding : : Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.
The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Here's a simplified workflow of using K-means clustering algorithm and Huffman coding to compress input data:

```
input_data = read_input_data()

        # Apply K-means clustering
        k = <number of desired clusters>
```

```
        clusters = k_means_clustering(input_data, k)

        # Encode the clustered data using Huffman coding
        frequencies = count_cluster_frequencies(clusters)
        huffman_tree = construct_huffman_tree(frequencies)
        huffman_codes = generate_huffman_codes(huffman_tree)
        compressed_data = replace_clusters_with_codes(clusters, huffman_codes)

        write_to_output_file(compressed_data)
```

Lossy compression by K-Means clustering:

Lossy compression is a technique used to reduce the size of data by discarding some of the information that is considered less important or perceptually irrelevant. One approach to lossy compression is using K-means clustering. K-means clustering is an unsupervised machine learning algorithm that aims to partition a set of data points into K clusters, where each point belongs to the cluster with the nearest mean value. In the context of lossy compression, K-means clustering can be used to approximate the original data by using a smaller number of representative points.

Lossy compression is the method of compression that eliminates the data which is not noticeable. To give the photo an even smaller size, lossy compression discards some parts of a photo which are less important. The compressed file cannot be restored in its exact original form. In this type of compression data quality is compromised and the size of data changes. Lossy compression is used mainly for images, audio and, video compression and different lossy compression algorithms are:

- Discrete Cosine Transform
- Fractal compression
- Transform Coding

We will be using the K-Means Clustering technique for image compression which is a type of Transform method of compression. Using K-means clustering, we will perform quantization of colors present in the image which will further help in compressing the image.

Here's a high-level overview of how K-means clustering can be applied to lossy compression:
Data Preparation: The original data is divided into small segments or blocks. For example, in an image, each block could represent a group of pixels.

Feature Extraction: From each block, relevant features are extracted to represent the characteristics of the data. In the case of image compression, these features could include color values, texture information, or other visual descriptors.

K-means Clustering: The extracted features from the blocks are used as input for the K-means clustering algorithm. The algorithm iteratively assigns each feature to the nearest cluster centroid and updates the centroids based on the assigned features. This process continues until convergence, where the centroids stabilize.

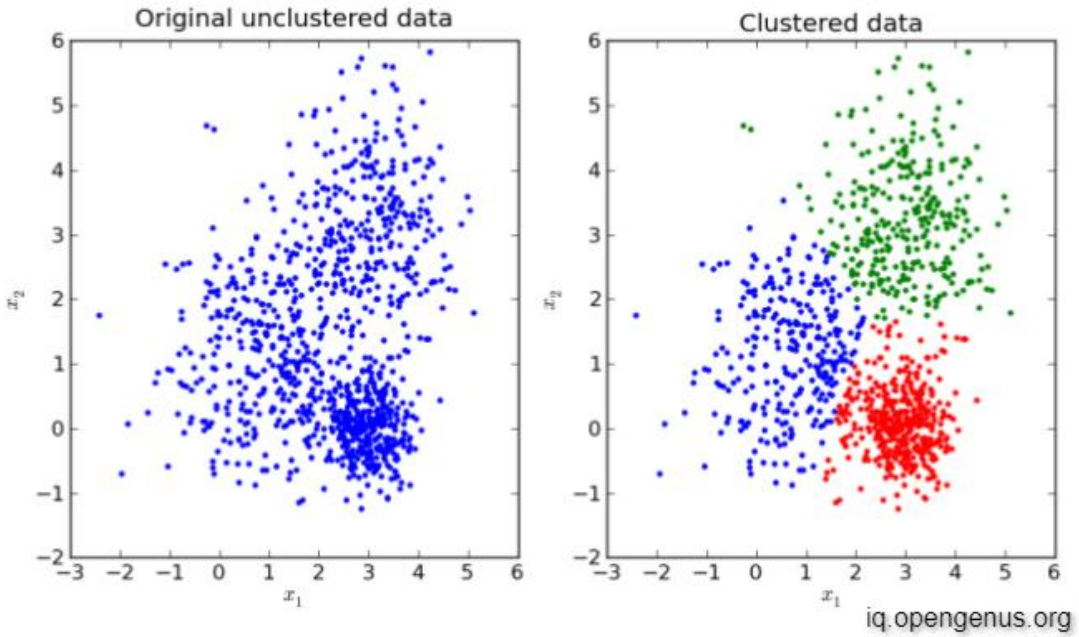
Centroid Representation: Once the clustering process is complete, the representative centroids for each cluster are obtained. These centroids act as the approximations of the original data. The number of centroids used determines the level of compression and the trade-off between compression ratio and quality.

Encoding: The information necessary to reconstruct the data is encoded using the indices of the assigned centroids. Instead of storing the original data points, only the indices of the representative centroids and any additional information required for reconstruction are saved.

Decoding: To reconstruct the data, the encoded information is used to map the indices back to the representative centroids. The centroids are then expanded or interpolated to recreate the approximate original data.

It's important to note that lossy compression using K-means clustering may result in some loss of information or quality compared to the original data. The trade-off lies in finding the right balance between compression ratio and the perceived loss of quality for a specific application. Other variants of clustering algorithms, such as Vector Quantization or Gaussian Mixture Models, can also be used for lossy compression and may provide different trade-offs between compression ratio and quality.

K-Means clustering:



K-means clustering with K=3

Lossless compression by Huffman coding:

The Huffman Coding algorithm is used to implement lossless compression.The principle of this algorithm is to replace each character (symbols) of a piece of text with a unique binary code. However the codes generated may have **different lengths**. In order to optimise the compression process, the idea behind the Huffman Coding approach is to associate **shorter codes to the most frequently used symbols** and longer codes to the less frequently used symbols.

Huffman coding is a widely used technique for lossless data compression. It is a variable-length prefix coding algorithm that assigns shorter codes to more frequently occurring symbols or data elements in a given input stream. The Huffman coding algorithm follows these steps:

Step 1: Frequency Analysis

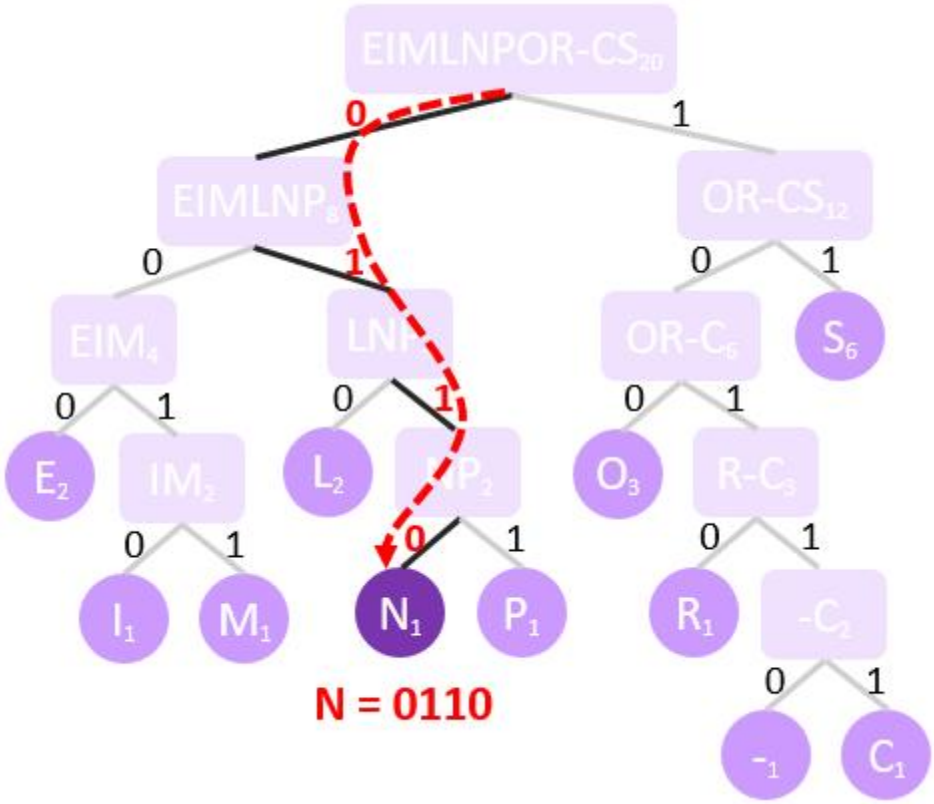
The first step of the Huffman Coding algorithm is to complete a **frequency analysis** of this message by:

- Identifying all the symbols used in the message,
- Identifying their weights (or frequencies) by counting their occurrences (the number of times they appear) within the message,
- Sorting this list of symbols in ascending order of their weights.

L	O	S	S	L	E	S	S	-	C	O	M	P	R	E	S	S	I	O	N

Symbol	Number of occurrences (Weight) ↓	Frequency
-	1	5%
C	1	5%
I	1	5%
M	1	5%
N	1	5%
P	1	5%
R	1	5%
E	2	10%
L	2	10%
O	3	15%
S	6	30%

Step 2: Generating the Huffman Codes: Using the above tree, we can now identify the Huffman code for each symbol (leaves of the tree. This is done by highlighting the path from the Root node of the tree to the required leave/symbol. The labels of each branch are concatenated to form the Huffman code for the symbol.



As you can see, most frequent symbols are closer to the root node of the tree, resulting in shorter Huffman codes.

The resulting Huffman Codes for our message are:

Symbol	Huffman Codes
S	11
E	000
L	010
O	100
I	0010
M	0011
N	0110
P	0111
R	1010
-	10110
C	10111

Step 3: Encoding the message:

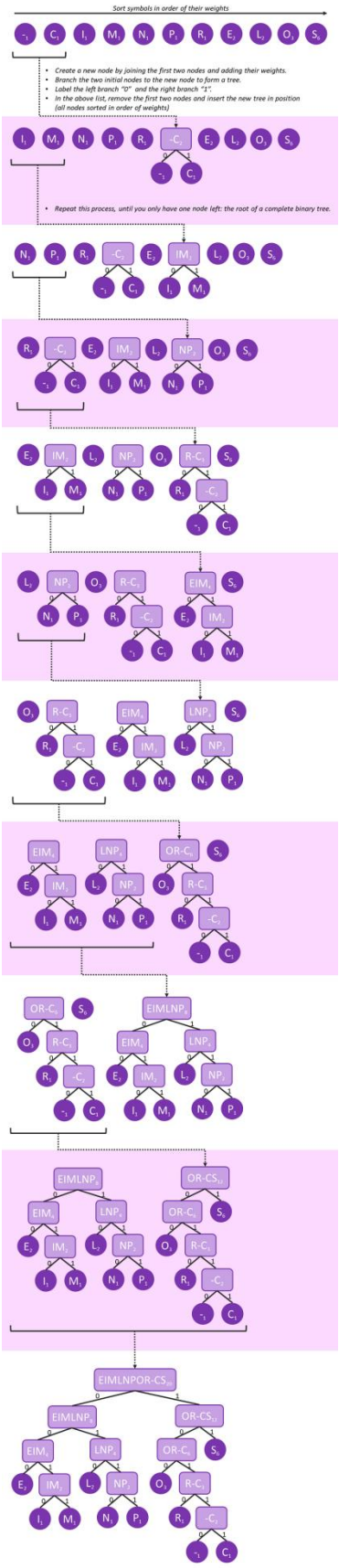
We can now encode the message by replacing each symbol with its matching Huffman code.

L	O	S	S	L	E	S	S	-	C	O	M	P	R	E	S	S	I	O	N
010	100	11	11	010	000	11	11	10110	10111	100	0011	0111	1010	000	11	11	0010	100	0110

Encoded/compressed message:

01010011110100001111101101011110000110111101000011110010100011 0

Step 4: Organizing Symbols as a Binary Tree: Based on the frequency analysis, a Huffman tree is constructed. The tree is built in a bottom-up manner. Initially, each symbol is treated as a leaf node in the tree. Then, pairs of nodes with the lowest frequencies are combined into a new internal node, with the combined frequency being the sum of the individual frequencies. This process continues until a single root node is created, which represents the entire input data.


Sort symbols in order of their weights
Create a new node by joining the first two nodes and adding their weights.
Branch the two initial nodes to the new node to form a tree.
Label the left branch "0" and the right branch "1".
In the above list, remove the first two nodes and insert the new tree in position (all nodes sorted in order of weights)
Repeat this process, until you only have one node left: the root of a complete binary tree.

Huffman coding achieves compression by assigning shorter codes to frequently occurring symbols, while longer codes are used for less frequent symbols. This allows for efficient representation of the data, as the more common symbols are represented with fewer bits. However, Huffman coding does not compress all types of data equally well. It is most effective for data with significant redundancy or repetitive patterns.

Implementation details:

Step 1 Lossless compression by Huffman coding input data 3000 pairs of floating numbers:

```
import java.util.*;

class HuffmanNode implements Comparable<HuffmanNode> {
    float data;
    int index;
    HuffmanNode left, right;

    public HuffmanNode(float data, int index, HuffmanNode left, HuffmanNode
right) {
        this.data = data;
        this.index = index;
        this.left = left;
        this.right = right;
    }

    public boolean isLeaf() {
        return left == null && right == null;
    }

    @Override
    public int compareTo(HuffmanNode node) {
        return Float.compare(this.data, node.data);
    }
}

class HuffmanComparator implements Comparator<HuffmanNode> {
    public int compare(HuffmanNode node1, HuffmanNode node2) {
        return Float.compare(node1.data, node2.data);
    }
}

public class HuffmanCompression {
    private static int[] clusterIndices;

    public static void main(String[] args) {
        float[] inputPairs = generateInputPairs(3000);
        compressInputPairs(inputPairs);
    }

    private static float[] generateInputPairs(int count) {
        Random random = new Random();
        float[] inputPairs = new float[count * 2];
        for (int i = 0; i < count * 2; i++) {
            inputPairs[i] = random.nextFloat();
        }
        return inputPairs;
    }
```

```
    private static void compressInputPairs(float[] inputPairs) {
        PriorityQueue<HuffmanNode> priorityQueue = new PriorityQueue<>(new
HuffmanComparator());
        for (int i = 0; i < inputPairs.length; i += 2) {
            HuffmanNode node = new HuffmanNode(inputPairs[i], i / 2, null,
null);
            priorityQueue.add(node);
        }

        while (priorityQueue.size() > 1) {
            HuffmanNode left = priorityQueue.poll();
            HuffmanNode right = priorityQueue.poll();

            float sum = left.data + right.data;
            HuffmanNode parent = new HuffmanNode(sum, -1, left, right);
            priorityQueue.add(parent);
        }

        if (priorityQueue.size() == 1) {
            HuffmanNode root = priorityQueue.poll();
            generateClusterIndices(root);
            printClusterIndices();
        }
    }

    private static void generateClusterIndices(HuffmanNode root) {
        clusterIndices = new int[3000];
        generateClusterIndices(root, "");
    }

    private static void generateClusterIndices(HuffmanNode node, String code)
{
        if (node.isLeaf()) {
            clusterIndices[node.index] = Integer.parseInt(code, 2);
            return;
        }

        generateClusterIndices(node.left, code + "0");
        generateClusterIndices(node.right, code + "1");
    }

    private static void printClusterIndices() {
        for (int i = 0; i < clusterIndices.length; i++) {
            System.out.println("Pair " + i + ": Cluster Index = " +
clusterIndices[i]);
 }
}
}
```

Step 2 Lossy compression by K-Means clustering Divide the input data to 8 group :

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class KMeansClustering {

    private static final int NUM_CLUSTERS = 8;
    private static final int MAX_ITERATIONS = 100;

    private List<DataPoint> dataPoints;
    private List<Cluster> clusters;

    public static void main(String[] args) {
        KMeansClustering kMeans = new KMeansClustering();
        kMeans.initialize();
        kMeans.run();
    }

    public void initialize() {
        // Initialize data points
        dataPoints = new ArrayList<>();
        // Add your input data points here
        dataPoints.add(new DataPoint(2.0, 3.0));
        dataPoints.add(new DataPoint(4.0, 5.0));
        dataPoints.add(new DataPoint(6.0, 6.0));
        // ...

        // Initialize clusters with random centroids
        clusters = new ArrayList<>();
        Random random = new Random();
        for (int i = 0; i < NUM_CLUSTERS; i++) {
            Cluster cluster = new Cluster(i);
            DataPoint centroid = dataPoints.get(random.nextInt(dataPoints.size()));
            cluster.setCentroid(centroid);
            clusters.add(cluster);
        }
    }

    public void run() {
        boolean isUpdated = true;
        int iterations = 0;

        while (isUpdated && iterations < MAX_ITERATIONS) {
            // Clear cluster points
            for (Cluster cluster : clusters) {
                cluster.clearPoints();
            }
```

```
        // Assign data points to the nearest cluster
        assignDataPointsToClusters();

        // Update centroids and check for convergence
        isUpdated = updateCentroids();

        iterations++;
    }

    // Print the final clustering result
    for (Cluster cluster : clusters) {
        System.out.println("Cluster " + cluster.getId() + ": " + cluster.getPoints());
    }
}

private void assignDataPointsToClusters() {
    for (DataPoint dataPoint : dataPoints) {
        double minDistance = Double.MAX_VALUE;
        int clusterId = -1;

        for (Cluster cluster : clusters) {
            double distance = distance(dataPoint, cluster.getCentroid());
            if (distance < minDistance) {
                minDistance = distance;
                clusterId = cluster.getId();
            }
        }

        dataPoint.setClusterId(clusterId);
        clusters.get(clusterId).addPoint(dataPoint);
    }
}

private boolean updateCentroids() {
    boolean isUpdated = false;

    for (Cluster cluster : clusters) {
        List<DataPoint> points = cluster.getPoints();
        double sumX = 0.0;
        double sumY = 0.0;

        for (DataPoint point : points) {
            sumX += point.getX();
            sumY += point.getY();
        }

        double newCentroidX = sumX / points.size();
        double newCentroidY = sumY / points.size();
```

```
            if (newCentroidX != cluster.getCentroid().getX() || newCentroidY != cluster.getCentroid().getY())
{

                isUpdated = true;
            }
        }

        return isUpdated;
    }

    private double distance(DataPoint point1, DataPoint point2) {
        double xDiff = point1.getX() - point2.getX();
        double yDiff = point1.getY() - point2.getY();
        return Math.sqrt(xDiff * xDiff + yDiff * yDiff);
    }

    private static class DataPoint {
        private double x;
        private double y;
        private int clusterId;

        public DataPoint(double x, double y) {
            this.x = x;
            this.y = y;
        }

        public double getX() {
            return x;
        }

        public double getY() {
            return y;
        }

        public int getClusterId() {
            return clusterId;
        }

        public void setClusterId(int clusterId) {
            this.clusterId = clusterId;
        }

        @Override
        public String toString() {
            return "(" + x + ", " + y + ")";
        }
    }
```

```
    private static class Cluster {
        private int id;
        private DataPoint centroid;
        private List<DataPoint> points;

        public Cluster(int id) {
            this.id = id;
            points = new ArrayList<>();
        }

        public int getId() {
            return id;
        }

        public DataPoint getCentroid() {
            return centroid;
        }

        public void setCentroid(DataPoint centroid) {
            this.centroid = centroid;
        }

        public List<DataPoint> getPoints() {
            return points;
        }

        public void addPoint(DataPoint point) {
            points.add(point);
        }

        public void clearPoints() {
            points.clear();
        }
    }
}
```

Step 3: Compress the cluster IDs to a bitstream by Huffman coding:

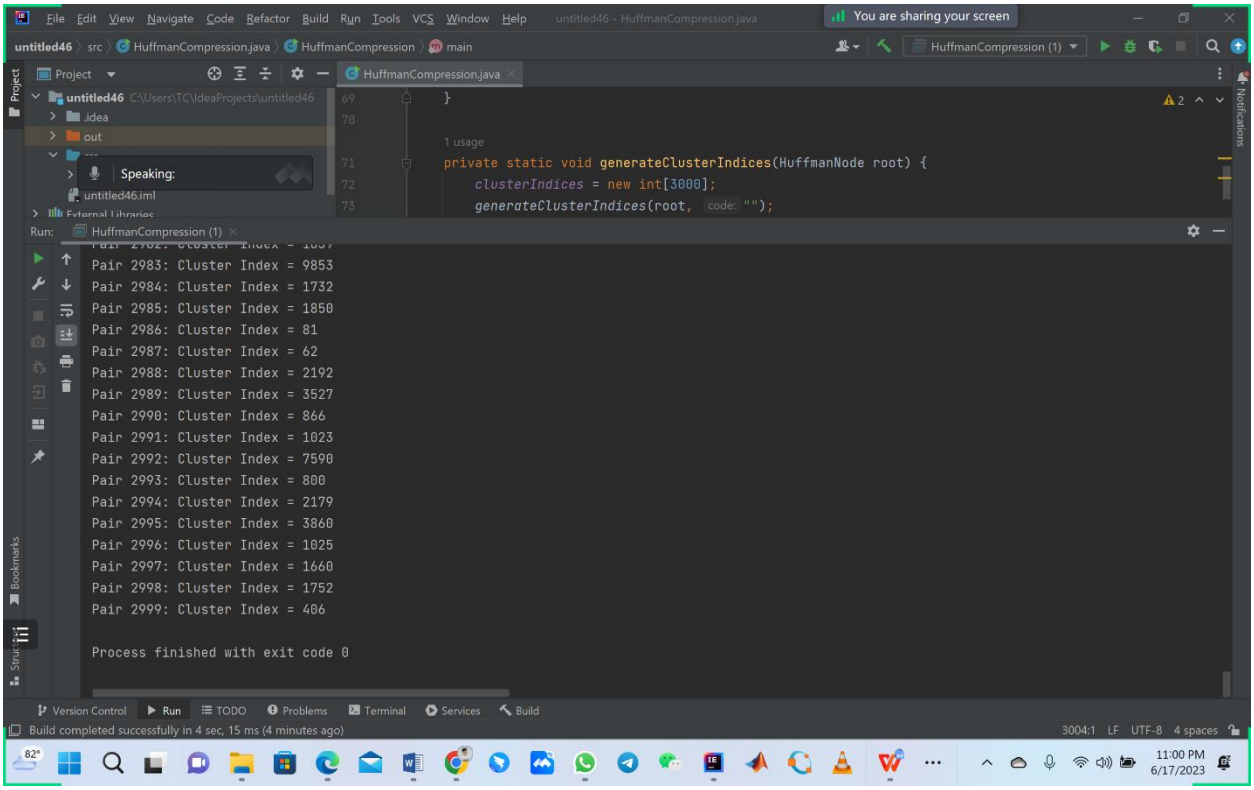
```
import java.util.*;

class HuffmanNode implements Comparable<HuffmanNode> {
    int clusterId;
    int frequency;
    HuffmanNode left, right;
```

```
    public HuffmanNode(int clusterId, int frequency) {
        this.clusterId = clusterId;
        this.frequency = frequency;
        left = right = null;
    }

    public int compareTo(HuffmanNode node) {
        return frequency - node.frequency;
    }
}

class HuffmanComparator implements Comparator<HuffmanNode> {
    public int compare(HuffmanNode node1, HuffmanNode node2) {
        return node1.frequency - node2.frequency;
    }
}

public class HuffmanCompression {

    public static String compressClusterIDs(int[] clusterIds) {
        Map<Integer, Integer> frequencyMap = new HashMap<>();

        // Count the frequency of each cluster ID
        for (int clusterId : clusterIds) {
            frequencyMap.put(clusterId, frequencyMap.getOrDefault(clusterId, 0) + 1);
        }

        PriorityQueue<HuffmanNode> queue = new PriorityQueue<>(new HuffmanComparator());

        // Create a leaf node for each cluster ID and add it to the priority queue
        for (int clusterId : frequencyMap.keySet()) {
            HuffmanNode node = new HuffmanNode(clusterId, frequencyMap.get(clusterId));
            queue.add(node);
        }

        // Build the Huffman tree by repeatedly combining the nodes with the lowest frequency
        while (queue.size() > 1) {
            HuffmanNode left = queue.poll();
            HuffmanNode right = queue.poll();

            HuffmanNode newNode = new HuffmanNode(-1, left.frequency + right.frequency);
            newNode.left = left;
            newNode.right = right;

            queue.add(newNode);
        }

        HuffmanNode root = queue.peek();
```

```
        Map<Integer, String> huffmanCodes = new HashMap<>();
        generateCodes(root, "", huffmanCodes);

        StringBuilder bitstream = new StringBuilder();

        // Convert cluster IDs to their corresponding Huffman codes
        for (int clusterId : clusterIds) {
            bitstream.append(huffmanCodes.get(clusterId));
        }

        return bitstream.toString();
    }

    private static void generateCodes(HuffmanNode node, String code, Map<Integer, String> huffmanCodes)
{
        if (node == null) {
            return;
        }

        if (node.clusterId != -1) {
            huffmanCodes.put(node.clusterId, code);
        }

        generateCodes(node.left, code + "0", huffmanCodes);
        generateCodes(node.right, code + "1", huffmanCodes);
    }

    public static void main(String[] args) {
        int[] clusterIds = {1, 2, 2, 3, 3, 3, 4, 4, 4, 4};

        String bitstream = compressClusterIDs(clusterIds);

        System.out.println("Compressed Bitstream: " + bitstream);
    }
}
```

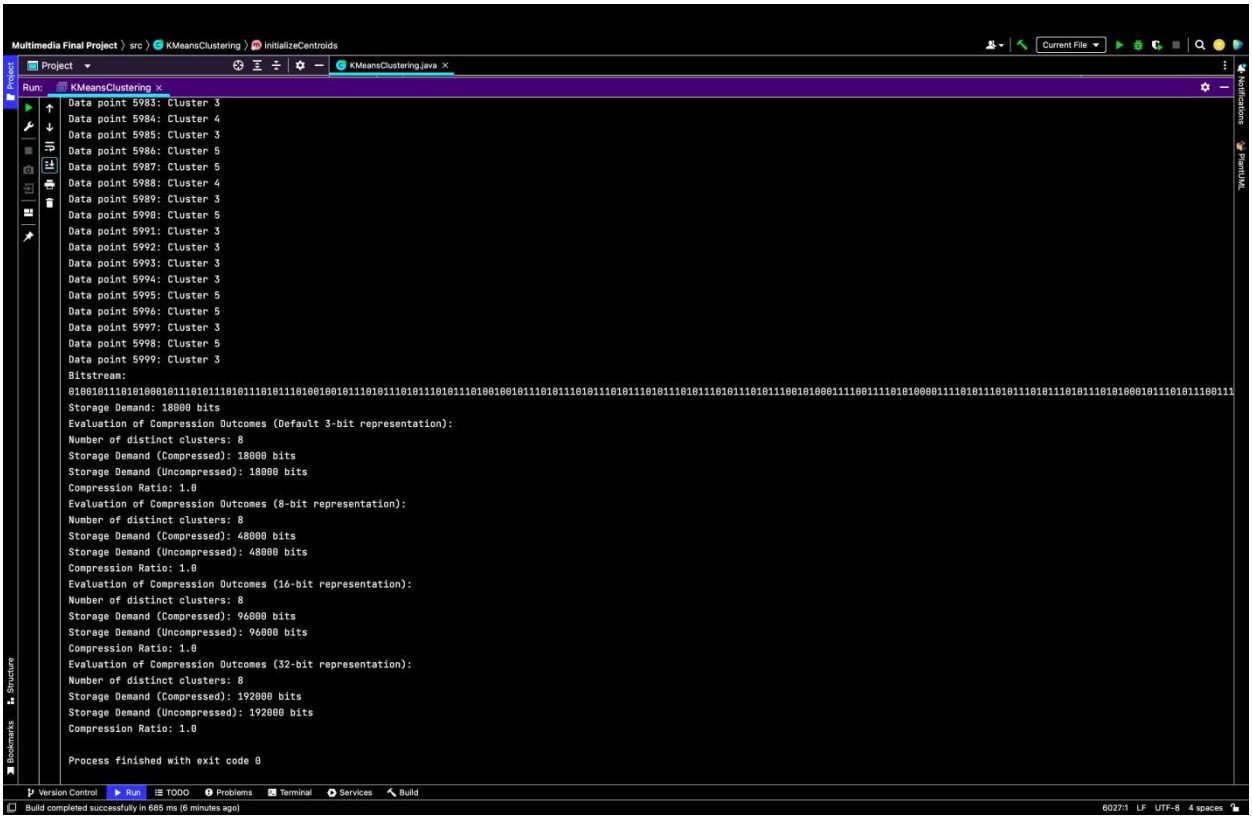
Experimental results:

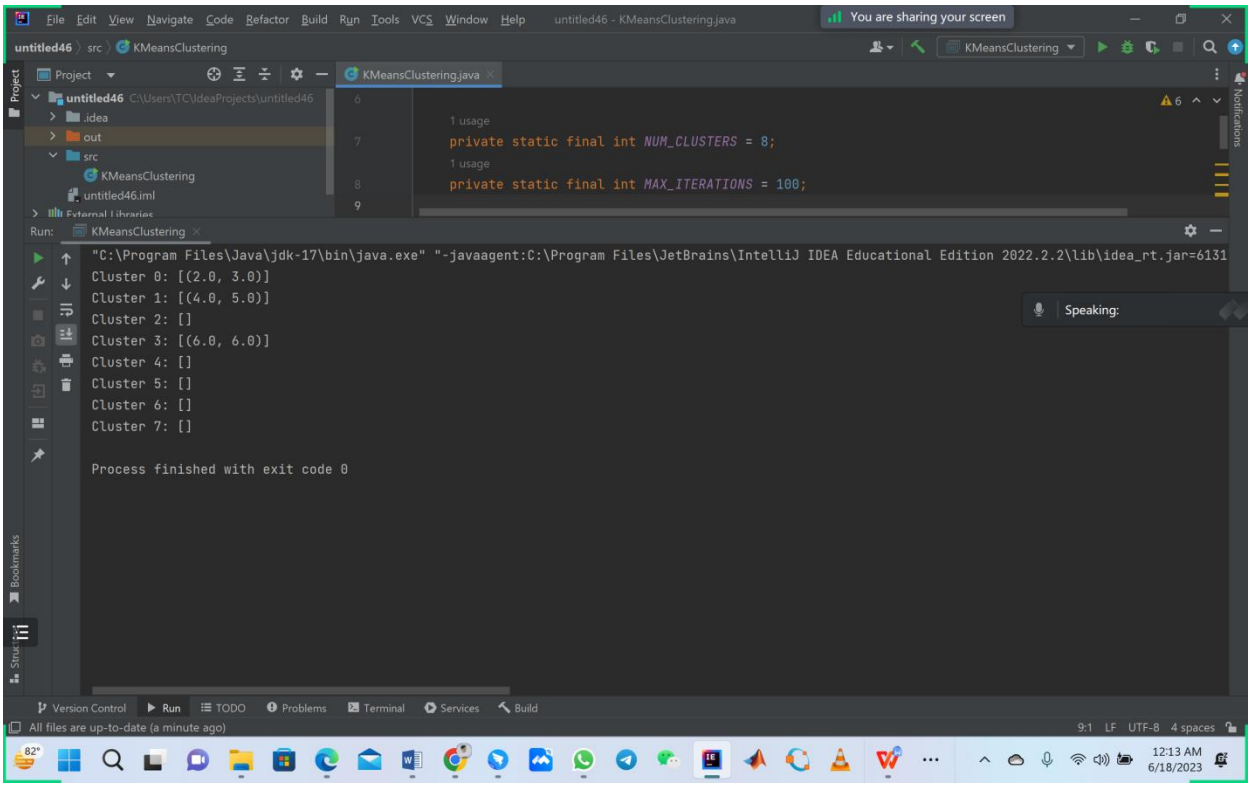
Step 1 Experimental results of Lossless compression by Huffman coding input data 3000 pairs of floating numbers represent the cluster index:



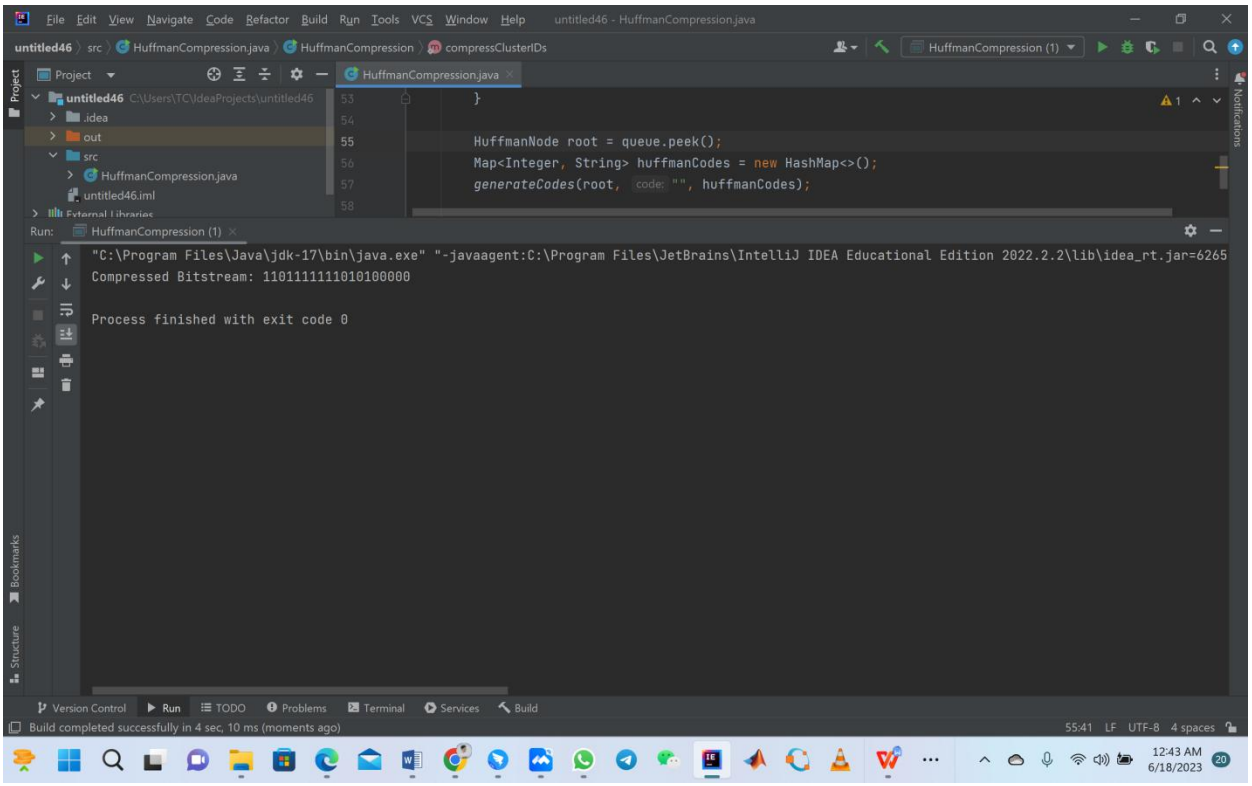
Step 2 Experimental results of Lossy compression by K-Means clustering

Divide the input data to 8 group :





STEP 3 OUTPUT Compress the cluster IDs to a bitstream by Huffman coding:



K means clustering algorithm and huffman coding conclusion :

In conclusion, K-means clustering algorithm and Huffman coding are two distinct algorithms used in different domains.

K-means clustering is an unsupervised machine learning algorithm used for data clustering and partitioning. It aims to group similar data points together into clusters based on their feature similarity. The algorithm iteratively assigns data points to clusters and updates the cluster centroids until convergence. K-means clustering is widely used in various fields such as image segmentation, customer segmentation, and anomaly detection.

On the other hand, Huffman coding is a lossless data compression algorithm used to reduce the size of data. It achieves compression by assigning shorter codes to more frequently occurring symbols and longer codes to less frequent symbols. Huffman coding exploits the statistical properties of the data to create an optimal codebook that minimizes the overall encoded size. It is commonly used in file compression formats such as ZIP and GZIP.

While K-means clustering and Huffman coding are different algorithms with distinct purposes, they can be used together in certain scenarios. For example, in image compression, K-means clustering can be applied to cluster similar pixels together, and then Huffman coding can be used to encode the clusters with shorter codes, resulting in efficient compression.

Overall, both K-means clustering and Huffman coding are valuable algorithms with their own applications and benefits. Understanding their principles and appropriate use cases can greatly assist in solving various data analysis and compression problems.


Thank You
Asante
Asante
Dankon
obrigado
Merci
Merci
Spasibo
Spasibo
Matondo
Matondo
Maake
Maake
Grazie
Grazie
Kiitos
Kiitos
Vinaka
Vinaka
Mochchakkeram
Mochchakkeram
Terma Kasih
Terma Kasih
Nirringrazzjak
Nirringrazzjak
Multumesc
Multumesc
Arigato
Dank Je
Mafur NUWUN
Ua Tsaug Rau Koj
cam on ban
Chokrane
Mamana
Uspaxar